UNIVERSIDAD DE CANTABRIA

PROGRAMA DE DOCTORADO EN CIENCIA Y TECNOLOGÍA

TESIS DOCTORAL

OPTIMIZACIÓN DEL RENDIMIENTO Y LA EFICIENCIA ENERGÉTICA EN SISTEMAS MASIVAMENTE PARALELOS

PHD THESIS

OPTIMIZING PERFORMANCE AND ENERGY EFFICIENCY IN MASSIVELY PARALLEL SYSTEMS

Raúl Nozal González

Dirigida por José Luis Bosque Orero

Escuela de Doctorado de la Universidad de Cantabria

Santander 2021

UNIVERSIDAD DE CANTABRIA

TESIS DOCTORAL

Optimización del Rendimiento y la Eficiencia Energética en Sistemas Masivamente Paralelos

PHD THESIS

Optimizing Performance and Energy Efficiency in Massively Parallel Systems

Raúl Nozal

Dirigido por José Luis Bosque

Grupo de Arquitectura y Tecnología de Computadores

Departamento de Ingeniería Informática y Electrónica

Noviembre de 2021

A la Familia,
a los seres queridos,
a las cosas buenas de la vida,
a los humanos comprensivos y de buen corazón,
a la gente trabajadora, humilde y honrada, a los que se esfuerzan,
a los que viven con pasión y curiosidad, especialmente a los que desean aprender,
y definitivamente, a los solidarios que luchan por construir un mundo mejor, más justo y sostenible.

Семье,
Любимым людям,
Хорошим вещам в жизни,
Понимающим людям с добрым сердцем,
Людям-труженикам, скромным и честным тем, кто не сдается
Тем, кто живет со страстью и любопытством и особенно тем, кто хочет учиться
И, определенно, тем солидарным, кто борется за построение лучшего мира, более справедливого и устойчивого.

إلى ا لأهل
إلى ا لمحبة
إلى ا لأشياء الجميلة في الحياة
إلى الرحماء أصحاب القلوب الطيبة
إلى المجتهدين المتواضعين الشرفاء
إلى الذين يعيشون شغف المعرفة والرغبة في التعلم
المتضامنون في كفاحهم لبناء عالم أفضل أكثر عدلًا واستدامة

为我的家庭，
为我的情人，
为好的东西生活里，
为工人们，谦逊的人们，为这些努力天天向上，为这些要学习，
致那些团结的人们为建设更美好，更公平和更可持续的世界而奋斗的人们。

To my Family,
to my loved ones,
to the good things in life,
to understanding and kind-hearted humans,
to hard-working, humble and honest people, to those who strive,
to those who live with passion and curiosity, especially those with the desire to learn,
and definitely, to those with principles of solidarity who fight to build a better, fairer and sustainable world.

Für meine Familie,
Für meine Lieben,
Für die guten Dinge im Leben,
Für die verständnisvollen und gutherzigen Menschen,
Für die fleißigen und ehrlichen Menschen,
Für jene, die nach mehr streben,
Für jene die leidenschaftlich und neugierig sind, besonders für jene, mit dem Verlangen zu lernen,
Und vor allem für jene, die solidarisch sind und für eine bessere, gerechtere und nachhaltige Welt kämpfen.

AGRADECIMIENTOS

Acknowledgements

Tras casi cinco años de esfuerzo, se da por cerrada otra etapa de mi vida. Bien es cierto que me hubiera gustado realizar una tesis doctoral como las de antes, pues me siento afín a ese espíritu de contribución vital, como proyecto de vida y con la consecución de un aporte que de verdad suponga un antes y un después. Ahora bien, también es cierto que si no me ponen límites, no quiero imaginar la de décadas y sacrificios que supondría. No obstante, la he tratado de elaborar con la mejor voluntad posible, fruto de ingentes esfuerzos y dedicación. A pesar de sentir que podía haber dado mucho más, mis condiciones materiales me han permitido compaginar la investigación con la docencia, parte de mis grandes vocaciones, pero sin perder el contacto con la perspectiva industrial y la aplicabilidad de cada contribución científica, el amplio abanico de aficiones, disciplinas y curiosidades por explorar, y en ocasiones, algo de vida social y personal. Este largo viaje ha estado acompañado de muchas grandes personas. He aquí mi reconocimiento y agradecimiento a todas ellas, a pesar de dejarme otras tantas, fruto de las intensas jornadas que han ido machacando mi organismo. Espero que me disculpen.

- ★ A Jose Luis. Mi tutor, consejero y director. Por acogerme, desgajar nuevos frentes, saltar conmigo a las trincheras y darme manga ancha, nunca limitando mis elecciones. Por entenderme y tener cerca de ti a alguien tan incandescente, obsesivo y que se desenvuelve entre montañas de ansiedad. Las cosas buenas de mí, tanto como persona, así como docente e investigador, las dejo a tu discreción ;). Hice muy bien escogiéndote para esta travesía doctoral y periodo de mi vida. Gracias.
- ❧ A María, piedra angular de mi futura Familia. Nuevo bastión y matriz engendradora. Proyectaremos nuevas generaciones y plasmaremos nuestras propias bases para tratar de hacer de este un mundo mejor. Tú has llevado el peso de esta tesis, me has ayudado, animado y comprendido. Soportas la carga y presión de mis altas jornadas y malas rachas, me has cuidado y mimado, empatizando con mi estrés e incertidumbre. Sin tu apoyo esto habría sido muchísimo más duro, triste, desesperante y vacío. Gracias.
- ❧ A mis progenitores y educadores, Don Luis Angel y Doña Julia, padre y madre. La Familia generadora y base de mi ser, origen de mi carácter y muchos de mis principios. Tanto de mis preocupaciones y ansiedad, como de mi alegría, curiosidad y capacidad de sacrificio. Habéis proporcionado un entorno propicio para desarrollarse, reconociendo el aprendizaje y el esfuerzo como algunos de los pilares de nuestro hogar.
  Gracias a los dos.

  (Madre, te toca ir en segunda posición, pues llevas ventaja tras dedicarte hace 8 años el software de reconocimiento de árboles y plantas que tanto me llevó)

- A los compañeros de investigación y gente de la academia, sobre todo a las personas de Arquitectura y Tecnología de Computadores, compartiendo seminarios, docencia y congresos, pero también entretenidas celebraciones, vermuts y cenas; y especialmente a Mon y Esteban, por su compañía y asesoramiento. Me arrepiento de no haber podido ir más a las clases de Mon en su día, pues con los años, cuanto más horas de clase doy a mis alumnos, más voy relacionando los contenidos con anécdotas propias, tanto de academia como de industria, recordándome más a su forma de impartir clase. Por cierto, Enrique, por si nunca te lo he dicho, una pena no haberte tenido como profesor, seguro que me hubieses gustado. A Rafa, Fernando, Chus y Carmen, seguro que si hubiese dado todas las asignaturas relacionadas con computadores con mi nivel actual de madurez y aprendizaje, las disfrutaría aún más. También a mis compañeros de trinchera, con los que fui compartiendo despacho y progresiva amistad, ahora ya todos liberados: Pablo, Borja, Iván y Mariano. Los paseos, bromas, charlas, discusiones y cervezas siguen cayendo, y espero que así continúen. También a Cristóbal, por compartir enseñanzas y técnicas. Me siento orgulloso de que seas uno de mis primeros *evangelizados* al Óxido.
- Al resto de compañeros de docencia y entorno universitario, especialmente a Javi (Gutiérrez), Jose Ángel (Herrero), Pablo (Abad), Amparo y Alex. A los colegas del grupo de Arquitectura de Computadores de Zaragoza, porque se hace muy fácil colaborar con vosotros y porque allí me sentí como en mi tierruca, salvo por el frío, el viento y la aridez. Gracias, Darío, Rubén, María y Angélica. Saltando a tierra de campos y compartiendo múltiples congresos, a Arturo, por el buen trato y su jocosa personalidad. Hablando de cachondeo, a los albaceteños del grupo de ATC con los que he compartido varias salidas nocturnas y muy buen rollo, además de maravillosos descubrimientos culinarios, como el queso al romero. A Francisco, Pedro, Jesús y Paco.
- A mis amigos más duraderos e importantes. Fran, es todo un placer compartir tantas experiencias, formas de ser y afinidades, tanto en lo personal como en lo profesional. El río, los cafés y cervezas, los periodos de trinchera y las palizas; muchos buenos y duros momentos. Jou, más de media vida juntos, y esto parece que continúa igual de bien, compartiendo hasta nuevos gustos. Debería aprender de ti a ser más sereno y aislar los «problemas» para mejorar en salud. Tienes un gran poder.
  Espero que nuestra amistad continúe por siempre. Gracias.
- Jaled, como representante de los nuevos amigos. Un gran *descubrimiento* y una bellísima persona. Poco más de tres años de amistad, y como si llevásemos desde siempre. Que suerte tuve al conocerte. Gracias por soportar mis consultas de electrónica y sistemas, pero es que eres mi Ingeniero Electrónico favorito. Echo de menos nuestras largas conversaciones en la intemperie, así que habrá que retomarlas algún día. Gracias.

▲ A mi familia ampliada, tanto las abuelas, como tíos y primos. Es un placer estar con vosotros, sintiéndome siempre tranquilo y alegre. Especial mención a Javi y Jose Ángel, carpintero y marinero, ambos lobos, uno de tierra y otro de mar. Gracias.

▲ Por supuesto, a mi tío salmantino, Don Miguel. Creo que no conozco a una persona tan peculiar. Eres una persona muy inteligente, pero destacaría tu experiencia de vida, tu cocina y tu vino. Si hubiese sido hijo tuyo, probablemente hubiese tenido carácter de superviviente, astuto y *ratero*, en el buen sentido del término. Valoro cómo de consecuente eres con tu pensamiento. Gracias.

♢ A los amigos del *viernes*, Borja, Chapi, Ksenia y Jessy. Me habéis ayudado mucho desconectando periódicamente, siempre bien acompañado, tanto por vosotros, como por buenos costillares y *valencianos*. Gracias.

♢ A los amigos del pueblo, los de siempre. Yonny, Héctor, Toci, Miguel, Guio y Juako, sois grandes y (generalmente :P) humildes personas, y me siento agradecido de estar junto a vosotros. Pocas cosas me gustan tanto como estar metiéndonos juntos unas buenas nachadas y hamburguesas del Marcos un viernes, y unas pizzas tradicionales en el Medieval el sábado. Eso sí, luego pasa factura, pero en buena compañía. Gracias a todos.

♢ A amigos fruto de aquel paso por Topografía, especialmente a Salva e Iñaki. Salva, un verdadero placer debatir y discutir (incluso con *pólvora* por medio), tu compañía y amistad siempre es bienvenida. Además, es un placer pasear por Burgos de tu lado. Iñaki, probablemente la persona más bonachona de esta lista, y compitiendo en tranquilidad con Jou. Como echo de menos nuestros tiempos de café y apuesta bien rodeados. Gracias a los dos.

♦ A las nuevas amistades, como mis leoneses Yony y Jessy. Sois unas personas encantadoras, y he disfrutado mucho de nuestras conversaciones y compañía. Espero que sigamos manteniendo esta relación tan buena y compartiendo futuras barbacoas y experiencias gastronómicas. Gracias.

△ A los colegas de la carrera con los que todavía se mantiene la llama viva. Especial mención a Gracy y Aída, así como a David. A ver si seguimos viajando y compartiendo gustos. Va siendo hora de probar el *airsoft*. Mencionando los estudios y el deporte, una mención para mi colega Aitziber, la *diseña-madora*, Aitziber. Gracias.

△ A las amistades por retomar. Pablo («Punto»), disfrutaba de tu compañía, y todavía recuerdo la efervescencia cuando hubo el debate *funcionariado-autónomos*. Aportas mucho y hace varios años que no nos vemos, por lo que ya es hora. Gracias.

★ A los amigos internacionales, ya con una trayectoria de casi una década. Peter, ahora padre *redneck*, muchas gracias por compartir tan buenos momentos. Gracias por aguantar mis duros periodos de escritura de papers en mitad de nuestras vacaciones. Anhelo nuestras rutas y cervezas acompañados de un buen *Haxe*. Gracias.

★ A las antiguas amistades, ahora retomadas. Es un placer poder volver a estar con vosotros y conseguir aislarme de mi rutina tan intensa. A Mario, un *cabra loca* que sigue siendo tan buena persona como siempre. A Borja y Manu, porque a pesar de ser unos tipos duros y repartir a diestro y siniestro (algunos se merecen un cachete, lo reconozco), tenéis coco y profundidad de ideas. Es un placer compartir análisis y debatir. Creo que ya es tradición una celebración anual con un buen chuletón en Ucieda. Un placer. Gracias.

★ A los colegas del gimnasio, y en especial a Javi, mi entrenador y amigo, por soportar mis burradas durante los primeros 3 años. Esos bombardeos con *HardBass* siempre se recordarán, y si no me he destrozado más el cuerpo es porque vino la pandemia mundial. Echo de menos nuestros pucheros de cocido y carajillos de anís en la Vega de Pas. Gracias por hacer del gimnasio mi segundo hogar.

✦ A aquellas amistades que trascienden oficios y relaciones. Susana, disfruto muchísimo de nuestro tiempo juntos, pateadas e intercambio de pensamientos. Tienes todavía mucho que enseñarme. Me apasionaba cómo dabas historia en bachiller, y me sigue apasionando cómo la cuentas hoy día. Espero pronto poder sacar algo de tiempo para aprender de arte. Gema, mi enfermera favorita y grande conversadora. Espero que podamos retomar algún día aquellas conversaciones profundas, pues eran inspiradoras. Gracias a las dos.

✦ A los compañeros y amigos del *Club de lectura*, en especial a Óscar y Chopi. Me complace poder compartir esos periodos juntos, aislándonos del resto de vicisitudes. Oscar, eres de los tíos más majos que conozco, y ha sido una agradable sorpresa oír tus reflexiones. Chopi, qué ganas de poder volver a juntarnos para nuestro domingo de comilona. Señor de oficios y principios, sin olvidar su naturaleza. Echo de menos nuestros trotes y conversaciones. El tío más agradable de todo Ramales. Gracias a todos.

✧ A todas las amistades de Ucrania, pues hicisteis una maravilla el tiempo que pasé allí. A mis amigos rusos y ucranianos, en especial a Sergiy (Gogol :P), que parece que nuestros caminos están *destinados* a cruzarse en diferentes localizaciones, compartiendo muy buenos momentos (y techo); pero por supuesto también a Alexander (Kyriy), Anastasiia (Kasprova), Alina (Kucheriavenko), Marianna (Kovalova), Vadim (Popov), Ramil (Nabiev), Kamil (Tokmakov), Yevgeniya (Kovalenko) y Anastasiia (Shamakina). Gracias a todos.

- A mis compañeros de estancia en el HLRS. A Jose Gracia, el jefe y compañero de trabajo que siempre quise tener. Un gusto compartir filosofía y «banquetes». A Thomas (Kloss), Joseph (Schuchart), Christoph (Niethammer), Venkata (Ayyalasomayajula), Kingshuk (Halder), Fabian (Dembski), Denis (Altmann), Soheil (Soltani), Alexey (Cheptsov), Ayse (Bagbaba) y Dmitry (Khabi). Siempre que pueda seguiré enviando sorpresitas culinarias. Ya sabemos que los sobados, las quesadas y el jamón apasionan tanto a nacionales como extranjeros. Gracias a todos.

- A Alexandros (Andreou), Stathis (Dimitrios), Georgios (S. Bousdras), Kleovoulos (Kalaitzidis) y Anna (Molinet), por nuestra afinidad, visitas, juergas y tertulias, tanto en Italia como Barcelona. Hicimos muy buenas migas, y la verdad es que nunca hubiera imaginado sentirme tan bien entre griegos, cretenses y chipriotas. He aquí nuestros lazos como orgullosos *mediterráneos*. Por supuesto, sin olvidar a Xavi (Salazar), ahora ya padre y comunicador. Gracias a todos.
- A Pilar (Alesón), de la EDUC, por ayudarme tanto y no protestar ante tanta pregunta, correo y llamada. Hemos conseguido paralelizar al máximo el proceso, todo un reto. Periodos muy estresantes, por los que te compensaré.
- A Fernando, el profesional del servicio de reprografía. Has soportado todo tipo de preguntas y detalles, aguantando a una persona tan obsesiva como yo, incluso con los detalles más «ridículos». Este documento no podría salir tan bien si no es por tu labor al otro lado del mostrador. Gracias.

- A la gente del campo y del *campo*, con los que comparto formas de vida y aficiones. Gracias a todos.
- De todos los lugares de travesía durante la tesis, a aquellos que he podido explorar y disfrutar, mostrándome la grandeza del hombre y la naturaleza, ayudándome a valorar más bondades de la vida. A Salamanca, Córdoba, Toledo, Segovia, Burgos, Mérida, los Alcornocales, Barbate, Cádiz, Madrid, Barcelona, Las Navas, Herrera de Pisuerga, los Valles Pasiegos, y sobre todo, Ramales de la Victoria. Al Palatinado, especialmente sus colinas y castillos, desde Speyer hasta Eußerthal pasando por Göcklinguen y Landau in der Pfälz. A Baviera y Austria, sobre todo los Alpes y Königssee. A Ucrania, especialmente a Lviv. A Irlanda, en su totalidad, pero particularmente a Dublín y el Munster, tanto Clare-Galway (Moher) como Cork.

Gracias a todos.

ABSTRACT

In recent years, with the emergence of a diversity of computational devices and architectures, it is becoming possible to tackle an interesting variety of problems. The constant quest to improve the performance and energy efficiency of computational platforms has embraced the concept of heterogeneous systems, offering solutions never seen before. The present and coming decades of computing cannot be understood without these types of massively parallel systems.

With the emergence of new specialized devices, a world of possibilities is opening up, allowing the combination of HPC devices, such as multi-core CPUs powered by integrated graphics processors, FPGAs, discrete GPUs, DSPs and even specialized accelerators for machine learning inference, graph analysis and visual processing, termed xPU devices. This heterogeneity is being observed in more and more places, ranging from adding specialized units to the execution cores within GPU architectures, to building SoCs with independent accelerator units along with general-purpose processors on the same die. All these solutions have in common their excellent cost-performance ratio and energy efficiency, which allow the acceleration of a wide range of massively data-parallel applications, such as deep learning, big data analysis, sound processing, video streaming, image processing or financial applications, among others.

However, hardware heterogeneity complicates the development of efficient and portable software, especially when specialized components from various suppliers are used, since many require their own programming languages. Languages and frameworks converge in programming models seeking to be progressively more expressive and abstract. OpenCL emerged as an open standard programming model for writing portable programs across heterogeneous platforms. However, it has a very low level of abstraction and leaves to programmers the complex tasks of problem partitioning and data transferral between the CPU and devices. Nevertheless, it continues to be the de facto open standard, and new manufacturers continue to offer OpenCL drivers for their devices.

Moreover, market trends and industrial applications indicate a strong predominance of languages such as C++ that call for higher level alternatives to OpenCL. For instance, SYCL is a cross-platform abstraction layer that originally builds on top of OpenCL, enabling the host and kernel code to be contained in the same source file . In this context, solutions such as oneAPI and the DPC++ compiler have emerged, offering a set of domain-specific modules and C++ support for programming heterogeneous systems, although initially limited to Intel architectures.

All these systems are based on the host-device programming model, thereby offloading computational regions to the system accelerator and leaving the CPU for management tasks. In more sophisticated cases, task-based offloading is carried out, where each device executes an independent function, but having to be orchestrated and managed by the developer along with a capable runtime. On the other hand, the co-execution technique allows all devices, including the CPU, to operate on the same problem, consuming less time and energy to

solve it. Again, the problem is that the programmer must take care of all the host and device management, penalizing maintainability and increasing engineering efforts.

In order to exploit co-execution, it must be easy to use, regardless of the number and type of the devices in the system. Moreover, the code must be portable, avoiding having to transform it in order to operate with other devices and systems. The programmer has to be abstracted from the underlying architecture and the optimal working strategies for each device or manufacturer, as well as the partitioning of the workload among the devices. This is definitely a complex task and needs to be done in the best possible manner, adapting to the behavior of the problem and the heterogeneous conditions of the system, in order to guarantee the best performance portability.

This dissertation offers different contributions to improve the performance and energy efficiency of these massively parallel systems, making proposals that address generally conflicting aspects, such as usability improvements and increased abstraction with performance, energy efficiency and optimization. Two proposals for runtime systems with radically different approaches are conceptualized, designed and evaluated.

The first, *EngineCL*, presented in Chapter 3, is an OpenCL-based runtime and API that facilitates very high-level operations, greatly improving programmability, while ensuring maximum portability and compatibility between heterogeneous systems, reducing all the management that a programmer must do. In addition, it provides a extensible pluggable scheduler system that enhances the performance of all kinds of problems in nodes with devices of all types and manufacturers.

Chapter 4 describes two major extensions performed to EngineCL that enhance its capabilities, revealing the versatility of its design. The first improves the runtime and one of its load balancing algorithms to facilitate time-constrained executions, one of the most counterproductive scenarios for these programming paradigms and accelerators. The second one allows EngineCL to efficiently execute problems for which OpenCL technology is not appropriate, as part of a real scientific software used for molecular dynamics simulations. It is extended to support hybrid programming models, exploiting the system resources efficiently while preserving the rest of the runtime functionalities.

Finally, Chapter 5 presents the second runtime proposal, *CoexecutorRuntime*, that takes on a different perspective. It focuses on providing oneAPI technology with co-execution support, enabling and optimizing the use of dynamic strategies to ensure efficient adaptability to irregular problems. Although it incorporates a number of extensions that bring programmer closer to the domain of the problem, its design principles allow extensibility while offering a oneAPI/SYCL-compatible API.

These contributions facilitate programmability, reduce integration efforts and achieve efficient exploitation of heterogeneous systems. In short, these multi-objective proposals help to extract the maximum performance and energy efficiency out of current and future massively parallel systems.

RESUMEN

Spanish extended abstract

La computación heterogénea es un término que ha ido cogiendo relevancia en los últimos años, impregnando tanto congresos académicos como eventos industriales, por no decir un sinfín de aplicaciones y usos con implicaciones tanto en software como en hardware. Este término, acotado al campo de aplicación de esta tesis, se refiere a la ejecución de aplicaciones en una plataforma de cómputo compuesta por dispositivos de procesamiento con diferentes arquitecturas, siendo esta diversidad lo que proporciona un gran potencial, aunque a la vez incrementa la complejidad del sistema y su programación.

Con la aparición de nuevos dispositivos especializados, se abre un mundo de posibilidades tanto en la computación de altas prestaciones (HPC) como en soportes embebidos y computadores domésticos. La combinación de dispositivos no para de crecer, incluyendo CPUs complementadas por procesadores gráficos integrados, FPGAs, GPUs discretas, DSPs e incluso dispositivos XPU, especializados en inferencia de aprendizaje automático (TPUs), reconocimiento de patrones (APs), y aceleradores de procesamiento visual (VPUs). Esta heterogeneidad se observa cada vez en más lugares, desde la incorporación de unidades especializadas a los núcleos de ejecución dentro de las arquitecturas, hasta la fabricación de *SoCs* con unidades aceleradoras independientes y procesadores de propósito general en el mismo chip, ofreciendo soluciones muy versátiles.

Desde el punto de vista hardware, es algo claramente beneficioso, pues cada dispositivo de cómputo es más adecuado para diferentes tipos de aplicaciones. Esta eficiencia puede ser tanto en términos de rendimiento, como en menor consumo energético y su consecuente implicación en la reducción de costes. Estas características tan buenas de eficiencia, costes y versatilidad son necesarias en la era del *big data*, la inteligencia artificial, y las simulaciones científicas en *clusters* con grandes consumos energéticos. En definitiva, en el camino a la computación exascale. De esta forma, siendo capaces de utilizar esta heterogeneidad y haciendo que cada dispositivo especializado contribuya en pos de una labor más ambiciosa, se facilitará alcanzar los beneficios esperados, sobrepasando las limitaciones actuales.

Por otro lado, desde la perspectiva software, es donde se encuentran los mayores retos para explotar al máximo las ventajas de este tipo de computación. La programación se complica radicalmente, alejandose de la computación tradicional homogénea. Para aprovechar los sistemas es determinante conocer los tipos de dispositivos de cómputo, su arquitectura y su modelo de programación. El código y los propios algoritmos se ven afectados por estas características, complicando la portabilidad y pudiendo aparecer nuevos cuellos de botella antes no presentes. Las vicisitudes son diversas, yendo desde las sobrecargas en la gestión de los dispositivos y la comunicación entre diferentes unidades de cómputo, pasando por la transferencia de datos entre memorias y sus patrones de acceso, hasta las optimizaciones arquitecturales y problemas de compatibilidad tecnológica.

Estos problemas de portabilidad software se acucian especialmente al utilizar componentes especializados de varios fabricantes, ya que muchos requieren sus propios modelos de

programación. OpenCL surgió como un lenguaje de programación estándar abierto para escribir programas portables en plataformas heterogéneas. Sin embargo, tiene un nivel de abstracción muy bajo y deja en manos de los programadores la partición y transferencia de datos y resultados entre la CPU y los dispositivos. Además, el soporte por parte de los distintos proveedores ha sido variable tanto en compatibilidad como en rendimiento. Aun así, sigue siendo el estándar abierto de facto, y los fabricantes siguen incorporando controladores de OpenCL en plataformas innovadoras.

Además, las tendencias del mercado y las aplicaciones industriales indican un fuerte predominio de lenguajes como C++, favoreciendo alternativas de mayor nivel de abstracción. Por ejemplo, SYCL es una capa de abstracción multiplataforma que originalmente se construye sobre OpenCL, permitiendo que el código a ejecutar en el host (CPU) y en el dispositivo estén contenidos en el mismo archivo fuente, gestionando las dependencias en un grafo de tareas. En este contexto, han surgido soluciones como Intel oneAPI y la implementación DPC++ de SYCL, que ofrecen un conjunto de módulos específicos de dominio y soporte C++/SYCL para la programación de sistemas heterogéneos, siendo adaptado cada vez más por distintos fabricantes.

Sin embargo, todos estos sistemas se basan en el modelo de programación *host-device*, favoreciendo la descarga de regiones computacionales (*kernels*) al acelerador del sistema, dejando la CPU para las tareas de gestión. En casos más sofisticados, se lleva a cabo una descarga basada en tareas, en la que cada dispositivo computa una función independiente, pero que debe ser orquestada y gestionada por el desarrollador junto con un runtime que posibilite dicho modo de ejecución. Por otro lado, la técnica de co-ejecución permite que todos los dispositivos, incluida la CPU, operen sobre el mismo problema para aprovechar el uso de los recursos del sistema, consumiendo menos tiempo y energía para resolverlo. El problema es que el programador tiene que encargarse de toda la gestión manualmente, penalizando la mantenibilidad y aumentando los esfuerzos de ingeniería.

Para explotar la co-ejecución, debe ser fácil de usar, independientemente del número y tipo de dispositivos del sistema. Además, el código debe ser portable, evitando tener que transformarlo para operar con otros dispositivos y sistemas. El programador debe abstraerse de la arquitectura subyacente y de las estrategias de utilización óptimas de cada dispositivo o fabricante, así como de las decisiones de partición del problema y de asignación de la carga de trabajo entre los dispositivos, logrando una planificación y un equilibrio de carga eficientes. Se trata, sin duda, de una tarea compleja que debe realizarse de la mejor manera posible, adaptándose al comportamiento del problema y a la heterogeneidad del sistema, para garantizar la mejor portabilidad del rendimiento.

Esta tesis presenta un conjunto de contribuciones enfocadas a posibilitar y optimizar la co-ejecución en sistemas heterogéneos, todo ello aliviando al programador en la toma de decisiones y fomentando una alta usabilidad. Se conceptualizan runtimes que abstraen los sistemas subyacentes, orquestan todas las operaciones con los dispositivos existentes y facilitan la portabilidad de rendimiento. Además, se incorporan, evalúan y optimizan las estrategias

de planificación, así como los algoritmos de balanceo de carga para poder explotar eficientemente los problemas a resolver. El enfoque de las propuestas es siempre multi-objetivo, tratando de exprimir el máximo rendimiento y eficiencia energética, así como mejorando la usabilidad. El propósito no es otro que aprovechar todos los dispositivos de cómputo disponibles para resolver las tareas de forma cooperativa. Las experimentaciones y validaciones efectuadas a lo largo de la tesis se han realizado siempre sobre máquinas reales. Además, durante la conceptualización, depuración e implementación de las propuestas se han utilizado otros sistemas y plataformas no presentes explícitamente en este documento. Sin embargo, han servido para verificar las implementaciones, ofrecer más evaluaciones en otras arquitecturas, ampliar la compatibilidad entre fabricantes, optimizar funcionalidades e incluso ampliar el análisis de resultados al detallar validaciones concretas de este documento. Aun así, el mayor aporte ha venido dado tanto por las máquinas del *grupo de investigación de Arquitectura y Tecnología de Computadores (ATC)* de la Universidad de Cantabria, como el *Centro de investigación internacional para la Computación de Alto Rendimiento (HLRS)* de Stuttgart, en Alemania. Adicionalmente, fruto de la estancia de investigación en el centro HLRS se ha podido contar tanto con el soporte científico y técnico, como con el simulador de dinámica molecular *ls1-MarDyn*, posibilitando una de las contribuciones propuestas.

En primer lugar, se propone EngineCL como runtime flexible y portable para sistemas heterogéneos. Es ideado, diseñado e implementado con dos principios en mente, la usabilidad y el rendimiento. Se nutre de todo el potencial de OpenCL, obteniendo una alta compatibilidad, pero mejorando su aplicabilidad al fomentar técnicas que potencian su rendimiento. El motor ha sido ideado para explotar de forma sencilla todos los dispositivos y arquitecturas diversas que existan en un nodo heterogéneo. Su arquitectura y principios de diseño han sido conceptualizados y optimizados con el objetivo de facilitar la programabilidad y mantenibilidad de los programas. Se ofrece una API simplificada sobre el framework de OpenCL, facilitando su uso y explotación, teniendo en cuenta las características comunes de ejecución de programas HPC y con necesidades de aceleración. Además, el runtime es modular y extensible, incluyendo también un sistema de *plugins* para planificadores altamente optimizado para reducir los overheads de gestión. Para demostrar la efectividad de EngineCL, se ha validado tanto bajo criterios de mantenibilidad como de rendimiento y eficiencia energética, utilizando múltiples tipos de dispositivos y arquitecturas. Los resultados experimentales muestran una sobrecarga prácticamente despreciable junto con una alta usabilidad comparado con OpenCL, además de ofrecer un rendimiento muy cercano al máximo teórico, aprovechando eficientemente el sistema heterogéneo.

La segunda contribución extiende e integra el runtime EngineCL en dos escenarios diferentes, incluyendo una aplicación real. Se muestra la versatilidad del runtime diseñado, y cómo se adapta a situaciones para las que no fue ideado inicialmente. Por un lado, EngineCL ha sido extendido y optimizado para adaptarse a servidores de servicios y nodos de tipo *commodity* donde las ejecuciones están limitadas por tiempo. Estas peticiones de cómputo tienen generalmente una duración de pocos segundos, además de ofrecer modos de actuación

contraproducentes para los runtimes pesados y donde los drivers tienen elevados periodos de inicialización. El modelo *host-device* suele sufrir penalizaciones ante este tipo de escenarios, por lo que en esta propuesta se detallan las optimizaciones realizadas sobre el motor para hacer frente a estas situaciones. Además, se realizan optimizaciones algorítmicas para mejorar la eficiencia al explotar las distintas arquitecturas presentes en el nodo. Por otro lado, se realiza una integración que combina distintos modelos de programación, realizando una co-ejecución híbrida y extendiendo los núcleos de ejecución del runtime, explotando una aplicación real utilizada en el centro de investigación HLRS. Los problemas de dinámica molecular del simulador sufren una fuerte penalización al utilizar OpenCL, invalidando toda posibilidad de mejorar el rendimiento al utilizar los distintos dispositivos del nodo, limitándose a un modelo *host-device* de descarga a aceleradores. Por este motivo, se extiende EngineCL para dar soporte a formas híbridas de co-ejecución que permiten explotar los problemas apropiadamente. La propuesta consigue mantener consistencia con la API ideada, a la vez que se dota de mayor funcionalidad a los tipos de ejecución soportados, aumentando la versatilidad del runtime. Gracias a esta propuesta es posible combinar múltiples tipos de ejecución, manteniendo el soporte al resto de funcionalidades del runtime, incluidos los algoritmos de planificación. Asimismo, ahora se le proporcionan al programador mecanismos para poder establecer diferentes tipos y orígenes de kernels de lanzamiento, incluso de forma dinámica en tiempo de ejecución. Esta propuesta facilita el camino para poder integrar otras tecnologías y modelos de programación, como por ejemplo CUDA, código máquina de aceleradores u otros modelos especializados. En ambas propuestas se consigue mejorar el rendimiento y la eficiencia energética tras la integración producida sin penalizar el diseño y API original, posibilitando la ejecución de problemas de menor duración en el primer caso, y ampliando su versatilidad a la vez que se explota eficientemente la CPU del sistema junto con los nuevos aceleradores en la segunda integración.

La tercera contribución es la creación del CoexecutorRuntime para la computación heterogénea basada en estándares modernos. Se facilita la co-ejecución de sistemas heterogéneos con soporte a la tecnología oneAPI, por lo que la API proporcionada es de alto nivel, compatible con C++ y SYCL. La propuesta proporciona un diseño cercano al dominio del problema, combinando la facilidad en la gestión con las características proporcionadas por oneAPI. Se sube el nivel de abstracción, enmascarando la gestión de los dispositivos, los datos y la planificación, pero se mantienen los principios de un único código fuente y la versatilidad de C++ en la definición de las operaciones y su arquitectura. Las principales hitos de este motor son su facilidad de modificación y la adecuación a mecanismos dinámicos de planificación, previamente inexistentes. La limitación de oneAPI respecto a la co-ejecución ha sido superada gracias a la incorporación de una arquitectura asíncrona multi-hilo que explota la ejecución simultánea de dispositivos, beneficiándose del potencial de las estrategias dinámicas e implementando diversos algoritmos de balanceo de carga. Además, cualquier programador de C++/SYCL puede incorporar nuevas características sin depender de otras tecnologías, por ser una arquitectura desacoplada y sin dependencias, facilitando su integra-

ción en todo tipo de herramientas y softwares. El motor se beneficia de nuevas arquitecturas hardware sin necesidad de realizar cambios, siempre que se proporcione un driver compatible. Por otro lado, el motor ha sido diseñado a medida que oneAPI evolucionaba y se liberaba, por lo que su arquitectura ha sido ideada pensando en la adaptabilidad a los cambios. Por este motivo, las extensiones de oneAPI han sido incorporadas, extendiendo la API para que el programador pueda explotar dichas funcionalidades. Además, la propuesta realiza una exploración y validación del comportamiento en diversas arquitecturas, tanto *commodity* como HPC, resaltando el rendimiento y la eficiencia energética conseguida, sobre todo al utilizar la memoria unificada entre CPU y GPU, así como los algoritmos de planificación adaptativos.

Aunque esta tesis haya obtenido unas propuestas claras con resultados tangibles tanto en términos de programabilidad como rendimiento y eficiencia energética, el camino exploratorio continúa. El problema es ambicioso y solo hace falta observar cómo se lleva décadas proponiendo soluciones arquitecturales, modelos de programación, lenguajes de explotación paralela, frameworks de abstracción y runtimes de ejecución, perfilando potenciales soluciones, nutriéndose unas propuestas de otras, convergiendo y construyendo soluciones cada vez mejores. Aun así, se destacan tres grandes frentes de actuación a partir de esta tesis. En primer lugar, hay múltiples líneas futuras de relativas a los runtimes y modelos de programación, pues la diversidad es enorme. Es importante explorar nuevas propuestas y extensiones, tanto de runtimes previos como tecnologías nuevas, introduciendo aquellas más relevantes con el objetivo de construir soluciones integrales lo más versátiles posibles. Lo que a priori parece una limitación en poco tiempo se convierte en un nuevo framework de cómputo heterogéneo aplicable a muchas arquitecturas. Sin la suficiente eclosión y combinación de posibilidades no se genera tracción y adopción, proporcionando innovaciones nunca vistas. Los modelos híbridos de programación son la antesala a nuevos lenguajes y a la incorporación de funcionalidades en motores existentes. Por ejemplo, considerando los dos motores ideados, se podría diseñar y explotar una solución combinada entre ambas propuestas, con lo mejor de cada una de ellas: la compatibilidad y usabilidad proporcionada por EngineCL junto con la flexibilidad y fácil adaptabilidad de CoexecutorRuntime. Por otro lado, la segunda vertiente se centraría en el diseño de algoritmos de planificación, especialmente diseñados para explotar las diferencias arquitecturales de los tipos de dispositivos. Estos algoritmos de balanceo de carga no están desacoplados de las implementaciones, por lo que es importante elaborar propuestas considerando sus runtimes y tecnologías de ejecución desde la propia concepción, empíricamente validadas y no siendo consideradas como algoritmos puramente teóricos. Por último, un campo de aplicación futuro de gran impacto estudiaría aprovechar las propuestas construidas amplificando su ámbito de aplicación, elevando el potencial de aprovechamiento al ser extendidas para *clusters* heterogéneos. La integración entre modelos de programación híbridos y extensión de núcleos de ejecución da un primer paso en esta dirección, salvo que se encuentra acotada a un solo nodo. El motor debería extenderse para posibilitar replicarse entre nodos, aumentando las capacidades

de comunicación y distribución de la carga, a la vez que se explotan patrones de cómputo híbridos con tecnologías bien asentadas, como MPI. Esta solución podría ser abordada considerando todas las tecnologías y lenguajes como partes internas, con el objetivo de establecer nuevos patrones y optimizar su uso, descargando al programador de las decisiones complejas. Además, podrían considerarse algoritmos de planificación a dos niveles, los que reparten el trabajo entre nodos, e intra-nodos, los que lo hacen dentro del nodo, cada uno atendiendo a consideraciones distintas para exprimir al máximo el conjunto de nodos.

Finalmente, teniendo en cuenta que la computación heterogénea ha venido a quedarse, convirtiéndose en la norma y no en la excepción, es importante considerarla prioritaria y ofrecer soluciones que permitan exprimir todo el potencial que ofrece. Esta búsqueda constante por mejorar el rendimiento y la eficiencia energética está ofreciendo soluciones nunca vistas gracias a estos sistemas masivamente paralelos. El futuro de la computación no puede entenderse sin ellos, pero es necesario ofrecer soluciones integrales de alta usabilidad, eficientes, con posibilidad de extensión y portables entre todo tipo de dispositivos y sistemas.

Palabras clave

Computación heterogénea · Co-ejecución · HPC · Programación paralela · Portabilidad del rendimiento · Aceleradores · Runtimes · Usabilidad · Mantenibilidad · Balanceo de carga · Planificación · OpenCL · Intel oneAPI · SYCL · C++ · Diseño API · Arquitectura software · Lenguajes de programación · Paradigmas de programación

Contents

List of Figures

List of Tables

List of Equations

List of Code Listings

Introduction

CHAPTER 1

Introduction

Decades go by and computational performance demands continue to grow, regardless of the type of device, including those with smaller form factors and energy efficient needs. Over the years, the industrial response has been driven by increasing processor power, usually by raising the clock frequency and adding many more transistors. However, being aware of the physical limits that processors are reaching, it has become necessary to achieve such quotas of computational demand and energy efficiency through the adoption of heterogeneous computing. The ubiquity and variety of machines and systems composed of processors of various kinds, gives a unique specialization to solving specific tasks with maximum performance and low energy consumption. As a result, in recent years the variety of devices and accelerators present in systems has been increasing,

However, heterogeneous computing offers multiple challenges that transcend various areas of knowledge, from software engineering to computer architecture, affecting both academia and industry. Academia, due to the feasibility of establishing portable solutions applicable in industry, as well as the existing technological variety and the complexity of hardware architectures. Industry, because of the difficulty of incorporating academic proposals into maintainable and usable products, as well as the enormous efforts to produce reliable, integrable, compatible and efficient solutions with multiple manufacturers of accelerators.

This chapter highlights the relevance of heterogeneous computing and introduces the challenges it poses, as well as the approach of this dissertation to optimize performance and energy efficiency in these massively parallel systems.

Chapter contents

1.1 Heterogeneous Systems

To understand the importance of heterogeneous systems it is necessary to establish a bit of context and what has made them so important. *Moore's law*, a prediction made more than 50 years ago, has been in force until relatively recently [1]. This prediction stated that the number of transistors on a chip doubled every 18 months. The more transistors, the more functionality and, in general, the higher the performance. In most cases, a software developer received, without much effort, an increase in performance thanks to the work of the computer architects. Technology improved, and clock frequencies increased, producing faster and faster processors.

However, a little over a decade ago, everything began to grind to a halt. The *Dennard scaling* condition, which had worked so well since 1974, ended [2]. This stated that the voltage and current should be proportional to the linear dimensions of the transistors, i.e., as the transistor size was reduced, increasing the speed and reducing the current and voltage, the power remained virtually constant. However, over the years, both the leakage current and the voltage threshold for performing switches have become difficult to reduce any further, causing the power density to increase with the number of transistors.

There were many strategies to improve single-thread execution and to add more features to processors, including pipelining, speculative execution and superscalar techniques [3–8]. When all these strategies reached their limits, architects resorted to the drastic approach of increasing the number of cores per chip, generally at a lower frequency. By using simpler cores at lower frequencies, power dissipation and energy consumption were reduced. This led to all kinds of strategies, including the creation of logical cores and the exploitation of *simultaneous multithreading (SMT)*, leading to multicore systems [9, 10]. With these processors now ubiquitous, single-threaded programs are no longer an energy- and resource-efficient option, albeit at the cost of complicating the work of software developers [11].

Despite the fact that there may be other types of heterogeneity that affect common processors, they are generally implicit for the programmer. Regardless of the *performance levels* of the operating system, the memory hierarchy or even the possibility of having cores with different capacities and computational powers (ARM big.LITTLE, Intel Alder Lake), they are usually programmed under the same sequential programming model [12–15]. Each thread or process is executed by each core in an apparently sequential manner, even if internally there is some parallelism between instructions. However, with the emergence of new types of computing devices and completely different architectures, this is no longer possible. Simultaneously to the previous developments, the market and users demanded to be able to process huge amounts of data, transforming the field of *high performance computing* (HPC). There was a shift from the predominance of HPC focused on large applications and parallel algorithms to massive data computing. It is then when programmers and the HPC community realized how powerful graphics cards (GPUs) are for data processing.

GPUs, originally dedicated to graphics and multimedia computing, have become part of the most widely used accelerator in heterogeneous computing. They were adapted to general purpose computing (GPGPU), and used for a myriad of applications, from training and inference in machine learning, through scientific and generalist applications, to being exploited in industry and service cloud servers [16–22]. They boast efficiencies orders of magnitude better than traditional processors, mainly due to the adoption of a different paradigm, the *single-instruction multiple-threads (SIMT)* model. Although CPUs have vector units, they fall far short of the power offered by GPUs, being able to perform thousands of operations simultaneously on large data sets. It is important to note that both Nvidia and AMD were promoting GPGPU computing, but Nvidia's CUDA, which emerged more than a decade ago, drove mainstream adoption of accelerators. Indeed, the impact of Nvidia has been so compelling that it is still used globally today, even though having some limitations. This coins the term heterogeneous system as it is understood today, consisting of a CPU and an accelerator, often termed *device*. An accelerator is an independent entity connected by network, bus or even assembled on the same chip. Nowadays there are devices of many kinds GPUs, MICs, FGPAs, DSPs and even TPUs. Depending on the vendor and type of accelerator, it varies the internal architecture. For instance, GPUs, are generally composed of *compute units*, each of which has multiple *processing elements* (ALUs and SIMD units), scheduling hundreds of thousands of *threads* to compute regions (*kernels*). However, they need to be managed by a CPU, usually called the host. This forces the adoption of a different programming model, as will be seen later. Nevertheless, this popularity led to the incorporation of many types of accelerators.

A second group of heterogeneous devices are *many integrated core (MIC)* coprocessors such as the Intel Xeon Phi, often classified as many-core accelerators together with GPUs [23–25]. These accelerators are characterized by including many single cores, so that they take on the advantages of traditional multicore programming, but with alternative architectures and interconnections, in some cases resembling GPU designs. MIC coprocessor are usually aimed at the HPC and server markets, therefore their cores use vector units of greater capabilities than mainstream processors. Although they are heterogeneous devices, this proposal has tried to favor all the existing knowledge about traditional processors, from the implication of the memory hierarchy to the microarchitecture itself, including the programming paradigms. Although coprocessors like the Intel Xeon Phi are no longer available on the market, all the work done by both Intel and the scientific community on this type of systems has been useful [26–32]. Moreover, these efforts have served to evaluate and optimize technologies that are currently used in Xeon processors with dozens of cores. Nevertheless, other vector engine processors are being developed, such as the NEC SX-Aurora TSUBASA, used in supercomputers [33–36], as well as custom hardware designs and SoCs [37–42]. Therefore, these accelerators are relevant architectures that continue to be leveraged in HPC.

Further developments of the heterogeneous system concept was extended beyond GPUs and many-core coprocessors, incorporating *field-programmable gate arrays (FPGAs)*, which are a major leap in the heterogeneous computing paradigm [43–48]. To understand their rise and use, it is necessary to understand the problems of *application-specific integrated circuits (ASICs)* [49–53]. Their power efficiency and performance can be very good, but their design and fabrication is expensive, very complex and inflexible, with slow iteration and debugging periods, limiting general-purpose computing to what the chip is specifically designed for. For this reason, FPGAs offer a great deal of versatility in terms of their possibilities, placing their application in an intermediate position between the recommended uses of a CPU and a GPU. These devices are a circuit that can be configured by a programmer to implement a specific program or function, achieving a fairly precise determinism in the final reconfigured characteristics, both in performance and power consumption. These devices provide great flexibility to the system, although they require a lot of time to synthesize and build the circuit configuration bitstream.

There are more types of heterogeneous devices. For example, *automata processors (AP)* have recently appeared as specialized pattern recognition nodes, highly recommended for data analysis, pattern matching, graph structure analysis and even statistical processes [54–56]. Although generally implemented in FPGAs, their core has been expressly designed to be efficient in processing regular expressions. *Digital signal processors (DSPs)* also have their place in the heterogeneous world, becoming very powerful when applied to specific niches such as analog, audio and video signal processing [57–59]. They are capable of applying multiple mathematical equations on sampled signal streams. Their energy efficiency but limited applicability makes them good coprocessors, often embedded on-chip. This is also the case with *tensor* or *neural processing units (TPUs or NPUs)*, being specific to support artificial intelligence applications, accelerating neural networks in machine learning [60–66]. Some *systems-on-chip (SoCs)* such as the Qualcomm Snapdragon 845 or the Amlogic A311D include specific units for inference or augmented reality, although they can also be found as external attachable units. Examples are the Coral USB Accelerator or the Intel Neural Compute Stick, among other *visual processing units (VPUs)* [37, 67–71].

As a result of all this effervescence, heterogeneous systems have all kinds of devices and accelerators specialized in computing certain types of workloads. For this reason, it is increasingly common to find systems composed of multiple accelerators, even with different types of architectures, achieving outstanding levels of performance and energy efficiency. Examples range from HPC systems with the Nvidia DGX-1/2, having 8/16 GPUs and currently used in supercomputers[1,2] to embedded SoCs for low-power *edge computing* platforms with CPU, GPU and NPU, among others [61, 62, 71–75]. For instance, considering a multi-device *multi-processor system-on-chip* (MPSoC), the Xilinx Zynq Ultrascale+ is composed of APU

[1] https://www.top500.org/lists/green500/2021/06/
[2] https://www.top500.org/lists/top500/2021/06/

cores, Real-Time processing units, an ARM GPU and even a FPGA in the same chip, as it is depicted in Figure 1-1 [76–79]. Not to mention the proliferation of configurations that are emerging as a result of such diversity, with heterogeneous systems increasingly common in both embedded IoT and commodity nodes, as well as HPC and cloud services servers [80]. Ultimately, these type of solutions faces the challenge of exa-scale computing, improving *FLOPs-per-watt* and *per-monetary cost* ratios [81–85]. However, the questions to be asked are how these systems can be programmed comfortably or whether a code can be ported to a specific accelerator achieving its best efficiency.

In short, heterogeneous systems are here to stay [86], but this should come as no surprise, since heterogeneous computing has been with us, in various fields and in a more or less explicit way, for a few years now. As has been seen, many types of devices and architectures are available, and new ones will appear in the future, so the emergence of even *quantum computing units (QCUs)* should come as no surprise [87–89].

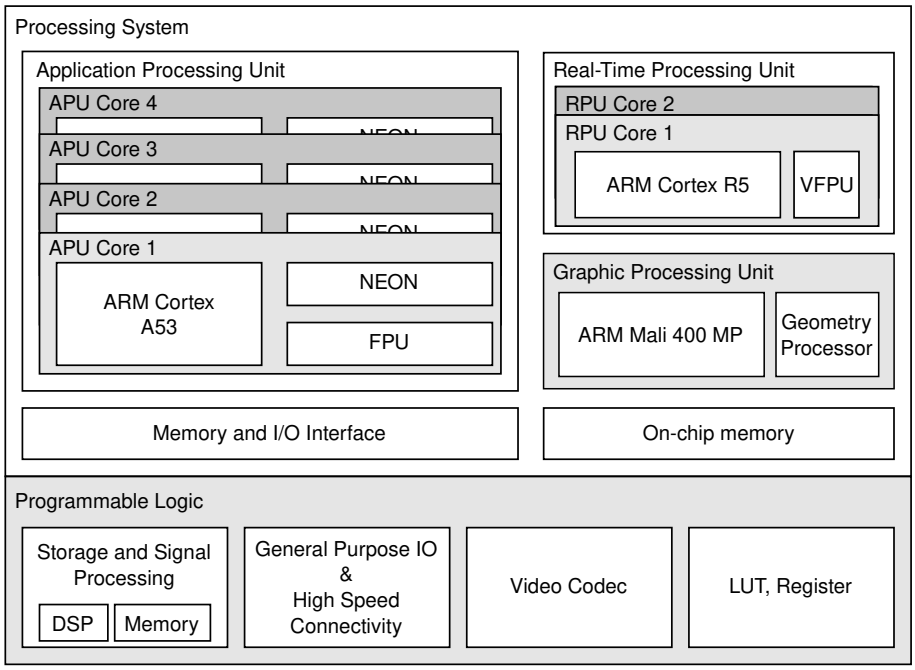


Figure 1-1: Xilinx Zinq Ultrascale+ MPSoC composed of 4 types of architectures: CPU, GPU, RPU and FPGA.

1.2 Programming Models & Languages

One of the fundamental aspects that determine the success of a hardware design is its programmability. It does not matter the speed or energy efficiency of a machine if it is not easy to program and implement the business logic or scientific application. There is no such thing as a perfect programming language, because the needs and applications are so varied, even more so if the variety of existing hardware devices is included. Moreover, if this were the

case, academic and industrial proposals would not continue to be generated, both for APIs and complete languages, as well as abstractions and facilitating frameworks.

In general, a programming language has to allow programmers to achieve the expected functional and non-functional requirements. It must have tools that give the programmer access to libraries of highly optimized functions to incorporate in their programs avoiding the need to reinvent the wheel. In addition, a great success is to enable the exploitation of lower level functions, obtaining a greater control over the machines, but without hindering productivity. Another fundamental aspect is the possibility of porting code, so that previous work can be reused in different architectures or devices. Since heterogeneous computing focuses on improving performance and specialization, such portability does not necessarily translate into performance portability [90–93], that is, a piece of code will execute successfully on another device, but may not achieve the best performance. Hence, it is also advisable to provide mechanisms to take advantage of the characteristics of other systems where such code will be executed. Finally, since heterogeneity is complicated, the programming language should present a point of view as homogeneous as possible, so that there is a clear decoupling between the code and underlying architecture.

In short, a programmer must be able to think at a high level, allowing to translate the problem domain and reduce the time-to-market, but must also be able to act on the low-level details, in order to achieve maximum efficiency [23, 94, 95]. The best strategy is still to provide options, either language improvements or new creations, so that programmers have enough choice. As with technology stacks and generalist programming languages, despite the specialized application niches, there are always some more versatile ones that end up being used widely, spreading to many domains. However, effervescence and variety are key to evolution, as some are nourished by others in ideas, expressiveness and potential for use.

The paradigm used in heterogeneous systems is the so-called *host-device programming model*, since the processor (host) offloads computational regions or functions, sometimes called *kernels*, to an accelerator (device) [24, 95]. They are independent entities connected by network, bus or even assembled on the same chip, but they need the management of a CPU to be used. Generally, the work is sent through a system driver that acts as a bridge of operations. The de facto language for heterogeneous programming using the host-device paradigm is *Open Computing Language (OpenCL)*[3], already established after a decade [96, 97]. Figure 1-2 depicts this programming model showing an abstraction on top of the MPSoC presented in Figure 1-1 of previous Section 1.1. The host acts as an orchestrator of the system, managing the devices and its internal *compute units* through offloading operations, as it will be explained in detail in Chapter 2. The main drawback of this model is that the programmer is in charge of performing all device management and initialization, transferring problem data between separate memory spaces, manually partitioning and offloading compute regions, and collecting results. Other solutions have appeared over the years trying

[3] `https://www.khronos.org/opencl/`

to raise the level of abstraction over OpenCL, such as *SYCL*[4] or *oneAPI* [98]. However, they all fall under the host-device model with all the drawbacks mentioned above, although to a different degree and with other peculiarities, such as, for example, less initial compatibility and portability.

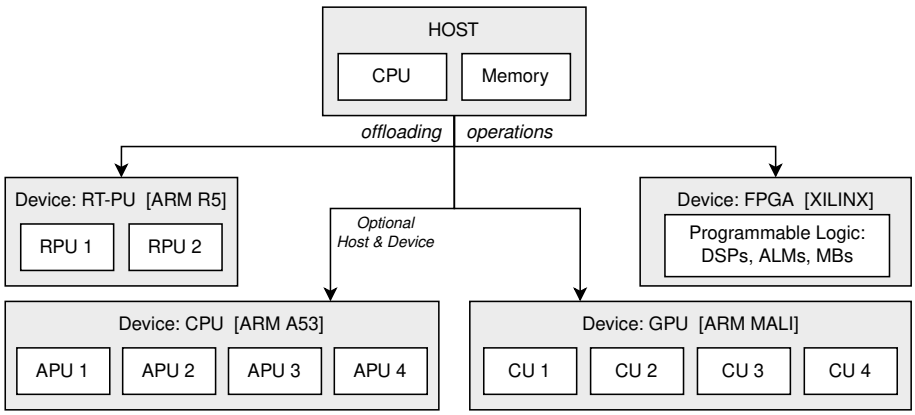


Figure 1-2: Host-device programming model applied on the Zinq Ultrascale+ MPSoC and its 4 devices.

One of the main problems with the host-device model is its initialization times, generally resulting in fixed execution costs. This, in combination with delays when offloading kernels to accelerators, is prohibitive for some situations. This is a serious problem, as many systems have devices that are not being leveraged. There are a multitude of applications that could benefit from optimizing execution for problems that require a total computation time of very few seconds, from embedded and commodity devices specialized in facial recognition, to medical analytics processors and sensors, to service servers and cloud provider platforms [99–110]. All these systems are of great relevance, since they affect emerging fields, from the *Internet of Things (IoT)* [111–113], through all kinds of web servers and cloud infrastructures, nourishing the technological base of fast serverless frameworks and lambda services [114, 115]. These types of scenarios are generally referred to as *time-constrained*.

Furthermore, there are situations in which a technology does not behave properly, either because of the programming model itself, the hardware architecture or the type of application. To face these problems, it is important to explore new ways to achieve efficient execution, even going as far as using other parallel programming paradigms. For example, allowing other technologies to be used when executing such applications, either by migrating kernels to devices or by including new accelerators and runtimes. This is another complex scenario that needs to be provided with versatile solutions that can perform technological combinations, since there are languages or paradigms that are better suited to certain situations or architectures. This is a problem found in specialized applications highly optimized through parallelism and vectorization. Even focused on hardware architectures such as Intel

[4] https://www.khronos.org/sycl/

Xeon and Intel Xeon Phi, finding cases in molecular dynamics simulators, among which is ls1-MarDyn [116–121]. When these systems are equipped with mixtures of programming languages and paradigms, as well as computational technologies, they are often referred to as *hybrid programming models*.

In addition to these issues, when more accelerators are incorporated into a heterogeneous system, the software does not scale, making it necessary to perform all these operations for each device in the system. The big drawback is that the programmer must take into account the adaptation of the code when porting it to other systems or when modifying the types of devices or their properties. This operation is tedious and hardly portable, making it difficult to use all devices properly. This orchestration of resources makes it difficult to take proper advantage of the features offered by the accelerators, and in many cases, energy is wasted in the process, either because there are idle devices or because the porting of code did not meet the needs of another architecture.

Although there are several programming languages and frameworks that have been adapted to use accelerators, they generally have a number of drawbacks. Among the most prominent, CUDA is the most widely used language for GPGPUs, but it has always been tightly coupled to Nvidia GPUs. OpenACC and OpenMP are proposals that raise the abstraction level via offloading directives. However, they compromise their flexibility, and do not always have support for new accelerators. The diversity is very large, and new proposals are constantly emerging, as Chapter 2 will show. However, to date, OpenCL continues to be the de facto standard with the best support for all types of accelerators.

Summing up, OpenCL provides many mechanisms that meet the above premises, but the programmer still has a tremendous responsibility when programming these systems. In addition, a conventional use in this paradigm is based on per-device tasking, offloading kernels to specific accelerators. This is an appropriate strategy when tasks require a high degree of specialization atomically exploitable by a specific accelerator. This is especially addressable when the dependencies between them are minimal, decoupled by the programmer, as well as when the knowledge of device availability and execution flow duration is available. However, when these conditions are not present, in many other occasions, it is an inappropriate solution, so co-execution techniques are promoted.

1.3 Co-execution

Co-execution is the parallel programming strategy whereby several processing units join their computational resources to simultaneously solve the same kernel [122–124]. In other words, the different devices of the heterogeneous system compute the same problem at the same time. To do this, each device needs to receive a portion of the complete problem, which is necessarily handled by the host system and its CPU. The programmers must keep track of the regions transferred and how each kernel acts on its data. Additionally, they are

also responsible for collecting the partial results computed by each device and joining them together to form the complete result of the computation.

One of the advantages of co-execution is that accelerators are generally good at exploiting the data parallelism inherent in the computational regions offloaded to these types of devices, especially when they can take advantage of multiple vector units and massive parallelism. This technique is able to extract the full potential of the system, in terms of performance and energy efficiency, since all accelerators contribute in solving the problem, including the CPU [93, 125–130]. In the host-device programming paradigm, it often happens that the main processor is used as the manager, acting as a global orchestrator fully involved in the selection, partitioning and allocation of data, as well as the collection of results. However, the CPU continues to consume power, not contributing to the computation, and in many occasions standing by to send and receive requests with the devices. This situation is becoming increasingly relevant, not only in types of devices used in a system, but also regarding the increasing number of CPU cores, wasting computational power. For example, there are already AMD Epyc and Intel Xeon Platinum processors with up to 64 cores, and the trend is raising [131–135].

Figure 1-3 shows an example of an HPC heterogeneous system following a traditional offload host-device programming model, where co-execution has not yet been enabled. The architecture of the heterogeneous system is shown on the left, consisting of an Intel i7 7700 with an integrated HD 630 Graphics to which two coprocessors, an Nvidia Titan X GPU and

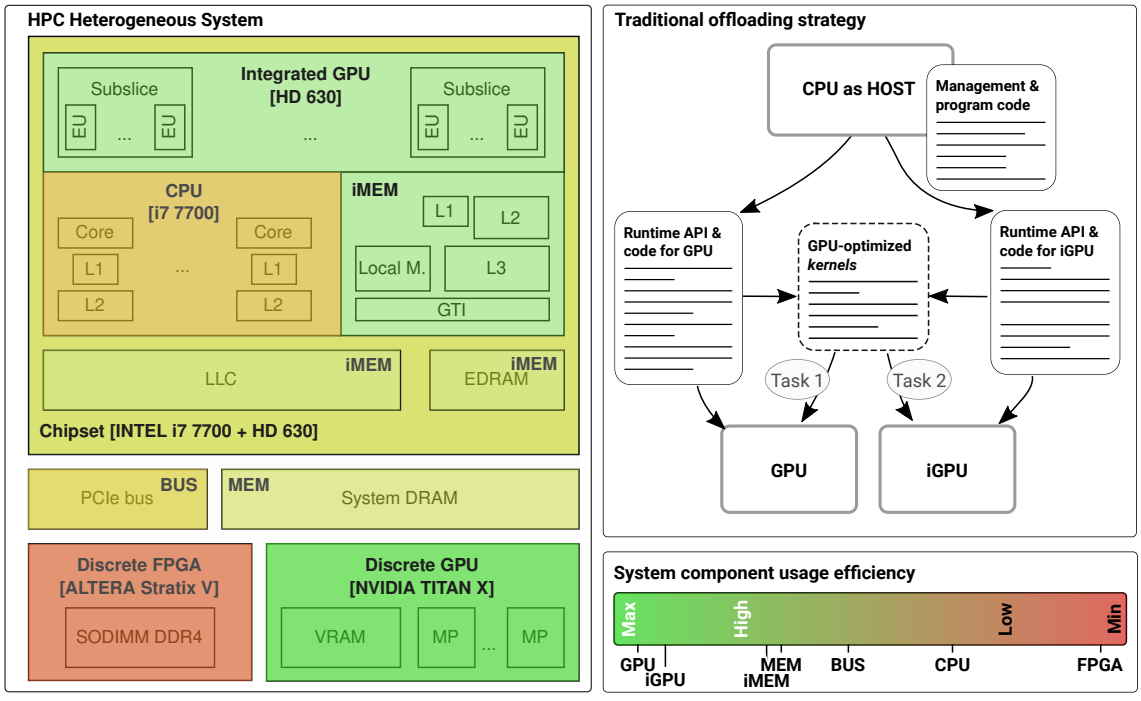


Figure 1-3: Traditional under-utilized heterogeneous system composed of an Intel chip with integrated GPU, with discrete FPGA and GPU connected via PCIe.

an Altera Stratix V FPGA have been attached. This type of nodes with mixed configurations is becoming more and more common, not only due to the upgradeability or the availability of reusable devices and inventory, but also because of the advantages in the execution of specific problems [48, 125, 128, 136–148]. A traditional programming scheme is represented in the upper right-hand side, whose implementation is complex due to the existing diversity and where programmers often choose to perform task-based offloads. It requires abundant management logic, code associated with the program to be computed, as well as specific software regions for each device. Moreover, in this case, the programmer chooses kernels optimized for GPUs to improve their *occupancy*, offloading to the discrete and integrated GPUs, each with an independent task. The efficiency of exploitation of each component of the heterogeneous system is shown on the bottom right, ranging from total under-utilization (minimum, in white) to effective exploitation (maximum, in black). As can be seen, devices such as the FPGA or the CPU are not exploited, while others such as the integrated GPU, or the PCIe communication buses and memory still have room for improvement in their utilization. Furthermore, even if all devices were being utilized through task-based parallelism, this representation would still be valid for a specific moment of the execution, since the devices may present stalls and idle periods, waiting to receive data or a notification after a previous dependent task has finished. The system is not leveraged, consuming power by devices in idle (FPGA), not computing part of the problem (CPU, static power) or not optimized for architectural variations (integrated GPU). Without co-execution and a portable, cross-platform compatible language, energy efficiency and performance are being lost.

These losses are measurable, as can be seen in Figure 1-4. The above HPC system is analyzed by considering two scenarios with respect to energy consumption. Both graphs show the devices on the abscissa axis and the consumption in watts on the ordinate axis. On the

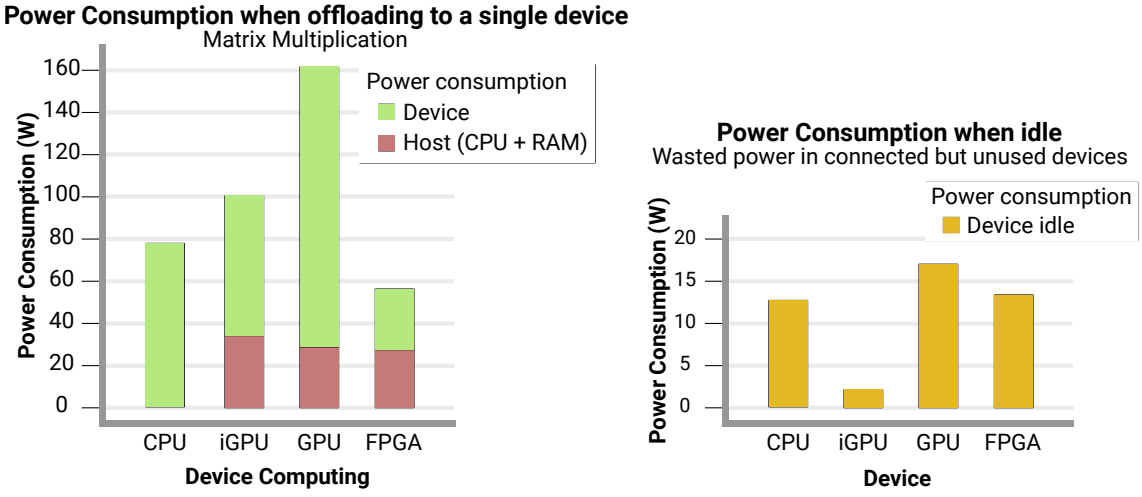


Figure 1-4: Power consumption of an HPC heterogeneous system with 4 devices when co-execution is not enabled or when devices are idle.

left, the power consumption is calculated when a single device is being offloaded to compute the matrix multiplication. In this case the offloading is performed to a single accelerator, a typical strategy to avoid increasing the implementation and optimization efforts. For each case, the PCIe-connected accelerators that are not used in the computation are being removed from the node. Results show how the host is still consuming (*CPU + RAM*), up to 37W without contributing, reaching cases where the CPU consumes as much as the FPGA, who is computing the whole problem. On the other hand, the right graph shows the power consumption of each device without doing anything, just for being connected and idle. Not to mention this presents a configurable heterogeneous system, but those nodes fixed by form factor or with acceleration units directly on chip cannot be unplugged.

For this reason, it is important that all devices in the heterogeneous system contribute as much as possible, in order to speed up problems and reduce energy consumption. In general, co-execution is beneficial to a programmer, as it is a more convenient method to apply to existing problems. Since it focuses on the partitioning of data per device, the programmer does not have to study complex dependencies and can exploit the accelerators in a natural fashion. On the other hand, in task-based parallelism execution, the scheduler has to transform a problem into multiple tasks of different granularity, trace the dependencies between them and manage the execution of each one on each device, with the drawback of having tasks (and devices) waiting for others.

However, task-based heterogeneous programming is a problem generally addressed and supported by programming languages, frameworks and even other runtimes, while co-execution is a problem that has so far been implemented manually by programmers [149–151]. The problem with these technologies is that to express heterogeneous co-execution it has to be implemented in terms of task parallelism. This introduces problems in the distribution of data between devices, complicates the transformation and mapping between programming models, and varies the way of scheduling such tasks. Furthermore, it may even introduce overheads, since tasks are usually treated as encapsulated entities to guarantee their scheduling based on input and output dependencies.

In short, co-execution provides advantages when using accelerators in heterogeneous systems, enabling data parallelism and facilitating the effort of the programmer with respect to the problem domain. However, the work to achieve this strategy has to be minimal on the part of a programmer, to facilitate its adoption, and is usually determined by two aspects, the abstraction of the heterogeneous system and the workload balancing.

1.4 Abstraction & Load balancing

The abstraction of the heterogeneous system tries to reduce the managing and operating of the different devices of the node. Efforts in achieving co-execution are also driven by and closely linked to the work done in achieving good accelerator management, masking the

system orchestration from the programmer. It is important that this abstraction layer hides the details of the underlying architecture, offering a simplified and uniform facade for the different devices. This point is key, as it also allows a programmer to operate with the devices independently of the system where the application is executed, enhancing portability.

Moreover, a system that encapsulates the necessary device operations, from initialization and configuration through data management and manual download of computational kernels, facilitates both independence and optimization of operations between accelerators. Leveraging strategies can be exploited both across device types, such as coarse-grained tasks for FPGAs and range-based tasks for GPUs, and across manufacturers, such as performing memory optimizations when building buffers on AMD graphics cards [102, 125, 128, 152–154]. System abstraction facilitates these types of operations, freeing the programmer and favoring both usability and portability between devices, architectures, manufacturers and systems.

The other fundamental aspect of co-execution is workload balancing, the purpose of which is to distribute the computational problem appropriately among all the devices that make up the system [122–124, 155–157]. The objective is to obtain the appropriate proportion of work for each device contributing to the job, so that they all have a similar execution time, generally finishing simultaneously. A correct distribution technique and scheduling algorithm is the way to minimize the waiting times of the devices, so that they are always sending, computing or receiving data. Derived from this objective is the need to overlap computation and communication, using as many transmission channels as are available and making the devices always compute while preparing (or receiving) the next data to be computed, maximizing global parallelism.

Two important concepts that influence load balancing arise from here. On the one hand, the types of problems to be computed, being *regular* or *irregular*. Regular problems are those in which the execution time is determined only by the size of the data to be computed on

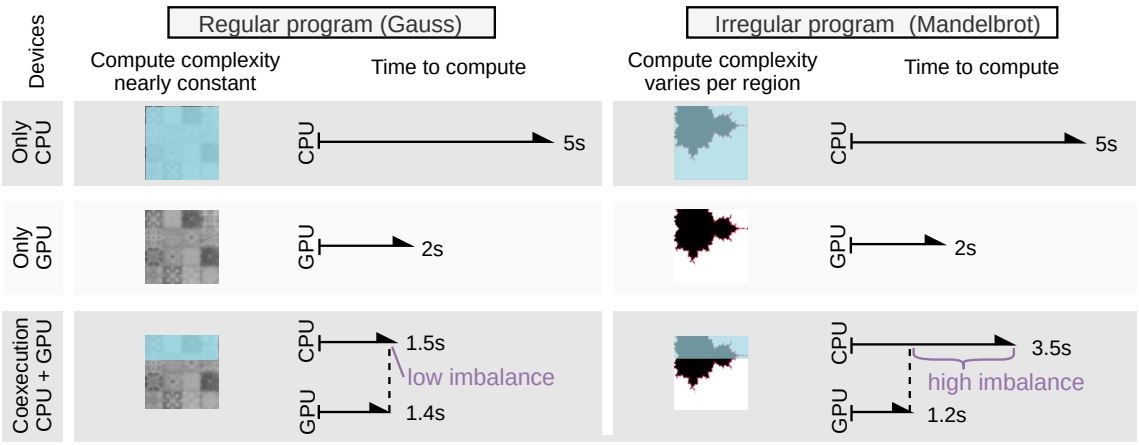


Figure 1-5: Static co-execution for regular and irregular programs.

each device [129]. The programmer needs to estimate offline how much workload to allocate to each device so that all finish at the same time, thus obtaining a balanced execution. This is depicted in the left part of Figure 1-5, when computing the Gaussian kernel (see Section 1.7.2 for more details about the kernels) with two devices, CPU and GPU. In this case, the kernel execution time is 5 seconds on the CPU and 2 seconds on the GPU, which means that the GPU has 2.5x the performance of the CPU. Each device takes about the same amount of time to compute each pixel, regardless of color or intensity, so it is a regular problem. Therefore, assigning the work to devices proportionally to their computing capabilities, a balanced distribution is obtained and the execution time is reduced to approximately 1.5 seconds.

However, irregular applications, where the processing time of a data set depends not only on its size, but also on the nature of the data, cause a challenging task. Thus, different portions of data of the same size can generate different response times. This is shown in the right part of Figure 1-5, which presents the execution of a Mandelbrot fractal computation in two devices. Each device varies the time to compute each pixel, due to the computational region of the mandelbrot function, with the darkest and reddest areas being the most computationally intensive data regions. Performing the same static balancing as in the regular case, it has coincided that the most computationally heavy regions have been executed by the CPU (slower device). This results in a significant imbalance, with the CPU taking 3.5 seconds while the GPU took only 1.2 seconds. This situation can only be addressed with *dynamic balancing algorithms* that allocate portions of work to the devices on demand [158, 159].

On the other hand, the heterogeneity of the devices themselves makes them more appropriate for different types of tasks, determining their granularity, recommended durations and the types of operations in which they excel. The capabilities of accelerators as well as their architectures determine the size of the minimum (and maximum) working blocks they are comfortable with, beyond which they suffer penalties. For example, a GPU needs up to hundreds of thousands of threads to obtain a full occupancy of its resources, so small job sizes will underutilize such a powerful device [160–163]. However, giving a modest workload to a CPU could saturate it, slowing down the overall execution time and even affecting the rest of the system. For this reason, load balancing and its decisions are complex, and it is necessary to tailor these partitioning decisions taking into account both the type of hardware and the amount and type of load to be allocated. For this reason it is necessary for the programmer to free himself from this type of decisions, since they complicate the programming and the slightest failure can mean absolute penalties in the final execution.

1.5 Hypothesis

Considering all of the above, this dissertation evaluates the following hypothesis:

The design and implementation of runtime systems that allow co-execution is the best way to leverage the full computational capacity of heterogeneous systems, while providing a sufficient level of abstraction for a programmer to use it properly. The approach to provide maximum performance, scalability and energy efficiency will come through the design of efficient and extensible runtime architectures, as well as the implementation of scheduling algorithms that maximize device utilization in the face of any type of application and heterogeneous system.

To achieve this, four cross-cutting issues need to be addressed: performance portability, technology compatibility, system abstraction and load balancing.

- *Performance portability*: providing mechanisms to execute as efficiently as possible in each type of architecture, relieving the programmer of the complications of devices and their operating modes, in a transparent manner.
- *Technology compatibility*: offering operating alternatives, easily adapting to new trends, without being tied to a single technology and allowing compatibility between programming models, exploiting each device in the most convenient manner.
- *System abstraction*: encapsulating technological and architectural complexities, providing a convenient, maintainable and extensible API and management system.
- *Load balancing*: offering efficient and diverse scheduling algorithms to exploit all types of problems, being extensible and optimized as an integral part of the runtime architecture.

To validate the effectiveness in overcoming these issues, they will be designed with different usability perspectives, evaluating programming models, software architectures, scheduling algorithms and emerging technologies for heterogeneous computing. The ultimate goal is to facilitate programmability and compatibility to exploit efficiently co-execution in all types of heterogeneous HPC and commodity systems.

1.6 Major dissertation contributions

The most remarkable contributions of this thesis to study the optimization of performance and energy efficiency in massively parallel systems are listed below. Each of them is presented in depth in the different chapters of the document.

- Proposing EngineCL as a flexible and portable heterogeneous runtime system. Its main aim is to reconcile usability and performance. Therefore, it integrates performance enhancing techniques and abstractions that allow to effortlessly combine the computing capabilities of several devices. Management overheads are reduced by implementing a highly optimized scheduler, which is customizable through hook functions. It is built on top of OpenCL with a modular architecture that is easily extendable. From OpenCL it takes advantage of much of its potential, like the high device compatibility. To promote maintainability of applications, it presents an easy-to-use API designed to integrate well in many scenarios, from desktop applications to HPC. A thorough evaluation of EngineCL is presented including performance, energy efficiency, scalability, portability and maintainability.

- Extending and optimizing EngineCL for time-constrained scenarios. There are computing request-response based computing services where the response time is limited, generally to a few seconds. The host-device model usually suffers penalties in such scenarios, as they rely on counterproductive performance modes for heavy runtimes and where drivers have long initialization times. So this proposal details the runtime and algorithmic optimizations performed on EngineCL to allow it to cope with these situations. The multi-threaded software architecture has been enhanced to adapt to different devices and architectures commonly found in commodity servers today. This extension confirms the versatility of EngineCL as it was easily adapted to situations for which it was not initially conceived.

- Enhancing EngineCL to extend execution types and exploit hybrid co-execution allowing the integration with the ls1-MarDyn simulator from HLRS. The ls1-MarDyn molecular dynamics simulator suffer a strong penalty when using OpenCL, which makes negligible the performance increase of combining different devices. For this reason, EngineCL is extended to support hybrid forms of co-execution that allow executing these problems efficiently. The new extension is able to maintain consistency at the API, while providing more functionality to the supported execution types, increasing the versatility of the runtime. For instance, it is possible to combine acceleration technologies and programming models such as OpenMP, while supporting the rest of the runtime functionality, including scheduling algorithms. In addition, the programmer is now provided with mechanisms to supply different types of execution

sources and kernels, even dynamically at runtime. This proposal simplifies the integration with other technologies and programming models, such as CUDA, machine code of accelerators or other specialized programming models.

- Proposing Coexecutor Runtime as a modern C++ co-execution runtime. If EngineCL relies on lower level technologies, like OpenCL, which is common in the scientific world, Coexecutor Runtime is aimed at C++ programmers using oneAPI, more prevalent in industry. It is a runtime that facilitates co-execution in heterogeneous systems supporting Intel oneAPI technology. It maintains the principle of single source code and leverages the versatility of C++ in the definition of operations. Its decoupled architecture allows any C++/SYCL programmer to incorporate new features without the need for other technologies or dependencies, facilitating its integration in all types of software. The runtime hides behind an abstraction layer cumbersome tasks, like the management of devices, data and scheduling. Through its asynchronous concurrent execution architecture it overcomes the limitation of oneAPI regarding co-execution. Like EngineCL, it includes various load balancing algorithms. However, Coexecutor Runtime goes further in harnessing dynamic scheduling mechanisms, yielding high efficiencies. The runtime can leverage new hardware architectures without the need to make changes, as long as a compatible driver is provided. Furthermore, the proposal explores the behavior in different architectures, both commodity and HPC, highlighting the performance and energy efficiency achieved by using the latest oneAPI extensions and adaptive scheduling algorithms.

1.7 Methodology

The proposals explained in this dissertation have been validated using the methodology proposed in this section. The results presented throughout the document are based on experimental data obtained from executions in real systems. This methodology section is common to all chapters, detailing the heterogeneous machines and devices used, the benchmarks evaluated, the metrics analyzed and tools used. Those chapters where variations in the methodology have been applied, extension of the benchmarked applications or peculiarities of the validation, will expand the methodology of their respective sections with the details particular to their experiments.

1.7.1 Platforms & Devices

The chapters of this dissertation present different heterogeneous nodes and devices used for the experiments, as detailed below. There are 11 different architectures, comprising Intel and AMD CPUs, Intel and AMD discrete and integrated GPUs, as well as an Intel Xeon Phi many-core coprocessor. The diversity of machines and devices serves the technological and validation needs of the runtimes built and techniques developed. For example, oneAPI can currently only be used on devices of the latest Intel generations, so specific heterogeneous systems are required.

Each chapter of the thesis refers to the machines used, as well as peculiarities that have been applied for such experimentations and validations. In any case, the characteristics of each node are detailed in Table 1-1 and they are summarized below:

Batel and **Trainera** are heterogeneous systems with identical properties regarding memory, system and processor. They are composed of two Intel Xeon E5-2620 CPUs with six cores that can run two threads each at 2.0 GHz and 16 GBs of DDR3 memory. The CPUs are connected via QPI, which allows OpenCL to detect them as a single device.

Batel has two accelerators, a GPU and a MIC. The discrete GPU is a Nvidia Kepler K20m with 13 SIMD lanes (or SMs in Nvidia terminology) and 5 GBytes of VRAM. On the other side, the coprocessor is an Intel Xeon Phi KNC 7120P, with 61 cores and 244 threads. These are connected to the system using independent PCI 2.0 slots.

Trainera is configured with a modern discrete GPU, an AMD RX5700XT. It is a Navi 10 XT generation, with RDNA 1.0 architecture, exposing 40 compute units at 1905 Mhz and with 8GB of video memory at 1750 MHz, offering a bandwidth of 448.0 GB/s.

Remo is a machine composed of an AMD A10-7850K APU and Nvidia GeForce GTX 950 GPU. The CPU has 2 cores and 2 threads per core at 3142 Mhz with only two cache levels, exposing 4 OpenCL compute units. The APU's on-chip GPU is a GCN 2.0 Kaveri R7 DDR3 with 512 cores at 720 Mhz with 8 compute units. Finally, a discrete Nvidia GPU is attached, providing 6 compute units, 768 cores at 1240 Mhz and 2 GiB of DDR5.

Table 1-1: Heterogeneous systems, platforms and devices.

	Batel	Trainera	Remo	Desktop	DevCloud
Processor	**Intel CPU**		**AMD CPU**	**Intel CPU**	**Intel CPU**
Model	Xeon E5-2620		A10-7850K	Core i5-7500	Xeon E-2176G
Architecture	Sandy Bridge		Kaveri APU	Kaby Lake	Coffee Lake
Specs	2.0 GHz		3.7 GHz	3.4 GHz	3.7 GHz
# CPU	2 (QPI)		1	1	1
# Cores	6		4	4	6
# Threads/Core	2		1	1	2
Cache	L1 32K(i) 32K(d) L2 256K L3 16M		L1 16K(i) 96K(d) L2 2M	L1 32K(i) 32K(d) L2 256K L3 6M	L1 32K(i) 32K(d) L2 256K L3 12M
Compute Units	24		4	4	12
Pref. wg size	128		1	128	128
Drivers	Proprietary (Intel Runtime 14.2) OpenCL 1.2		Open-source (mesa, clover) OpenCL 1.2	Mixed (oneAPI, ocl) OpenCL 2.0	Mixed (oneAPI, ocl) OpenCL 3.0
Memory	16 GiB DDR3		8 GiB DDR3	8 GiB DDR4	64 GiB DDR4
System	CentOS 7 Kernel 3.10		ArchLinux Custom Kernel 4.19	Ubuntu 20.04 Custom Kernel 5.4	Ubuntu 20.04 Kernel 5.4

	Batel	Trainera	Remo	Desktop	DevCloud
Accelerator	**Nvidia GPU**	**AMD GPU**	**AMD iGPU**	**Intel iGPU**	**Intel iGPU**
Type	Discrete	Discrete	Integrated	Integrated	Integrated
Model	K20m	RX5700XT	Radeon R7	HD Graphics 630	UHD Graphics P630
Architecture	Kepler	RDNA 1.0 Navi 10	Spectre 200 Series GCN 2.0	Gen 9.5 GT2 IGP HD Graphics-M	Gen 9.5 GT2 IGP HD Graphics-W
Specs	13 SMs 2496 cores 706 MHz CUDA Cap. 3.5	40 CUs 2560 cores 1605 MHz	8 CUs 512 Shaders 720 MHz	24 EUs 192 cores *600 MHz*	24 EUs 192 cores 1200 MHz
Memory	5 GiB DDR5 320-bit bus 1300 MHz	8 GiB DDR6 256-bit bus 1750 MHz	*system shared* fast bus GPU-DDR3	*system shared* 6 MiB LLC shared	*system shared* 12 MiB LLC shared
Compute Units	13	40	8	24	24
Pref. wg size	32 (Warp)	32 (Wavefront)	64 (Wavefront)	32 (EU threads)	32 (EU threads)
Drivers	Proprietary (CUDA 460) OpenCL 1.2	Proprietary (amdgpu-pro) OpenCL 2.1	Mixed (amdgpu, catalyst) OpenCL 2.0	Mixed (oneAPI, ocl) OpenCL 3.0	Mixed (oneAPI, ocl) OpenCL 3.0
Accelerator	**Intel MIC**		**Nvidia GPU**		
Type	Discrete		Discrete		
Model	Xeon Phi 7120P		GeForce GTX 950		
Architecture	Knights Corner KNC		Maxwell 2.0 GM 206		
Specs	61 cores 244 threads 1333 MHz AVX2		6 SMs 768 cores 1240MHz CUDA Cap. 5.2		
Memory	16 GiB DDR5 512-bit bus 1375 MHz		2 GiB DDR5 128-bit bus 1653 MHz		
Compute Units	240		6		
Pref. wg size	128		32 (Warp)		
Drivers	Proprietary (Intel Runtime 14.2) OpenCL 1.2		Proprietary (CUDA 455) OpenCL 1.2		

Desktop is a computer with an Intel Core i5-7500 Kaby Lake architecture processor, with 4 cores at 3400 MHz, one thread per core and three cache levels. Kaby Lake's on-chip GPU is an Intel HD Graphics 630, a mid-range (GT2) IGP integrated graphics processor, member of the family of Gen 9.5 GT2 IGP, with 24 execution units running between 350 and 1100 MHz. It is configured to be run at 600 Mhz for stability purposes regarding experimentation. An LLC cache of 6 MB is shared between CPU and GPU.

DevCloud is an Intel server node with an Intel Xeon E-2176G processor, with 12 logical cores at 3700 MHz, two threads per physical core and three cache levels. Coffee Lake's on-chip GPU is an Intel UHD Graphics P630 with 24 execution units running up to 1200 MHz and sharing a 12 MiB LLC cache with the CPU. This heterogeneous system is provided by the Intel DevCloud services, with the machine fully reserved for the experimentations.

Table 1-1 details the heterogeneous systems, platforms and devices used. Considering the properties, exposed in rows, each node contains a processor and one or two accelerators.

When a property differs between systems that a priori would not have to be different, it is due to the needs of the devices and the compatibility between their drivers. For example, each node has a Linux operating system, and it is impossible to change the versions, since some device would not work. If a Linux kernel has been patched, it is due to the needs of the driver itself, as it happens in the case of *catalyst* for *Remo* or the adaptation of the GPU to be able to analyze the hardware counters in *Desktop*. Furthermore, when using open-source or mixed drivers, it is because this combination offered the best performances (*Remo CPU*). It usually means that the driver vendors have neglected them (*Remo iGPU*) or they are in a release process still with proprietary dependencies (*Desktop* and *DevCloud*).

Considering OpenCL terms and concepts, two properties of interest are listed for each device, the *Compute Units* and the preferred work-group size (*Pref. wg size*). The former refers to the processing units that may be scheduling and executing in parallel, while the latter refers to the number of work-items (*kernel instantiations*) that are recommended for each of those processing units. Architectural and technological variation can be observed, such as the evolution from GCN 2.0 to RDNA 1.0 (*Remo GPU* to *Trainera GPU*) and the generalized versatility of CPUs and MICs, associating Compute Units to system threads and work-items to sets of SIMD (128) vector operations.

1.7.2 Benchmarks

Throughout the document, up to 8 different benchmarks are used. In addition, there is a real application, explained in Integration II 4.2, as it is only applicable to that extension and integration aspect. The benchmarks are briefly described in the following lines.

- *Gaussian* creates an output image by calculating the gaussian blur of an input image, as one of the most common filters and effects found in image and video processing applications.

- *MatMul* calculates the matrix multiplication using matricces with random values.
- *Binomial*, used in finance, is a binomial options pricing model (BOPM) which provides a generalizable numerical method for the valuation of financial options, using a discrete-time model (lattice based).
- *Taylor* performs a bi-dimensional Taylor approximation for a set of points.
- *NBody* simulates physics experiments and predicts the individual motions of a group of objects interacting with each other.
- *Ray* renders an image by modeling the light transport in a provided scene, composed of objects, walls and lights. The ray tracing technique computes the color values of each pixel of the image, used wildly in 3D rendering applications and games.
- *RAP* implements a Resource Allocation Problem, where the indirections cannot be predicted since they are solved at runtime.
- *Mandelbrot* computes a particular instance of a fractal set, displaced over the center of the produced image.

Since there is a strong validation work throughout the thesis, benchmarks with a remarkable variety of characteristics have been chosen, as can be seen in Table 1-2. The objective is to contrast the behavior in different types of situations that can be encountered when using these technologies in real applications. The benchmarks have been selected on the basis of three fundamental criteria.

Firstly, the major distinction is the use of two types of problems, regular and irregular, as it is introduced previously. A regular problem is one in which two work packages of the same size take similar time to compute for the same device. However, irregular kernels have a more complex and unpredictable behavior, where it is very important to have dynamic

Table 1-2: Selected benchmarks and their variety of properties.

Property	Gaussian	MatMul	Binomial	Taylor	NBody	Ray	Rap	Mandelbrot
Local Work Size	128	1,64	255	64	64	128	128	256
# Read buffers	2	2	1	3	2	1	2	0
# Write buffers	1	1	1	2	2	1	1	1
# Kernel args	6	5	5	7	7	11	4	8
Out pattern	1:1	1:1	1:255	1:1	4:1	1:1	1:1	1:1
Local memory		✓	✓	✓		✓		
Barriers			✓	✓				
Built-in functions	✓		✓	✓	✓	✓	✓	✓
Extensions			✓	✓				✓
Custom types						✓		
Main data types	uchar4 float	float	float4	double	float4 float	float int	int	uchar4 double
Classification	regular	regular	regular	regular	regular	irregular	irregular	irregular

adaptive algorithms that distribute the workload during execution and adapt to the behavior of the application, as it will be shown in later chapters.

Secondly, they provide enough variety in terms of development issues, regarding many parameter types, local and global memory usage, custom structs and types, number of buffers and arguments, different local work sizes and compute-write patterns. The amount of properties, computing patterns and use cases are relevant because they provide enough diversity to compare technologies and runtimes.

Thirdly, trends, both industrial and research, as well as implementations and compatibilities with the technologies used. The selected benchmarks are provided by the accelerator manufacturers and driver developers themselves, so they are optimized to be used in the base technologies against which they are evaluated. In this way, the engines, techniques and solutions provided in this document are using the optimization recommendations, guidelines and examples [164–171], offering a fair comparison.

It is important to note that there are two well differentiated technological aspects, the one based on OpenCL and the one based on oneAPI. For this reason, some properties are added, modified or discarded because they are not applicable. A notable case of this fact is the use of specific "local work size", something specific to OpenCL but not to oneAPI. This property has been experimented with the CoexecutorRuntime engine, since it offers a way to force a manual assignment of values in the kernel instantiation. No significant differences in behavior were found for the problems and platforms studied, so it was decided to simplify its use and perform an automatic adjustment.

Nevertheless, there are cases where a benchmark is not evaluated with a specific technology, such as Binomial in oneAPI. This is due to the features of DPC++ compiler that are under development and optimization. This work-in-progress implementations of the specification are not yet properly solved, penalizing the usage of typical OpenCL-like constructs, such as `parallel_for_work_group`, `sycl::group` and `parallel_for_work_item`, used in such benchmarks. Therefore, neither the raw oneAPI program nor the exploitation of CoexecutorRuntime make sense until these issues are resolved in future compiler releases.

1.7.3 Metrics

The main metrics used and referred to throughout the dissertation are detailed here, focusing on the two fundamental aspects, performance and energy consumption. On the other hand, those chapters that evaluate other specific metrics are detailed in the corresponding sections, as is the case with all the usability and programmability metrics to validate the runtime of the Chapter 3.

Performance has been evaluated using the response time of the selected benchmarks. Measurement includes the time required by the communication between the host and the devices, comprising device initialization and management, input and output data transfers,

as well as the execution time of the co-executed workload itself. The benchmarks are executed in two scenarios, the *heterogeneous system*, taking advantage of all the respective devices listed in Section 1.7.1, and the *baseline*, that only uses the fastest device of the system. The baseline device is generally the GPU, but is specified in each experiment. It is chosen to compare against the best possible case, either with OpenCL or oneAPI, performing a full offload to the fastest device and avoiding any complexity derived from co-execution, scheduling and its programmability.

Based on these response times, three metrics are analyzed. The first is the *speedup* (S) for each benchmark when comparing the baseline and the heterogeneous system response times. Since the speedup is calculated with respect to the most powerful device, the maximum achievable speedup using n devices will not be n, as it occurs in homogeneous systems. Thus, the value will always be less than n, a fraction depending on the computational power of each device in the system. Equation 1-1 shows the speedup, considering the response time of the fastest device and the co-execution when using the devices in the heterogeneous system.

$$S = \frac{T_{fastest}}{T_{co-execution}} \tag{1-1}$$

The speedup for each application using a perfectly balanced distribution has also been used to give an idea of advantage of using the complete system. They were derived from the response time T_i of each device as shown in Equation 1-2.

$$S_{max} = T_{baseline} \sum_{i=1}^{n} \frac{1}{T_i} \tag{1-2}$$

The second metric is the *heterogeneous efficiency* [172–174], obtained by dividing the obtained speedup by the speedup for the perfectly balanced distribution of the workload, as shown in Equation 1-3. The obtained value ranges between 0 and 1, giving an idea of the usage of the heterogeneous system. Efficiencies close to 1 indicate the best usage of the system is being made. The measured values do not reach this ideal because of the communication overheads and host-device interactions.

$$Eff = \frac{S_{real}}{S_{max}} \tag{1-3}$$

Finally, a third metric that relates to device behavior, utilization and effective performance is considered. This metric is interesting as it helps to understand the behavior of the load balancing algorithms and the usage of the devices during the final stages of execution [127]. The *balancing efficiency*, shown in Equation 1-4, measures the effectiveness of load balancing regarding the workload distribution towards the end of the execution.

$$B_{Eff} = \frac{T_{first}}{T_{last}} \tag{1-4}$$

Being T_{first} and T_{last} the execution time of the device that finish at first and last, respectively. Thus, the optimal is 1.0, meaning both devices finish simultaneously without imbalancing the workload.

Regarding **energy**, the two cases seen in the performance metrics are also considered. However, the baseline also considers the power consumption of the processor and the system. This is because an accelerator generally cannot run by itself and depends on the CPU, which acts as a manager. Apart from the total consumed energy, the Energy Delay Product (EDP) [175, 176] is computed, shown in Equation 1-5, combining performance (T, time) and energy (E, power consumption) to evaluate the efficiency of the system. The following section details the tool developed to obtain the energy consumption.

$$EDP = T \cdot E \tag{1-5}$$

1.7.4 Tools

Working with heterogeneous systems and especially with technologies such as OpenCL or oneAPI has made it necessary to perform a number of extra steps not required when using other more common technologies or devices, such as OpenMP or processors [177]. The summary table in Section 1.7.1 reflecting the heterogeneous nodes used in this document already shows a clear variation of operating systems, kernel versions and drivers. This diversity and technological complexity require the need to inspect and validate executions on different architectures, including the study of hardware counters to clarify the most complex behaviors.

On the one hand, the systems have been adapted to be analyzed by **profiling, API tracing and debugging tools**, such as AMD CodeXL [178] and Intel VTune [179]. The developed runtimes have modules and components that facilitate the inspection and analysis of the behaviors, as will be detailed in their respective sections. The need to create inspectors associated with the runtime is due to the fact that existing tools offer very detailed solutions, useful in the evaluation and profiling of specific problems and single executions, as well as during the elaboration of the software architecture, its main design principles and optimizations. However, they do not help to analyze the runtime system, its components and management stages, involving multiple executions and considering common, statistically relevant phenomena. Existing tools cannot discriminate performance losses due to runtime management and its multi-threaded architecture, highlight inefficiencies in load

balancing or show overheads associated with specific device management drivers, among others. Therefore, it has been necessary to use tools from the manufacturers during the conceptualization, construction and evaluation processes, since they offer more detailed information on their architectures. The most popular tools for debugging and profiling in OpenCL suffer from inadequate support from the manufacturers, both AMD CodeXL and Intel VTune. This is mainly due to poor documentation and community support, configuration problems and discontinuation of the software, and even technological incompatibility and outdated dependencies. For this reason, to this day, it is still impossible to properly inspect co-execution behavior when involving a CPU (VTune) or both CPU and GPU (CodeXL), either because the OpenCL support from Intel is focused only on its GPUs, or AMD only on the most recent generations that are compatible with ROCm tools. There are other generalist profiling projects and tools that have been gaining relative popularity during the last years, such as Intercept Layer for OpenCL [180], PTI for GPU [181], CLTracer [182] or LPGPU2 [183]. The main problem with all these tools, excluding their ongoing development, is that they are generally supporting a single vendor or focused on the analysis of a specific vendor or platform type. For instance, Low-Power Parallel Computing on GPUs 2 (LPGPU2), despite inheriting the CodeXL features, it is closely linked to the analysis of embedded platforms. In short, all these proposals have drawbacks regarding the analysis of co-execution, the scheduling system and evaluation considering statistical aspects. Nevertheless, they have been used during the implementation of the runtime systems, especially CodeXL and VTune.

On the other hand, the **measurement of the power consumed** by the system has required the creation of a specific tool, called Sauna [156, 184], to obtain the power consumed by each device. The main differences with respect to third-party and vendor tools, associated with their drivers or devices, is that they provide a top-like operation, offering real-time measurements of the entire device, not a specific process. They do not provide a configurable periodic query, they hog system processor threads, and make it difficult to measure energy simultaneously in the different devices of the node.

Its design is based on a very light software layer, in order to avoid overheads when performing a periodic query. The idea is to monitor multiple devices at the same time, standardizing their measurements, since each manufacturer and device has peculiarities to which it has to adapt. For example, Intel and AMD CPUs can be queried thanks to the Linux kernel module that provides access to the *Running Average Power Limit (RAPL)* [185] registers, which provide cumulative power readings. On the other hand, both the Intel Xeon Phi MIC coprocessor and GPUs from Nvidia and AMD provide instant power measurements. The many-core Xeon Phi offers a library and querying API using the *mpss-micmgmt* handler, communicating with the coprocessor driver. AMD GPUs supported by the ROCm tools *rocm-smi* allow querying of their associated devices, while the CUDA driver, together with the *NVIDIA Management Library (NVML)* [186], is used to obtain information about Nvidia

GPUs in the system. In addition, it is enhanced providing support for any device that uses the *sysfs* driver, through which the power values of generic devices attached to that Linux interface can be obtained. In any case, Sauna normalizes the magnitudes and generates consistent results throughout the execution. Finally, it is important to note that it allows to select the sampling frequency, as each device and application has specific conditions. If high sampling frequencies are used, delays and overheads may be incurred in the execution, while if they are low, the capture of events and variations in consumption may be lost, generating inaccurate results. In general, sampling frequencies range, depending on the experiment and the heterogeneous node, between 50 and 150ms, always trying to balance the overhead with the accuracy of the results.

1.8 Document structure

Once the Introduction is concluded, the dissertation is structured as follows:

Chapter 2 presents the fundamental concepts to understand the proposal of this thesis, in addition to providing a review of related work. The background covers the most common technologies and programming languages used, co-execution algorithms for load balancing and an overview of runtimes and their principles.

Chapter 3 presents the EngineCL runtime, a fundamental part of the dissertation, and built on top of the OpenCL technology. This runtime has a multi-purpose objective, combining high usability with high performance. This chapter presents the design principles, the abstractions built over OpenCL and the architectural system, detailing some of its modules and components. Since it is an engine that has been devised to be extended as technologies evolve, internal details regarding the scheduling stages are provided. Finally, an exhaustive validation on multiple architectures is performed, both in maintainability and performance, highlighting the ease of use and the negligible overheads with respect to OpenCL.

Chapter 4 shows how EngineCL is extended and integrated for two specific uses, so its potential for adaptability, ease of use and efficiency can be appreciated. The chapter is divided into two parts, starting with integration for time-constrained systems. EngineCL is optimized to be used in more modest environments, typical of commodity computers, where executions of short duration are performed. These applications are generally recommended for CPUs, and not for models such as OpenCL that offload to accelerators. However, EngineCL is extended, improving efficiencies, after applying various algorithmic and runtime strategies. On the other hand, the integration with hybrid technologies to exploit molecular dynamics problems is presented. The problem arises from a collaboration in an international research center, where the ease of porting parts of a molecular dynamics simulator to OpenCL technology is evaluated. With the advantages of EngineCL related to the exploitation of heterogeneous systems, and knowing the drawbacks of OpenCL applied to this simulator, the runtime is extended. The combination of programming models and a ver-

satility in the way of executing these problems, preserving co-execution and high usability, makes it possible to efficiently exploit the kernels of the simulator.

Chapter 5 proposes the CoexecutorRuntime runtime system to exploit co-execution using the novel oneAPI technology. This chapter details the approach chosen for its conceptualization, much more cohesive with the underlying technology itself. Focused on a high-level programming model but with all the extensions provided by the oneAPI framework and its modules. It is a runtime that has evolved as new versions of oneAPI have been released, a technology in constant development. For this reason, this chapter presents a series of design decisions different from those seen in Chapter 3, facilitating its adaptability and tuning to the oneAPI APIs and extensions. In addition, the potential of the dynamic mechanisms that have been propitiated in the proposed runtime, previously non-existent, is highlighted. Then, the runtime is validated in commodity and HPC architectures, exposing the benefits that this approach provide.

Finally, the thesis ends with Chapter 6, where conclusions as well as future lines of work are presented.

Background & Related Work

CHAPTER 2

Background & Related Work

The emergence of heterogeneous devices and specialized architectures has led to the development of programming languages for the efficient use of these systems. The variety of applications and characteristics of the accelerators, leads to the existence of various applicable programming models. These models end up materializing, over the years, de facto standards used by all types of manufacturers in heterogeneous computing. Since computing devices are used to squeeze out maximum efficiency, scheduling and co-execution algorithms are designed to adapt to the problems and applications to be solved, as well as to the performance properties of the devices. This complexity, together with the limitations and density of the programming required to efficiently use these architectures, leads to the conceptualization and development of runtime systems to facilitate the work of programmers. This chapter presents the technologies, programming models and algorithms used in the thesis, as well as exposing the work related to the research carried out.

Chapter contents

2.1 Technologies & Programming languages

Solutions to make parallel programming simpler and more suitable for programmers are often focused on exploiting the various existing programming models that abstract away from the complexities of parallelism, concurrency and the underlying architectures. Implementations of such models often reach down to lower layers of utilization and optimization, making it possible to manipulate different drawbacks that affect efficiency, as discussed in Chapter 1.

This section provides an overview of parallel programming languages and frameworks for heterogeneous systems, focusing on the two exploited throughout the dissertation, OpenCL and oneAPI, explained in more detail.

2.1.1 Overview

There are many variations of languages, new proposals and even combinations of several existing ones, making it difficult in many occasions to delimit which are the innovations they introduce due to the real complexity of programming ecosystems. It is an incipient field that is constantly nourished by ideas, but some programming models and implementations stand out from the rest, either by versatility, performance or popularity in certain areas. Thus, the most widely known technologies and languages for programming heterogeneous systems are the following.

CUDA, **Compute Unified Device Architecture**[1], is a proprietary general purpose computing programming language for graphics cards (GPGPU), created by Nvidia for its devices [24, 187, 188]. Being one of the first and intimately linked to the growth and potential of the Nvidia GPUs, it is mature and well-supported, being popular worldwide. It offers low-level and high-level APIs, and its runtime offers a software layer that allows the use of a virtual set of GPU instructions, being generally exploited from C, C++, Fortran and Python. This GPGPU approach was one of the first to dispense with shader and texture-based graphics programming, which is low level and not very portable, facilitating the exploitation of scientific, engineering and industrial problems [189–197]. CUDA is so useful, established and important in GPU processing that there are bindings from most generalist programming languages. The philosophy is similar to OpenCL, offering a host-device model of kernel offloading, with comparable execution and memory management principles, albeit with different terms. Each thread executes the same kernel, but with different data, so it is based on the SPMD model. The latest versions support unified memory models, shared memory, multi-GPU support, optimized memory transfer strategies, tensor cores support, dynamic parallelism and half-precision floating-point operations. Moreover, they have multiple optimized libraries for bitwise and linear algebra exploitation, graph analysis,

[1] `https://developer.nvidia.com/cuda-zone`

FFT calculation, image processing or neural networks, among others. The major limitation is its adherence to Nvidia technologies, being designed specifically for their GPUs, and not any type of accelerator and manufacturer.

OpenMP, **Open Multiprocessing**[2], the API and runtime focused on shared memory architectures and used for more than two decades, with great consolidation in all types of environments, especially HPC [198–200]. Originally focused on the fork-join model, which manages the parallel execution of a set of threads within a process, sharing the address space. It has directives, functions and environment variables, typical components of a language based on pragmas. These components greatly facilitates programmability, easing its integration as part of other libraries, both native and as wrappers for other languages. Two of its main advantages are ease of use and incremental parallelization and porting, incorporating parallel regions gradually, in addition to having support for sequential execution as fallback compatibility feature. In this way, this framework for parallel programming is not very intrusive, facilitating the exploitation of these techniques in an optional fashion, wherever there are exploitable resources. As with all languages, new programming modes have been incorporated over the years, including task-based parallelism, extension of synchronization primitives, support for vector directives or atomic operations, among others. With respect to heterogeneous computing, it is from OpenMP 4.0 and 4.5 versions onwards where directives for the support of accelerators are offered [199]. The features offered focus on two areas, program execution and data management, using constructs such as `target`, `teams` and `map`, among others. It is gradually gaining more adoption, due to its simplicity and worldwide use, but there is still work to be done in order to have an adequate support for the existing heterogeneity [201–210].

OpenACC, **Open Accelerators**[3], is another programming standard specifically focused on heterogeneous computing, although originally centered on CPU-GPU systems [95, 211]. Like OpenMP, it is based on the annotation of C, C++ or Fortran code through the use of compiler pragmas and directives, enabling the movement of memory and the launch of computational code on devices. In fact, members of the OpenACC committee have acted as members of the *OpenMP standard group* to join forces and create common specifications for accelerator support. In addition to having directives for data movement (`data`, `declare`) or region execution (`loop`, `kernels`), it has support for a runtime API, facilitating the extension of some functionalities without the need for pragmas. This standard aims to facilitate the programmability of heterogeneous systems, with a philosophy similar to OpenMP in terms of degree of abstraction and incremental acceleration [193, 201, 208, 212–215]. However, this abstraction limits the control of the programmer, although some mechanisms are provided to orchestrate grouped and more complex operations (`workers`, `gangs`, `vectors`), with ideas close to the OpenCL standard. On the other hand, some of the

[2] `https://www.openmp.org`
[3] `https://www.openacc.org`

complications of this standard are the difficulty of incorporating these specifications in implementations, the limitation in the support of accelerator types and manufacturers, as well as the experimental degree of some of its contributions in the most well-known compilers, hindering its popularity and support .

HSA, **Heterogeneous System Architecture**[4], is a cross-vendor initiative to enhance portability by defining a set of specifications that facilitate data movement and execution between CPUs and accelerators, such as FPGAs or GPUs [216, 217]. This hardware platform and stack software focuses on enabling processors of different types to work cooperatively and efficiently through shared memory, although it was initially conceived for CPU-GPU work. This proposal defines the management and dispatch protocol for kernels, so that compilers can map constructs that describe parallelism and behavior. This abstraction makes it possible for other languages to express themselves in these terms and then provide abstraction over the kernel code itself. The code is compiled to an intermediate language as a virtual ISA, called *heterogeneous system architecture intermediate language (HSAIL)*, offering an abstraction on the types of manufacturers and devices, and allowing the subsequent optimization and construction of the ISA for each of the devices finally used. One of the disadvantages is the complexity of offering HSAIL transformations for other devices that do not support instruction-based executions (GPUs), complicating the mapping to FPGAs and DSPs [216, 218–222].

TBB, **Threading Building Blocks**[5], now transformed to **oneTBB** after the importance acquired by oneAPI, is a C++ template library developed by Intel to exploit parallel programming on multi-core processors [223, 224]. It provides a runtime and API with a set of primitives, operations, algorithm skeletons and data structures optimized for parallelism. The programmer has to divide problems into tasks, being able to group them, establish dependencies and schedule them for execution [225, 226]. TBB offers load balancing and work stealing strategies, easing the life of the programmer while trying to take advantage of all cores. Although it initially had bottlenecks and overheads due to dynamic capabilities, it is a technology that continues to be used and optimized to this day. One of the advantages of this template-based technology is the possibility of exploiting high-level strategies, such as polymorphism, incurring in low overheads. Another is the cohesion with other Intel technologies focused on other architectures, problem types and libraries, facilitating their exploitation in a cooperative way, as is observed with oneAPI, oneMKL or oneDNN [23, 227–230]. However, although oneTBB is a proposal in a different direction and with a great trajectory, it is focused on multi-core exploitation and is closely linked to the manufacturer, limiting heterogeneous exploitation [231–234].

SYCL[6] is a Khronos standard that offers a cross-platform abstraction layer that builds

[4] `https://hsafoundation.com`
[5] `https://intel.com/oneTBB`
[6] `https://www.khronos.org/sycl/`

on the concepts, efficiency and portability provided by OpenCL for programming heterogeneous platforms [98]. Its philosophy is to use a single-source code to program the heterogeneous devices, directly using C++. Simplification and abstraction over heterogeneity is achieved through the work of the compiler and runtime, although they depend on the very specific implementations of the standard. In any case, they consolidate a task graph as the fundamental piece to orchestrate devices and offload kernels. Originally conceived as an abstraction on OpenCL, over the years it has been maturing and allowing the encapsulation of other technologies and systems such as **Level Zero** [235], vector computers [236] and **Vulkan** platforms [237]. Thanks to the efforts and proposals of the different implementations, it has been possible to make a leap with the latest version of the SYCL 2020 standard, allowing to generalize the original backends model [90, 91]. In addition, the latest changes have favored extensions to simplify its programmability, apply parallel reductions, exploit pointers and shared memory, or improve interoperability between technologies, among others. SYCL has been gaining relevance over time, precisely because of its proximity to C++, being used in most industrial and scientific applications [92, 238–241]. For this reason, libraries and compilers have emerged around SYCL, both from industry firms, such as ComputeCpp or oneAPI [237, 242, 243], and community and academia proposals such as hipSYCL or triSYCL [91, 238]. In fact, this initiative is becoming so important that the second proposed runtime, presented in Chapter 5, is based on one of the SYCL implementations.

C++ AMP, **Accelerated Massive Parallelism**[7], is a programming language that extends C++ and its runtime library to support GPUs [244]. Proposed by Microsoft and with an original implementation based on DirectX 11, but with integrations in proposals from other vendors, like a HSA implementation of AMD. It was initially focused on GPGPUs, offering algorithms based on data parallelism and modifications to the C++ language to address the limitations of such hardware architectures. It uses C++ lambda functions for kernel code generation, transferring data implicitly through the use of `array_view` objects to represent contiguous memory regions to be used during execution on GPUs. Efforts are focused on providing the same code for CPU and GPU, checking the feasibility of kernel execution on the accelerator, supported language features and providing fallbacks for the CPU [245–250].

OmpSs[8] is a pragma-based model with design principles similar to OpenMP, but focused on task-based parallelism [150]. It facilitates the programming of heterogeneous architectures using extensions and backends, supporting other programming languages such as OpenCL or CUDA [251, 252]. The programmer must indicate the tasks and their interrelationships, so that the runtime is able to orchestrate the kernels, their dependencies and the compute nodes involved. The source code is compiled according to the processing element of the underlying system, producing different object codes that are scheduled at

[7] `https://docs.microsoft.com/en-us/cpp/parallel/amp/cpp-amp-overview`
[8] `https://pm.bsc.es/ompss`

runtime [252–256].

Just as Nvidia has always promoted CUDA, AMD has done the same with different initiatives, but always in a cross-cutting and continuous way, contributing with OpenCL support.

ROCm[9], **Radeon Open Compute**, is one of the latest initiatives as part of their *Boltzmann Initiative* and the *open computing platform*[10]. It has a modular design that allows any hardware manufacturer to adapt its drivers to the ROCm stack, integrating programming languages such as OpenCL or HIP [257–260]. Moreover, the ROCm runtime is implemented on top of a HSA-compliant language-independent runtime [261]. However, this platform only supports some operating systems, some modern CPUs and a subset of AMD GPUs, limiting its usefulness for now, although offering performance similar to Nvidia GPUs [262]. AMD worked on other initiatives over the years, being *Compute Abstraction Layer* (CAL) and *Accelerated Parallel Programming* (AMD APP) among the most relevant [263–265]. Some of these frameworks and libraries provide lower level features, while others offer more abstract interfaces close to C++, but always focused on exploiting the use of their GPUs. Nevertheless, these efforts have offered and improved OpenCL backends to benefit from the same drivers and optimization techniques.

HC C++, **Heterogeneous Compute C++**[11], is an AMD initiative inspired by C++ AMP, SYCL and C++17, with a proposed API for heterogeneous computing with C++ [266]. It makes modifications to the C++ AMP language, providing access to lower-level constructs, such as the synchronization primitives and custom memory transfers [267, 268]. Their goal is to take advantage of cutting edge language features and computing characteristics of devices while enabling increased productivity. However, while uses are still being found, such is the movement between proposals that AMD has ended up favoring these efforts around OpenCL and HIP. **C++ Heterogeneous-Compute Interface for Portability**[12] (**HIP**) and the HC language share the same compilation technology, so many features of the language exploited in kernels are leveraged from HIP [269]. This proposal has two ways to compile the code depending on the execution platform, exploiting both Nvidia CUDA and AMD ROCm. The runtime API and language have been designed to be CUDA-compatible, enabling performance on par with that offered by the CUDA platforms while providing additional low-level features. CUDA is so widely used that this proposal is important to enhance code portability, including conversions to OpenCL platforms not supported by CUDA or ROCm [268, 270, 271].

These languages offer different drawbacks for heterogeneous computing and the objectives of this thesis. On the one hand, all those based on pragmas and directives, such as OpenMP, OpenACC or OmpSs, are relatively easy to port, generate an additional layer of abstraction with respect to the code. Directive statements are generally simplified *domain*

[9] https://github.com/RadeonOpenCompute/ROCm

[10] https://gpuopen.com https://www.amd.com/en/graphics/servers-solutions-rocm-hpc

[11] https://github.com/RadeonOpenCompute/hcc/wiki

[12] https://github.com/ROCm-Developer-Tools/HIP

specific languages (DSLs), transforming automatically the code to do complex behaviors with just a few words in a single line. However, the main disadvantage is the limited flexibility, making development, compilation and evaluation cycles more difficult. These DSLs are language and version dependent and cannot offer all the versatility of the language in which the rest of the code is expressed, so they are often constrained. Additionally, the programmer has no way to extend such syntax or provide variations on the structures or behaviors provided. Hence, programmers that want to make derivations of features present in the standard, all the functionality should have implemented from scratch, in another technology or in low-level primitives, if they are exposed at all.

On the other hand, many of these languages are tied to the vendors and the hardware they manufacture, limiting code portability and real usability. The broader the scope, the more abstraction is required, and in general the more difficult it is to achieve adequate performance. However, in such cases, the proposals will be more useful and applicable to all types of architectures and applications. When vendor-specific proposals gain interest in the community, attracting frameworks and other vendors, they end up being incorporated in other technologies. For instance, this has happened with CUDA through HIP transformations as well as interoperability backends with SYCL.

There are other languages and frameworks that focus on technologies limited to specific operating systems (Android, Windows), software stacks (Renderscript, DirectX, JVM) or even types of problems and their scope (image processing, multi-block structured grids, sparse linear systems, deep learning) [272–278]. These are proposals of interest for these limited environments, but not for generalist proposals of maximum applicability.

Finally, it is important to highlight that even proposals such as OpenMP, known worldwide and supported by most compilers, require years and collaborative efforts by the community and manufacturers to support certain accelerators and the latest stardard features. Therefore, to date, OpenCL is still the proposal that offers the greatest compatibility.

Nevertheless, there are many other languages, libraries, frameworks or specifications for parallel programming in heterogeneous environments, or at least, gradually adapting to them and incorporating support for different types of accelerators. However, the base technologies and programming models on which to build the proposals developed during this thesis have been OpenCL and oneAPI.

2.1.2 OpenCL

Open Computing Language[13] (**OpenCL**) is a parallel programming language and framework for heterogeneous environments on cross-vendor and cross-platform hardware [94, 279]. Khronos Group defined the first specification of this standard in 2008, and since then, the functionalities have been increased up to the recent OpenCL 3.0, which is slowly start-

[13] `https://www.khronos.org/opencl/`

ing to be implemented. Recent contributions have included shared virtual memory, nested parallelism, support for subgroups, extensions for embedded and support for asynchronous DMA operations, increased debugging information, synchronization events or interoperability with other languages such as Vulkan, among others.

The purpose of this open programming standard has been to model a framework that concentrates the features and needs of many types of manufacturers, devices and applications, providing a language that is acceptable for a wide range of competing needs. Therefore, it entails an effort to improve programmability and code portability between different heterogeneous systems. Roughly put, it consists of an extension to C/C++ that allows programmers to shift parts of their code to the accelerators, introducing the Host-Device programming model, already presented in previous Chapter. This API is sufficiently generic to be used in a wide number of different architectures while being adaptable to each hardware platform, achieving high performance. It is important to highlight that the fundamental feature of OpenCL is its adaptability. Using the kernel language as well as the API, a program designed for one device can run not only on different hardware but also on different types of devices, as long as an OpenCL driver is provided. Therefore, it can run on a variety of performance CPUs [94, 233, 280], GPUs [24, 281–283], MIC coprocessors [23, 128, 284], FPGAs [125, 285, 286], DSPs [58, 59, 287] and even accelerators [288].

OpenCL consists of platform, execution, programming, compilation and memory models, explained below.

2.1.2.1 Platform model

The OpenCL architecture consists of a CPU-based host that controls a set of Devices, often denoted as *Compute Devices*, as it is depicted in Figure 2-1. Each of these devices is composed of execution units called *Compute Units* and these, in turn, of *Processing Elements*, which are in charge of executing OpenCL kernels. It is precisely the definition of these entities that allows the abstraction provided on the different types of devices, since it is a decision made by the manufacturers and their drivers. For instance, Compute Units in Nvidia are Stream Multiprocessors (SMs), Stream Cores or SIMD Engines for AMD GPUs and in CPUs, they are generally associated to the logical cores of the system.

The platform model is key for the development of applications that have to be portable between OpenCL-capable systems, even from the same manufacturer or device type. A *platform* is often thought of as a common interface for a specific OpenCL runtime, implemented through a driver. For example, in Section 1.7.1 of Chapter 1, the table of heterogeneous systems shows devices such as *Remo* that even having AMD CPUs and GPUs, presents two different platforms because they contain different drivers. This also applies to *Desktop* and even in *Batel*, where the CPU and Xeon Phi share the OpenCL version and its driver, but not the platforms due to different implementations regarding the hardware interaction. However, throughout this dissertation there are cases where the same platform encompasses two

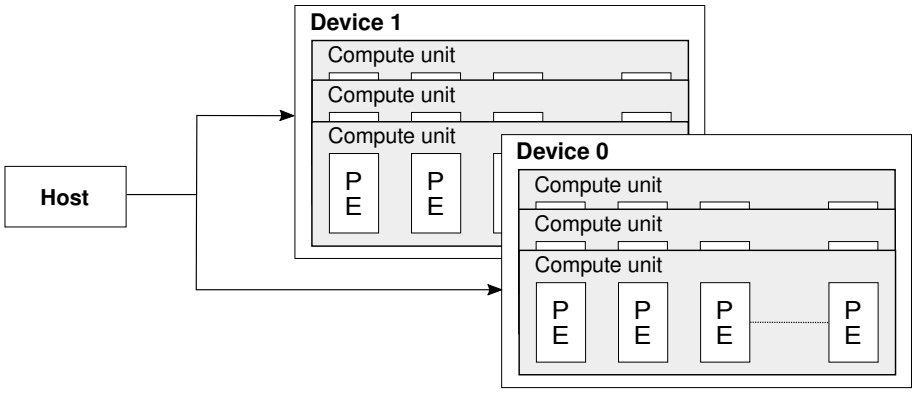


Figure 2-1: Platform model of OpenCL showing a system with two compute devices.

different devices, favoring the sharing of some OpenCL primitives per manufacturer.

This design allows manufacturers to translate this architectural abstraction to physical hardware, fostering one of the fundamental aspects of this model, where each compute unit is functionally independent.

2.1.2.2 Execution model

A *context* is an abstraction on which the operations associated with the devices, both memory management and execution, are coordinated. It manages all the interaction mechanisms between host and device, the control of memory objects, programs and kernels to be executed on each device. A context can contain different devices and resources that can operate between them, facilitating data sharing.

A *command-queue* is a communication mechanism between each device and the host, generally launching operation requests to a specific device through its respective queue. This is a requirement, as each request to the queue is implicitly assigned to the device with which it operates. There are both in-order and out-of-order queues, fetching operation commands in the order received, or depending on the driver and being rearranged at runtime. However, in-order queues are the most commonly used, since out-of-order queues are a mechanism that depends on more interaction from the programmer in terms of OpenCL mechanisms and primitives (event handling and chaining), as well as requiring support by the OpenCL implementations involved. Other common operations requested on command queues are synchronization commands (barriers), event commands and memory transfers.

Generally, *events* are objects used to specify dependencies between commands or to perform queries on OpenCL operations, although custom events can also be created. Asynchronous OpenCL API operations provide support for events that can be used to establish dependencies for future events, as well as to query driver information or wait for state changes.

These primitives and objects are depicted in Figure 2-2. It shows an OpenCL runtime

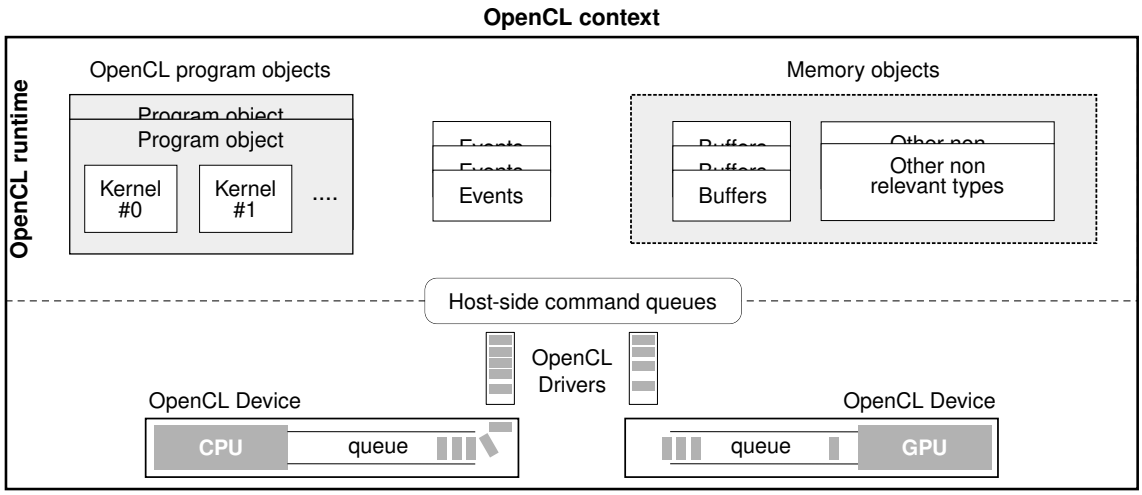


Figure 2-2: Execution model showcasing an OpenCL context managing two compute devices (CPU and GPU) and a set of OpenCL primitives to interact with.

with a context in which two compute devices are managed, sending requests to the CPU and GPU through the command queues (host and device-side). Additionally, other OpenCL primitives are allocated and used as part of the context, such as events, kernels, as well as memory and program objects.

2.1.2.3 Programming model

Code using the OpenCL runtime API executes on the CPU, just as in a classical sequential programming model. However, device code requires more stages to be executed, mainly due to the abstraction offered by this programming model. The code executed on the devices is encapsulated in data-parallel, C-like functions, which are known as *kernels*. Although most kernels are expressed in a subset of C99, it has language extensions, both vendor-specific or language defined, built-in functions and even the possibility of being expressed in C++ with recent versions [94, 289] and proposals [290].

When a kernel is offloaded to a device, OpenCL launches multiple instances of the kernel, each with a different portion of the data, under the Single Instruction Multiple Thread (SIMT) paradigm. Each instance is called a *work-item*, being the unit of concurrent execution in OpenCL C. The programmer can decide how many items are launched by setting a parameter called *global work size*. Work-items are launched in teams so they can cooperate and synchronize with each other. OpenCL ensures that the work-items of each team, or *work-group*, are launched simultaneously in the same compute unit. However, work-groups are run concurrently in the compute units, as a device may not have enough resources to execute them all at once. Work-group size can be defined through the *local work size* parameter.

The hierarchical concurrency model implemented by OpenCL ensures scalability in exe-

cution, allowing a very large number of work-items to be launched. The programmer must set the number by specifying an n-dimensional range (NDRange), although there are mechanisms for launching coarse-grained parallelism (task). The dimensional space of the work-items is mapped to the input and output buffers, depending on the dimensions used, and up to NDRange with a 3-dimensional index space can be used.

A fundamental aspect of operations within a work-group is that internally launched work-items can be synchronized (barriers) and have access to the shared memory address space. Since work-group sizes are fixed per dispatch, communication costs do not rise as for larger dispatches, which is essential to maintain scalability. The actual mapping of work-groups to hardware components is both architecture and OpenCL implementation dependent. On the other side, the synchronization between work-items belonging to different work-groups is undefined. OpenCL devices provide intrinsic functions that allow identifying execution space elements, such as work items, work groups and other concepts, such as relative indices and sizes of the n-dimensional execution space.

The goal is to represent it in a fine-grained parallelism. In this way, the OpenCL interface and the low-level language of the kernels allows a mapping to a diverse set of devices. Conceptually, it is very similar to the parallelism inherent in OpenMP data-parallel loops or functional language map operations.

2.1.2.4 Compilation model

An OpenCL program is a set of OpenCL C kernels, functions and data. OpenCL source code is compiled using runtime APIs, making it possible to create data structures and primitives defined by OpenCL. Later, these kernels are assigned with execution arguments and subsequently can be launched for execution through the command queues. This compilation procedure, although cumbersome, provides the opportunity to optimize OpenCL kernels for the devices to be exploited, which may be previously unknown. This facet is important because it guarantees code portability, since the complete program is compiled in two stages. On the one hand, the main program using the OpenCL API. On the other hand, the JIT-compiled kernel code by the OpenCL drivers and the installable client driver (ICD) loader.

In this manner, independence is provided with the types of devices and manufacturers involved in the target system, delegating the optimizations and heterogeneous exploitation through dynamic mechanisms. One of the advantages of this process is that the construction process can generate both the final binary and intermediate representations, being able to serialize them as binary objects. In this way, the programmer can also store the compiled binaries and reuse them in subsequent executions, as a kernel compilation at compile time, reducing the runtime overhead.

2.1.2.5 Memory model

The memory spaces of hardware devices in a heterogeneous system have generally been separated from the host, and although there are cases of shared memory and techniques to take advantage of these systems, it is not common. For this reason, OpenCL establishes an abstraction on the memory model, considering separate memory spaces and delegating to drivers and future extensions the possibilities of exploiting other situations.

OpenCL distinguish two main types of memory, host and device. The first one involves the available memory for the program, its data structures, OpenCL runtime and primitives. And on the other side, in a separated address space, the memory allocated in the devices accessed by the running kernels. Data is moved between the host and the devices using functions defined by the API, ensuring sufficient memory at both ends. Due to the differentiated memory address space, kernel launches must be preceded by an input data copy phase, from the main memory to the device memory, and followed by another in the opposite direction for the results. For these operations OpenCL uses the concept of *buffers*, which are a host representation of the memory of the devices in a context, being an address that is valid in the memory of the device. There are functions to create and manage buffers on devices, enqueing read or write operations on the associated command queues. Moreover, data transfers can also be blocking or non-blocking, controlling whether the host has to wait for the transfers to complete or can continue executing. However, these copy phases must be explicitly instructed by the programmer, which constitutes a tedious and error-prone task.

OpenCL divides device memory into four regions, *global*, *constant*, *local* and *private*, being associated within a kernel by keywords and identifying the location of variables or arguments. Since the memory regions are logically disjoint, by definition of the memory model, the kernel programmer is in charge of allocations and transfers. An example of these regions and their mapping to two GPU architectures is depicted in Figure 2-3. It depicts the mappings for the *AMD RX5700XT RDNA* and *AMD A10-7850K APU GCN* devices, used in the *Trainera* and *Remo* heterogeneous systems presented in Section 1.7.1. Global memory is visible to all work-items instantiating the kernel, and any memory transferred between the host and the device first passes through global memory. Constant memory is read-only data that is modeled within global memory, but is specifically designed for data that has to be accessed simultaneously by all work-items. Global and constant memory are mapped to GPU video memory in the GCN architecture. Local memory restricts its use to work-items in a work-group, and is generally mapped to on-chip memory, providing shorter latencies and higher bandwidths. This is commonly used as a scratchpad for fast collaboration and data sharing. It is placed in the local data share region, as part of every vector processor of the GPU architecture shown. Finally, private memory is that which is reserved by each work-item, being mainly local variables and arguments of primitive types (nonpointer arguments). Considering each vector processor of the GCN architecture, local memory is

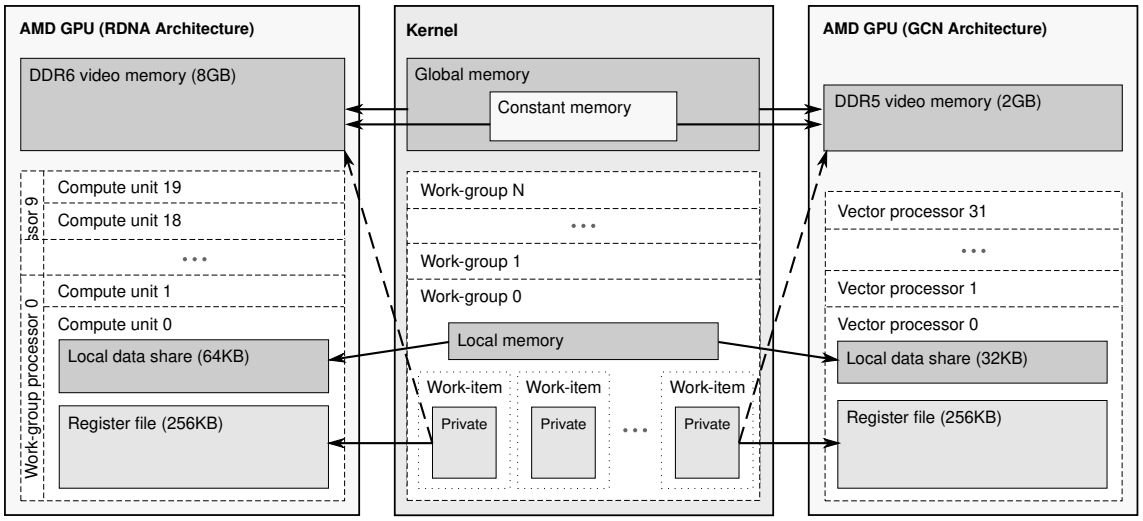


Figure 2-3: Memory model mapping of OpenCL regions to AMD GPU regions (RDNA & GCN architectures).

mapped to the local data share region, while private memory *uses* the register file. However, constant variables of the private memory may be stored in the global video memory.

This abstraction over the memory model provides flexibility, since the mapping of memory spaces to actual hardware is implementation dependent.

2.1.3 Intel oneAPI

Intel **oneAPI**[14] is based on the SYCL specification, although it provides its own extensions to accelerate the computation and facilitate the development [98]. The programming language is called **Data Parallel C++** (**DPC++**), making a leap in abstraction and promoting interoperability with host code, compared to the OpenCL language. DPC++ is a community-driven, standards-based language built on ISO C++ and Khronos SYCL, allowing developers to reuse code across hardware targets. DPC++ allows the host and the device code as part of the same compilation unit, feature called single source property, that allows potential optimizations across the boundary between both codes. Due to this property, DPC++ establishes three types of scope to distinguish between host (application), host-device interface (command group) and device (kernel). OneAPI comprises four models based on SYCL, each of which is part of the operations that a developer has to perform when using oneAPI. Since SYCL was originally designed as an abstraction over OpenCL, it inherits most of the concepts and abstractions highly detailed in the previous, Section 2.1.2. Hence, the key points applicable in oneAPI with respect to the existing models are exposed below.

[14] https://intel.com/oneAPI https://www.oneapi.io/

2.1.3.1 Platform model

Define a host that manages one or more devices and coordinates the application and command group scopes. A device can be an accelerator or the CPU itself, each of which contains a set of Compute Units. In the same manner, each of these provides one or more *Processing Elements*. The complete system could have multiple platforms, since the composition of drivers in execution platforms is determined by the drivers and their implementation.

2.1.3.2 Execution model

It defines and specifies how kernels execute on the devices and interact with the host. It is subdivided in host and device execution models. The data management and execution between host and devices are coordinated by the host execution model via command groups. These are groupings of commands like kernel invocation and memory access (*accessor*, to be detailed later), which are submitted to queues for execution. The device execution model specifies how computation is accomplished on the accelerator, specifying range data sets. These are allocated across a hierarchy of ND-ranges, work-groups, sub-groups, and work-items, easing the programming patterns and their composition. This facilitates the memory and compute operation relationships, giving the programmer flexibility to express the algorithms.

A fundamental concept in the SYCL execution model is the Directed Acyclic Graph (DAG). Each node contains an action to be performed on a device, such as kernel invocation or data movements. The SYCL runtime controls, asynchronously, the resolution of dependencies and triggering of node executions. Thus, it tracks and orchestrates actions and their dependencies to perform in the devices, safely executing each operation when the requirements are met. On the other side, if the handler is not used, the code executes synchronously by the CPU as part of the host program, bypassing the DAG.

2.1.3.3 Memory model

It coordinates the allocation and management of memory between the host and devices, and how they interact. Memory resides upon and is owned by either the host or the device and is specified by declaring a memory object. *Accessors* define the interaction of these memory objects between host and device, communicating the desired location and access mode.

An extension to the standard SYCL memory model is Unified Shared Memory (USM), which enables the sharing of memory between the host and devices without explicit accessors. It manages access and enforces dependencies with explicit functions to wait on events or by signaling a dependency relationship between events. Another important feature of USM is that it provides a C++ pointer-based alternative to the buffer programming model (SYCL Buffers), which increases the abstraction by leaving the migration of memory to the underlying runtime and device drivers. On the other side, since it does not rely on accessors,

dependencies between command group operations must be specified using events to help the compiler determine the data dependencies and patterns.

2.1.3.4 Kernel programming model

The kernel is the computing function instantiated to be executed by every processing element of the accelerator. It allows the programmer to determine what code executes on the host and device, giving an explicit computing function via lambda expression, functor or kernel class. Therefore, the separation of host and device codes is straightforward, without language extensions. Device code can specify the parallelism mechanism with a coarse-grained task, data-parallel work or data-parallel construct taking into consideration the hierarchical range of the execution model. It supports the *single source* property, meaning the host code and device code can be in the same source file. Therefore, it improves usability, safety between host and device boundaries (matching kernel arguments) and optimization strategies due to better understanding of the execution context (aliasing inference and propagating constants). Finally, DPC++ kernels execute asynchronously via forced allocations of kernel class instances, implicit waits of C++ destructors or explicit queue waits.

2.2 Load Balancing Algorithms

A fundamental consideration for successful co-execution is an effective workload distribution between the host and the devices. Therefore, the load balancing algorithms used in the experiments along this dissertation are briefly described.

It is necessary to have a sufficient variety of algorithms, since there is generally no scheduling strategy for data parallelism that is always the best for all types of situations. Moreover, it is important to contrast the behavior of the different proposals, runtimes and technologies to know their differences and implications for their utilization and exploitation. Algorithms can be divided into static or dynamic. Static algorithms are usually simpler and easier to implement, generally achieving low overhead and synchronization since the partitioning decisions are made in advance [156]. However, they are less adaptable to the type of workload, so they tend to suffer with irregular problems. On the other hand, dynamic algorithms have more synchronization issues because they distribute the workload on demand, adapting themselves at runtime [82, 126, 291]. Three load balancing algorithms are chosen, one static and two dynamic. These offer enough diversity to study the behavior of the problems, runtimes and the heterogeneity of the system. To facilitate understanding the behavior of the algorithms detailed below, Figure 2-4 depicts a comparison among the three load balancing algorithms in real executions. The Y axis shows every algorithm and the package distribution per device, using the host device (*CPU*) and an accelerator (*ACC*), while the X axis reflects the execution time per benchmark (Ray and NBody). Every rectangle is a

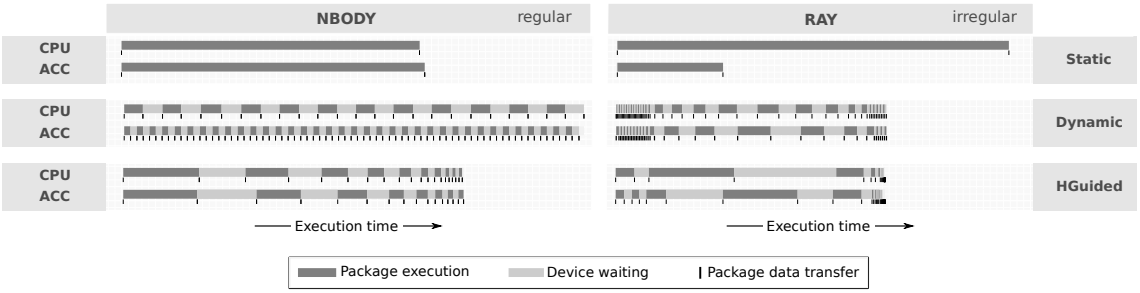


Figure 2-4: Package distribution in real executions for irregular and regular problems using the Static, Dynamic and HGuided load balancing algorithms.

work package launched to a specific device. As can be seen, every algorithm balances perfectly the load. For the sake of completeness, this chart represents an ideal situation in which all the algorithms have achieved a perfect balance efficiency, simultaneously finishing both devices. Additionally, the packages data transfer, both writing and reading, are almost negligible compared with the packages execution and device idle times. This is a real behavior found, but it can vary drastically due to the complexity of the heterogeneous systems, architectures and technologies involved, as will be seen throughout the experimentations of the dissertation.

2.2.1 Static algorithm

This algorithm works before the kernel is executed by dividing the data-set in as many *packages* as devices are in the system. To take heterogeneity into account, the division relies on knowing the computing power of the devices in advance. Then the execution time of each device can be equalized by proportionally dividing the data-set among the devices.

Considering a heterogeneous system with n devices. Each device i has *computational power* P_i, which is defined as the amount of work that a device can complete per time unit, including the communication overhead. These powers are parameters that must be given to the algorithm and can be extracted by profiling. Then, the total computational power of the heterogeneous system is the sum of the individual powers of the devices $P_H = \sum_{i=1}^{n} P_i$.

The application will execute a kernel over W work-items, grouped in G work-groups of fixed size $L_s = \frac{W}{G}$. Since the work-groups cannot communicate among themselves, it makes sense to distribute the workload taking the work-group as the atomic unit. Each device will have an execution time of T_i. Then the execution time of the heterogeneous system will be that of the last device to finish its work, or $T_H = max_{i=1}^{n} T_i$.

The goal of the Static algorithm is to determine the number of work-groups to assign each device, so that all the devices finish their work at the same time. This means finding a tuple $\{\alpha_1, ...\alpha_n\}$, where α_i is the number of work-groups assigned to the device i. Therefore, the expression used by the algorithm is:

$$\alpha_i = \left\lfloor \frac{P_i G}{\sum_{j=1}^{n} P_j} \right\rfloor \tag{2-1}$$

If there is not an exact solution with integers then $\sum_{i=1}^{n} \alpha_i < G$. In this case, the remaining work-groups are assigned to the most powerful devices.

The beauty of the Static algorithm is that it minimizes the number of synchronization points. This makes it perform well when facing regular loads with known computing powers that are stable throughout the data-set. However, it is not adaptable, so its performance might not be as good with irregular loads.

2.2.2 Dynamic algorithm

Some applications do not present a constant load during their executions. To adapt to their irregularities, the Dynamic algorithm divides the data-set (*work-groups* of OpenCL) in small packages of equal size. The number of packages is well above the number of devices in the heterogeneous system. During the execution of the kernel, a master thread in the host is in charge of assigning packages to the different devices, including the CPU, following the steps shown below:

Input: G number of work-groups,
N devices,
package_size package size multiple of work-group size

$G_r \leftarrow G$ *(Number of remaining work-groups)*
$C_r \leftarrow 0$ *(Number of work-groups computing)*
for $j \leftarrow 1$ ***to*** N **do**
  $c_i \leftarrow min(package\_size, G_r)$
  if $c_i > 0$ **then**
    Schedule c_i work-groups on device d_i
    $G_r \leftarrow G_r - c_i$
    $C_r \leftarrow C_r + c_i$

while $C_r > 0$ **do**
  $(d_i, c_i, r_i) \leftarrow$ Wait for any device
  Merge partial results r_i in host memory
  $C_r \leftarrow C_r - c_i$
  if $G_r > 0$ **then**
    $c_i \leftarrow min(package\_size, G_r)$
    Schedule c_i work-groups on device d_i
    $G_r \leftarrow G_r - c_i$
    $C_r \leftarrow C_r + c_i$

The master splits the G work-groups in packages, each with the package size *package_size*

specified by the programmer. This number must be a multiple of the work-group size. If the number of work-items is not divisible by the package size, the last package will be smaller. In an initial stage, the master launches one package c_i on each device, including the host itself if it is desired. Then, it waits for the completion of a package given to any device until there are no more pending work-groups computing. When device d_i completes the execution of a package:

1. The device returns the partial results corresponding to the processed package.
2. The master stores the partial results in host memory, merging them with previous ones.
3. If there are remaining packages, a new package c_i is launched on device d_i.

This algorithm adapts to the irregular behaviour of some applications. However, each completed package represents a synchronization point between the device and the host, where data is exchanged and a new package is launched. This overhead has a noticeable impact on performance. The Dynamic algorithm takes the size of the packages as a parameter. The time to process a package of equal size is the same in regular applications, while it is not in irregular ones, like it is depicted in *Dynamic* in the Figure 2-4.

2.2.3 HGuided algorithm

The previous strategies have their strong points and their weak spots. Although neither is the best for every application, both give hints toward an optimal data-division algorithm. The Heterogeneous Guided algorithm (*HGuided*) is an attempt to reduce the synchronization points of the Dynamic while retaining most of its adaptiveness.

The same algorithm used in the Dynamic approach is applicable to the HGuided, except for how the data-set is divided. The HGuided algorithm makes larger packages at the beginning and reduces the size of the subsequent ones as the execution progresses. This reduces the number of synchronization points and the corresponding overhead, while retaining a small package granularity towards the end of the execution to allow all devices to finish simultaneously, as can be seen in *HGuided* in the Figure 2-4.

Since it is an algorithm for heterogeneous systems the size of the packages is also dependent on the computing power of the devices. The size of the package for device *i* is calculated as follows:

$$package\_size_i = max\left(\left\lfloor \frac{G_r P_i}{k \sum_{j=1}^{n} P_j} \right\rfloor, min\_package\_size\right) \quad (2\text{-}2)$$

where k is an arbitrary constant, and the smaller the constant, the slower decreases the package size. Setting this constant appropriately prevents too large package sizes when there are only a few devices, with cases such as giving half the workload in the first package to a device, unbalancing the load. Generally, a value of k between 2 and 3 provides the best results. G_r is the number of pending work-groups and is updated with every package launch.

P_i is the computational power of the device i, while P_i and P_j obtains the computational power ratio compared with the n devices of used in the computation. Finally, HGuided uses the computing powers of the devices and the minimum package size as its input parameters, being the minimum package size a lower bound for the $package\_size_i$. Furthermore, this algorithm is slightly improved for a specific scenario by performing parameter tuning during the Integration I of Chapter 4.

2.3 Related Work

The rise of heterogeneous computing and the proliferation of a rich variety of architectures have led to research and propose a huge number of programming languages, frameworks and libraries to exploit these heterogeneous systems comfortably and efficiently. Such is the complexity and variety that in-depth studies are periodically carried out to provide the state of the art and help other researchers and developers, both from industry and academia, to understand the implications of the different existing technologies [261, 265, 278, 292–295]. In addition to establishing an overview of the hardware and software tradeoffs in heterogeneous computing and the importance in the present and future [86, 217], specific visions related to development patterns in parallel programming [296], by device [297], by programming paradigm [298] or by execution environment [299], among others, have been discussed.

This multi-objective problem, both due of the implications in maintainability and programmability, generally opposed to a low level of detail and accessible optimization, together with the possibilities of extension and compatibility, determine the portability of performance and effective exploitation of heterogeneous systems. Moreover, on many occasions, as a result of the work on facilitating programmability, mechanisms are provided that determine the way of interacting with the devices, or algorithmic proposals are made directly to facilitate load distribution, with even greater implications on the performance obtained. For this reason, three large blocks of work are distinguished: programming models and languages as integral solutions, frameworks and proposals to improve abstraction, and those related to load balancing and scheduling optimization.

2.3.1 Programming models

The innovation in parallel programming models and languages for heterogeneous systems offers an ambitious approach, generating an integral solution from which implementations, derived technologies and runtime libraries emerge. The languages and solutions listed here fall into the category of high-level languages, as they offer a higher level of abstraction, generating an intermediate layer to facilitate programmability. Low-level languages, sometimes referred to as native languages, have been described in the Section 1.2 for their relevance

in programming heterogeneous devices, and have generally served as the basis for building abstraction proposals.

High-level programming models can be categorized under different application scopes. On the one hand, those based on specific languages, intimately associated with their syntax and expressiveness, as in the case of C++ [300–305], due to its strong popularity in industrial projects. These proposals respect the modern development philosophy and incorporate the features of the latest standards, sometimes unifying host and device code, offering type inference and favoring the use of templates, among others. This language-speficic approach also arises with other languages, such as Rust [306], Java [307, 308], Julia [309], Javascript [310], or Python [311], although some are often associated with specific frameworks and acceleration uses, such as Tensorflow or PyTorch [312, 313]. The advantages of this variant are the enhancement of code reusability and maintainability, as well as a level of abstraction and performance similar to that provided by the level given by the language itself, being something beneficial in the first listings.

Since C++ has so much relevance, in many instances it is possible to differentiate its exploratory aspects based on the *Standard Template Library* (STL), such as one of the best known and previously presented, TBB. There are generic runtimes based on asynchronous mechanisms for task execution to different backends [314, 315], using only specific GPUs [316], facilitating programmability based on common patterns [317], but also including the managing of concurrent data structures [318].

There are also proposals focused on programming models based on skeletons, often referred to as *algorithmic skeletons* since they are high-order functions that implement common computational patterns, such as *map*, *reduce* or *scan*. These languages provide their own predefined data structures and operations, determining dependencies and performing code transformations to different backends [319, 320] or to a particular technology [321–323]. However, this approach suffers complications when porting existing code or expressing certain non-trivial programming patterns with the provided skeletons.

Finally, there are other models with less common work lines, such as directive-based and domain-specific programming models. In addition to the most well-known standards and models previously explained, there are academic works that offer code constructs and annotations, delegating to compilers and runtime systems the work of offloading, optimization and parallelization [324–327]. These works improve productivity and enable incremental porting, but there are often difficulties when integrating and adapting certain parallel patterns. Considering specific environments and applications, there are several works that facilitate heterogeneous programming using domain-specific languages, from the application of multi-block grids to solve sparse linear systems [328, 329], to facilitating parallel image processing [330], to defining high-level parallel languages for graph analysis and task scheduling [331, 332]. There are also instances where interoperability efforts are made between technologies and languages, such as the possibility of exploiting mobile devices and

their low-level languages for GPU acceleration [333] or the possibility of describing high-level structures centered on the object-oriented paradigm, but with layers of translation, finally mapping to OpenCL and CUDA code [334, 335].

2.3.2 Abstraction

Abstracting the heterogeneous system and the underlying hardware architectures is a task addressed from different perspectives, but generally focused on providing a runtime or framework that facilitates programmability. Most proposals are centered on providing runtimes that exploit specific technologies [336–341], although there are also solutions that choose to combine them [123, 124, 339, 342–350]. There are works that focus on kernel modifications and source-to-source transformations, detecting memory access patterns and extending the code to support more devices [123, 124, 320, 336, 342, 348, 351, 352].

Most proposals address solutions that orchestrate the system by facilitating the exploitation of task parallelism [323, 353], through the use of heuristics and reuse of historical information [339, 343, 354], through prediction models [129, 337] or replacement techniques [346, 347]. On the other hand, work focused on co-execution techniques on improving programmability by identifying computational patterns and distributing the workload [123, 124, 351, 352], as well as facilitating the merge of results from different devices [342].

As detailed in the programming models approach, there are many projects aiming at high-level parallel programming in C++, although two major blocks can be distinguished to provide abstraction to the heterogeneous system and facilitate programmability. There are those that offer an STL-like API [314, 355–357], sometimes being applied as backends of other technologies [315, 358, 359], increasing the efficiency for modern runtimes. And on the other side, common approaches to provide abstraction are the pattern-based proposals and algorithmic skeletons [319–321, 323, 353, 360, 361]. However, there are also lower level proposals with simpler toolchains, which make efforts to abstract the system using only the C language [291, 362, 363].

2.3.3 Load balancing

One of the fundamental aspects to improve the efficiency in the exploitation of the heterogeneous system is the scheduling and load distribution mechanisms among the available resources. There are authors who have proposed techniques to address both objectives of energy consumption and performance, based on monitoring their devices and tasks and doing frequency throttling [364, 365] and task-based scheduling based on energy efficiency tracking [345], but most proposals are focused on performance improvement. Several works are based on leveraging the task-based paradigm, where devices use coarse-grained kernels, with the runtime acting as an orchestrator, often taking into account execution history to de-

cide future assignments. Some are centered on performing profiling of devices, kernels and data transfers [339, 366, 367], others by using regression models [337], classification [368], machine learning [369, 370] and greedy algorithms [343]. Under this prism there are works that focus on performing concurrent secondary kernel enqueueing to devices, if previous kernels do not perform well and achieve full occupation of the available resources [348, 349, 371, 372], as well as applying work-stealing techniques [346, 347].

However, co-execution involves another set of challenges, usually associated with the efficient partitioning of data among hardware resources, often performed with static approaches [122–124, 351, 352, 373, 374]. The main problem, as stated in Section 1.3 of Chapter 1, is that these strategies lack the adaptativeness to efficiently exploit all kinds of heterogeneous problems and systems. Although there is work focused on prior training and performance modeling [375–377], they continue to suffer penalties in the face of new irregular problems.

There are works that propose dynamically modifying the assigned workload, either by performing throughput measurement strategies of the packages sent to a device [342, 378], using heuristic algorithms [379], or by studying the behavior in the first packages to determine the rest of the problem [155, 157]. Some work focuses on defining a weight-based scheduling [155, 157], relying on studies that determine the behavior of GPUs for mainly regular problems [158], which indicates that the throughput follows a logarithmic curve with respect to the package size. However, there are also proposals that adapt this study to deal with irregularity by creating a model that predicts the throughput of future packages based on the logarithmic approximation of the throughput of previous packages [159]. Finally, recent works propose an adaptive effortless load balancing algorithm, by measuring the performance of the devices and tunning internal parameters, following a logistic function to set package sizes at runtime [127].

EngineCL

CHAPTER 3

EngineCL

This chapter proposes *EngineCL* as a runtime to satisfy the main problems of heterogeneous computing while serving as a tool to exploit new scheduling proposals. It is a comprehensive solution with a modular architecture focused on programming efficiently heterogeneous devices. It is originally built as an abstraction over OpenCL, managing by itself all the operations necessary to make use of the existing devices in the system, without a programmer having to use the low level framework. The runtime has been built with two principles in mind: usability and performance. Its purpose is to overcome these conflicting objectives, offering high maintainability while exploiting the heterogeneous system in the best possible way.

The proposal is validated exhaustively in terms of usability, performance and energy efficiency. Regarding usability, a wide variety of well-known and widely used Software Engineering metrics have been analyzed. Hence, the maintainability offered by the EngineCL API has been contrasted with respect to OpenCL. The performance has been validated on two different nodes, an HPC system and a commodity system, with six different architectures to show the compatibility and efficiency of EngineCL. A set of programs have been evaluated both using a single device under the host-device programming model and co-executing with all the devices in the system, exploiting the load balancing algorithms implemented in the software architecture. Finally, an evaluation of the energy efficiency achieved by the runtime is exposed, highlighting the improvements over using the most energy-efficient device.

Chapter contents

3.1 Motivation

As it is exposed in Section 1.5, the main objective to increase performance and energy efficiency is to simplify and enhance *heterogeneous co-execution*, that is, the simultaneous execution of a single massive data-parallel kernel in a set of devices with different architecture and computing capacity. This objective plans a range of challenges to be addressed and solved, which can be grouped under three fundamental concepts: abstraction, performance portability and usability.

Challenge 1: Abstraction. OpenCL forces the programmers to manipulate a set of low-level operations that require a thorough knowledge of the underlying architecture of the heterogeneous system [125, 291]. Thus, they are burdened with discovering the available platforms and devices, defining buffers and distributing data among all devices, launching the execution of kernels, as well as collecting partial results and organizing them properly. All this greatly complicates the programming of heterogeneous systems, reducing productivity and making it very prone to errors. Specifically, data management is a very complex aspect, since in general devices have separate memories [123, 124, 342, 351, 352]. Moreover, even if there are devices with shared memory and other architectural strategies, the technology may limit these features [126]. Therefore, developers must create and manage buffers for each device and memory region, allocate a part of the data, retrieve partial results and organize them properly to obtain the final result of the application.

To minimize the co-execution effort, it is necessary that programmers do not know many of the details of the underlying architecture present in the heterogeneous system. The solution to these problems is to offer tools that provide a higher level of abstraction. These tools must take care of all the tasks mentioned above, in a completely transparent way for them, or at most with a minimum specification support on their part.

Challenge 2: Performance portability. OpenCL solves code portability, meaning that the same kernel can run on different heterogeneous systems. However, performance portability goes one step further. The idea is that the same code harnesses the computational capacity of processing resources of different heterogeneous systems. For this, two key problems have been identified: load balancing and differences in the architecture of accelerators.

Considering the load balancing problem, its objective is to distribute the workload among all the devices in the system proportionally to their computing power. To do this, it is necessary to consider both the heterogeneity of the system and the behavior of kernels that can be regular or irregular, as it was detailed in Section 1.3 [128, 291]. In the former, two workloads of the same size always spend the same time on the same device. However, in irregular kernels this does not happen, so it is necessary to have a dynamic and adaptive algorithm that is able to distribute the workload at runtime and can adapt to the behavior of the application.

Regarding the architecture of the accelerators, these hardware devices are designed to accelerate the execution of applications with specific properties. For instance, GPUs favor the execution of massively data-parallel kernels through using multi-threading, while FPGAs favor the execution of kernels with deeply segmented implementations. This means that it is often necessary to adapt a kernel implementation to achieve maximum performance from a particular device. On the other hand, compiling for some devices, such as FPGAs, is time consuming, thus it is necessary to pre-compile and provide the binary code at runtime. However, in others, on-line compilation can provide advantages, as some parameters can be tuned for the specific architecture to be used. Therefore, it is important to offer the possibility to manage device-specific kernels, as well as the possibility to work with binary kernels or source code.

Challenge 3: Usability. OpenCL is a powerful programming language since it allows code portability between different devices. However, this feature increases the complexity of the language and its effective utilization. The current OpenCL API is very tedious and presents a wide variety of functions with multiple and complex parameters, as can be seen below for a simple memory operation [291].

```
clBuffs[0] = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_ALLOC_HOST
  _PTR, sizeof(float) * numEntries, buffHostPtr, &err);
clBuffersMap[0] = clEnqueueMapBuffer(cmdQueue, clBuffs[0], false, CL_MAP
  _READ, offset, sizeof(float) * numEntries, numEventLst, eventLst,
  &event, &err);
```

On the other hand, there are currently several OpenCL specifications available and different devices support one specific version or another. The differences between these specifications are noteworthy, so that an application programmed to run on an OpenCL 2.0 device cannot run on a device that only supports the OpenCL 1.0 specification. For instance, the basic function for queueing a kernel launch, `enqueueNDRangeKernel`, has a different number of parameters in OpenCL 1.0 and 2.0 [125]. Another example is the support of mechanisms for the synchronization between host and device, since some specifications support asynchronous callbacks, while others force the use of blocking communication mechanisms [48].

For all these reasons, it is necessary to provide programmers of heterogeneous systems with a simpler and intuitive API, simplifying the set of functions and their parameters. It is also necessary to have a runtime that internally manages the possible differences in configuration and functionality of the devices, regarding their OpenCL support. And most importantly, the multi-objective perspective, to achieve all this without penalizing performance.

To overcome these challenges, this chapter proposes EngineCL, a new runtime and API based on OpenCL that notably simplifies the programming of heterogeneous systems. The abstraction level is increased because it frees the programmers from tasks that require a

specific knowledge of the underlying architecture, and that are very error prone. It ensures performance portability thanks to the integration of schedulers that successfully distribute the workload among the devices, adapting both to the heterogeneity of the system and to the behavior of the applications. And finally, the simplified and extensible API has a great impact on their usability, productivity and maintainability.

3.2 Overview of EngineCL

EngineCL is an OpenCL-based runtime system, built with modern standards and designed under the principles of usability and performance. Its design decisions are based on previous work related to heterogeneous systems, OpenCL and other parallel programming models [128, 139, 291]. The designed architecture is modular and flexible enough to extend the internal core and runtime functionalities. Although it has been initially created to facilitate co-execution in OpenCL, its structure allows extending the internal management and computation modules. One of the modules favored from the beginning has been the scheduling system, which allows experimenting with load balancing algorithms thanks to the scheduling pluggable system.

The main ideas of EngineCL are:

- Enable easy integration and porting of applications to be used in heterogeneous systems. The designed API has to be flexible but easy to use by a novice programmer. One of the key points is the maintainability of the applications, and EngineCL must not be a burden on the development and evolution efforts of the software that uses it.

- Facilitate the design, implementation, porting and validation of scheduling algorithms. One of the fundamental aspects of performance portability is the ability to exploit new architectures and applications without the need for the programmer to be involved in decisions or to develop complicated workload management strategies. Therefore, the system must facilitate the exploitation of new load balancing algorithms. At the same time, it must make it possible to study and optimize the behavior of these schedulers, knowing at all times the low-level operations and the implications they have on systems and applications.

- Offer an extensible and modular architecture, thoroughly tested by profiling under different architectures and systems. OpenCL is a complex language that requires significant effort to master correctly, and is often tied to the specific behavior of architectures, device types or even systems and drivers. This experience determines the design decisions, and makes the effort of implementing structures that correctly exploit these optimizations very tedious.

3.3 EngineCL

EngineCL is a runtime with a multi-threaded architecture and multiple modules that endows a programmer with the ability to properly exploit heterogeneous systems. The basics explained in this chapter focus on four sections to understand the most important parts and functionalities provided by the runtime:

- General design principles, to understand its conception and key ideas.
- The architecture, including its tiers, contexts and modules, to expose its composition.
- OpenCL abstractions and how the platform, execution, memory and programming models are encapsulated and exploited.
- The schedulers provided, the pluggable scheduling system and its main features regarding extensibility and optimization.

3.3.1 Principles of design

EngineCL is tightly coupled to OpenCL and how it works. Therefore, it is not intended to replace it, but to act as a wrapper over it. The system modules and their relationships have been defined according to the most efficient and stable patterns. Every major design decision has been benchmarked and profiled to achieve the most optimal solution in every of its parts, but mainly promoting the modules related with the data management, synchronization and API abstraction.

Four main decisions have been applied since the very beginning:

- OpenCL should be isolated to improve the compatibility of the runtime. It allows high adaptation to new technologies while preserving the runtime API and its main schedulers. It is slightly independent to OpenCL, but still promoting it since it is the best technology to support heterogeneous devices.

- It should be managed to be easily extensible while providing the best average performance between all the available devices. It ensures the best performance and efficiency independently of the new devices to be incorporated, solving many issues found when adapting new architectures to runtime systems. The usage of callback mechanisms mixed with events is one of the core aspects to boost the performance of the system, while being able to improve the expressiveness of the schedulers. This strategy helps the device drivers to optimize the enqueued operations, such as recognizing data movements or creating sub-queues, when the driver programmers consider such scenarios.

- Asynchronous inside, synchronous outside. Although this design decision hardly complicates the internal implementation of the runtime, due to its asynchronous nature, it allows the incorporation of unconstrained scheduling mechanisms. OpenCL

was originally created not providing thread-safe API operations, limiting its operations to basic synchronous offloading execution loops. However, such constraints limited the load balancing and algorithmic benefits. EngineCL exposes an API with synchronous operations to facilitate its adoption for common use cases, but it can easily be extended to expose asynchronous behaviors to the programmer.

- In case an increase in maintainability implies a performance penalty, being verified via profiling, an implementation with the minimum performance overhead is chosen. The runtime addresses usability and performance, but the latter is more critical and should be promoted.

These design decisions allow incorporate any type of scheduling algorithm, providing asynchronous operations between the runtime and the devices, enhancing the general efficiency for every program. After all, EngineCL excels the more devices it has to manage, exploiting simultaneously all of them to compute a problem.

The overview of the scheduling design in relation to the runtime technology is presented in Figure 3-1. It represents two of above design principles. On the one hand, OpenCL has been isolated as an execution core associated to each device. On the other hand, the interaction between the scheduling mechanisms and the devices is done through interfaces and custom software protocols, being job notification queues the most common internally. The purpose of these is facilitating the extension of the runtime, especially with regard to load balancers. This example shows two threads creating and managing their devices. Generally, each system thread manages one device, although this is a parameter configurable by the developer and the runtime. Each has a stage to configure everything related with OpenCL

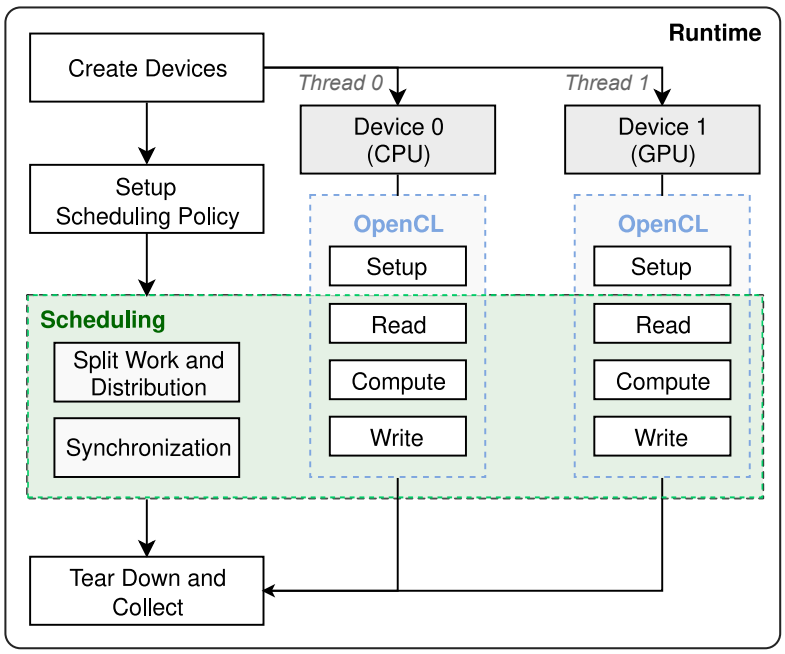


Figure 3-1: Technology encapsulation that isolates its interaction with the scheduling mechanisms.

for each device (*Setup*), containing multiple operations. This is isolated to avoid propagating technology decisions through the software architecture. Once the cooperative work with the scheduler begins, a iterative cycle of work occurs. This cycle continuously requests new data, computes kernels and sends results to the host (*read, compute, write*), ending when the scheduler notifies the devices to finish. The general idea of this design is to decouple the load distribution algorithms from the devices, as well as to isolate the entire runtime from the base technology. However, the architecture design and computing patterns are focused on the needs of OpenCL and also by the provided abstractions. Therefore, although a modular architecture has been provided, many structures, interfaces and classes have been created with OpenCL in mind. This can be seen when reusing concepts from the execution model, such as work-items and work-groups, as well as when propagating data types from its kernel programming language, as will be seen in the following sections.

3.3.2 Architecture

EngineCL has been developed in C++, mostly using C++11 modern features to reduce the overhead and code size introduced by providing a higher abstraction level. It has a multi-threaded architecture that combines the best measured techniques regarding OpenCL management of queues, devices and buffers. Some of the decisions involve atomic queues, parallel operations, custom buffer implementations, reusability of costly OpenCL functions, efficient asynchronous enqueueing of operations based on callbacks and event chaining. These mechanisms are used internally by the runtime and hidden from the programmer to achieve efficient executions and transparent management of devices and data.

It redefines the concept of *program* to facilitate its usage and the understanding of a kernel execution. Because a program is associated with the application domain, it has data inputs and outputs, a kernel and an output pattern. The data are materialized as C++ containers (like *vector*), memory regions (C pointers) and kernel arguments (POD-like types, pointers or custom types). The kernel accepts directly an OpenCL-kernel string, and the output pattern is the relation between the *global work size* and the size of the output buffer written by the kernel. The default value is 1 : 1, because every work-item (thread-kernel instantiation) writes to a single position in the output buffers.

It is designed to support massive data-parallel kernels, but thanks to the program abstraction the runtime will be able to orchestrate multi-kernel executions (task-parallelism), prefetching of data inputs, optimal data transfer distribution, iterative kernels and track kernel dependencies and act accordingly. Therefore, the architecture of the runtime is not constrained to a single program.

The runtime follows Architectural Principles with well-known Design Patterns [380] to strengthen the flexibility in the face of changes. As it is depicted in Figure 3-2, the runtime is layered in three tiers, and its implementation serves the following purposes: Tier-1 and

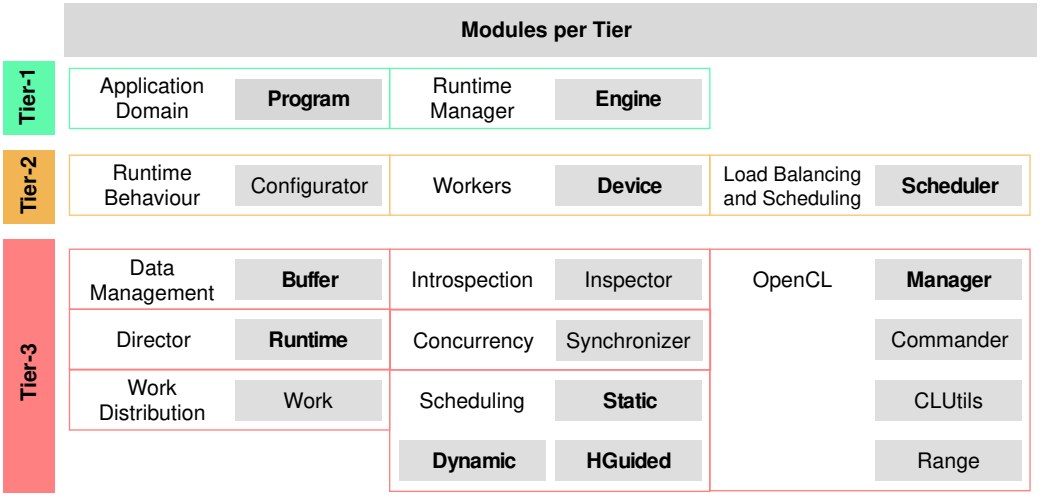


Figure 3-2: EngineCL building blocks: tiers, contexts and modules (main are highlighted).

Tier-2 are accessible by the programmer. The lower the Tier, the more functionalities and advanced features can be manipulated. Most programs can be implemented in EngineCL with just the Tier-1, by using the `EngineCL` and `Program` modules. The Tier-2 is accessed when the programmer selects a specific `Device` and provides a specialized kernel, uses the `Configurator` to obtain statistics and optimize the internal behavior of the runtime, or sets options for the `Scheduler`, among others. Tier-3 contains the hidden inner parts of the runtime that allows a flexible system regarding memory management, pluggable schedulers, work distribution, high concurrency and OpenCL encapsulation. It also depicts *contexts*, which are groupings of modules with a common purpose, usually by making them coupled. For example, `OpenCL` has several modules, but all of them with dependencies to the base technology, so a change in one could affect the rest. Another context of several modules is `Scheduling`, although this one does not have coupling between `Static`, `Dynamic` and `HGuided` modules, but rather they present similar functionality and a common interface. On the other hand, between different contexts there is only the cohesion inherent to any software architecture, thereby making the system easy to extend.

Having seen *tiers* and *contexts*, the lowest representation in this overview are modules, such as `Program`, `Device` or `Buffer`. These group together functionality for a specific purpose, and their components are tightly coupled. For example, the `Inspector` module has structures, functions and constants to collect and present profiling and execution debugging information. If any signature, structure or set of operations change, the rest of the module will probably have to be adapted in order to be used correctly. An important aspect to all modules is that there is generally one class with the same name, although the module can be implemented using several classes. This facilitates its use, acting as the primary class or as an entry point to the module. For example, as will be seen in the API Section 3.4, the developer imports `EngineCL`, `Program`, `Device` or `Scheduler` classes.

The `Program` and `Engine` modules are focused on encapsulating the global behavior of

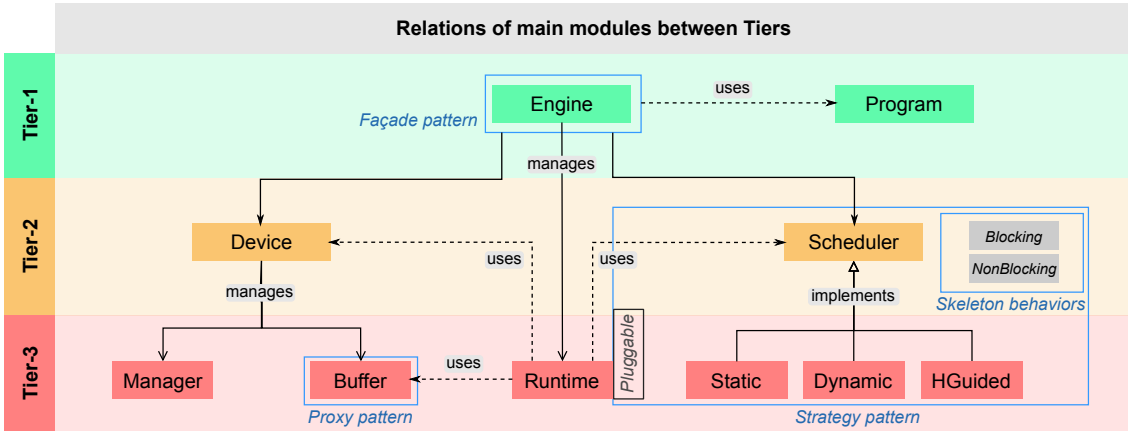


Figure 3-3: Relations and Design Patterns of the main modules to provide encapsulation and extensibility.

both the application domain and the rest of the runtime functionalities, from devices to schedulers. `Device` represents all interaction with the hardware devices, acting as a management module. However, its functionalities are found in lower level classes, mainly under the `OpenCL` context. The `Scheduler` module supports the use of schedulers, helping to hook load balancers and configure them. From Tier-3, the `Buffer`, `Runtime` and `Manager` modules stand out. `Buffer` is responsible for managing memory and providing associations with scheduler data, such as C++ containers. `Runtime` acts as the orchestrator of the system and its multi-threaded architecture. It uses the synchronization primitives of the `Concurrency` context and organizes the devices and their initialization and teardown stages, among others. Finally, `Manager` is in charge of establishing and managing OpenCL-related functions and primitives, instructing the `Commander` module to make certain requests and process data receptions, as well as making it easier for other external modules to request operations of the `OpenCL` context.

Figure 3-3 depicts how the Tier-1 API has been provided mainly as a *Facade Pattern*, facilitating the use and readability of the Tier-2 modules, reducing the signature of the higher-level API with the most common usage patterns. The Buffer is implemented as a *Proxy Pattern* to provide extra management features and a common interface for different type of containers, independently of the nature (C pointers, C++ containers) and its locality (host or device memory). Finally, the *Strategy Pattern* combined with *skeleton* behaviors have been used in the pluggable scheduling system, where each scheduler is encapsulated as a strategy that can be easily interchangeable within the family of algorithms. Because of its common interface, new schedulers can be provided to the runtime. They have an extensible skeleton-based design that implement *behaviors*, as will be detailed in Section 3.3.4.

In summary, EngineCL is designed following an API and feature-driven development to achieve high external usability and internal adaptability to support new runtime features when the performance is not penalized. This is accomplished through a layered architecture and a set of core modules well profiled and encapsulated.

3.3.3 OpenCL Abstractions

All OpenCL primitives and concepts that interfere with kernel offloading have been redefined and encapsulated as part of the EngineCL architecture. The layered design encapsulates new concepts, non-existent in OpenCL, to provide the runtime with sufficient flexibility and usability. Going down through the layers, the concept of *Device* can be found, as an independent execution unit that materializes an *execution core*. This core implements all the functionality associated with the underlying technology. In this way, multiple levels of abstraction can be identified, and the programmer never handles the concepts of program, device, kernel or buffer, as they are necessary in OpenCL.

Nevertheless, the following abstractions or encapsulations produced on OpenCL concepts are identified:

- Platform and Execution models: *Device*, *Platform* and *Context*.
- Execution model: *CommandQueue* and *Event*.
- Execution, Programming and Compilation models: *Program*, *Kernel*, *EnqueueNDRangeKernel* and other execution primitives.
- Memory and Execution models: *Buffer*, *EnqueueWriteBuffer* and other memory management primitives.

`Device`, `Platform` and `Context` are treated independently as *{Device,Platform}* tuples, since the same system can experience different OpenCL driver implementations. Thus, the same device can be used in different ways depending on its identification tuple. Furthermore, this independence facilitates the management of devices, since they are not restricted to OpenCL working contexts determined by the manufacturers and their drivers. For example, two AMD devices, a CPU and a GPU, are considered independent devices and their drivers are unaware that there is another device in the system managed by it. This favors encapsulation and generalization in the use of devices and their interconnections. It is EngineCL who has complete control of drivers, implementations and devices, limiting the scope of knowledge and management of each OpenCL driver. Moreover, such independence enables sophisticated management strategies, such as using a specific driver to compute memory-limited kernels, and another driver for compute-limited kernels, on the same device. It also allows to combine different OpenCL versions, and even different types of implementations and origins, for instance, open-source and proprietary.

`CommandQueue` and `Event` are primitives used as part of kernel launch operations, at a higher level of abstraction, providing utilization mechanisms. The execution core contains this type of primitives, being able to manage the verbosity of OpenCL queues and events. Nevertheless, EngineCL command queues are configurable by the programmer, since different devices may have different behaviors when taking advantage of concurrent execution channels. However, over time, manufacturers have favored low-level queuing strategies, as well as limitations on effective command queues, all implemented within the driver. For

this reason, one command queue per concurrent read, execute or write operation is generally recommended and is the default choice in the runtime. Moreover, it is not uncommon to find modern devices and drivers that no longer suffer penalties when using a single command queue per device, determined by the validation and experimentation performed in many devices while developing the runtime. Even so, being an empirical process depending on the system, drivers, problem and devices used, EngineCL makes it possible to set the number of command queues used, being at least one command queue per device to a pattern of up to two command queues per buffer and one command queue per launched kernel.

`Program`, `Kernel`, `EnqueueNDRangeKernel` and other execution primitives are grouped and encapsulated to make it possible to traverse the domain of the program to be computed. Since EngineCL has to manage the dependencies between needed memory regions and compute kernels, as well as the relationship of work-items and data used, a comprehensive solution that takes into account all these relationships within the runtime is needed. The program now denotes a much higher abstraction, the *program domain*, which can encompass several OpenCL programs and origins, such as pre-computed binaries, specialized code per architecture or generic source code compiled at runtime per device, among others. On the other hand, the EngineCL Kernel has associations to the programs it comes from, arguments with information about the types used, as well as the identification of dependencies in terms of memory transfer. These distinctions and encapsulations in the operations make it possible to distribute the work, instantiating kernels among the devices of the node, taking into account the displacements and quantities of work-items launched according to the needs of the problem. In addition, since EngineCL is prepared to be executed on devices of various kinds, there is an operation translation layer, taking into account both initial versions of OpenCL that did not have support for offsets in launched work-items or where the N-dimensional instantiation of kernels is performed internally to the OpenCL kernel, using OpenCL APIs such as `clEnqueueTask`, as in the case of FPGAs [125, 381].

`Buffer`, `EnqueueWriteBuffer` and other memory management primitives are encapsulated as elements of the memory of the program to be computed, being accessed by the execution mechanisms indicated above. EngineCL implements its own management buffers in order to recognize the types of data used in memory, the size of requested and occupied regions, the locations and offsets assigned to kernel launches, as well as the relationship between computation and writing pattern (*output pattern*) of the programs. Thanks to metaprogramming and a flexible architecture, it is possible to provide programmers with a simple API. It is able to establish direct associations with containers and structures typical of the C++ standard, but also with raw pointers. Internally, the runtime instantiates the OpenCL structures (`Buffer`) necessary to transfer the used data to the device memories, without requiring programmer intervention, except for identifying the mode of the buffers (if they are read, write or read and write). There are several mechanisms to transfer the data, being configurable from the runtime and allowing an extensible behavior. There are two

main parameters to configure a buffer that requires reading, that is, the one that will need the data in the device memory to be computed by the kernel:

- transfer type: complete or by region, specifying the amount of buffer data to send and receive.
- transfer mode: synchronous or asynchronous, determining the concurrent behavior as part of the multi-threaded architecture.

Both can be configured in those buffers that require their reading, that is, those that need the data in the device memory to be computed by any kernel. The synchronous mode facilitates the data transfer prior to the initialization of the computation by the runtime, being a blocking operation, while the asynchronous mode will only transfer the memory when it needs to be computed. These behaviors have implications for program performance, and are largely determined by the desired behavior of a programmer. As an example, sending a full buffer to the device is beneficial if there is sufficient memory and if memory queuing management generates unacceptable delays in the computation, even more so if many work packages are generated. Even so, it is generally advisable to use asynchronous sending mode by regions, optimizing the use of memory and the speed of operations, especially if there are multiple concurrent command queues and the sending regions are not extremely large, facilitating

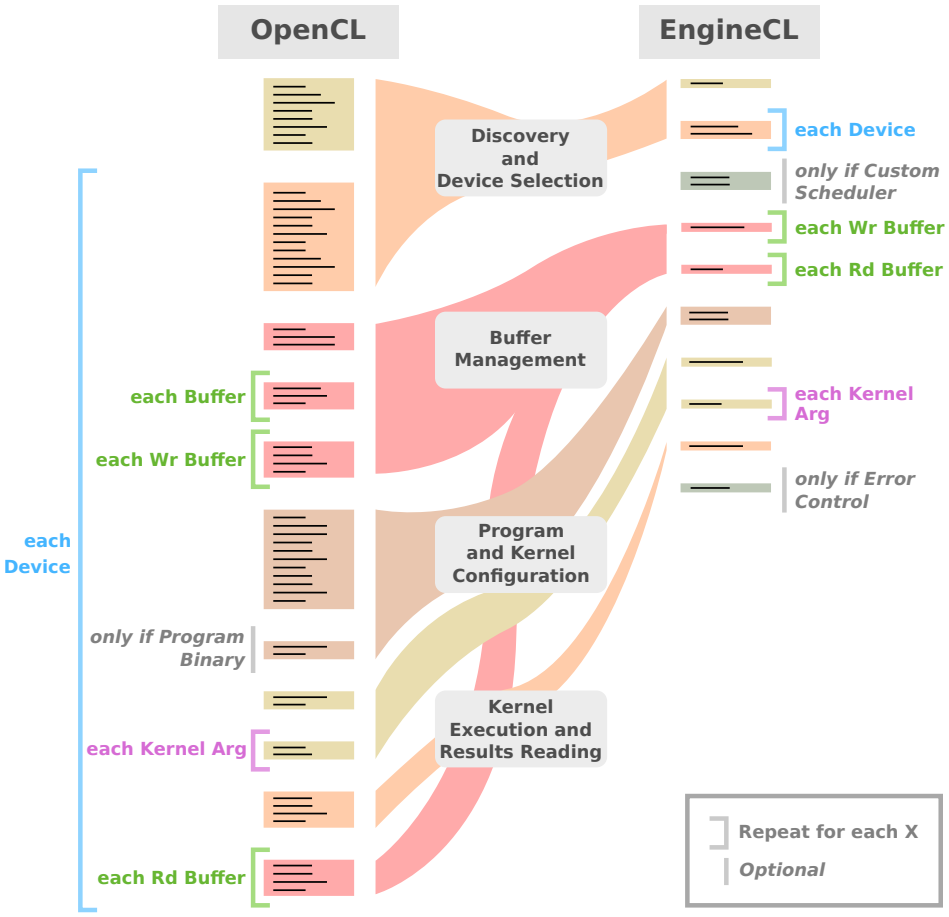


Figure 3-4: Overview of the portability and migration of a generic OpenCL program to EngineCL.

the overlap of computation and communication. As for the output buffers, only the transfer mode parameter is configurable, being synchronous or asynchronous. The former reduces intermediate management overheads to delay sending computed data to the host until all computation is complete, but increases memory usage to keep track of all blocks pending sending. Asynchronous mode, the default behavior, sends regions of already computed data to the host as soon as the computation is finished, but without establishing control mechanisms over the operations to avoid unnecessary waits. Finally, API extensions are available to facilitate the creation of local memory on each device, for those kernels that require it in their kernel arguments.

Thanks to all these abstractions it is possible to design an API that is able to offer a reduction of code like the one shown in Figure 3-4. The height of every rectangle has the same proportions in lines of code as the real program. OpenCL involves more code density and repeats almost all phases per device used. As will be seen in the usability validation, this simplification has very clear programmability advantages.

3.3.4 Schedulers

The EngineCL architecture allows to easily incorporate a set of schedulers, as it is shown in Figure 3-3. The runtime provides by default three well-known schedulers, implementing the load balancing algorithms described in Section 2.2 of Chapter 2, validated in many works [82, 93, 125, 252, 291]. Programmers decide which one to use in each case, depending on the characteristics and knowledge they have of the system architecture and the application.

The `Scheduling` context provides a skeleton from which to model the general behavior of schedulers, building *behaviors* as specialized skeletons. However, new types can be created, allowing new ways of interacting with the runtime and devices. Currently, two types of *behaviors* are provided, `Blocking` and `NonBlocking`. These are the basis on which the implemented algorithms are built.

`Blocking` behavior focuses on achieving maximum compatibility and minimum code execution to resolve load distribution. The main purpose of it this behavior is to be able to use the implemented skeleton approaches in any case, even in old versions or draft implementations of OpenCL drivers. This is achieved by reducing interaction with other modules and restricting requests to the OpenCL execution context. In this way, fewer calls are triggered to the OpenCL runtime API. The variety of OpenCL uses, drivers and versions complicates the use of devices, overriding more sophisticated usage patterns, such as NonBlocking. However, the Blocking behavior is lightweight and offers a clear advantage based on minimal overhead, favoring execution in other types of environments, as will be seen in the Integrations Chapter 4. For instance, there are embedded architectures that rule out the use of non-blocking strategies, restricting asynchronous and extended OpenCL API calls.

This is also found for FPGAs and embedded SoCs, as briefly discussed in related works at the end of such Integrations.

The other behavior is `NonBlocking`, being much more complex but offering much more versatility and potential throughput utilization. These launch mechanisms are governed by the most efficient patterns found with the hardware architectures and drivers studied. This operation structure relies on the runtime multi-threaded architecture to provide the coarse-grained parallelism, while favoring fine-grained parallelism via event chaining between operations (synchronous and asynchronous). Furthermore, every thread manages private scheduling primitives, but interact with other parts of the architecture via synchronization primitives and OpenCL callbacks. This behavior is appropriate when it is necessary to implement more complex workload distribution algorithms or the hardware architecture allows to leverage more efficiency in the overlapping of operations. The disadvantage is the complexity of implementation and its limitation of use in some drivers or architectures, such as those mentioned above.

The load balancing algorithms have been implemented using these two *behaviors*. `Static` is implemented using the `Blocking` behavior, designed to be as simple and efficient as possible. It focuses on the same principles as its skeleton, in order to offer maximum compatibility and portability between platforms. In addition, the operations and OpenCL primitives used are limited, simplifying its design but coupling it with a oracle-like algorithm. Therefore, it is a limited scheduler and is not intended to be extended. On the other hand, both `Dynamic` and `HGuided` are implemented with the same common root, the `NonBlocking` behavior. They use chained events, multiple callbacks (write, read and execute events), a concurrent queue for the queuing of work packages, as well as configurable operation mechanisms to provide flexibility to each scheduler. For instance, they can be capture and dump profiling data at runtime, provide overlapping mechanisms, such as combining execution with data sending, or setup device-host data consistency during the execution, among others. The main difference between both algorithms with respect to the use of the `NonBlocking` *behavior* is the way the package size is calculated, which is more complex and time consuming in `HGuided`. However, as can be seen in the Section 3.6, this complexity is worthwhile.

Related to the schedulers and the software architecture itself, the `Introspection` context enables a series of utilities that can be attached to different parts of the runtime. This provides greater detail on the behavior of the system, and in particular, of the schedulers. This is a very versatile mechanism, since it allows schedulers to mutate their behavior during execution based on states and information provided by `Introspection` functions. In other words, it is not only useful for the development and debugging of new schedulers, but it can also be used to dump execution traces for profiling purposes. For example, the programmer can design a dynamic algorithm based on throughputs provided by the EngineCL context itself, conditioning the distribution of work and the performance of the devices. This has

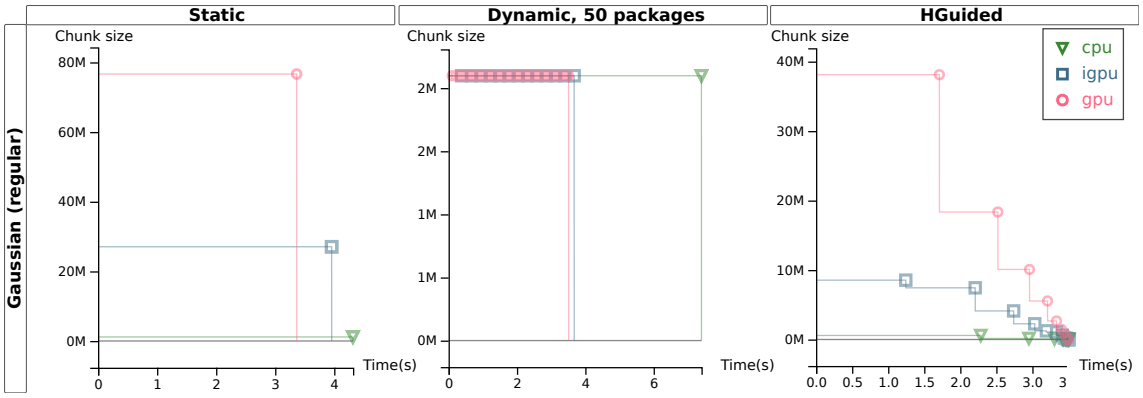


Figure 3-5: Introspection utils showing the package distribution for every load balancing algorithm in a regular program.

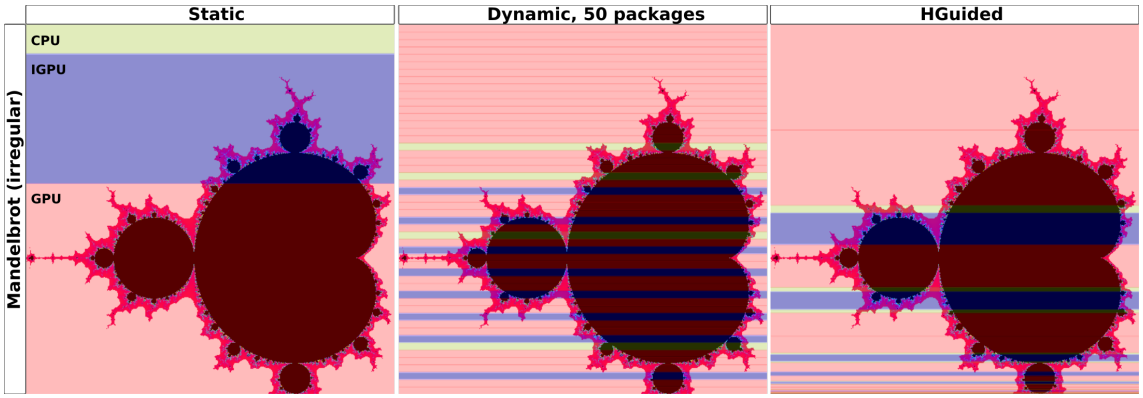


Figure 3-6: Introspection utils representing visually the package distribution in terms of the output computed by Mandelbrot.

clear advantages, since the throughputs are real at the software level, based on the implementation conditions and efficiency of the runtime itself, as well as providing normalized values. Therefore, it avoids performing costly calls to OpenCL drivers from different manufacturers during execution, if supported at all. For all these reasons, the `Introspection` context is very useful to study and optimize the behavior of the algorithms.

In relation to this context, Figures 3-5 and 3-6 show the package distribution produced by the three load balancing algorithms of EngineCL. In both cases the `Introspection` utilities are used, specifically thanks to its `Inspector` module, since both representations are built once the executions are performed, and not during them. The `Inspector` shows more information, such as memory allocations and its data transfers, debugging of the computed values, runtime management overheads, types of data and kernels involved, device properties and runtime configurations. Thanks to this information it is possible to analyze results of the Section 3.6, as for example in the study of the Binomial timings before the computation phase or the effective distribution of work. It is also used during the optimizations of the Integrations Chapter. Figure 3-5 depicts the size of each package and the time during the Gaussian execution, detailed in Section 1.7.2 of Chapter 2. Every marker on the chart

represents when a specific device computes a package. As it is a regular problem, it is clear how each algorithm works. The software architecture and optimizations performed are important for adaptive algorithms, mainly due to the amount of synchronizations. Dynamic delivers packages regularly per device, while HGuided increases the number at the end of the execution, balancing the computation.

On the other hand, Figure 3-6 shows the visual representation constructed based on the Mandelbrot program output. This problem has been computed from top to bottom for each of the images. These show how the workload has been distributed in the three implemented algorithms. For instance, the Static algorithm delivered three regions to the devices, the first for the CPU, then the iGPU and finally the GPU. The colored horizontal regions represent the chunk sizes computed by each device, overlapping the real computation performed, the mandelbrot fractal, in black and red. Since this is an irregular problem, the adaptive dynamic algorithms tend to perform better. This is appreciated in Dynamic and HGuided considering the amount of work processed by the faster devices, such as the GPU, with respect to the rest.

Summing up, the multi-threaded architecture and scheduling abstractions provided by EngineCL are flexible enough to enable blocking and non-blocking mechanisms. Since the runtime architecture provides parallelism as part of its tiers and modules, the co-execution drawbacks of the static approaches, based on chaining consecutive operations in synchronous fashion, are mainly overcame. On the other side, the more sophisticated load balancing algorithms are implemented using the non-blocking mechanisms, and its code and runtime complexity is compensated by the algorithmic improvements.

3.4 API Design

This section describes two use cases of the EngineCL API. As the Section 3.3.2 describes, the runtime has been thought from the beginning to provide a straightforward and flexible API from the point of view of the programmer. Both examples are real use cases, but they have been modified intentionally to show different API calls for demonstration purposes. As an example, the programmer will usually prefer a single call to `work_items` than two consecutive calls to `global_work_items` and `local_work_items`.

The programmer starts by initializing the `EngineCL` and `Program`. Both primitives are common to all programs using EngineCL. The instantiated `engine` is the main element of the system because it manages devices, the application domain and extended features such as schedulers and introspection data, such as statistics and profiling of the execution. It handles well-known OpenCL concepts, such as the number of global and local work items. Additionally, it allows setting the devices to be used by masks (CPUs, GPUs, Accelerators, All devices in the system, any mixed combination, etc) or explicitly setting the platform and device. The latter mode is commonly used not only under development but also in production systems with many driver implementations (Pocl, Beignet, vendor specific, etc.)

and when the programmer needs custom sets of devices or kernel specializations.

The concept of `Program` is decoupled from the runtime to help the programmer to understand it as an independent entity to be modified and to be easily extended to support multi-kernel executions. Therefore, it will allow establishing new parameters such as the concurrency of execution (many kernels at the same time) or linked buffers between programs (shared).

The API can be extended to support new features or to expose Tier-3 functionality to the above tiers, being able to use them directly without the need to access the EngineCL internal code. Finally, the API is evolving as EngineCL integrates or supports new applications, data types, OpenCL features or devices, such as FPGAs, but the current examples show the core of its expressiveness and functionality.

3.4.1 Case 1: Using only one device

Listing 1 shows how EngineCL is used to compute the benchmark Binomial Options with only a single device, the GPU. This example shows the explicit versions of some calls, such as `global` and `local` work items and a mixture of positional and aggregate kernel arguments. This is usually the first step to port OpenCL programs to EngineCL.

It starts reading the kernel, defining variables, containers (C++ *vectors*) and OpenCL values like *local* and *global work size* (`lws`, `gws`), in lines 1 to 6 (*L1-6*). Then, the program is initialized, setting all the variables and filling the containers with the appropriate data. The helper function `binomial_init_setup` is used to simplify the example with everything that is not related to EngineCL (*L8*). The rest of the program is where EngineCL is instantiated, used and released. The instantiated `engine` (*L10*) uses the preferred GPU of the system by using a `DeviceMask` (*L11*). In case there is no graphics card in the system, it has automatic fallback mechanisms to use the CPU, so the application is guaranteed to be computed (`GPU_FALLBACK`). Then, the *gws* and *lws* are given by explicit methods (*L13,14*). The application domain starts by creating the `program` and setting the input and output containers with methods `in` and `out` (*L16-18*). With these statements the runtime manages and synchronizes the input and output data before and after the computation. The `out_pattern` is set because the implementation of the Binomial OpenCL kernel uses a writing pattern of 1 : 255 (*L20*), that is, 255 work-items compute a single out index. Then, the kernel is configured by setting its source code string, name and arguments. Assignments are highly flexible, supporting aggregate and positional forms, and above all, it is possible to transparently use the variables and native containers (*L22-27*). The enumerated `LocalAlloc` is used to determine that the value represents the bytes of local memory that will be reserved, reducing the complexity of the API. Finally, the runtime consumes the program and all the computation is performed (*L29,31*). When the `run` method finishes, the output values are in the containers. Optionally, errors can be checked and processed easily.

```
auto kernel = file_read("binomial.cl");
auto samples = 16777216; auto steps = 254; auto steps1 = steps + 1;
auto samplesBy4 = samples / 4; auto lws = steps1; auto gws = lws * samplesBy4;

vector<cl_float4> in(samplesBy4);
vector<cl_float4> out(samplesBy4);

binomial_init_setup(samplesBy4, in, out, steps);

ecl::EngineCL engine;
engine.use(ecl::DeviceMask::GPU_FALLBACK_CPU);

engine.global_work_items(gws);
engine.local_work_items(lws);

ecl::Program program;
program.in(in);
program.out(out);

program.out_pattern(1, lws);

program.kernel(kernel, "binomial_opts");
program.arg(0, steps); // positional by index
program.arg(in); // aggregate
program.arg(out);
program.arg(steps1 * sizeof(cl_float4), ecl::Arg::LocalAlloc);
program.arg(4, steps * sizeof(cl_float4), ecl::Arg::LocalAlloc);

engine.use(std::move(program));

engine.run();

/*
* if (engine.has_errors()) // [Optional lines]
*   for (auto& err : engine.get_errors())
*     show or process errors
*/
```

Listing 1: EngineCL computing Binomial Options benchmark using the GPU while supporting the CPU as a fallback at runtime.

3.4.2 Case 2: Using several devices

Using a single device provides a much more convenient and maintainable API than OpenCL, but as the number of devices and the complexity of the system configuration increases, EngineCL excels even more. Listing 2 depicts EngineCL computing the NBody benchmark using three devices of the system: CPU, GPU and Xeon Phi. Because of that, the engine is configured to use two of the provided schedulers: Static and HGuided. The scheduler used is selected at runtime by means of a boolean environment flag. Moreover, it uses kernel specialization for different devices, getting the maximum performance per device, but also using the generic kernel for maximum compatibility.

Like in the previous use case, the benchmark is initialized up to line 12. In this example, the programmer has specified two kernel specializations along with the common version: a specific implementation for GPUs and a binary kernel built for the Xeon Phi (*L2-4*). The `Device` class from the Tier-2 allows more features like platform and device selection by index (`platform, device`), device discovery flags, as well as specialization of kernels and building options. Three specific devices are instantiated, two of them with custom kernels (source and binary) by just giving to them the file contents (*L18,19*). After setting the work-items in a single method (*L20*), the runtime is configured to use the `Static` or `HGuided` schedulers with different work distributions for the CPU, Phi and GPU (*L21,22*). Finally, the program is instantiated without any out pattern, because every work-item computes a

```
1  using namespace ecl;
2  auto kernel = file_read("nbody.cl");
3  auto gpu_kernel = file_read("nbody.gpu.cl");
4  auto phi_kernel_bin = file_read_binary("nbody.phi.cl.bin");
5  auto bodies = 512000; auto del_t = 0.005f; auto esp_sqr = 500.0f;
6  auto lws = 64; auto gws = bodies;
7  vector<cl_float4> in_pos(bodies);
8  vector<cl_float4> in_vel(bodies);
9  vector<cl_float4> out_pos(bodies);
10 vector<cl_float4> out_vel(bodies);
11
12 nbody_init_setup(bodies, del_t, esp_sqr, in_pos, in_vel, out_pos, out_vel);
13
14 auto props = { 0.08, 0.3 };
15
16 EngineCL engine;
17 engine.use(Device(0, 0),
18            Device(0, 1, phi_kernel_bin),
19            Device(1, 0, gpu_kernel))
20       .work_items(gws, lws)
21       .scheduler(env_bool_flag("ECL_SCHEDULER_ST") ?
22                  Scheduler::Static(props) : Scheduler::HGuided(props))
23 ;
24
25 Program program;
26 program.in(in_pos)
27        .in(in_vel)
28        .out(out_pos)
29        .out(out_vel)
30        .kernel(kernel, "nbody")
31        .args(in_pos, in_vel, bodies, del_t, esp_sqr, out_pos, out_vel)
32 ;
33
34 engine.program(std::move(program));
35
36 engine.run();
```

Listing 2: EngineCL computing NBody benchmark using a runtime-decided load balancing approach to exploit CPU, GPU and Intel Xeon Phi.

single output value. After data containers are mapped (*L26-29*), the seven kernel arguments are set in a single method, increasing the productivity even further (*L31*).

As it is shown, EngineCL manages both programs with an easy and similar API, but completely changes the way it behaves: Binomial is executed completely in the CPU, while NBody is computed using the CPU, Xeon Phi and GPU with different kernel specializations and workloads. Platform and device discovery, data management, compilation, specialization, synchronization and computation are performed transparently for the programmer in a few lines. As it was depicted in Section 3.3.3 in Figure 3-4 and later exposed in Section 3.6, EngineCL saves hundreds to thousands of lines of code to manage all the operations here exposed to compute each program, but even more when using all the available resources of the heterogeneous system. EngineCL only needs a single line to incorporate a new device to the co-execution. For instance, assuming a simplification of functionality, case 1 and case 2 could involve about 750 and 3200 lines of OpenCL, respectively. All these without providing execution information and profiling, runtime fallbacks and compatibility between versions, as EngineCL provides. Furthermore, OpenCL does not offer an optimized multi-threaded architecture, asynchronous mechanisms between devices and the scheduling system, partial memory transfers or direct consumption of C++ structures.

3.5 Methodology

EngineCL has been validated both in terms of usability and performance. The experiments have been carried out using the machines *Batel* and *Remo*, defined in Section 1.7.1. It is interesting to emphasize that with these two nodes it is possible to test the versatility of EngineCL for 6 different types of devices: Intel CPU, AMD CPU, commodity GPU, HPC GPU, integrated GPU and Intel Xeon Phi.

Five benchmarks have been used to show a variety of scenarios regarding the ease of use, overheads compared with a native version in OpenCL C++ and performance gains when multiple heterogeneous devices are co-executed. Section 1.7.2 contains the properties of the selected benchmarks: Gaussian, Binomial, Mandelbrot, NBody and Ray. Moreover, Ray contains three different raytracing scenes, containing sets of lights and objects with enough complexity to provide a variety of irregular computations. Furthermore, the amount of properties, computing patterns and use cases are relevant because they provide enough diversity to compare EngineCL with OpenCL both in terms of overheads and usability. The worst-case scenario is considered for EngineCL in terms of exploiting its potential, since only one device is used for these comparisons.

The **validation of usability** is performed with eight metrics based on a set of Software Engineering studies [382–385]. These metrics determine the usability of a system and the programmer productivity, because the more complex the API is, the harder it is to use and maintain the program.

The *McCabe's cyclomatic complexity* (*CC*) measures the number of linearly independent paths. It is the only metric that is better the closer it gets to 1, whereas for the rest a greater value supposes a greater complexity. The *number of C++ tokens* (*TOK*) and *lines of code* (*LOC*, via *tokei*) determine the amount of code. The *Operation Argument Complexity* (*OAC*) gives a summation of the complexity of all the parameters types of a method, while *Interface Size* (*IS*) measures the complexity of a method based on a combination of the types and number of parameters. The maintainability worsens the more parameters and more complex data types are manipulated. On the other side, *INST* and *MET* measure the number of *Structs/Classes* instantiated and methods used, respectively. Finally, the *error control sections* (*ERRC*) measures the amount of sections involved with error checking.

To facilitate the understanding of the impact on usability that EngineCL has with respect to the OpenCL base technology, a ratio per benchmark and metric analyzed is given. The only exceptional case is the CC metric, since it has a quantitative nature, with zero being the best possible value. The Equation 3-1 reflects the improvement ratio per metric:

$$Usability_{ratio} = \frac{Usability_{OCL}}{Usability_{ECL}} \tag{3-1}$$

Regarding the **performance evaluation** two types of experiments are presented. The first analyzes the **overheads** and **scalability** of EngineCL with respect to OpenCL C++ when using a single device. Due to the difficulty in measuring stable overheads, these executions are carried out by removing every non-necessary process of the system, establishing user-defined CPU governors with fixed frequencies and increasing the batch of executions to reduce the noise of the system. The minimum problem sizes are selected based on the computing power of every device, being reasonable for each benchmark. Then, the size increases per device and benchmark until the overheads are stabilized or when the execution time is prohibitive, such as CPU reaching more than 100 seconds of execution or GPU being memory-bounded. As a result, they represent the overall trend.

The time overhead, expressed as percentage, is computed as the ratio between the difference of the response times in the execution of the same kernel for both EngineCL (T_{ECL}) and the OpenCL version (T_{OCL}), as shown in Equation 3-2:

$$Overhead = \frac{T_{ECL} - T_{OCL}}{T_{OCL}} \cdot 100 \tag{3-2}$$

To evaluate the **co-execution performance** of *EngineCL* and its load balancing algorithms, the total response time, as well as the response time of each of the devices, are measured, including kernel computing and data transfer. Each program uses a single problem size, given by the completion time of around 10 seconds in the fastest device (GPU) for *Batel*, and 7 seconds in the fastest device (GPU) for *Remo*. Then, as it is exposed in Section 1.7.3,

three metrics are calculated: balancing efficiency, speedup and heterogeneous efficiency. The comparison is performed with respect to a pure OpenCL C++ solution using its host-device programming model. Hence, *EngineCL* is evaluated against the fastest device, which is the GPU for all the cases studied.

Furthermore, the energy efficiency analysis is performed by measuring the total time and the energy consumption in Joules. It is measured using the *sauna* tool, detailed in Section 1.7.4. With these two values the EDP is calculated, obtaining the energy efficiency. Since the EDP results are benchmark and device dependent, a related metric is used to better understand the impact of the runtime and co-execution. Equation 3-3 presents the improvement ratio of co-execution, per scheduler and benchmark, with respect to execution on the most energy efficient device, which is the GPU.

$$EDP_{ratio} = \frac{EDP_{GPU}}{EDP_{co-execution}} \tag{3-3}$$

The scheduling configurations are grouped by algorithm. The first two bars represent the Static algorithm varying the order of delivering the packages to the devices. The one labelled *St* delivers the first chunk to the CPU, the second to the iGPU/PHI (depending on the node) and the last one to the GPU, while in the *St Rev* the order is *GPU-iGPU/PHI-CPU*. The next two show the Dynamic scheduler configured to run with 50 and 150 chunks (*Dyn 50*, *Dyn 150*), being chosen for offering an acceptable compromise between adaptability and synchronization for this load balancer. Finally, the latter presents the HGuided algorithm, labelled *Hg*. The Static algorithm requires an exhaustive search for the best load distribution, performing work prior to the actual execution that is time consuming for the programmer. On the other hand, with adaptive algorithms such as Dynamic or HGuided it is not necessary, being generally effective with different values in the input parameters, both in number of packages and assigned computational powers, respectively.

To guarantee integrity of the results when doing the load balancing experiments, 60 executions are performed per case, divided in 3 sets of no consecutive executions. Every set of executions performs 20 iterations contiguously without a wait period, discarding an initial execution to avoid warm-up penalties in some OpenCL drivers and devices. When measuring the overheads, the experiments are modified to 300 executions, 2 sets and 100 iterations.

3.6 Validation

This section provides a comprehensive validation of EngineCL with respect to OpenCL. The first two validations detail the worst-case scenario for EngineCL, which is when only a single device is used to offload the computation.

- Determine the usability and maintainability of the EngineCL API compared with OpenCL, thanks to the use of a set of metrics that are the state of the art in research in Software Engineering.
- Expose the overhead and scalability of EngineCL compared with OpenCL, indicating the penalization of all the runtime features and management.

Then, the next three validations expose the power of EngineCL when using the full system, considering the co-execution of all devices, finally leveraging the EngineCL runtime, its design decisions and load balancing algorithms.

- Exposing the balancing efficiency of EngineCL when distributing the workload between them.
- The performance and efficiency obtained when the heterogeneous system is fully exploited, compared with the fastest device.
- An energy efficiency evaluation compared with the most energy efficient device of the system.

3.6.1 Usability

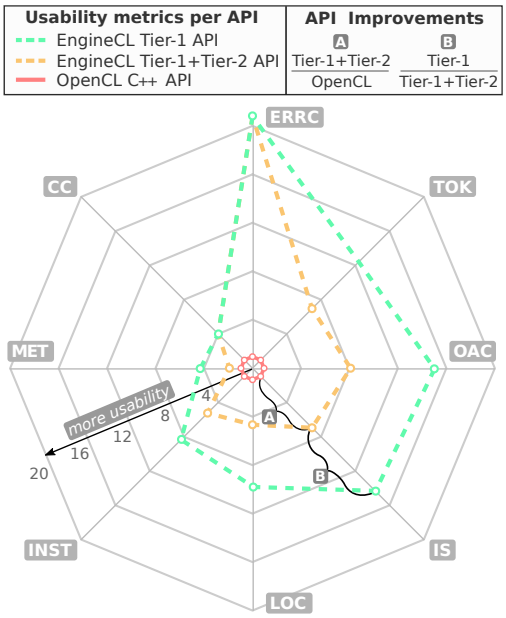
This section shows the experiments performed to evaluate the usability provided by EngineCL. For this purpose, the listed benchmarks are implemented using both OpenCL and EngineCL. Only one device is used to perform the computation, otherwise it would be tedious to implement all the functionality and load balancing in OpenCL. The benchmarks have been selected based on a variety of different properties, computation patterns and use cases in order to show sufficient diversity in the use of the APIs. Table 3-1 presents the values obtained for every benchmark (rows) in every of the eight metrics (columns) when using the EngineCL Tier-1+Tier-2 API, compared with the OpenCL C++ API. Additionally, the average of each ratio per metric, considering the set of programs, is depicted as a chart. This chart shows the base case, OpenCL C++ API, the ratios presented in the table on the left for the EngineCL Tier-1+Tier-2 API, and finally, the impact in ratios if using only the EngineCL Tier-1 API. The further away from the center of the radar chart a marker is located, the better the level of usability for the metric in question, represented by each axis. For instance, the *IS* metric improvement of Tier-1+Tier-2 over OpenCL is the region denoted in the chart with the label `A` (7.3x), while Tier-1 over Tier-1+Tier-2 is outlined by the letter `B` (1.97x).

For every program, the maintainability and testing effort is reduced drastically, as can be seen with *CC*, reaching the ideal cyclomatic complexity, or *ERRC*. The savings in error checking are on average 21 times better by using EngineCL, reducing the visual complexity of alternate paths for error control.

The code density and complexity of the operations are reduced between 7.3 to 8.5 times compared with OpenCL, as it is shown with TOK, OAC and IS. In programs like Ray and Binomial the *OAC* ratio is greater than in *TOK*, because the number of parameters grows in

Table 3-1: Comparison of usability metrics for a set of programs implemented in OpenCL C++ and EngineCL Tier-1+Tier-2 APIs (left) and their average ratios for Tier-1, Tier-1+Tier-2 and OpenCL (right).

Program	Runtime	CC	ERRC	TOK	OAC	IS	LOC	INST	MET
Gaussian	OpenCL	4	22	585	312	433	87	17	28
	EngineCL	1	1	60	33	53	15	3	13
	ratio	4:1	22.0	9.8	9.5	8.2	5.8	5.7	2.2
Ray	OpenCL	4	21	618	307	424	89	17	27
	EngineCL	1	1	191	40	65	24	3	17
	ratio	4:1	21.0	3.2	7.7	6.5	3.7	5.7	1.6
Binomial	OpenCL	4	18	522	255	355	77	16	24
	EngineCL	1	1	81	28	48	18	3	11
	ratio	4:1	18.0	6.4	9.1	7.4	4.3	5.3	2.2
Mandelbrot	OpenCL	4	18	473	222	313	74	15	24
	EngineCL	1	1	65	35	55	15	3	13
	ratio	4:1	18.0	7.3	6.3	5.7	4.9	5.0	1.8
NBody	OpenCL	4	26	658	373	517	96	18	32
	EngineCL	1	1	66	38	60	16	3	15
	ratio	4:1	26.0	10.0	9.8	8.6	6.0	6.0	2.1
	ratio	**4:1**	**21.0**	**7.3**	**8.5**	**7.3**	**4.9**	**5.5**	**2.0**



both implementations, but managing complex types is harder in OpenCL.

As it is exposed with *INST* and *MET*, the number of classes instantiated and methods employed by the programmer are around 5 and 2 times less than in the OpenCL implementation, mainly because it has been deliberately instantiated Tier-2 classes.

The EngineCL code is using the explicit `Device` class, as in Listing 2, but also one argument per line, as in Listing 1, with all the `program.arg` calls. Therefore, the comparison is performed using Tier-1 and Tier-2 of EngineCL, instead of benefiting of the higher-level functions and classes, using only the simplified Tier-1 API. For this reason, to get an overview of the impact, the chart shows the improvement if only the Tier-1 API is used. Therefore, metrics affected by the use of more explicit calls and lines of code benefit when using only the Tier-1. Therefore, TOK, OAC, IS, LOC, INST and MET achieve improvement ratios of 12.5, 15.1, 14.4, 9.8, 8.3 and 4.3, respectively, over the OpenCL C++ API. The only metrics not affected are CC and ERRC, since conditional flow and error control management cannot be further reduced. Thus, for the analyzed metrics and benchmarks, it can be seen how the Tier-1 API can become on average 1.84x better than using Tier-1+Tier-2 in terms of usability. Certainly, this is applicable if there are no special needs only accessible through the use of Tier-2, as detailed in the Architecture Section 3.3.2 and demonstrated with two examples in the API Section 3.4.

Summing up, EngineCL has excellent results in maintainability, implying less development effort. Thanks to its API usability, the programmer is able to focus on the application domain, and its productivity is boosted by hiding complex decisions, operations and checks

related with OpenCL. Moreover, all this considering that EngineCL is employed to exploit only one device. The advantage is that just by adding one more line of code the programmer would have support for an additional device, and so forth. In OpenCL, on the other hand, multiple new code regions would be required to achieve that, as shown in real code proportions in Figure 3-4.

3.6.2 Overhead of EngineCL

This section presents results of experiments performed to evaluate the overhead introduced by EngineCL compared with OpenCL when a single kernel is executed in a single device. On the one hand, a scalability analysis is presented for some specific cases, in order to understand the behavior of the runtime as the size of the problem grows. On the other hand, an overhead analysis is offered, showing the worst combinations of devices and benchmarks, as a detailed study.

Figure 3-7 shows the execution times for both OpenCL and EngineCL for different problem sizes. Since there are 6 architectures and 5 benchmarks, the representative cases regarding the progression of overheads are considered. For this purpose, Binomial is shown as regular benchmark, and Ray as an irregular one. They have been selected since they present the worst overheads, Binomial in Batel and Ray in Remo. Each chart shows on the abscissa axis the different problem sizes, and on the ordinate axis the execution time. They show the behavior when executing each of the problems on a device, in columns, while nodes are on each row. Finally, there are two types of graphs, each showing different problem sizes.

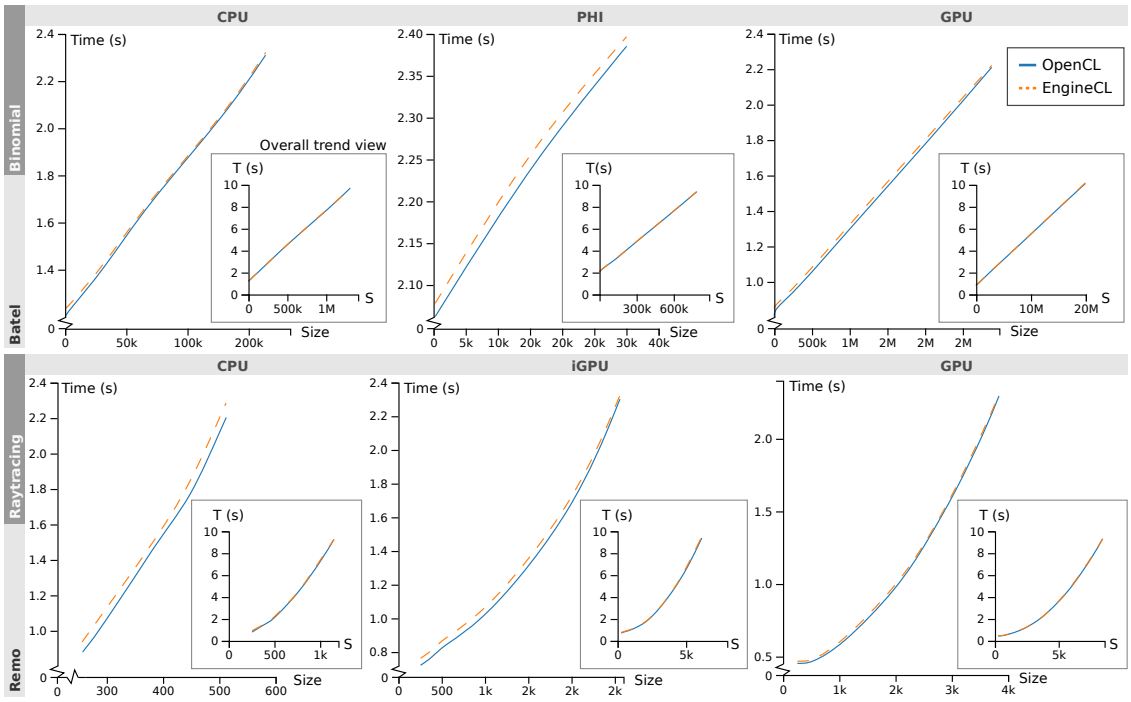


Figure 3-7: Scalability of EngineCL compared with OpenCL for each device in the system.

The maximum overhead value is produced with the CPU in the Remo node, with very small problem sizes. Its overhead value is 2.8%, while the average value obtained for the minimum problem size for all benchmarks is 1.3%. Inside each chart there is another smaller chart showing the overall trend with larger problem sizes.However, the main charts highlight the execution times for the smallest problem sizes, exposing the slightly differences between EngineCL and OpenCL.

For example, in the larger chart view, presenting the smaller sizes, the Phi in Batel has execution times that appear further apart from each other. However, these are absolute time values, therefore they do not represent a more significant amount between EngineCL and OpenCL than what happens on the CPU for Remo.

For this reason it is important to introduce the summary of the worst overheads encountered, showing localized points. Figure 3-8 depicts the maximum overheads per device and benchmark, including the variability (standard deviation). The general trend is that overheads decrease with longer execution times, so the larger the problem size the more amortization of the absolute overhead of the runtime. Each bar represents the overhead when computing problem sizes that takes the execution times indicated. For example, 1% at *+5s* means a specific device presents 1% of overhead in EngineCL compared with OpenCL when computing a particular benchmark that takes around 5 seconds to complete.

Analyzing each device separately, it can be observed that the worst results are obtained in the Remo CPU. This is reasonable since EngineCL also runs on the CPU, that has only 2 cores and 4 threads. Therefore, its multi-threaded architecture interferes with the execution of benchmarks, stealing them computing capacity. This behavior is highly mitigated in the Batel CPU, where the threads used by the runtime does not penalize since there are 24 available threads on the CPU. Regarding the discrete devices, the differences between them are mainly dependent on the driver implementation and how it is affected by the multi-threaded and optimized architecture of EngineCL. The commodity Remo GPU has the highest overhead between the discrete devices, up to 1.59%, but quickly reducing it with larger problem sizes. There are cases, like the Xeon Phi, in which the driver and device produces high variability in the results, probably produced by the amount of host threads that are spawned.

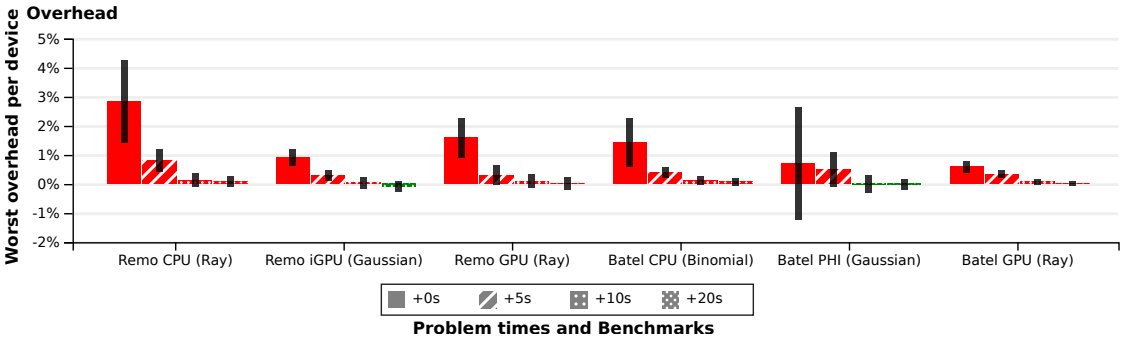


Figure 3-8: Worst overheads found per device and benchmark.

Two conclusions can be drawn from the results as a whole. On the one hand, the overhead introduced by the EngineCL runtime is negligible for all evaluated devices. On the other hand, that EngineCL scales very well with the execution time, so that the overhead decreases significantly as the execution time of the application increases.

3.6.3 Load Balancing

Starting from this Section, EngineCL is evaluated performing co-execution using all the devices of the heterogeneous system. For this purpose, five configurations of the schedulers provided by the runtime are analyzed to execute five benchmarks, including three different irregular scenes for Ray, as presented in Section 3.5.

The first metric analyzed determines whether EngineCL successfully distributes the workload among the devices. To this end, Figure 3-9 presents the *Load Balance*, defined as the ratio of the response times of the first and last devices to conclude its work. The ideal value for this metric is one, meaning that all devices finish simultaneously and the maximum utilization of the machine is attained.

Based on these results, three general conclusions can be outlined. Firstly, EngineCL successfully balances the workload in the two systems analyzed. The mean value of the balance is 0.96, very close to 1.0, with maximum values of 0.98, for example in Gaussian (Batel) and Ray1 (Remo). Secondly, HGuided is the algorithm that offers the best results in all the

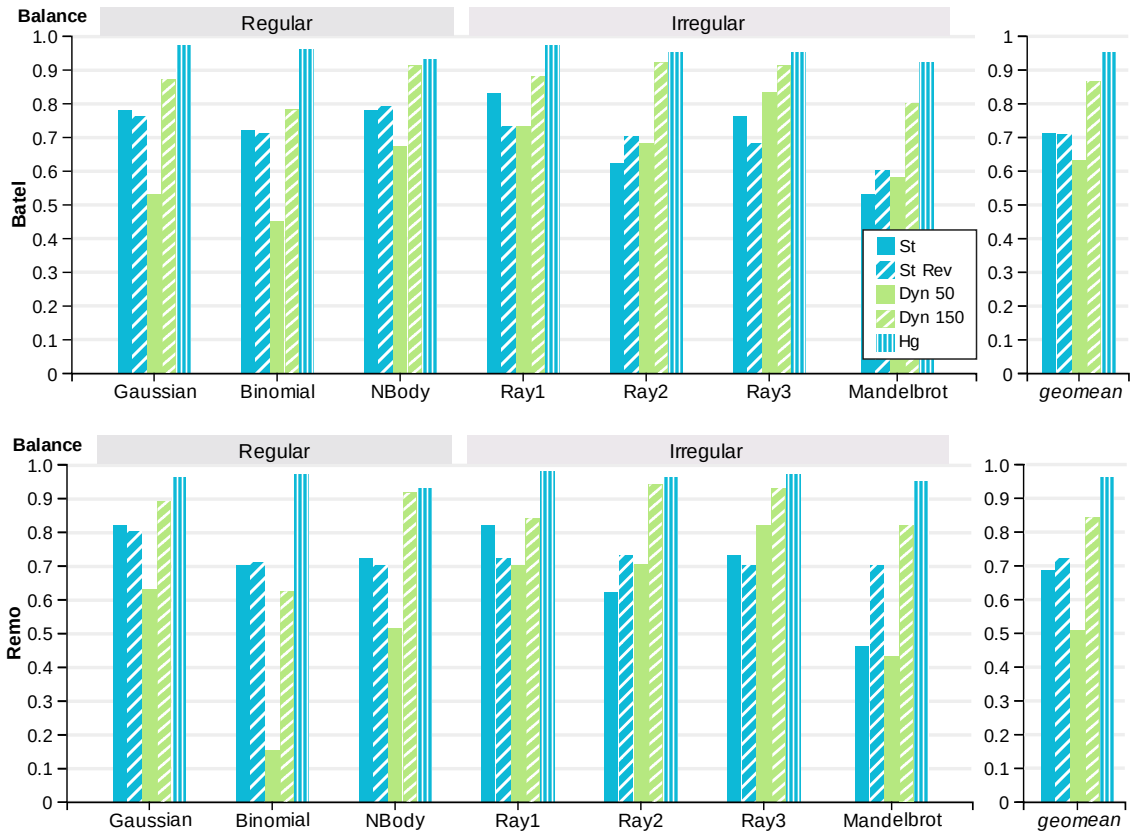


Figure 3-9: Balancing of the system per benchmark and scheduling configuration.

scenarios studied, in both Batel and Remo, and for both regular and irregular applications. Finally, it can also be seen the great relevance of selecting a suitable load balancing algorithm, since otherwise very large imbalances can occur as shown in the cases of static algorithms in Mandelbrot or dynamic approaches with a few packages for Binomial.

Regarding the rest of the algorithms, it can be observed that both static algorithms have a very similar behavior in regular applications, as expected. Nevertheless, they present important differences in the irregular ones, for instance Mandelbrot in Remo. Besides, their behavior depends completely on each case, as can be seen in the cases of Ray1 (static is better) and Ray2, where the reverse gets better results. Finally, the dynamic algorithm always achieves the best-balanced results with the greatest number of packages. However, as it can be seen later, this does not always mean the best performance.

3.6.4 Performance

The performance results achieved in the heterogeneous systems with different load balancing algorithms are shown in Figure 3-10 and 3-11, where the speedups and efficiency are depicted, respectively. The speedups are due to the co-execution when using all the devices of the heterogeneous system, compared with only using the fastest device in each node, that is the GPU on both heterogeneous systems. The efficiency gives an idea of how the system is utilized. A value of 1.0 represents that all the devices have been working all the time. Both metrics are detailed in Section 1.7.3.

The main conclusion that can be drawn is that, for all benchmarks and both nodes, EngineCL achieves better performance than the baseline. This is achieved thanks to co-execution, balancing the workload among the devices, but also due to the low overhead introduced by the runtime and its efficient management. The magnitude of the improvements will depend on the computing power of the devices of the system. On the other hand, efficiency figures show that EngineCL can exploit co-execution very efficiently. This is an excellent result, taking into account the great difference in computing power that exists between the devices of the nodes employed.

However, to achieve these improvements, it is necessary to select an appropriate load balancing algorithm. As can be seen in the figures, HGuided achieves the best results for all the scenarios analyzed, with an average efficiency of 0.89 in Batel and 0.82 in Remo. Therefore, EngineCL can adapt to different kinds of loads and computing nodes, obtaining outstanding performance. It is important to note that efficiencies will tend to improve the larger the problems to be computed, since EngineCL and its schedulers have more time to amortize the impact of load management and distribution. Moreover, as seen in the overhead analysis, the penalty on OpenCL is reduced as the problem size increases. Hence, EngineCL stands out even more in HPC environments such as the Batel heterogeneous system.

Analyzing the speedups and efficiencies in detail, Static delivers good results in regular

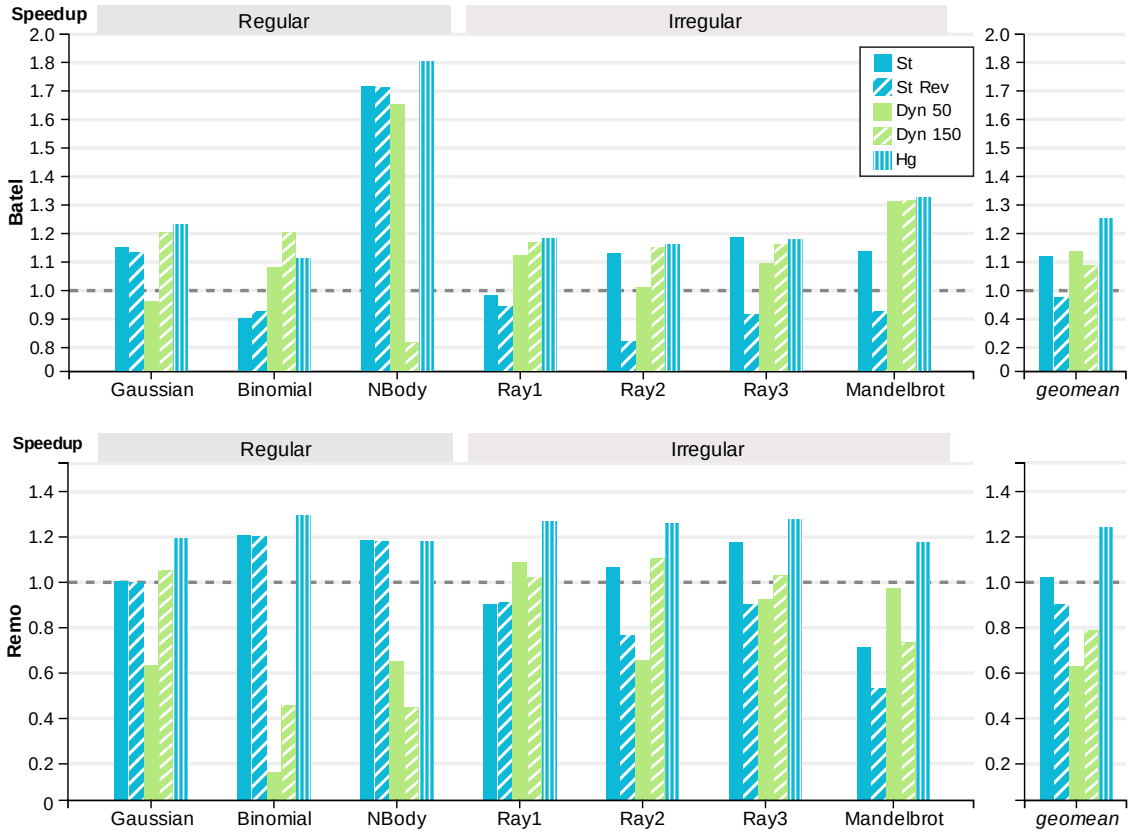


Figure 3-10: Speedups for every scheduler compared with the fastest device (GPU).

applications, with consistent efficiencies between 0.73 (Remo) and 0.87 (Batel), regardless of the order of the devices. Binomial in Batel is an exception that will be explained later, due to the Xeon Phi. However, in irregular applications the results are much more erratic, because it does not adapt to these irregularities, such as Ray1 (0.76) and Ray2 (0.92) in Batel or Ray1 (0.58) and Ray3 (0.75) in Remo. Furthermore, the order in which the devices are considered also has a significant impact on efficiency, as it is shown in Ray2, Ray3 and Mandelbrot. When a slower device processes regions of problems with less computational load, its speed increases compared to other regions, unbalancing the execution.

The Dynamic algorithm has good results in most irregular applications when every device can provide enough computing capacity (Batel), achieving a geometric mean efficiency of 0.81, but suffers in benchmarks like NBody and Gaussian. They are sensitive to the number of chunks and their size, increasing the overhead of communication and usage of slow devices, respectively. Therefore, it is important to accurately determine the number of packages to get the best results in each benchmark. In Remo, the Dynamic algorithm suffers these penalizations due to its CPU with low computing power, which imbalances the co-execution when a wrong package size is giving to the slow device.

An analysis of the work size distribution among the devices is shown in Figure 3-12. It depicts the percentage of work distribution given to each device, taking into account each scheduler and benchmark. Each bar has three regions with the work size given to each

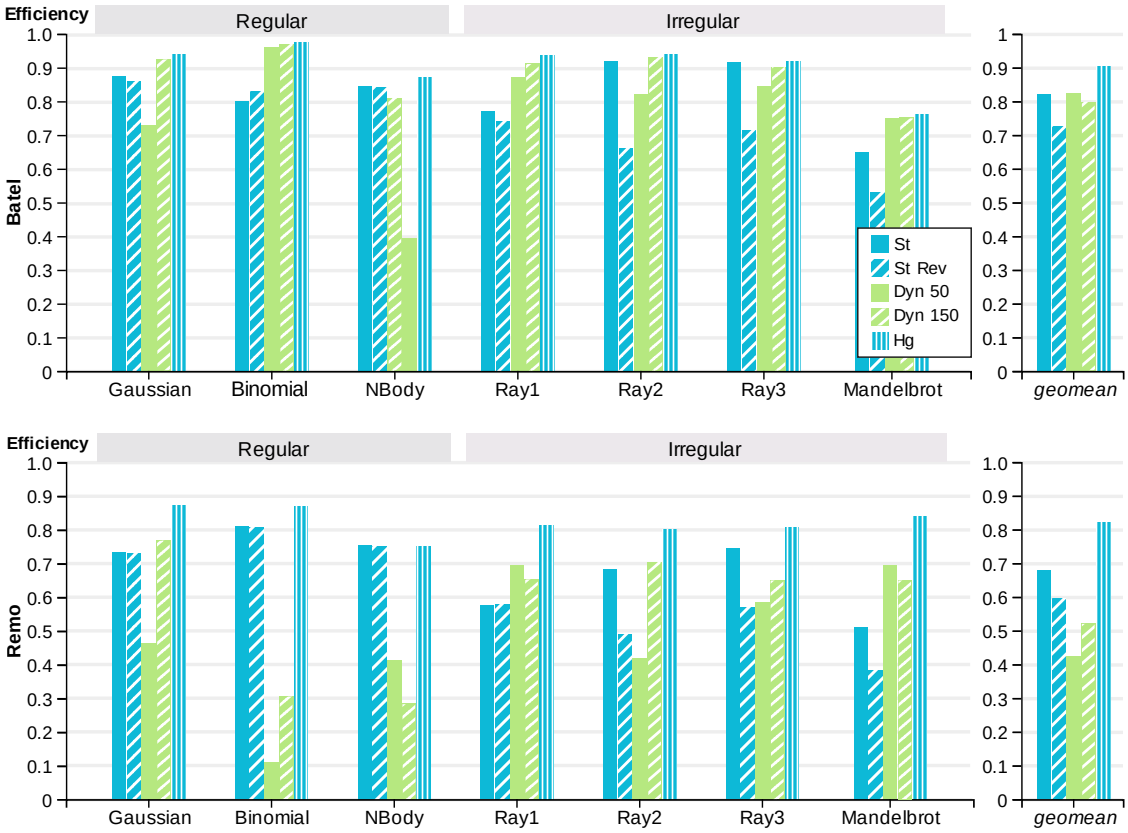


Figure 3-11: Efficiency for every scheduler compared with the fastest device (GPU).

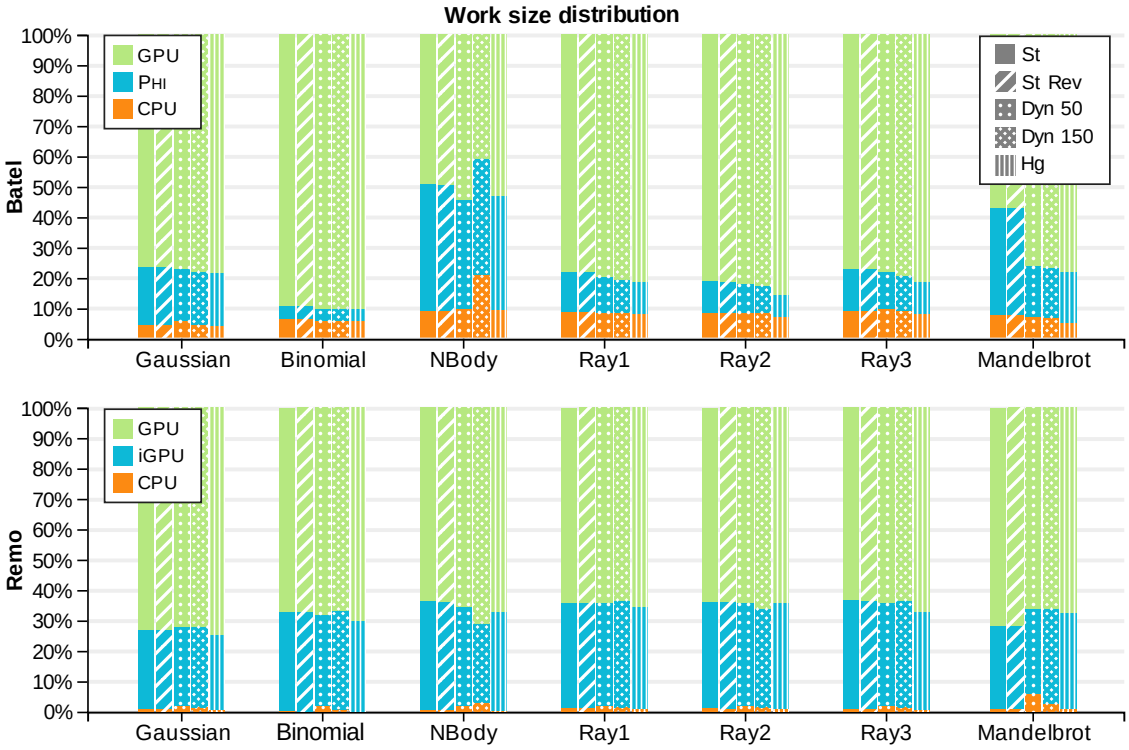


Figure 3-12: Work size distribution per device, benchmark and scheduler.

device. Every scheduling configuration distributes a similar workload for each device, except NBody and Mandelbrot, in Batel. The CPU takes more workload as the number of packages increases in NBody, introducing synchronization overheads that are negligible with fewer packages. Also, Mandelbrot shows how the Phi processed too much amount of work for the part of the image given in the Static, being more complex to calculate than the expected when computing the complete image. Also, Remo work distribution shows how the CPU penalized the whole execution in Dynamic due to large work sizes.

As it was introduced, the GPU in Binomial outperforms the CPU and Xeon Phi, as can be seen in the Static work size distributions. Therefore, a slightly variation in the completion time for any of these devices will imbalance the execution. Another important point is introduced to the analysis, regarding low-level drawbacks from driver implementations. When using the CPU in co-execution, the OpenCL driver of the Xeon Phi requests high CPU usage to initialize, configure and operate with the Phi. Therefore, when the Intel Xeon and the Intel Xeon Phi are being used simultaneously, the CPU is shared by both drivers without any coordination, despite being part of the same OpenCL platform. This introduces time variations during the initialization and new overheads in the final completion times. This behavior is depicted in Figure 3-13, on the top side, showing the average times from initialization for all the executions in Binomial. The abscissa axis shows the base case (single device) and each scheduling configuration, with a bar showing the behavior for each device. The ordinate axis shows the time since EngineCL started. Using only the Phi needs around 1800 ms. to initialize and start computing, while it is up to 2700 ms. when using it as Single, that is, without co-executing. This variation combined with the small amount of work given

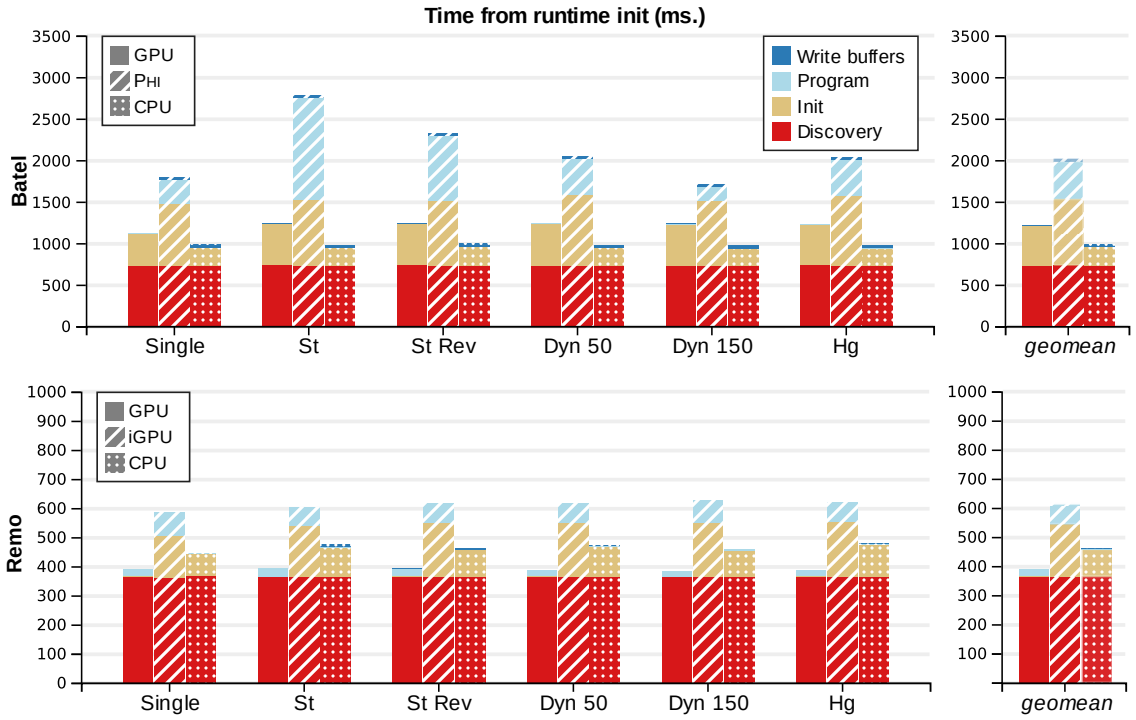


Figure 3-13: Binomial timings before the computation phase.

to the CPU and Phi produces enough imbalance to not achieve the goal. On the other side, the Dynamic approach it is much worth for two reasons. First, it releases the CPU intermittently between the delivered chunks, so that both CPU and Phi drivers are interleaved. Second, thanks to its adaptability, it solves initialization variations by giving more chunks to the GPU, as shown. Drivers and their management are relevant for OpenCL computation and to achieve efficient co-execution. This can be seen in the bottom of the figure, where Remo drivers and devices are completely stable compared to the Xeon Phi, used in Batel.

3.6.5 Energy

This section presents the energy efficiency analysis performed on the Batel heterogeneous system. Remo AMD APU does not allow measuring energy consumption as it does not have accessible hardware counters or alternative mechanisms to measure it, so the study of energy efficiency would be very limited with only the discrete GPU.

To measure the energy efficiency, the EDP is used, as discussed in Section 3.5. For ease of understanding, Figure 3-14 shows the improvement ratio obtained by using co-execution with EngineCL with respect to executing the benchmark on the GPU, that is, the most energy efficient device. Thus, values above 1.0 indicate that the co-execution is more energy efficient than the GPU.

The main conclusion that can be drawn is that for all the benchmarks studied, EngineCL obtains improvements in energy efficiency compared with the GPU. This is satisfied as long as an appropriate load balancing algorithm is chosen, with the HGuided scheduler being the best in all cases, except in Ray3. This is an exceptional situation, since the *Static* algorithm, as contrasted with its *Static Rev* version, gives very different results depending on the computational regions assigned to it. Thus, it can be severely penalized by irregular programs, and the fact that it has managed to be more energy efficient than HGuided is due to the specific layout of the raytracing scene. The slightest change in the scene or in the order of the devices in the system could cause Static to achieve a very high EDP value, and therefore very low energy efficiency.

The HGuided algorithm is able to obtain a geometric mean of 1.37 for all the benchmarks studied, around 1.36 for the regular ones, and 1.39 for the irregular ones. The rest of the algorithms have an average improvement of around 1.1, except in the case of *Static Rev*, which suffers a major penalty due to the irregularity of the Ray and Mandelbrot problems. This is a curious phenomenon, since generally the Xeon Phi has higher power consumption when launching the entire problem. However, when acting in co-execution and especially when exploiting sophisticated dynamic algorithms, such as HGuided, it tends to be more energy efficient. However, if there is a larger number of packages and management penalties, as with *Dyn 50* and *Dyn 150*, the energy consumption increases and the resulting EDP is not as advantageous. For instance, NBody only achieved an improvement of 1.49 over the GPU,

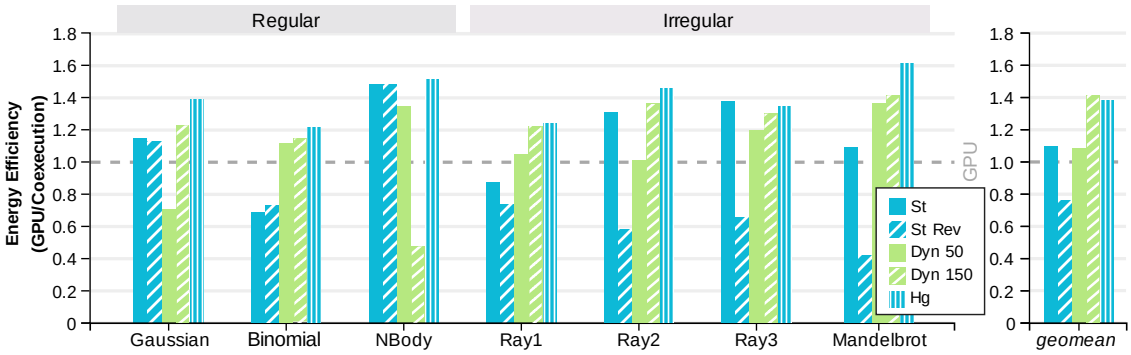


Figure 3-14: Energy Efficiency compared with GPU in Batel (more is better).

when its speedup was as high as 1.80, so it was clearly penalized by its power consumption. NBody is a benchmark that requires a lot of synchronizations, and although the Xeon Phi has taken almost 38% of the work, as represented in Figure 3-12, it has consumed too much power. It is a less efficient device and although it contributes to the time reduction, it significantly increases the power consumption.

The most significant improvement has been achieved with HGuided for Mandelbrot, becoming 1.59 times more efficient than the GPU. However, it is a benchmark that has obtained the lowest heterogeneous efficiency, but in terms of EDP it is the one that has achieved a balanced performance. It has reduced the total time without consuming much power. Considering the Binomial case demonstrated in the previous section, the conclusions are reaffirmed here, indicating how the energy efficiency is degraded due to the CPU saturation by the OpenCL driver, wasting both energy and time.

In summary, it can be seen how performance and energy efficiency are related. However, there are situations where peculiar behaviors are obtained. Despite the GPU being a more energy efficient device than the Xeon Phi, when co-execution and different load balancing algorithms come into play, there are situations where the whole heterogeneous system is better leveraged. For this reason, it is important to have a compromise between computational power, management overheads, power consumption and efficient load balancing algorithms, as all of these influence whether all devices co-executing can be more energy efficient, despite having units such as the Xeon Phi.

In summary, EngineCL can execute a single massive data-parallel kernel simultaneously on all devices in a heterogeneous system with negligible overhead. In addition, thanks to the load balancing algorithms, it yields excellent efficiencies, both in terms of performance and energy.

3.7 Conclusions

This Chapter introduces EngineCL, a powerful OpenCL-based tool that greatly simplifies the programming of applications for heterogeneous systems. This runtime frees the pro-

grammer from tasks that require a specific knowledge of the underlying architecture, and that are very error prone, with a great impact on their productivity. Moreover, the runtime is designed and profiled to provide internal flexibility to support new features, high performance to avoid any overheads compared with OpenCL and a pluggable scheduling system to efficiently use all the available resources with custom load balancers. Hence, for these reasons and with appropriate schedulers, it favors performance portability. The API provided to the programmer is very simple, thus improving the usability of heterogeneous systems.

These statements are corroborated by the exhaustive validation that is presented, both in usability and efficiency. Regarding usability, a large variety of well-known and widely used Software Engineering metrics has been analyzed, achieving excellent results in all of them. Considering 5 implementations of programs both in EngineCL and OpenCL when using a single device, the worst-case scenario for EngineCL, the maintainability improvement ratios range from 2 to 21 times more usable than using OpenCL technology and its runtime API.

The efficiency has been validated in two different nodes, one HPC and one commodity system, with six different architectures to show the compatibility and capabilities of EngineCL. Three important conclusions can be drawn. First, the careful design and implementation of EngineCL allows negligible overheads with respect to the native OpenCL version, always below 2.8% in all the cases studied. Moreover, considering the worst values per device, the average overhead is 1.3%. Furthermore, EngineCL scales very well with the size of the problem, so overheads vanish for large problem sizes. Second, it is critical to select the right scheduler, especially for irregular applications, where it needs to be dynamic and adaptive. Among the schedulers implemented and integrated in EngineCL, HGuided provides the best results, being able to balance both regular and irregular applications, with an average efficiency of 0.89 and 0.82 for the HPC and desktop system, respectively. Furthermore, the multi-threaded architecture and co-execution mechanisms allow leveraging the heterogeneous system in terms of energy efficiency. On average, for the cases studied, EngineCL achieves an improvement of 1.37 over the most energy efficient accelerator, the GPU, if the appropriate load balancing algorithm is selected. Finally, thanks to all the above, EngineCL is able to provide the programmer with effortless co-execution, thus ensuring performance portability between very different heterogeneous systems.

EngineCL Integrations

CHAPTER 4

EngineCL Integrations

The existing variety of applications, execution environments, architectures and technologies, provides enough situations where a single technology, such as OpenCL or OpenMP, is not able to cope. It is not just a matter of squeezing out maximum performance, but a multitude of cases where it is impossible to use it. It may be that the computing device does not have software support for a specific technology, such as incomplete, non-existent, incompatible or non-thread safe drivers; that it is not accessible from a custom location, operating system or execution node, as occurs when running desktop graphics applications or multi-node web servers; or simply because the problem execution environment was never intended to exploit heterogeneous systems, limiting computation to the CPU, as occurs in many sandbox environments, virtual machines, web browsers or runtimes of programming languages.

The approach lies in extending and exploiting EngineCL, presented in the Chapter 3, supporting more complex programming models and situations for which they were not initially conceived. However, thanks to their architecture, design principles and efficiency, they offer a working environment suitable to be extended, increasing their functionality. There are cases in which it is impossible to exploit problems or devices using the OpenCL technology, since the limitation is at a lower level, as in the case of drivers and firmwares. In these cases, it is important to obtain more sophisticated solutions that allow mixing programming models in order to meet performance and flexibility needs. This is a multi-objective purpose, since these solutions generally increase the complexity of the system and its programmability, requiring to enhance the architecture, achieving low coupling among the runtime modules, all without affecting the original effective performance. The most important aspect of the whole extension process is to be able to preserve all the features for which they were designed, that is, to add new functionality without penalizing the original capabilities already validated.

EngineCL has been applied to different scenarios, among which two integrations of the runtime are detailed. Each integration refers to the exploitation of EngineCL to solve a problem in a specific environment, implying that the application and adaptation involves an extension of the runtime system and its design decisions. The first one is focused on optimizations for time-constrained applications, and the second one is applied to exploit a molecular dynamics simulator, supporting hybrid programming models.

Chapter contents

4.1 Integration I: time-constrained scenarios

This integration addresses time-constrained scenarios where applications run on commodity nodes and service servers for a very short period of time. This is a problem for host-device programming models, where drivers and loader systems are a bottleneck for such constraints. The OpenCL execution infrastructure, known for its verbosity and complexity, requires many steps to launch kernels to accelerators, penalizing its utilization. EngineCL, despite its already proven efficiency in HPC, suffers from the impediments of the underlying technology. However, it is an appropriate framework to optimize both the runtime architecture and the algorithmic aspects of co-execution for such scenarios.

4.1.1 Motivation

Modest heterogeneous systems are often used to exploit problems that transcend high-performance computing and fall under other types of challenges, such as multimedia workloads, video encoding, image filtering and inference in machine learning. Low cost and ease of acquisition are some of their strengths, providing versatility to these ubiquitous systems. It is increasingly common to find desktop computers and embedded devices composed of integrated heterogeneous systems, CPU cores and GPU compute units in a single chip. Along with them, it is common to attach discrete GPUs. The availability of these systems in both commodity infrastructures and medium-sized service servers facilitate a new field of work, but involves great challenges [80].

Challenge 1: Technology overheads. These systems and applications require minimizing the overhead introduced by EngineCL and its underlying technology, OpenCL. The offloading kernels and everything related to their execution should be performed in hundreds of milliseconds, sometimes a few seconds, where every management operation or the minimum overhead completely penalizes the offloading to devices.

This type of computations are included as part of a *Processing System* (*PS*), since they are not generally independent applications, such as CLI or GUI programs. However, throughout the Integration the PS will be referred to as *program* for ease of understanding, since the *main program* would refer to the service process, server or desktop application, excluded from this Integration work because it is unique to the host and has no association with acceleration technologies like OpenCL. For instance, a Binomial Options program mean that a PS implements such kernel computation as one of its main externally callable functions, so the main program can launch it.

For this reason, the margin for improvement is very small, since the problem is isolated to the download technology. Even so, it is important to lighten the computation process with accelerators to extend the chances of execution. It is not possible to optimize commu-

nications and transfers between the PS and the main program, or anything related with the host. However, both OpenCL and EngineCL have a number of execution stages, runtime primitives and mechanisms for transferring memory that must be addressed to lighten the overhead produced by the programming model.

Challenge 2: Nature of the executions. These type of computations are typically carried out in two operating modes. First, the most widespread usage and termed *Region of Interest execution* (*ROI*) to simplify, integrated in a main program, directly consuming the data and making only the transfer and computation through the use of other technologies. This function is executed in parallel while the main program continues operating, such as the server managing requests or the GUI rendering charts. Second, by launching the computation function as a process independently to the main program, serializing and exchanging the data and results. This mode is simpler for the programmer, does not involve extra complexity due to decoupling, and it is often referred to as *binary* or *full-process execution*.

This work addresses both modes, although the most important one is the ROI operating mode, since it is the normal operation mode in many desktop applications and service servers, like Nginx or Apache, often delegated through C++ plugins or scripting dynamic languages, such as PHP, CGI, uWSGI or Lua. In this case, the server application is previously up, and the PS has three main functions to be called by the main program:

- init y setup: the main program is initialized together with the connected plugins and processing systems. At this stage the PS is also called to prepare for subsequent computation calls.
- compute: the PS is eventually called, performing memory transfers and dispatching computational functions. These have to be performed as quickly as possible, as they are usually part of requests to the main program that must be resolved with low latency.
- tear down: the main program restarts, suspends or shuts down, and therefore notifies the PS to act accordingly, freeing memory, preparing execution snapshots or storing profiling data, among others.

For this reason, it is necessary to take into account the impact on both modes of operation and study their behavior in the face of optimizations, as it will determine the types of PS that can be implemented.

Challenge 3: Algorithmic tuning. Since the ultimate goal is to perform efficient co-execution in systems where it is not common to take advantage of available accelerators, it is not only necessary to focus on optimizations of the software architecture and the use of its technology. HGuided, the most effective algorithm seen so far for adaptive co-execution, can be tuned for the environments where it is going to be used. There are certain parameters that can be adjusted, and performing an exploratory search and behavioral analysis can be

decisive in squeezing even more efficiency out of the system.

Therefore, it is important to take into account the context in which the applications are executed, knowing that this is the worst possible scenario to do co-execution. This has also been contrasted when integrating other accelerators, such as FPGAs, as it is briefly described in the Discussion, at the end of this Chapter. In short, it is necessary to exploit techniques and optimizations to allow an efficient execution that takes advantage of all the available resources. The original EngineCL proposal never focused on these types of applications, so co-execution suffers in these very limited scenarios. These types of optimizations do not have much impact on HPC applications where execution times are much longer, hence the reason why they have not been addressed so far. It has been the integration in this type of environments and applications that has revealed new requirements.

4.1.2 Optimizations

To overcome the above problems, focused on desktop systems and time-constrained scenarios, the runtime and the execution procedures have been enhanced. Two groups of optimizations have been researched: runtime-centered to allow using load balancing under more problem sizes, competing against the fastest device in the system, and algorithmically, improving the best algorithm used so far by tuning its parameters to boost the average efficiency in a wide range of program types.

Regarding the runtime and its software architecture, improvements are grouped in terms of execution-platform models and memory model. These optimizations have been incorporated to reduce overheads produced both in the initialization and closing stages of the program, mainly due to the use of OpenCL drivers in the analyzed infrastructures, as well as the management of OpenCL memory regions and primitives. They are tagged as *initialization* and *buffer optimizations*.

4.1.2.1 Execution & Platform models

This optimization focuses on the **initialization stages**, taking advantage of the discovery, listing and initialization of platforms and devices by the same thread (Runtime), as it is depicted in Figure 4-1 for the initialization phase. These stages involve both the execution model, mainly its first steps that need to be addressed in every execution, and the platform model, since it requires interaction with devices, drivers and their configuration. The figure shows vertically three layers of abstraction of the software architecture corresponding to the `Runtime` module, from an execution perspective, as time progresses horizontally to the right. Considering the initialization and configuration stages, the Runtime module encapsulates the `Runtime Setup`, `Devices Discovery`, `Devices Setup` and `Scheduler` blocks. The latter is also an independent module with the same name, but its functionality is simplified here as it is not important for this optimization. `Devices Discovery`

and `Devices Setup` are now practically parallel, hence they are on top of each other, considering the same layer (Module Blocks). Descending to the lowest layer, where the *Execution stages* are indicated, each of the operations performed by each block of the upper layer are represented. For example, `S. Ops` is a work region that occurs at the beginning of the `Devices Setup` block, while `D. Ops` occurs at the end of the `Devices Discovery` block. The discovery and acquisition of `Device 1` occurs when contacting the OpenCL `Platform 1`. Considering `Device 2` and `Device 3`, both depend on `Platform 2`, but the driver suffers a small delay that prevents a perfect parallelization in the discovery of `Device 3`. On the other hand, considering the timeline, it is observed how the initialization of the devices (e.g. `Setup Device 1`) depends on finishing its discovery and releasing the platform. This is also the case when configuring all devices once they are initialized (`Config D1`, `Config D2`, `Cfg D3`), needing all of them simultaneously to be ready to continue. This occurs mainly due to blocking requirements to establish synchronization primitives, initialize semaphore barriers, copy EngineCL Program domain structures, reuse OpenCL primitives or query device flags, among others.

To understand the change produced it is necessary to observe the rectangle of dashed lines around this area (*New parallel region*), and how the same discovery and configuration region is projected downwards using the original EngineCL perspective and behavior (*Original region*). The lower part shows the original steps and how the restrictions limited parallelism. There was only effective concurrency and overlapping operations for a short period of time when some devices were discovered and others initialized, as long as they belonged to different OpenCL platforms.

In parallel, both the thread in charge of load balancing (`Scheduler`) and the threads associated with devices (`Device`) take advantage of this time interval to start configuring and preparing their resources as part of the execution environment. These threads will wait only if they have finished their tasks independent of the OpenCL primitives, instantiated by the `Runtime`. It takes advantage of the same discovery and initialization structures to configure the devices before delegating them to the `Scheduler` and `Device` threads, which will be able to continue with the following stages. These optimizations reduce the execution time affecting the beginning and end of the program, due to the increase of the parallel fraction of the program as well as the reuse of the structures in memory, liberating the redundant OpenCL primitives.

Platforms and devices are now initialized and configured in parallel. Only some operations are necessarily sequential, although such regions have been reduced to a minimum. Some sequential examples are the mappings of OpenCL primitives, such as contexts, platforms and configuration flags, needed to be able to produce independent operations concurrently. This also happens in EngineCL modules, such as `Manager`, when setting and linking structures associated with the `Device`, encapsulating OpenCL device identifiers that should be unique to track all their resources. These mechanisms are vital to communicate with the

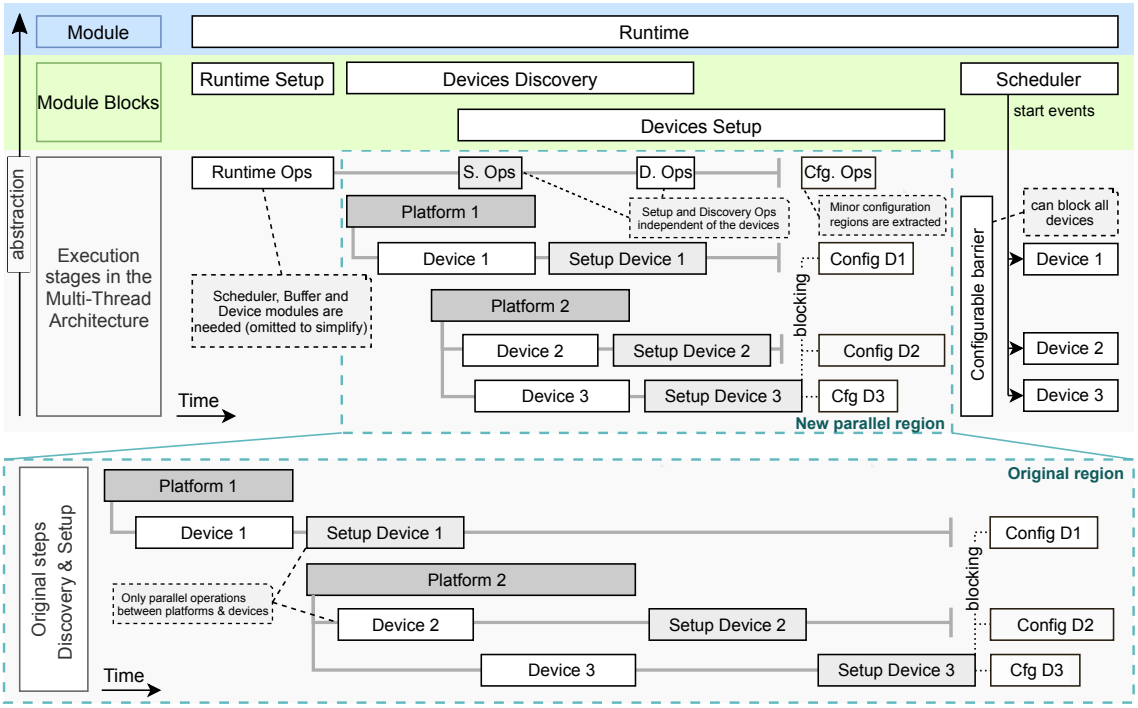


Figure 4-1: Diagram of the *Runtime* module showing the optimization in the device discovery and configuration blocks thanks to the parallelization of the initial execution stages, using two platforms and three devices as an example.

`Runtime` module. For example, if EngineCL is configured to operate with the GPUs and CPU of the node, and the programmer has a specialized kernel for AMD accelerators, the `Runtime` must necessarily consult and keep track of the discovered devices and configure them appropriately at runtime. This type of operations continue to restrict the runtime improvement, as exemplified in the figure with the *blocking* barrier (dotted lines) and the next operations (final setups). However, the gain is significant since there are multiple regions that benefit from overlapping stages. Even though the initialization process has increased and now it is more complex, the programmer is still unaware of the changes, as the external API has remained stable.

4.1.2.2 Memory model

Regarding the memory model, the software architecture has been modified to improve the usage of OpenCL memory containers, both when instantiating and sending data through input and output **buffers** (`Buffer`). The variety of architectures as well as the importance of OpenCL sharing memory strategies save costs when doing transfers and unnecessary complete bulk copies of memory regions. This occurs usually between main memory and device memory, but also between reserved parts of the same main memory (CPU - integrated GPU). Some of the most important features are the incorporation of fine-grained and coarse-grained Shared Virutal Memory (SVM) Buffers and the use of `clSVMAlloc` for explicit memory reservation without the need to create OpenCL Buffers. Although SVM

capabilities (OpenCL 2.0) allow further shared memory features, many vendors still apply the OpenCL 1.2 version, leaving such improvements for future implementations. In fact, OpenCL 3.0, already present in some devices of the *Desktop* and *DevCloud* machines presented in Section 1.7.1, does not mean that they have implemented full SVM support. This is relevant because from this version onwards the features are modular and optional. However, *Remo* node used in this Integration has an integrated GPU with partial support for OpenCL 2.0 and its SVM features, so these optimizations serve to achieve tighter synchronizations when accessing SVM memory. Therefore, all these variations in functionality and complexity due to the diversity in the implementation of capabilities and extensions of the specification highlight the importance of offering the maximum compatibility and versatility possible. In this case, offering the possibility of exploiting those memory models and optimizations present in the hardware and drivers that may appear.

For this reason, changes are made to the EngineCL memory model. Figure 4-2 shows the software design proposed for OpenCL memory allocations. The *Strategy pattern* is used to implement the different behaviors when interacting with OpenCL memory [380]. On the one hand, `GenericBufferStrategy`, is the original EngineCL mechanism, encapsulated as the common way of constructing and manipulating memory in case a more specialized one is not available. The advantage is that it is able to operate with any type of device, offering maximum portability, but at the cost of being less usable and without potential optimizations. The `PinnedMemoryStrategy` class implements memory mapping, relieving the programmer from having to program such verbose code, but being able to leverage Shared Physical Memory optimizations (SPM via IOMMU) with integrated GPUs (OpenCL 1.2). Additionally, this strategy enables also DMA transfers for OpenCL buffers when using discrete GPUs. Finally, with `SVMRegionStrategy` it is possible to use explicit SVM memory region allocations and OpenCL API functions, as expressed above. The drawback is that it requires OpenCL 2.0, only partially supported by the integrated GPU of the AMD APU (*Remo* system).

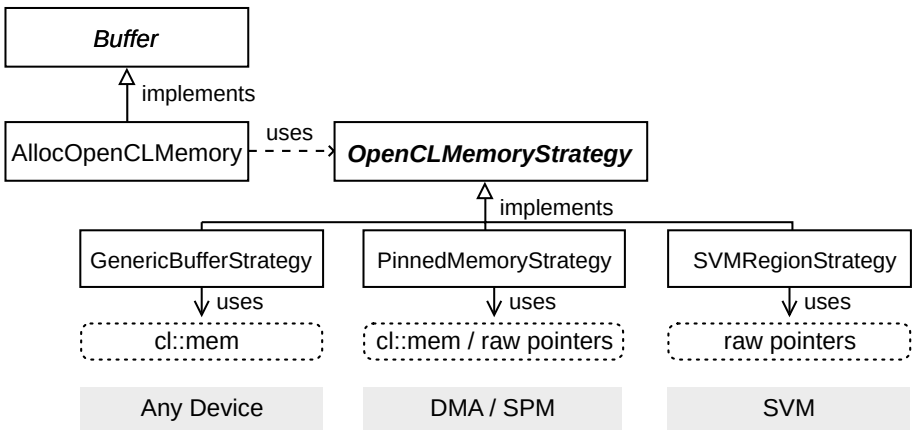


Figure 4-2: Memory model optimizations encapsulated as OpenCL strategies.

Finally, by tweaking OpenCL buffer flags that set the direction and use of the memory block with respect to the device and program, OpenCL drivers are able to apply underlying optimizations to the memory management.

4.1.2.3 Algorithmic optimizations

Finally, considering the algorithmic approach, an extension and exploration of two parameters of the best algorithm used so far, HGuided, has been performed in order to maximize the co-execution performance.

Considering the formula for splitting and assigning packages to devices in HGuided, as it is shown in Section 2.2 of the Background Chapter, the package size for device i is now computed as:

$$package\_size_i = max\left(\left\lfloor \frac{G_r P_i}{k_i \sum_{j=1}^{n} P_j} \right\rfloor, min\_package\_size_i\right) \quad (4\text{-}1)$$

As it is detailed in the HGuided Equation 2-1, G_r is the number of pending work-groups and is updated with every package launch. P_i is the computational power of the device i, while P_i and P_j obtains the computational power ratio compared with the n devices of used in the computation. The parameters of the HGuided are the computing powers and the minimum package size. However, performance variations have been found when using distinct minimum packages, as it affects the devices in different ways. Thus, new input parameters are allowed, being able to assign as many minimum sizes as devices are computing. Therefore, the parameter search for this scenario focuses on studying the relationship between the constant k and the minimum packet per device, keeping the rest of the parameters unchanged.

The minimum package size is a lower bound for the $package\_size_i$ and they are usually dependent on the computing power of the devices, being bigger package sizes in the most powerful devices. Moreover, k_i is an arbitrary constant. The smaller the k constant, the faster decreases the package size. Tweaking this constant prevents too large package sizes when there are only a few devices, with cases such as giving half the workload in the first package to a device, unbalancing the load.

HGuided is optimized by applying a combined tweaking to both the k constant and the minimum package size, inversely related. For each device, a pair of minimum package size and k_i constant is given. The former is a multiplier of the local work size and it increases with more powerful devices, while the latter decreases in such cases. The k_i, although related with the computing power of each device, it is established with values between 1 and 4 to avoid crossing the border penalties: neither too large nor too small packages.

4.1.3 Methodology

The experiments are carried in *Remo*, the heterogeneous commodity system composed of *CPU*, integrated GPU (*iGPU*) and *GPU*, as it is described in Section 1.7.1.

Five benchmarks have been used to show the performance gains of the optimizations. Table 1.7.2 shows the properties of the selected benchmarks: Gaussian, Binomial, Mandelbrot, NBody and Ray. Moreover, Ray is provided with two scenes with a variety of lights and objects to explore different irregular behaviors.

The performance evaluation of EngineCL for time-constrained scenarios is done by analyzing the co-execution with the optimizations in the heterogeneous system. As it is highly detailed in Section 2.2, Static and HGuided algorithms are evaluated, along with the new HGuided optimized version described in this Integration. The Dynamic algorithm is discarded because it delivers low performance, strongly penalized by the synchronization due to the short duration of the executions. Few packages generate a strong imbalance, whereas many impose high execution overheads. The scheduling configurations are grouped by algorithm. The first two bars represent the Static algorithm varying the order of delivering the packages to the devices. The one labelled *Static* delivers the first chunk to the CPU, the second to the iGPU and the last one to the GPU, while in the *Static rev* the order is *GPU-iGPU-CPU*. Then, the latter present the HGuided algorithm and its new optimized version.

To evaluate the performance of the load balancing algorithms the total response time is measured, as well as the response time of each of the devices. The measures include the kernel computation and buffer operations (reading and writing), but excluding program initialization and releasing, as part of a time-constrained scenario. Each program uses a single problem size, given by the completion time of around 1.5 seconds in the fastest device (GPU). Then, as it is exposed in Section 1.7.3, three metrics are calculated: balancing efficiency, speedup and heterogeneous efficiency.

To guarantee integrity of the results, 50 executions are performed per case. An initial execution is discarded for every set of iterations to avoid warm-up penalties in some OpenCL drivers and devices.

4.1.4 Results

This section presents results of experiments carried out to evaluate the performance when using all the devices in the system with the load balancing algorithms, including the new runtime and HGuided optimizations. In other words, all schedulers benefit from new design decisions, parallel initialization steps and memory management optimizations. As noted in Chapter 3 during the energy efficiency validation, *Remo* AMD APU does not allow measuring energy consumption as it does not have accessible hardware counters or alternative mechanisms to measure it, so the study of energy efficiency would be very limited with only the discrete GPU.

4.1.4.1 Performance Results

The performance results achieved in the heterogeneous system with different load balancing algorithms are shown in Figures 4-3 and 4-4, where the speedups and efficiency are depicted. The abscissa axis contains the benchmarks defined in Section 4.1.3, each one with four scheduling configurations. The last group of bars shows the geometric mean per scheduler.

The results reveal, as contrasted in Chapter 3, that HGuided scheduling configurations achieve the best results. However, there are situations in which some algorithmic configurations can reach or even slightly surpass the original version of the HGuided balancer, as in the case of the Static algorithm for the NBody computation.

Any extra operation can cause an overhead that penalizes the complete execution, especially in this type of scenarios with short execution times. For this reason, Static is the simplest algorithm and allows an efficient implementation, providing acceptable results since it generates low overheads due to minimum package and device synchronizations. However, HGuided is a much more complex algorithm to implement optimally, requires more operations to distribute the packages and involves more synchronization overheads. And yet, its algorithmic advantage most of the time overcomes the other negative points, as demon-

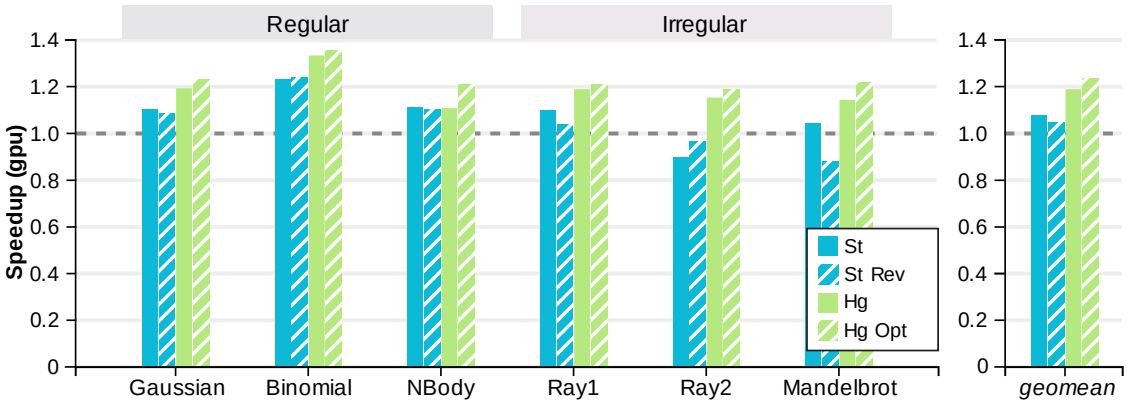


Figure 4-3: Speedups for every scheduler and program compared with a single GPU.

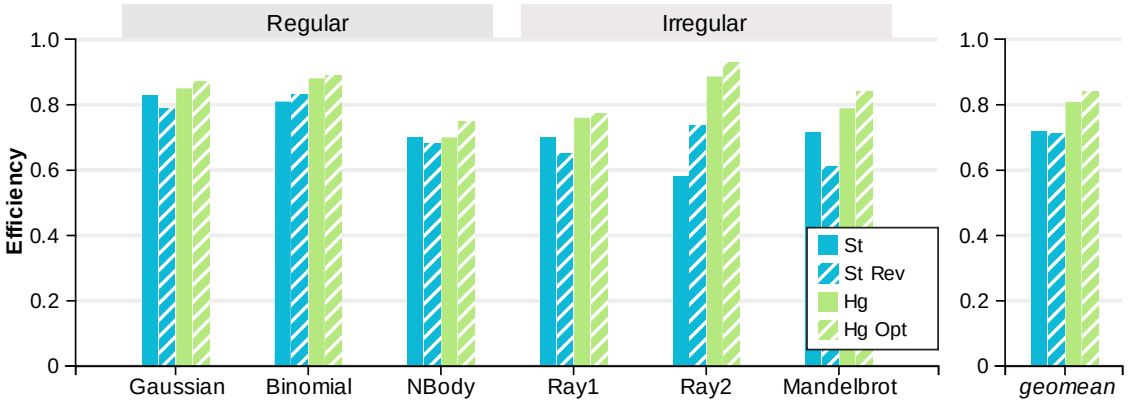


Figure 4-4: Efficiency for every scheduler and program compared with a single GPU.

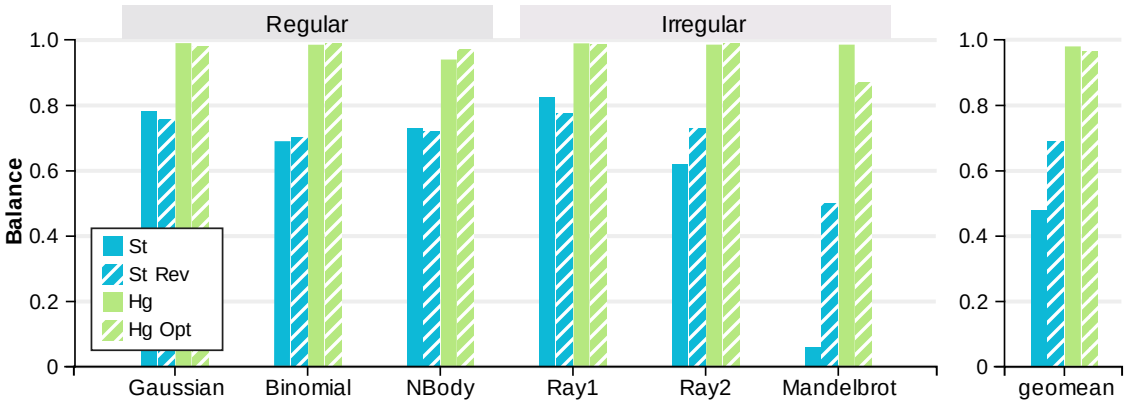


Figure 4-5: Balance for every scheduler compared with a single GPU.

strated in these and previous experiments.

The algorithmic optimizations of HGuided allow improved performance for all the cases studied, which is not the case with the previous version. Nevertheless, they require a detailed analysis of the execution scenario in order to establish the appropriate parameter tuning. However, since they are parameters with a range of values over which to perform the exploratory search, it is a task that does not require programmer intervention. Regarding its differences, HGuided achieves average efficiencies of 0.81 and 0.84, when using the default and the new optimized version, respectively. Due to the optimizations applied, the results of the HGuided improve 3% for regular problems and 3.5% for the irregular ones, being ahead of the rest scheduling configurations and reaching efficiencies of up to 0.89 and 0.93 for Binomial and Ray2, respectively.

Figure 4-5 depicts the balance metric obtained per scheduler, as it is defined in Section 4.1.3, achieving near the best balance when using HGuided, in all applications. This is a consequence of the computation of smaller packages at the end of the execution. For regular problems, the balance is directly related with the performance of its scheduling configuration, but it is not completely fulfilled for irregular ones, with cases like Mandelbrot. It has higher performance in Static than in its reversed version, but it suffers imbalance. These variations are produced because a slow device (CPU) finishes working and no more packages need to be computed, waiting for the other devices, but finishing the total problem computation in less time, because faster devices compute more work.

4.1.4.2 Optimizations Evaluation

Taking into consideration the global vision of the problem as well as the excellent results obtained with the new proposal of the HGuided, it is necessary to emphasize the optimization work performed at the algorithmic level as well as in the runtime, as it was detailed in Section 4.1.2.

Figure 4-6 shows how the performance is affected by the combination of pairs (m, k), being m the multiplier value to obtain the minimum package size and k the HGuided constant,

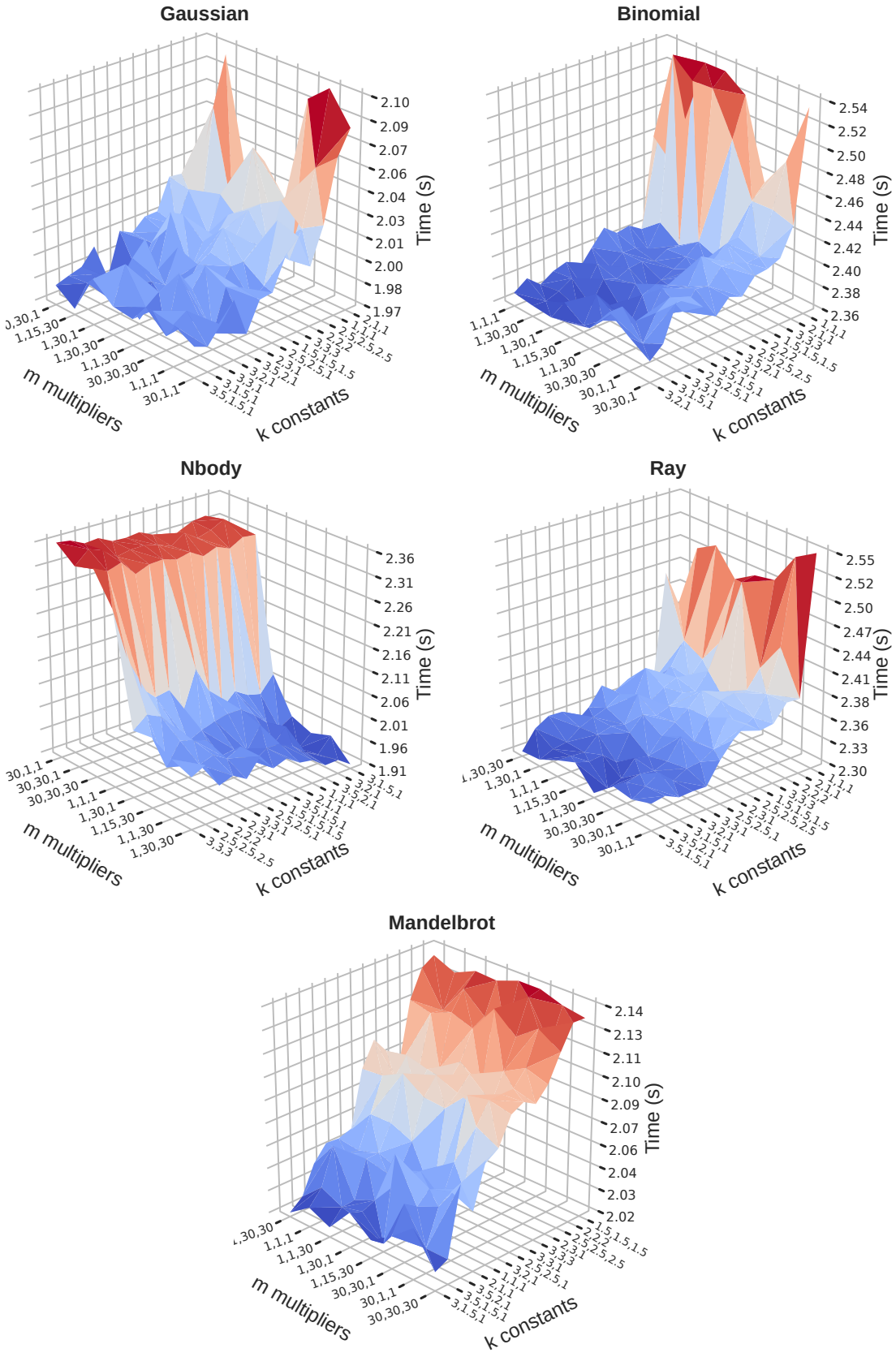


Figure 4-6: HGuided scheduler performance: m multiplier (minimum package size) and k constant parameter combination.

both for each device in the heterogeneous system. The Z axis sets the program execution time for the problem size indicated in the Section 4.1.3, while X and Y axes show the m and k values per device, respectively. The order of the three values represent the CPU, iGPU and GPU. For example, a Gaussian execution with the values $k\{3,3,1\}$ and $m\{1,15,30\}$ means that the HGuided has scheduled the workload by distributing smaller packages to the CPU and iGPU but larger to the GPU. This k variation affects the first packages delivered. In addition, as packages are distributed and their size decrease is when the minimum package size limitation occurs. In this case, because of the m selected values, the CPU is not limited, but the iGPU and GPU minimum package sizes are 15 and 30 times larger than the local work size, respectively. For instance, the local work size (*lws*) of Gaussian benchmark is 128, as it is detailed in Section 1.7.2.

For all the programs there is a correlation between the m multipliers and k constants, inversely related. The charts show how the k constants have a greater impact on performance than the minimum packet size, except in NBody, where the limitation of the minimum package size for the CPU is completely relevant to avoid synchronization overheads. The more packages are created, the more management needs to be performed, and both Runtime and Scheduler units are CPU-managed (host thread), incurring in more overheads when dealing with transfers and computations. Although this effect can be appreciated in every application, in NBody it is even more highlighted due to its communications.

Considering all programs, by classifying the results based on the performance average, several conclusions can be extracted to improve the general efficiency:

- The more powerful the device, the greater the minimum package size.
- The more powerful the device, the fewer the k constant.
- There is not a perfect choice for every program, but the combination of $m_i = \{1,15,30\}$ and $k_i = \{3.5,1.5,1\}$ give the best results.
- If a single k constant should be selected for every device, $k = 2$ is the best option.
- If the CPU is involved in the computation and no previous profiling can be performed, it should maintain $m = 1$ to avoid major penalties in performance.

On the other hand, taking into account the runtime, two tasks have been carried out: the *initialization* optimization affects the execution of the complete program (binary), while the *buffers* optimization improves both binary and based on the region of interest (ROI). The ROI discards the initialization and release stages, considering only the transfer and computation, where a minor management overhead or synchronization call highly penalizes the general performance.

Figure 4-7 depicts the evolution of the execution time as the problem size increases for each program, as well as the trade-off between single or multi-device execution. The abscissa axis represents the problem size expressed as total work items (gws). The green (top) and blue (bottom) lines show how the execution time increases as the size of the problem grows, for binary and ROI, respectively. Every vertical line represents a inflection point when it

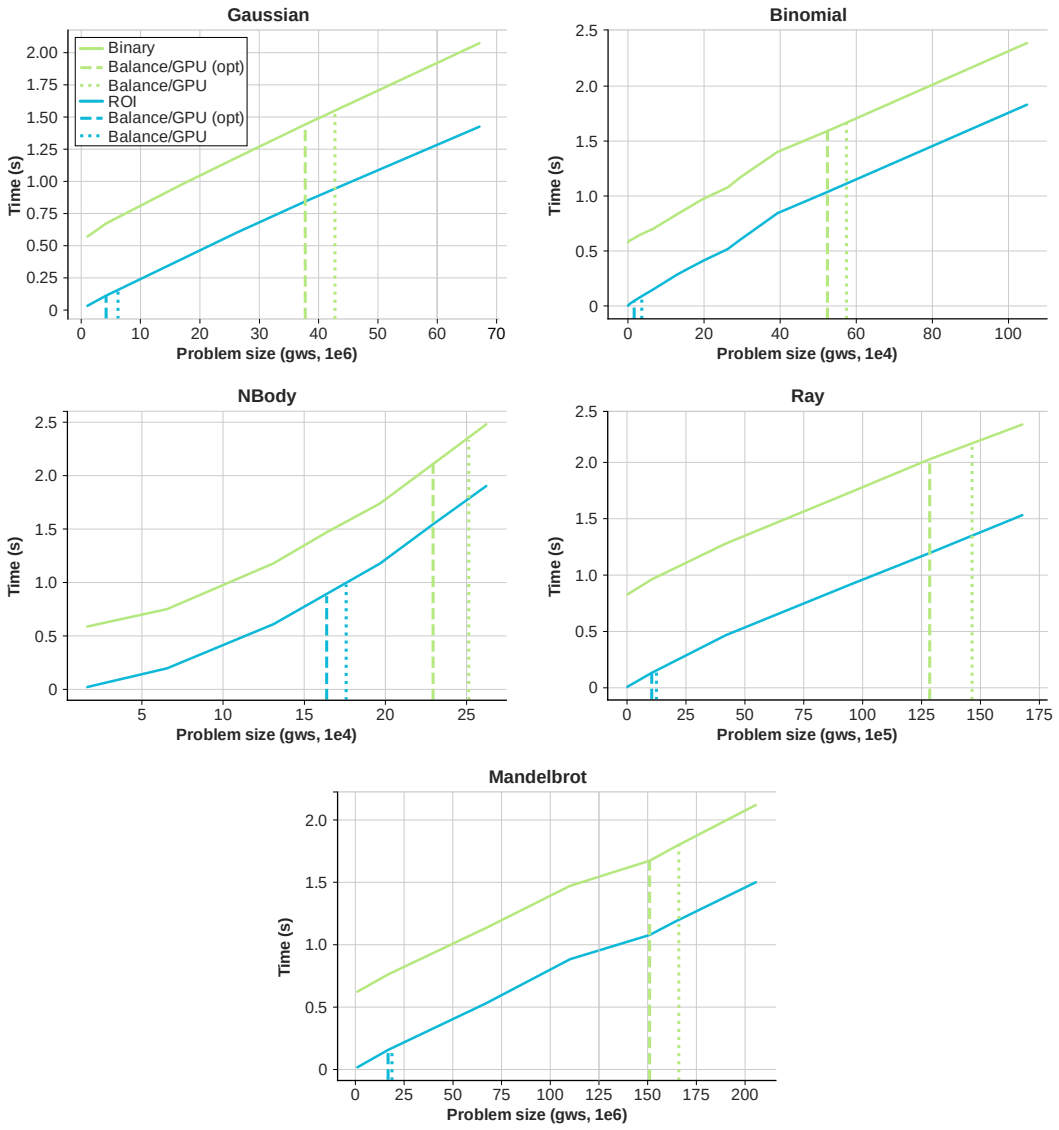


Figure 4-7: HGuided Opt execution time per problem size when launching the binary and only the region of interest (transfer and compute).

crosses the previous continuous lines associated with its execution type, indicating when it is worthwhile to balance the load with HGuided Opt, compared to executing only in the fastest device (GPU). The dotted lines show the previous version, while the dashed lines the new optimized one.

Analyzing the results in detail, all programs show a similar increasing linear trend in their first seconds of execution, except NBody, which grows exponentially. The difference between executions is practically a constant value, due to the amount of common operations during initialization and end of execution, although influenced by the type of kernel (compilation and arguments configuration) but also the amount and size of buffers involved.

The *initialization* optimization has a strong impact on its constant, saving 131 milliseconds on average when reusing OpenCL primitives and the configuration stages are highly parallelized until they are limited by their own dependencies (Buffer, Context, Queue, etc).

On the other hand, *buffers* optimization affects both execution modes, but has a greater impact on ROI, as it affects transfers. Devices that share the same main memory can reuse the buffers without penalty. Nevertheless, its impact is minimal for such small problem sizes, since the transfers are practically negligible compared to the rest operations.

The inflection points improve, on average and taking into account the two types of execution, 7.5% when optimizing the initialization and 17.4% when facilitating the recognition of buffers types and avoiding unnecessary copies. These two optimizations are fundamental considering the scenario in which these applications are evaluated.

Considering the experimental setup, a set of conclusions can be indicated:

- The average time it is worthwhile balancing the load has to exceed 15 milliseconds when considering the region of interest, and 1.75 seconds when executing the complete program.
- The bigger the problem size, the better it performs load balancing, and therefore, it excels over a single execution over the fastest device.
- The amount of potential optimizations and time savings that can be achieved in the runtime are limited by the gains obtained by applying optimizations on the slowest device and being executed alone (single mode).

In summary, both EngineCL and its load balancing algorithms, thanks to the exhaustive analysis and evaluation, support optimizations and fine-grained work focused on specific scenarios, significantly improving their results.

4.1.5 Conclusions

This Integration has highlighted the importance of optimizing both software and algorithmic designs, in order to properly exploit commodity systems and time-constrained scenarios. These situations cause high overheads in work-load distribution and device management, making it prohibitive for such short-duration problems, where the host-device model is heavily penalized. This reveals the importance of adaptability in runtime systems, reflected in the changes needed to exploit other types of scenarios for which it was not initially conceived.

Thus, considering these scenarios and the challenge of achieving efficient co-execution, the contributions focus on two main improvements. First, a number of optimizations have been designed and implemented on EngineCL that address the initialization stages, the reuse of primitives in the multi-threaded architecture and buffer management. Second, an effort has been made to optimize the HGuided scheduler, tunning the values of its parameters. For this purpose, an experimental evaluation has been carried out to evaluate the variations and relations of the parameters of this algorithm under the worst-case scenario for load balancing.

A number of conclusions can be drawn from the experimental results. Firstly, the pro-

posed optimizations reduce the execution time of a program by 7.5% and 17.4% to make co-execution successful for the binary and ROI operation modes, respectively. Regarding the HGuided parameters, it can be concluded that the more powerful the device, the larger the minimum package size and the lower the k parameter value should be. Thus, the best result is always obtained if the minimum package size is not limited for the CPU, when being the less powerful device in the system. Finally, thanks to all the optimizations, the new load balancing algorithm is always the fastest and most efficient scheduling configuration, yielding an average efficiency of 0.84. Thus, opportunities to compete in time-constrained scenarios against the fastest device increase.

4.2 Integration II: hybrid programming models

Molecular dynamics simulators are a relevant application field for heterogeneous systems due to their potential savings in execution times and reduction of energy consumption. For this reason, they have been optimized for years for multi-core HPC architectures, making it difficult to port them to other architectures that allow simultaneous computation on several devices.

In this Integration, EngineCL is extended to enable efficient co-execution of molecular dynamics kernels of the ls1-MarDyn simulator. Several contributions are made including support for a new execution core and a hybrid co-execution mode, solving the problems encountered when executing with OpenCL-based technologies. The architecture has been modified to offer greater flexibility and to combine programming models, offering new encapsulations and API adaptations. The changes produced allow similar CPU efficiency to the optimized and vectorized code currently used by the simulator, while exploiting simultaneously the accelerators of a heterogeneous system, reducing the total computation time.

4.2.1 Motivation

The advent of heterogeneous systems has opened up new optimization opportunities for applications that have traditionally been optimized to run on clusters of CPUs, like the ls1-MarDyn molecular dynamic simulator [116].

An interesting collaboration arises with researchers of the *Scalable Programming Models & Tools Department* (*SPMT*), located in the *High Performance Computing Center Stuttgart* (*HLRS*), the supercomputing and research center of Stuttgart, as a result of a stay during the second quarter of 2019. This Integration work is carried out until the beginning of 2021, focusing on the exploitation of ls1-MarDyn in heterogeneous systems. The essence of this work is that it is the first real scientific application of the dissertation, moving away from the benchmarks and synthetic or simplified applications, typically used in most works and benchmark suites. A feasibility study was carried out in the porting and exploitation of this software with OpenCL, finding many challenges and complexities derived from the software architecture and optimizations implemented to its core. It is a very ambitious simulator, developed for years by multiple physicists and expert computer optimizers. Since the rebuilding of the software architecture to fit accelerators is an unfeasible task, due to constraints regarding time and the access to the original architects and developers, the approach has shifted. A study of the design principles and the currently implemented parallelism has been carried out, with the aim of recognizing the most computationally intensive regions and incrementally porting these bottlenecks using EngineCL.

Two key insights emerge from this porting efforts. The first one, focused on the architectural modification of the simulator, profiling studies and proposals for adapting the base

parallelism, all of them of interest to ls1-MarDyn developers and optimizers. The second one, the extensions and optimizations applied to EngineCL, the impact of the underlying technologies and its experimental results in heterogeneous systems. For HPC interest and coherence with the rest of the dissertation, this Integration focuses on this second aspect. The following results focus on evaluating the behavior of the most intense and/or most used regions, present in different software layers of the simulator. Therefore, they serve as a reference in order to study the impact of the optimizations. They are called kernels for simplicity, considering the OpenCL/EngineCL terminology.

The use of heterogeneous systems and the OpenCL technology for ls1-MarDyn gives rise to two main challenges: device performance portability issues as the programming model varies and the programming complexity when extending the core of EngineCL.

Challenge 1: OpenCL performance portability. One of the key points of the performance of a device and the associated OpenCL programming model is determined by the quality of the driver and the optimizations provided by the vendor. This has been a serious problem encountered during the OpenCL technology applicability study in the kernels extracted from the ls1-MarDyn simulator. The Intel Xeon processor requires a degree of optimizations not achievable by its driver regarding these molecular dynamics kernels, causing a performance penalty. Thus, the feasibility study only allowed a correct use by the GPU, but ruled out any possibility of exploiting the CPU.

EngineCL facilitates incremental transformation and analysis of compute-intensive regions, thanks to its architecture, built-in schedulers and introspection capabilities. However, on the other hand, since EngineCL uses internally OpenCL technology, it still presents this problem. The ls1-MarDyn software uses parallelized and vectorized kernels, so OpenCL is not able to cope with such an optimized code, as it is depicted in Figure 4-8. It depicts the time to compute two types of kernels for different problem sizes. Gaussian kernel, a classical kernel and evaluated throughout the dissertation, is shown on the left. It shows how both OpenCL and OpenMP technologies offer similar performance. On the right, on the other

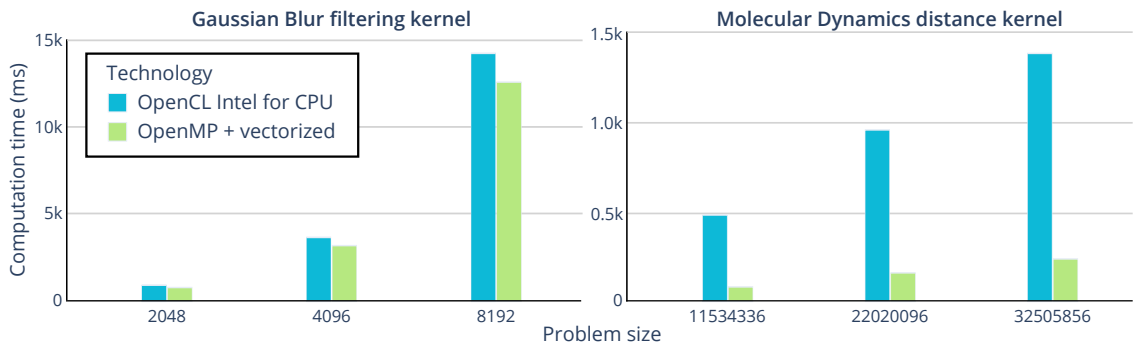


Figure 4-8: CPU computation times for classical and ls1-MarDyn kernels, using OpenCL and OpenMP technologies for a set of problem sizes.

hand, shows the penalization of OpenCL when computing a ls1-MarDyn kernel. For this reason, there is a need to improve the performance of OpenCL in comparison with the code optimized for CPUs.

Consequently, it is necessary to transform the core of EngineCL, enabling the exploitation of hybrid programming models. That is, the combination of different programming paradigms, languages or technologies, but being treated as an integral part in the resolution of a program (*Program Domain* in EngineCL nomenclature). The objective of this combination is to increase the runtime flexibility and be able to exploit different programming techniques and source code origins. All of this while enabling transparent co-execution between hybrid programming models to achieve maximum performance.

Challenge 2: Programming complexity. Due to the need of mixing technologies and programming models based on different philosophies, new drawbacks arise. The main one is the increased complexity of the whole system to enable all the advanced and innovative functionality. Such technological flexibility requires structural changes, affecting all software layers. However, EngineCL is a runtime system with proven adaptability and modularity. Therefore, it is necessary to adapt the runtime in order to maintain its premises and design principles, but to provide it with new computational capabilities more optimized for the CPU. To guarantee the abstraction and performance portability achieved by EngineCL and validated through the dissertation, the runtime must preserve its main features. In short, EngineCL should keep a negligible overhead with respect to OpenCL, allow efficient co-execution and maintain a clean API design, while providing new execution methods that are independent of OpenCL to properly exploit the simulator.

The goal is to achieve efficient co-execution between CPUs and accelerators, leveraging years of optimization efforts to exploit Intel Xeon CPUs. These have ranged from the application of vectorization and parallelization with threads, through memory hierarchy exploitation, embedded assembly code in specific regions, to the application of mapping and affinity strategies. In this particular case, the optimizations materialize in the execution of molecular dynamics kernels, but the strategies and extensions developed are applicable to other types of applications. Nevertheless, this work shows the impact of these optimizations and the use of different devices with respect to the initial implementations limited to CPUs.

4.2.2 Overview

The proposal of this extension focuses on enhancing EngineCL with more functionality, without compromising its usability. The main objective of this Integration has been to provide model support for hybrid heterogeneous computing models. In particular to the ls1-MarDyn simulator and the heterogeneous system evaluated, it means to be able to combine different computing technologies for CPU-GPU co-execution. Thereby, the runtime has

experienced three innovations from the functional point of view.

First, the entire system has been adapted to support new *execution cores*, such as the native and OpenCL. An execution core is an internal part of the software architecture of EngineCL. It is responsible for translating EngineCL commands into operations of the underlying technology. Until this Integration, OpenCL was the only usable technology. With this work, this concept is defined, amplified and encapsulated, taking full relevance. This has required an internal transformation, including the generation of new interfaces to encapsulate the distinct implementations of its behavior. Furthermore, it has also required a minimal modification of the external API, trying to preserve the high usability.

Secondly, a new execution core has been implemented, the *native execution core*, thanks to the architectural adaptation to support variants of the internal computational engine. This core is specialized in the execution of binary kernels for the CPU, as a native execution, instead of an execution core based on OpenCL.

Finally, the third one focuses on adapting the runtime to support hybrid co-execution, mixing native and OpenCL-based execution cores. In this way, the same kernel is computed simultaneously by two independent technologies, being EngineCL in charge of synchronization, workload distribution and resource management, regardless of the execution core.

The last two are clear optimizations in both offload-based executions with a single device *à la* pure host-device programming model, and co-execution powered by schedulers. The changes made to achieve these improvements are detailed below, focusing on the functional and architectural enhancements that directly affect the optimizations validated in Section 4.2.6.

4.2.3 Optimizations

The optimizations made in the runtime affect multiple system modules, mainly focused on decoupling the components related to OpenCL technology and providing efficient co-execution mechanisms for new execution cores, without requiring structures and mechanisms inspired by OpenCL's own execution and management models. Section 4.2.3.1 provides an overview of the affected modules and the most important design patterns applied.

Since the new native execution core introduces multiple changes in the initialization chain and association to the application domain, the execution model is detailed in Section 4.2.3.2. The divergent behavior created by each execution core is encapsulated, thanks to the proposed architecture. This allows reducing the number of common operations required, while ensuring independent operation flows per device and execution core used. Furthermore, this independence favors specialization strategies in the instantiation of primitives and structures used by the runtime, as is the case of the memory regions and the abstraction performed on the Buffers, highlighted in Section 4.2.3.3.

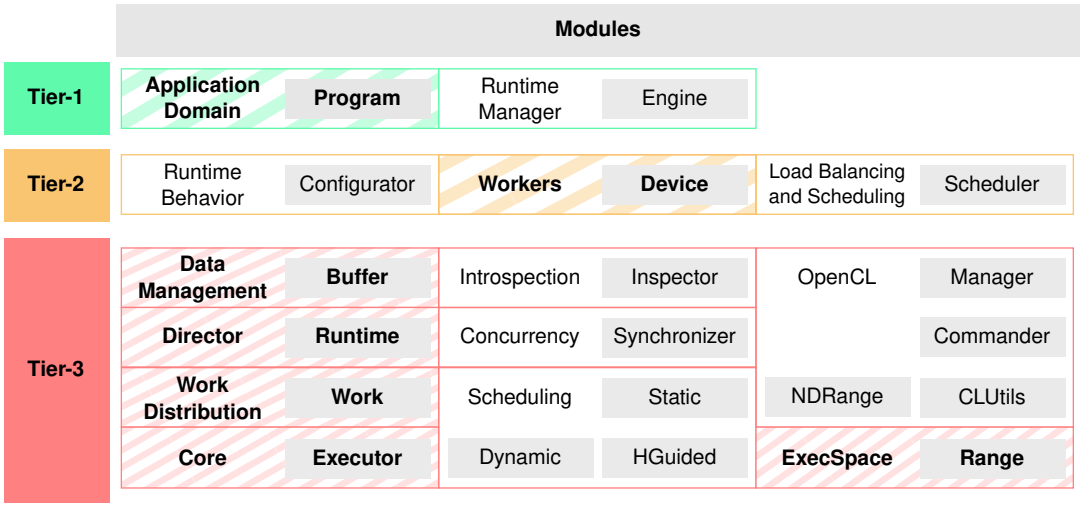


Figure 4-9: EngineCL contexts and main modules, highlighting those affected by the optimizations.

4.2.3.1 Architecture

As it has been detailed in Chapter 3, EngineCL offers a layered architecture in three tiers, increasing the functionality and degree of complexity the lower the tier. The changes of this Integration have been very relevant, producing structural modifications, since the base technology has changed. For this reason, it has been necessary to access Tier-3. As presented in Section 3.3.2, Figure 4-9 shows the layers, tiered horizontally, contexts and the main modules, highlighting those affected by the optimizations:

- `Program`, `Device`: API design modifications to allow using the low-level features of Tier-3.
- `Buffer`, `Runtime`, `Work`: modules extended to support further functionalities, while providing compatible interfaces with other modules.
- `Executor`: new module to deal with different execution technologies.
- `Range`: refactored functionality, extracted from the `OpenCL` context and created as a new isolated `ExecRange` context and `Range` module.

As can be seen, this layered design allows encapsulating the functionality provided by the modules of the lower layers, while favoring functionality reuse and a simplified API design. The original design principles and the resulting software architecture have been necessary to conveniently modify the runtime to support new features.

The adaptation of the runtime to support new execution paradigms has involved a refactoring of the execution engine, which until now was intended only for OpenCL. A new execution interface, `Executor`, has been incorporated, which determines the *execution core* of a device. In order to traverse the execution space, provided by the application domain (`Program`), the `Range` module has been refactored and isolated as a new one, which masks how the kernel is executed, independently of the execution core. In addition, both the `Runtime` and the `Work` distribution are adapted to understand the new abstract execution mechanism provided by Executor. The new component signatures influence how they

are manipulated and instantiated by each `Device` and the `Runtime` itself. These structural changes have not affected the API design, thanks to the layered architecture.

On the other hand, with the addition of a new native execution core, a slight modification of Tier-1 has been facilitated, as will be seen in Section 4.2.4. The runtime can provide a native optimized kernel for the CPU, using directly the `Program`, favoring independency between the computing technologies and the application domain. In this way, the variables and parameters provided to the `Program` class are internally associated to the structures needed by each internal execution core, regardless of the technology used.

Finally, new design principles have been provided to facilitate internal maintainability and extensibility. Three functionalities have been defined that can work independently of each other, the execution space (`Range`), the execution core (`Executor`) and the data management (`Buffer`). Since the behavior of each of the new interfaces depends on the instantiation of the chosen adapter, the *AbstractFactory* pattern is applied to simplify the composition of operating modes [380].

In this way, the mode of operation of the core does not determine the rest of the internal components on which it depends, the extensibility of the runtime and its internal execution mechanisms. *Abstract Factories* facilitate the construction of products with interchangeable parts. Therefore, the `NativeFactory` and `OpenCLFactory` are consolidated in this Integration, assisting when instantiating and manipulating native or OpenCL primitives and operations.

The `NativeFactory` builds the most optimal components for a native execution mode on CPU, instantiating an execution space based on C++ iterators, an execution core based on a executable code blob for CPU and a lightweight data management with direct access to host memory. Finally, `OpenCLFactory` offers the original EngineCL components, now refactored and encapsulated as independent instances. It builds an n-dimensional range-based execution space, an OpenCL-based execution kernel, and explicit reservation-based memory management with disjoint spaces.

4.2.3.2 Execution model

The execution model is explained focusing on the divergent tasks regarding the execution cores, helping to understand the main differences that allow supporting distinct executors including the novel native CPU processor. To simplify the model, it is taken into account the compilation and execution phases of a program when using the runtime. Moreover, an overview of the co-execution process is described, focusing on job scheduling and synchronization with devices that is performed internally.

Figure 4-10 shows the kernel compilation stage in the upper part, while the lower one summarizes the stages produced during the execution phase. The starting point is a source code with an OpenCL kernel, while the offline compiler is used to prepare the binary kernels to be used later during runtime execution.

Considering the compilation stages, the `clkernel` tool performs a compilation for the different devices present in the system, thanks to the *ICD loading mechanism* of OpenCL and the subsequent binary construction offered by its drivers. On the other hand, the `cl2native` tool performs a source code transformation to be compiled by the different backends present in the system. The programmer can include annotations to facilitate the conversion, provide his own optimized variant or even use his own binary file, as long as it maintains the appropriate signature to be consumed. The tool to provide automatic transformation serves as a straightforward and easy method to execute kernels with the new functionality, but hand-optimized variants would be the best option to reach the maximum performance. In any case the code will be provided with a wrapper that establishes a common interface to be called. This signature will be consistent with the specification needed by EngineCL at runtime. The programming model chosen by cl2native is OpenMP, but the programmer is free to use any other strategy and model. Finally, one of the established compilers, such as `icc` or `gcc`, will be used, building an optimized binary kernel with *position-independent execution* and without *name mangling*, to be used directly by the runtime. In both cases, the resulting files contain the pre-compiled programs with the code ready to be consumed by the different devices of the heterogeneous node.

Subsequently, during the execution phase, as it is shown in the lower part of Figure 4-10, as soon as the `Engine` module uses the `Program`, a series of steps occur, affecting the chosen execution mode. The `Runtime` creates and configures the devices through the `Device` interface, initiating their configuration and starting to operate independently. This behavior is

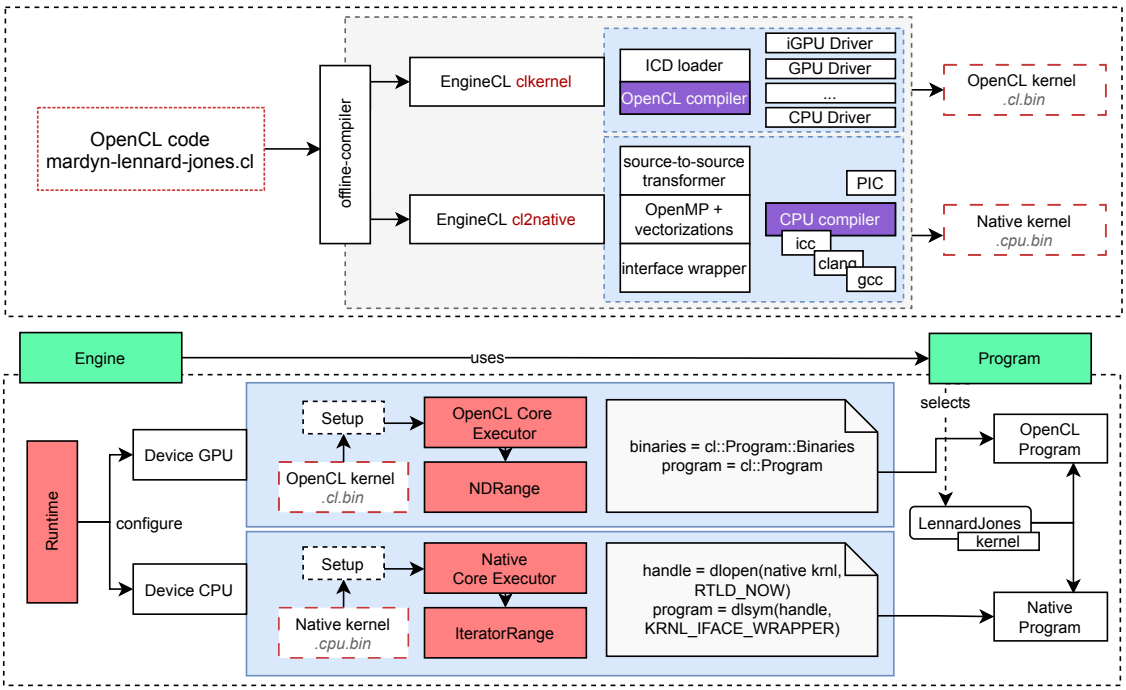


Figure 4-10: Kernel source code compilation process (above) and initialization during the EngineCL execution phase (below).

hidden from the rest of the runtime, while they perform a set of steps depending on the chosen mode of operation determined by the instantiated components (*Factories*, for instance `OpenCLFactory`). Figure 4-10 shows two paths at the bottom, one for the GPU and one for the CPU. The OpenCL binaries are initialized and assigned to the devices associated to the context managed by the Device, that is the GPU, who uses an execution engine based on OpenCL, an n-dimensional range and a program based on a low-level class of OpenCL. On the other hand, the CPU device uses the native execution core, its execution space is based on an iterator and initializes the program through a dynamic indirection mechanism. Finally, the previously constructed binary kernel is loaded dynamically in a blocking fashion, and the start function of the program is subsequently configured. Access to the appropriate symbol within the executable is provided by using a wrapper with a common signature.

Considering the major steps in the scheduling process, Figure 4-11 shows the overview when using the hybrid co-execution model. This represents the extension produced on the original design shown in Section 3.3.1 of Chapter 3. In this example, each device uses a different execution core, the CPU device exploits the native core, while the GPU keeps using the OpenCL core. The execution cores are encapsulated for the rest of the system, providing opaque operations for the runtime.

Thanks to the optimizations implemented, the native kernel configured with buffer bypass (`bypassed allocs`, `bypassed`) avoid a number of operations that occur in the OpenCL-based model (`allocs`). For example, the reservation of disjoint memory spaces (configuration phase), the discovery and configuration of platforms (`fast init`), the enqueuing of read and write operations, or the data exchange mechanisms during asynchronous op-

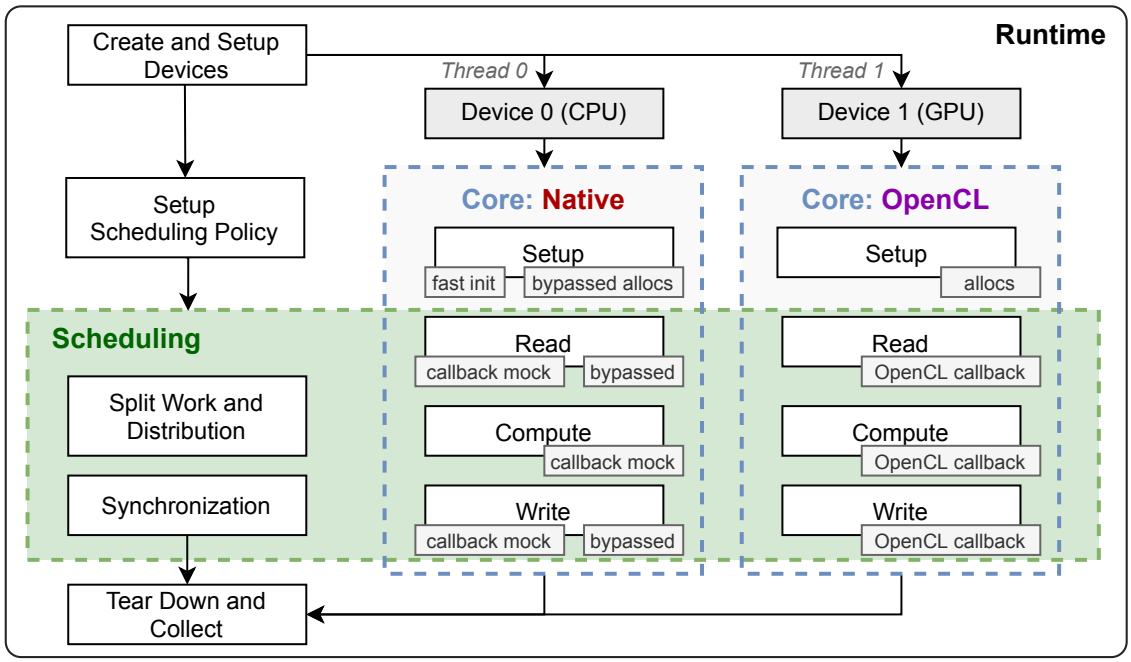


Figure 4-11: Technology encapsulation in relation to the scheduling mechanisms with the hybrid co-execution model.

erations. As the interface has to be consistent for communication with the rest of the runtime modules, some of the operations offer light layers for interaction, such as callbacks (`OpenCL callback`), synchronization or the queuing of work packages and their computation operations, which are not necessary in the native model (`callback mock`). The slowest operation of the native core execution process is the initialization and its blocking load. This phenomenon of blocking in the runtime initialization process has similarities with what happened initially EngineCL and how it was optimized in Section 4.1.2 of Integration I. Those enhancements contributed to speed up the initialization phases of the runtime when developing this hybrid mode. Thanks to the multi-thread architecture and optimized pipeline, the loading of resources by the native core does not affect the rest of the runtime phases, whether it is the device initialization, the scheduler configuration or the distribution of work packages. The scheduling stage of EngineCL is not affected by the different execution cores, so the division and allocation of work as well as the synchronization stages continue to maintain the same mode of operation as in the original engine.

4.2.3.3 Memory model

The outline of the memory model, regarding the modifications performed in this work, is shown in Figure 4-12. Originally there was only one `Buffer` class that encapsulated an OpenCL buffer, reserving one region of memory on the host and another on the device, except if it was the CPU, in which case there was only one region.

With the addition of the native mode and its optimizations, two new types of buffers have been offered for EngineCL. On the one hand, those based on memory region reservation, `BufferAlloc`, where two classes can be instantiated. The `AllocOpenCLMemory` acts as a *Proxy* pattern that delegates actions to an OpenCL buffer, preserving the initial EngineCL behavior for every OpenCL device [380]. The `AllocHostMemory` provides a host buffer that can be used and manipulated exclusively by a native execution core.

On the other hand, a `BufferBypass` class is provided as an optimization for the native core. This mechanism acts as a proxy with respect to the `AllocHostMemory` class, being able

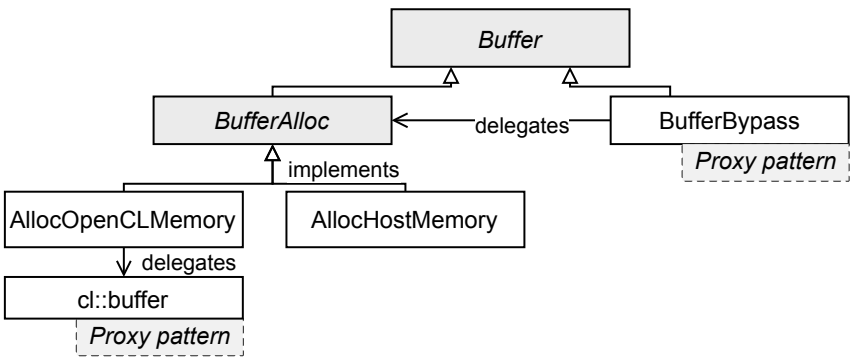


Figure 4-12: Hybrid memory model optimizations providing new classes and interfaces as an abstraction of the EngineCL Buffer.

to configure the behavior depending on how the memory region is accessed. It offers two configurable behaviors, either at compile time with a static policy or at run time based on dynamic conditions, such as how to access the memory region, including size, data type or position. Considering ls1-MarDyn and its kernels, it has been found that a pure bypass strategy is the most advantageous, since there is no reservation or copying of memory subregions. However, this bypass mechanism can be useful for other configurations, applications and systems that use the new execution core. That is, the `BufferBypass` class configured by the runtime to always reuse host memory and never make instances of the `AllocHostMemory` class under any conditions. Thereby any memory request acts directly on the C++ containers or the host memory regions provided by the application domain (`Program`) itself. It is worth mentioning that the possibility of beneficial use of `AllocHostMemory` could be found in kernels that mutate buffer contents or where more sophisticated execution space strategies are performed, taking advantage of the memory hierarchy, such as in stencil algorithms.

Finally, it should be noted that the use of the native model reduces the memory requirements of the runtime, since the OpenCL-based operating mode includes multiple primitives and structures to be able to use this technology. The OpenCL execution core and its increase in memory occupation is not directly related to the application domain data, but is influenced by the use of this programming model and the number and architecture of the devices. Examples are contexts, command queues, events or the callback payloads themselves. Therefore, by using the native execution core, the runtime is being lightened, providing it with higher performance, less memory footprint, and facilitating a more beneficial co-execution, as will be seen in the Section 4.2.6.

4.2.4 API Design

The design pillars of EngineCL are usability and performance, and as noted in the Architecture Section 4.2.3.1, the innovations introduced have not adversely affected the Tier-1 and Tier-2 used by programmers and users of the runtime. Listing 3 shows a block of code that computes one of the kernels exploited and optimized in this Integration, the Lennard-Jones potential for a set of molecules. The first two statements obtain the OpenCL kernels, both generalist and specialized for a GPU. Next, the problem data is reserved and initialized, as part of the ls1-MarDyn simulation process, masked in the `ljpotential_init_setup` function. The runtime `engine` is initialized and configured between lines 11 and 20 ($L11-20$), where the devices to be used and the scheduling algorithm are established. Subsequently, the application domain is built, where the input and output buffers are specified ($L23-25$), as well as the kernel and its arguments to be executed ($L27, 28$). Finally, the program is provided to the runtime to be executed in co-execution.

Regarding the high level API, the only noticeable variation after these innovations, is the one produced in lines 13 and 14, where the Engine is being requested to use the CPU natively

```
/* binary and source code file readers */
auto kernel = file_read("mardyn-lennard-jones.cl");
auto tesla_kernel = file_read("mardyn-lennard-jones.gpu-k20m.cl.bin");
/* omitted for brevity: data init, mesh, fields and potentials */
vector<cl_float4> in_pos(molecules);
vector<cl_float4> out_pos(molecules);
auto gws = molecules; auto lws = 64;

ljpotential_init_setup(molecules, /* ..., many args */ in_pos, out_pos);

ecl::EngineCL engine;
engine.use({
              ecl::DeviceMask::CPU,
              ecl::Device::CpuBlob("mardyn-lennard-jones.hand-opt-cpu.bin")
           },
           ecl::Device(2, 1),
           ecl::Device(1, 0, tesla_kernel));

engine.work_items(gws, lws);
engine.scheduler(ecl::Scheduler::Dynamic(120));

ecl::Program program;
program.in(in_pos);
/* rest of the application domain arguments */
program.out(out_vel);

program.kernel(kernel, "LennardJonesPot");
program.args(in_pos, /* ..., */ out_pos, molecules);

engine.program(std::move(program));

engine.run();
```

Listing 3: EngineCL API with hybrid co-execution mode for LennardJones computation.

through a binary execution object (`CpuBlob`). The rest of the code statements maintain the same API design as originally established, encapsulating all the functionality internally. It should be noted an important feature that allows further customization and potential benefits. The hybrid co-execution mode does not prevent a programmer from simultaneously requesting the CPU by two different execution cores (native and OpenCL), or even identical ones (e.g. two native ones), offering greater flexibility in the exploitation of optimizations or programming models for certain kernels.

4.2.5 Methodology

The experiments are carried out on the node *Trainera*, referred in Section 1.7.1. The first technology involved is the current ls1-MarDyn implementation, labelled *CPU-icc*. It is parallelized with OpenMP, vectorized and compiled with the Intel compiler (`icc`). Then, involving OpenCL technology and its drivers are *GPU* and *CPU-ocl*. Finally, the new hybrid

mode and its native execution core for the CPU, labelled *CPU-hy*.

Five kernels related to the computation of particles and their interactions have been selected as part of the computational core of ls1-MarDyn. Two of them, *md_dist* and *md_distn2* are related to the computation of distances between molecules. The former offers a flow-based interaction with low computational load, while the latter performs calculations based on indirections over all cells. On the other hand, *md_diststar* handles the minimum image convention while computing the distance between molecules. Finally, *md_bin* computes the associated indices for a set of cells in streaming mode, while *md_lj* obtains the potential and evaluates the force for the Lennard Jones 12-6 potential.

The validation of the proposal is done by analyzing the performance of the new native execution core as well as the hybrid co-execution compared with *CPU-icc*. First, a scalability analysis of each device and technology involved is presented, using only one device at a time. To do this, the total execution time each of the individual devices has been measured, increasing the size of the problems.

Then, the analysis of performance and energy efficiency of the hybrid co-execution is performed. The total response time is measured, including kernel computing and data transfer. Two EngineCL scheduling configurations are evaluated when co-executing, Static and HGuided, labelled as *St* and *Hg*, respectively. The Dynamic algorithm is discarded as it gives very poor results with these kernels, penalized by the synchronization of the packages. Speedup and energy efficiency metrics are used to measure performance, as it is detailed in Section 1.7.3. Nevertheless, Energy-Delay Product (*EDP*) is used to measure energy efficiency. To simplify the understanding of the results, the improvement of the hybrid co-execution over the *CPU-icc* version is provided, as it is shown in Equation 4-2.

$$EDP_{ratio} = \frac{EDP_{CPU-icc}}{EDP_{hybrid\ co-execution}} \tag{4-2}$$

To guarantee integrity of the results, the values reported are the arithmetic mean of 30 executions, discarding the first one to avoid warm-up penalties. The standard deviation is not shown because it is negligible in all cases. To measure the energy consumption, another 30 executions are performed to avoid introducing time delays due to the sampling overheads.

4.2.6 Validation

The execution times when using a single device to compute the whole problem are depicted in Figure 4-13, showing how each device scales as the problem size is increased. For all the kernels, the *CPU-ocl* obtains the worst results, making it pointless to use OpenCL on the CPU. These results are so poor that it limits the co-execution, penalizing the runtime management itself and preventing it from being competitive with respect to the *CPU-icc* version.

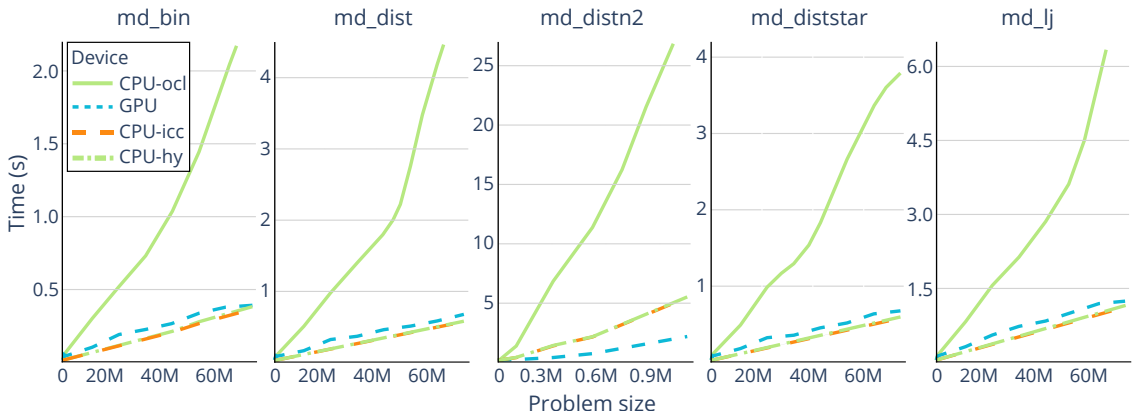


Figure 4-13: Scalability when launching the computation in a single device.

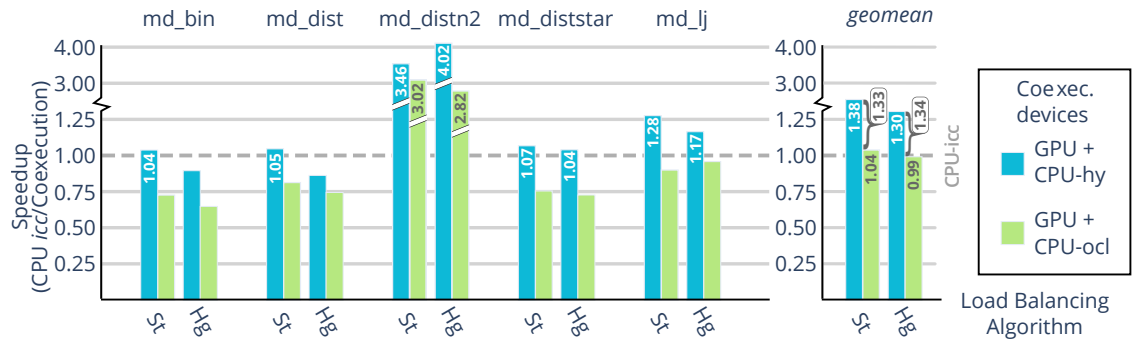


Figure 4-14: Speedups when co-executing compared with ls1-MarDyn technology (*CPU-icc*).

However, thanks to the contributions of this Integration, a new execution kernel for EngineCL is provided that offers very similar performance to the CPU optimized ls1-MarDyn version, as shown by *CPU-hy* and *CPU-icc*. As stated in Section 4.2.5, *CPU-hy* refers to the new native execution core, as in this experiment, but also to hybrid co-execution when using OpenCL and native execution cores at the same time, as will be seen later. On the other hand, the GPU obtains computation times close to these last two CPU modes, although being slightly slower except in the case of the *md_distn2* kernel. It computes 2.64 times faster than the best version of the CPU, when calculating the distances between one million molecules. These kernels are highly optimized for the CPU, taking advantage of the memory hierarchy and vectorizations. Thus, the GPU is not the fastest device, as has been the case in many other classical kernels.

It is now possible to properly exploit a heterogeneous environment thanks to the new execution core and the performance offered. These results show how the CPU with EngineCL is competitive and co-execution strategies may be possible.

Considering the co-execution in the heterogeneous system, Figure 4-14 shows the speedups when co-executing with respect to the CPU optimized version, *CPU-icc*. The abscissa axis shows the load balancing algorithms used for each of the kernels, including the geometric mean. The annotations of the most significant values have been rounded to the

second decimal place in both cases, speedups and energy efficiency charts. EngineCL performs co-execution using the CPU and GPU devices, but after these enhancements, the CPU device can run with OpenCL-based or native execution cores.

The results show that, using the right scheduler in each case, co-execution is always worthwhile, with the new hybrid execution model. This is not the case if the old OpenCL-based model is used, as can be seen by comparing the green bars with the blue ones. The average speedup is of 1.38x, and up to 4.02x on the *md_distn2* kernel. This is due to the new architecture and optimizations enabled by the hybrid mode. Therefore, it allows concurrent operation without incurring overheads that slow down execution, as is the case with the purely OpenCL-based mode. The *GPU + CPU-ocl* setup only becomes competitive with the *CPU-icc* version with a single kernel, due to its computational overhead.

It can be seen that the *md_distn2* and *md_lj* kernels are the ones that offer the highest performance in co-execution, due to the fact that they have a higher computational cost. The number and complexity of their operations, along with the memory regions used per kernel, increase the total computation time. On the other hand, kernels limited by memory or with a strong communication pattern compared with the computation time, are restricted in time. Therefore, dynamic balancing algorithms, such as *HGuided*, do not have enough time to amortize their cost by making decisions at runtime. The *Static* algorithm offers the best generalized performance, due to the simplification of management operations by the runtime, and the correct workload distribution. *Static* is adequate since the total computation time is low and the kernels present regular behaviors, balancing properly the workload. Since kernels are CPU intensive, it is counterproductive to take up management and scheduling time, as it slows down the final execution for such limited times. On the other hand, it is observed how in the case of *md_distn2*, where the execution is longer, the HGuided algorithm is able to amortize its synchronizations and CPU usage, obtaining shorter computation times than in the *Static* version. Since the total computation time is long enough, it benefits from the parallel operations provided by a strategy that generates multiple chunks at runtime, concurrently computing and doing data transfer. A scalability study would be appropriate to contrast the behavior of different algorithms with these molecular dynamics kernels. However, most of these kernels are memory-bound, and the heterogeneous node is not suitable to perform this analysis that requires long-term computations.

Therefore, it is important to highlight the advantage of having different scheduling algorithms, as each one offers beneficial exploitation situations. This is a situation that had not occurred in the validation of EngineCL in Chapter 3. However, it is positive that the runtime has different schedulers, exploits efficient implementations and benefits in diverse scenarios, including the possibility for the programmer to define his own strategies and balancers with the pluggable system.

Finally, Figure 4-15 shows the experimental results considering the energy efficiency of the co-execution with respect to the system using the *CPU-icc* version. Thus, it depicts the

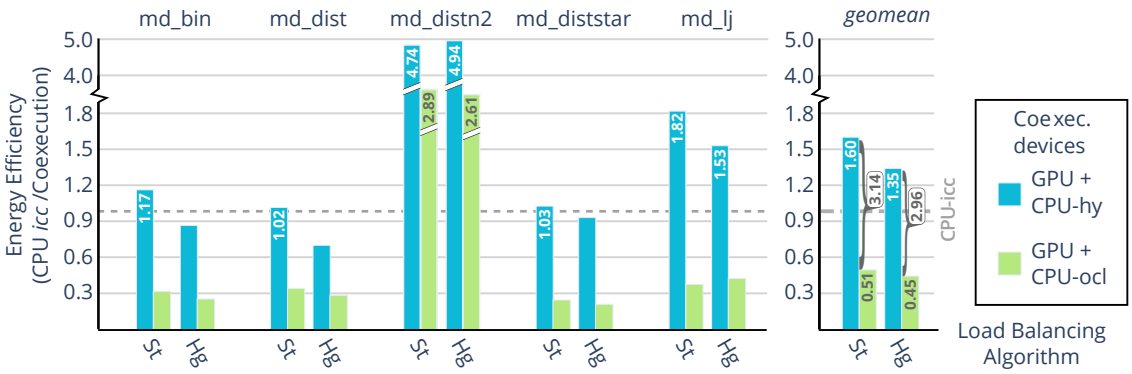


Figure 4-15: Energy efficiency when co-executing compared with ls1-MarDyn technology (*CPU-icc*).

gains in EDP when co-executing compared to using the current optimized version. The conclusions observed in the performance evaluation are accentuated since both energy consumption and response time are taken into account. The GPU is a very energy efficient device, so that in kernels where there is a higher computational load, the improvements with respect to the CPU optimized version are intensified, reaching up to 4.94 in *md_disnt2* and 1.82 in *md_lj*. On average, improvements of 1.60x are obtained with *Static* and 1.35x with *HGuided*, with respect to the *CPU-icc*. Regarding the differences between the *CPU-hy* and *CPU-ocl* based co-execution, the performance is up to 1.34x better on average with the new hybrid mode, while in energy efficiency they increase to 3.14x with *Static* and 2.96x with *HGuided*.

4.2.7 Conclusions

In this Integration, ls1-MarDyn, a highly optimized simulator for HPC processors, is chosen to exploit more efficient solutions that simultaneously take advantage of the different heterogeneous devices of a node, such as GPU and CPU. Since the OpenCL technology for CPU does not have an appropriate performance for a set of molecular dynamics kernels, a number of innovations on the EngineCL runtime is carried out in order to exploit the co-execution efficiently.

These contributions show the importance of having a modular architecture and the possibility of encapsulating the computational technology. This runtime system not only allows to squeeze heterogeneous systems and their performance, but also offers great flexibility in the mutation of its capabilities and programming models. The major extensions carried out have been the adaptation of the architecture to support new execution approaches, the new native *execution core* for the CPU and a hybrid method of co-execution.

An experimental evaluation is performed to compare both performance and energy efficiency with respect to the current parallelized and vectorized processing mode. Scalability analysis of the new CPU execution core shows similar performance to the optimized mode

used by ls1-MarDyn, improving over the OpenCL version in all cases. When performing co-execution there is always at least one scheduling mechanism that offers improvements over the CPU version, both in performance and energy efficiency. On average, improvements of up to 1.38x in performance and 1.60x in energy efficiency are obtained with respect to the current optimized version.

4.3 Discussion

EngineCL has proven to be a solid runtime system and capable of adapting to the needs that are encountered. This is evidenced by the two integrations detailed here. The first one, focused on short duration executions in low resource environments, one of the most challenging scenarios for the host-device programming model, and in particular for runtimes that offer high software layers. The second, showing how despite being focused on OpenCL technology, there are some situations in which it is not the most effective programming model. Thus, it is necessary to overhaul the guts of the engine, making it possible to exploit other technologies and hybrid forms of accelerator programming.

However, EngineCL has been integrated in other situations, although they have been omitted in this thesis because they are mostly carried out by collaborating groups, are being developed, or are beyond the scope and focus of this dissertation. Still, a brief mention may be of interest, considering the scope of this Chapter. Three other adaptation and integration works have been done throughout this dissertation, one of which has already been published in the scientific community.

On the one hand, the approach to mainstream computing, since the runtime, its building system and execution toolchain have been adapted for operating systems other than Linux, such as Android, by means of NDK with Java-C++ connectors through JNI, as well as Microsoft Windows and its Visual C++. This allows the exploitation of accelerators and devices based on other types of OpenCL drivers and loader systems, sometimes offering more optimized versions due to the efforts of vendors for the gaming communities. In addition, it provides the access to platforms with many other types of programs and use cases, ranging from efficient remote processing in mobile devices with low-power profiles, through the processing of geographic information systems, trading algorithms and rendering engines, to the acceleration of simulators and GUI software based on native operating system libraries, such as Android Window Manager, OpenGL Platform Runtime or Win32 API [386–388].

On the other hand, EngineCL has been integrated as a core part of two utilities for video and audio processing. The first one focused on embedded platforms such as Raspberry Pi, thanks to community open source drivers for VideoCore IV graphics cards [389], while the second one has been exploited in commodity environments with conventional video recorders. This work has shown the compatibility offered by the runtime system, since it can run in restricted environments and operating systems, Arm-based CPUs and with proprietary technologies. The fundamental aspect of this runtime extension is that it supports hybrid computing paradigms, fostering coarse-grained tasks and data-parallel processing within the same node. These efforts allow a user to manipulate and observe a graphical interface with asynchronous support for notifications, use IPC-based distributed computing, and stream processing using OpenMP and OpenCL technologies, all simultaneously. In addition, a fundamental aspect of this work is cross-platform compatibility achieved by em-

bracing such technological complexity and software stacks based on open standards, such as WebRTC and Web Audio API. The first steps of both developments and experiments have been reflected in these works [390, 391].

Finally, the most relevant work so far due to the implications in heterogeneous systems and their programmability. EngineCL has been integrated in heterogeneous systems composed of CPUs, GPUs and FPGAs, thanks to the collaborative effort with researchers from the Computer Architecture Group of the University of Zaragoza. The behavior of FPGAs is very different from other accelerators, requiring other ways of traversing the *Program Domain* and its iteration space, with limitations in the OpenCL runtime API and requiring greater efforts in the synthesis and use of kernels. Even so, the runtime system has been adapted and extended to support this type of architectures, increasing its compatibility and enabling the efficient exploitation of all node devices. It is an interesting and impactful work for the HPC community, precisely because it offers an appropriate performance portability to such heterogeneous machines with configurations that are increasingly popular [48, 125].

In short, this Chapter highlights the importance of thinking about both the present and future challenges. It is important to deliver high efficiency with current environments, devices and applications. However, it is even more critical to provide the foundations to easily extend the runtime capabilities and achieve exploitable solutions to future problems.

Coexecutor Runtime

CHAPTER 5

Coexecutor Runtime

As new technologies and programming models emerge, new possibilities open up in the use of heterogeneous systems. Intel oneAPI and its SYCL-based unified memory model is one of these, amplifying the ambitious horizon of high-level programming languages for heterogeneous computing. However, this proposal has some challenges, both in programmability and efficiency when exploiting diverse architectures.

In this chapter, Coexecutor Runtime is proposed as an abstraction on top of the oneAPI technology, increasing the expressiveness and guaranteeing co-execution capabilities. Its approach with a SYCL-compliant API facilitates portability and interoperability with the underlying technology, while providing architecture flexibility to incorporate extended features and new proposals. Its design decisions regarding dynamic mechanisms and the implementation of efficient schedulers allow to fully exploit applications on these systems.

The proposal is validated on two types of heterogeneous systems and with a set of diverse benchmarks, highlighting the improvements in performance, energy efficiency and scalability achieved with respect to oneAPI and its host-device programming model.

Chapter contents

5.1 Motivation

There are many proposals to simplify the programming and management of acceleration devices and multi-core CPUs, as it is exposed in Chapter 2. However, in many cases, portability and ease of use compromise the efficiency of different devices, even more so when co-executing. There are new languages that have gained traction in recent years, trying to increase the level of abstraction, while simplifying the work of the programmer.

One of the most promising programming models is SYCL, which allows to run a single C++ code in different architectures, improving the programmability and enabling code portability. Although initially conceived as an abstraction over OpenCL for C++ programmers, its adoption has been moderate, mainly due to support difficulties from manufacturers and compilers. However, since 2019, Intel oneAPI has emerged as a new and powerful standards-based unified programming model, built on top of SYCL. This commitment and continuous development until the release of the first stable version in the last quarter of 2020, has aroused great interest in the community. This approach has two key advantages over OpenCL. On the one hand, Intel has strongly endorsed it, attracting other manufacturers and communities. On the other hand, SYCL is a very suitable language for the industry, since C++ has stood out for decades. Thus, the adoption of C++/SYCL is straightforward and natural. With the support of major manufacturers and the scientific community, along with the proliferation of the SYCL 2020 standard and its extensions, an inflection point appears regarding the importance of SYCL for C++ programmers.

Therefore, considering this proposal and the new features offered by oneAPI, different challenges can be distinguished.

Challenge 1: Performance portability. As it is previously introduced, Intel oneAPI is based in the host-device programming model, where the compiler builds C++ regions (*kernels*) and the runtime offloads them to a set of hardware accelerators. Its runtime is able to manage complex applications composed of a set of kernels, even if they have dependencies between them, through a Directed Acyclic Graph (DAG), as it is detailed in Section 2.1.3. The assignment of a kernel to a particular device can be done by the programmer, so it is determined at compile time, or let oneAPI choose the device at runtime. In either case, a kernel can only be scheduled to a single device when the dependencies are satisfied.

The only possibility of co-execution with SYCL is for the programmer to split the work into several kernels, as many as devices in the system. Also, data partition and workload distribution must be done manually. Furthermore, the compiler must detect that these kernels are independent and schedule them simultaneously. This complicates the co-execution and, therefore, the exploitation of the whole system to solve a single kernel. Even if the programmer is willing to face this extra effort, an additional problem arises with workload balancing. Since the division of the workload is done at compile time, it is necessarily static. This does

not scale well, as has been demonstrated previously through Chapters 3 and 4, being necessary to address this situation with dynamic balancing algorithms. Nevertheless, as this is a novel technology for another type of architectures, it is necessary to exploit all possibilities and evaluate the behavior in the face of different types of strategies and optimizations.

Challenge 2: Abstraction. The oneAPI technology allows easy offloading to a device, but there are many problems when balancing the load between different devices, since it is still fundamentally a host-device programming model. All the management to achieve co-execution and dynamic mechanisms, as well as the support for a wide enough variety of devices, makes it necessary to establish abstractions to facilitate programmability. Although C++ operations are high-level, many of its code regions still suffer from verbosity and potential sources of error. This need is increased when efficient management interfaces have to be built on top of technologies with such expressiveness. For this reason, with oneAPI it is also necessary to provide mechanisms for programmers to exploit the heterogeneous system without having to interfere with all the operations needed to efficiently leverage all devices.

Challenge 3: Extensibility, scope and interoperability. The level of abstraction is given by both C++ and SYCL, but the definition of the applications and their interactions does not involve an abstraction as high as that achieved with EngineCL. In this case, SYCL is tied to a modern C++ programming style, so this association should not be decoupled in the abstraction being built, as it limits its adoption. Interoperability between C++, SYCL and the abstraction features of the runtime reduce the efforts of transforming and incorporating SYCL-compatible applications and libraries. Furthermore, the concept of program domain should be approached differently, since expert C++ programmers must be able to comfortably alter the code as if it were SYCL/oneAPI. Herein, it is important to only abstract the runtime and its management operations, while still keeping the scope of the problem directly visible to the programmers. In this way, they will feel under the same philosophy of SYCL, being able to compose operations and extend the behavior of the runtime without penalizing its performance.

Challenge 4: Adaptability. In the span of a year, oneAPI has evolved a lot, providing new features and extensions, deprecating functionalities and modifying the behavior of its DPC++ compiler. Experimental features may end up being included in the SYCL standard, as it happened before, so it is important to have them in order to exploit new capabilities. This standardization process allows other implementations to end up adapting this type of proposals, and it is a matter of time to have compilers with support for all types of manufacturers and devices. Thus, it is necessary that the proposal is able to incorporate experimental features and extensions from manufacturers. For example, one of the most relevant extensions of oneAPI is the possibility of exploiting unified shared memory as an alternative to

SYCL Buffers, as indicated in Section 2.1.3. For this reason, the architecture and API should easily be extended in order to allow programmers to leverage the full potential without interfering with the rest of the design decisions and optimizations.

5.2 Coexecutor Runtime

Considering all the previous reasons and challenges, *Coexecutor Runtime* addresses them by designing a proposal with a clearly distinct foundation from *EngineCL*. The latter focuses on compatibility and usability, uses the OpenCL framework and language, encapsulates the problem domain with a very high layer of abstraction and provides an integral solution to accelerate applications. However, with Coexecutor Runtime, the abstraction and API is higher than oneAPI, but fully compatible with C++ and SYCL. This runtime is being part of the problem domain, combining its ease of management aspects with the features provided by oneAPI. That is, programmers are aware of the fundamental parts of the process of co-execution, data dispatch and computation. This design decision relieves them from tedious tasks while allowing them to extend the lower level behaviors. Furthermore, its architecture has been designed with adaptability to changes in mind. For this reason, oneAPI extensions have been incorporated, extending the API so that the programmer can exploit these functionalities.

Coexecutor Runtime is based on the DPC++ compiler and runtime, hereafter referred to as oneAPI for simplicity. It is built on top of oneAPI as a runtime library, providing with this approach several architectural and adaptive advantages. Firstly, the design and implementation are based on open standards, both C++ and SYCL, following easily recognizable architectural patterns. Hence, any C++/SYCL programmer could extend its software architecture for their own purposes. Secondly, since it is drawing on previous standards such as OpenCL, it facilitates the adaptation for a whole repertoire of libraries and software generated over a decade, helping to benefit from co-execution. Thirdly, it serves as a skeleton upon which to apply different strategies and workload balancing algorithms for using oneAPI and SYCL. Finally, as it is designed from a sufficiently standardized and abstract approach, it allows the adaptation and extension to execution technologies and proposals created by other manufacturers, both compilers and accelerator drivers.

Therefore, in order that the runtime provides co-execution it is necessary the correct detection of a potential concurrent execution path by the oneAPI compiler and runtime. This materializes a parallel execution of several tasks of the DAG, thanks to the existence of totally independent hardware resources. In this way, the proposal is flexible enough to adapt to a variable number of computation entities, while simple enough to assist the compiler in the detection, favoring the creation of totally independent nodes recognizable by the runtime.

It is important to highlight that the architecture designed to provide the best efficiency in operations and to enable the exploitation of purely dynamic algorithms is based on a multi-

thread architecture with asynchronous mechanisms. Synchronous co-execution is the basis on which dynamic co-execution and the final architecture are built. This idea is generalized and provided with mechanisms to synchronize the launching of work packages, a system of notifications and events, or to enable more sophisticated schedulers, among others. Likewise, the synchronous proposal serves as a method to guarantee static co-execution in implementations that may offer less favorable behavior in the face of dynamic algorithms. Moreover, SYCL does not guarantee even static co-execution, so this synchronous mechanism, even less sophisticated and versatile than the dynamic and asynchronous one, is useful for other more constrained heterogeneous systems and SYCL implementations.

Finally, the three balancing algorithms presented in Section 2.2 are implemented, as performed with EngineCL. Hence, they are adapted to this architecture in order to provide the best possible efficiency and verify the technological trade-offs.

5.2.1 Synchronous static co-execution

The SYCL standard does not determine the behavior in the face of different computational regions used by independent devices, but the DPC++ implementation of the standard is not able to guarantee simultaneous execution. The main problem arises in the detection of disjointed memory regions when the same data structures are used by many oneAPI scopes. This problem occurs both when using independent or shared kernel execution regions, even though the programmer is able to recognize the independence between the execution and data spaces. For this reason, it is necessary to provide an architecture that facilitates the recognition and management of the system devices, as well as their transfer and computation regions, part of the oneAPI command queues and scopes. The main conceptual idea is to provide with a multi-threaded architecture that isolates every oneAPI scope, and therefore, each device used in the computation. This allows the underlying compiler and DPC++ runtime to recognize the disjoint spaces and be able to perform operations simultaneously. Since this promotes a static scheduling and workload distribution approach, it is necessary to establish a runtime layer that is as light as possible, reducing the management overhead, because it only schedules one work package per device.

The synchronous co-execution mechanism in the *Coexecutor Runtime* ensures that there is simultaneous execution among the devices, while reducing runtime management operations. However, this approach is limited in terms of implementing more sophisticated adaptive algorithms. For this reason, the synchronous mechanism focuses on static approaches. Nevertheless, since there is a part of the architecture that benefits from parallel operations in conjunction with the DPC++ runtime, an *asynchronous pattern* is developed internally to isolate each scope of oneAPI. Therefore, it provides the foundations for more advances scheduling strategies. C++ futures and its asynchronous mechanisms facilitate an acceptable degree of usability without the need to complicate the management code to solve co-

execution problems. This solution favors independency of regions and captures in a lambda region that makes the offload to another device asynchronously. It takes advantage of the primitives and variables previously initialized, reducing time and favoring reuse, thanks to being included in a bigger scope, that is, the parent scope of the lambda function. This is a key aspect if the programmer is using USM, since it allows reusing the memory regions directly, giving direct access to the original queue at any time.

Summing up, this mode of co-execution is limited to using static, oracle-style algorithms. Although internally it uses an asynchronous pattern, the programmer uses it synchronously, limiting the extensibility and the scheduling behaviors. The main advantage of this co-execution mode is that the solution is lightweight, in addition to guaranteeing better compatibility, making it suitable for limited environments.

5.2.2 Asynchronous dynamic co-execution

The dynamic co-execution is based on a generalization of the asynchronous pattern presented in the previous section. As it was shown in Chapter 3 with EngineCL, high efficiencies have been achieved by using strategies based on event chaining and multithreaded architectures with mixed management. That is, notifications based on callbacks and using workload management with standard C++ threads [93, 126, 128]. The main problem when trying to extrapolate these strategies, based on events, futures and C++ asynchronicity, is the limitation of expressiveness in the iterative distribution of workloads, complicating and preventing dynamic strategies and disabling all usability. Therefore, custom notification mechanisms have been developed and integrated as part of the runtime architecture, leaving those provided by the technology. This improves the resulting extensibility of the runtime, leveraging its effective compatibility. Thanks to this approach, it allows operating with architectures incompatible with callbacks, as is the case of FPGAs [125].

The strategy proposed for dynamic co-execution is to promote multithreaded management architectures based on the runtime of oneAPI. The *Coexecutor Runtime* enhances the isolation between devices, since one of the key points is to make it easier for the compiler to detect disjoint memory structures as well as the independence between queues and tasks. In addition, since oneAPI offers a sufficiently sophisticated and complete memory model, the management architecture must be adapted to favor both buffer management and the possibility of exploiting USM.

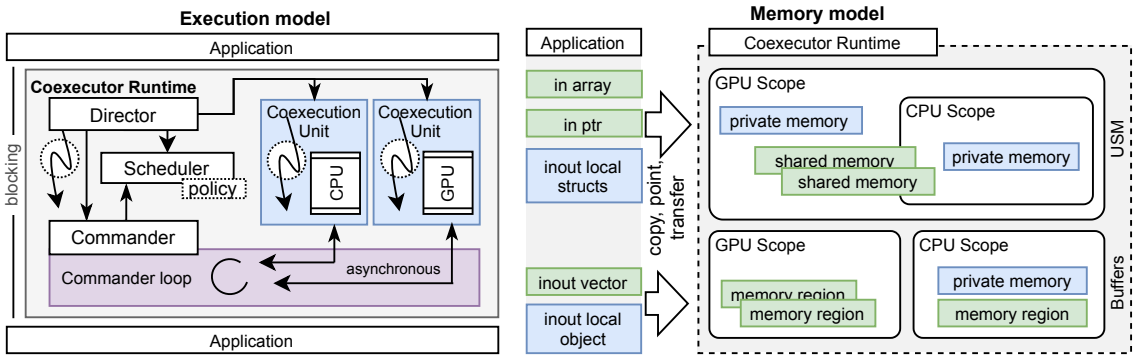
To define the proposal, three perspectives are considered, the execution model, from the memory point of view and the last one, the relationship of the *Coexecutor Runtime* with the runtime of oneAPI, as it is explained in the following sections.

5.2.2.1 Execution model

The *execution model* is shown in Figure 5-1(a), representing the interaction of the runtime as part of the execution process of an application. Execution is blocked from an application point of view, although internally it works asynchronously. In this way, the programmer only has to wait to finish the computation taking advantage of all its devices. However, it has the possibility to continue extending operations to be run on devices from the application side. It is done since the task graph is managed by the runtime of oneAPI. The *Coexecutor Runtime* is in charge of the creation and control of management threads (curved arrows), which will be part of the operation mechanisms of the co-execution architecture until the control is returned to the application. There is a thread belonging to the main manager, with its core component called *Director*. In addition, there are lightweight management threads, termed *Coexecution Units*, requiring one per computing device.

The *Director* configures the *Coexecution Units* and manages both the *Commander* and its communication with the rest of the entities. The *Scheduler* is instantiated and plugged in with a policy established by the programmer, using one of the schedulers explained in Section 5.2.3. The *Commander* is responsible of packaging the work, emitting tasks and receiving events, as part of the computation workflow with the *Coexecution Units*. This process is termed as *Commander loop*, and it follows the scheduling strategy defined by the *Scheduler*.

Regarding the *Coexecutor Runtime* internal workflow, the *Director* instantiates and configures oneAPI primitives and structures necessary both for the operation with oneAPI runtime and used by the *Scheduler* itself. Among these are work and queue entities, execution contexts and mapping of memory structures between the application and the runtime. These oneAPI structures and instantiations are shared by the components of the *Co-execution Runtime*, favoring the reuse and detection of data types by the oneAPI runtime. In parallel, the management threads of the *Coexecution Units* initialize the communication mechanisms within the runtime, as well as the request of devices and their configuration with oneAPI.



(a) Execution model as part of a blocking section of an application.

(b) Memory model example for USM and SYCL buffers when using CPU and GPU.

Figure 5-1: Coexecutor Runtime considering an example of CPU-GPU dynamic co-execution.

The communication is bidirectional between *Commander* and each *Coexecution Unit*, since it is co-executed with an independent scheduler that handles the decisions. As soon as there is a *Coexecution Unit* ready to receive work and the management thread has finished the initial phase, it establishes communication with the *Commander loop*. As the rest of the devices are completing their initialization, they incorporate into the loop, where the scheduling phase starts.

It is important to note that in multi-core CPUs, where oversubscription has a significant impact, it might be convenient to disable the *Director* management thread via its configurable behavior, and merge its management as part of the CPU Coexecution Unit. It is a compromise in terms of oversubscription overhead and runtime acceleration when overlapping Commander tasks with the computation or communication of Coexecution Units.

5.2.2.2 Memory model

The *memory model* is presented in Figure 5-1(b). It shows the separation between structures and memory containers, taking into account the two types of strategies used: USM or buffers of SYCL, although the *Coexecutor Runtime* supports the combination of both during the co-execution. On the left side are shown the structures, C++ containers and memory pointers used by the application, while the right side outlines the view of the runtime. The *Director* and its *Coexecution Units* handle the allocations and configuration of the memory space with oneAPI, and the programmer only has to request the use. The runtime will distribute them in the oneAPI memory model, either by transferring pointers, copying memory regions or sharing unified memory blocks.

Since the co-execution proposal is designed and evaluated to run on host and accelerator devices, two operating modes of the runtime are distinguished regarding OneAPI memory environments. If USM is used, the *Coexecutor Runtime* provides two scopes: a larger one for a device (GPU) and a smaller for another (CPU). This way, the memory spaces initialized by the GPU are reused in the CPU using oneAPI primitives. On the other hand, if SYCL buffers are used, the scope of each device will manage independent buffers with memory regions that will be part of a higher container or structure. Therefore, favoring the recognition of disjointed data spaces by the compiler. Private memory allocations can be made in both memory models, in the form of buffers and variables, where each field is controlled independently by each *Coexecution Unit* and its oneAPI scopes.

Finally, both ways of operating can be combined, since could be regions of the kernel that use the USM model and others that rely on buffers and variables. *Coexecutor Runtime* will reuse the scope of each device to map any C++ containers and memory regions, each of which will be governed by a memory model.

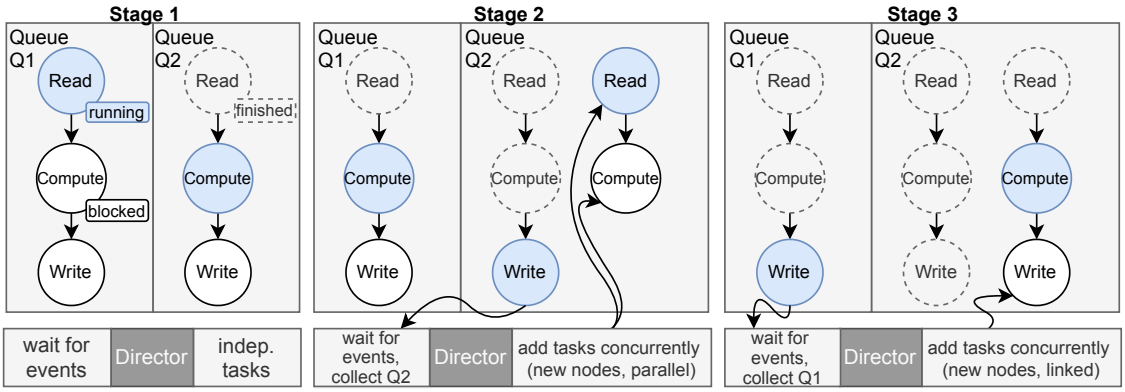


Figure 5-2: Example of interaction with the DAG from oneAPI's perspective while running a dynamic approach with two queues.

5.2.2.3 Runtimes interaction

The interaction between the *Coexecutor Runtime* and oneAPI is shown in Figure 5-2. Three stages are presented during the execution of the runtime, with two different queues Q1 and Q2. It starts from a situation where the runtime has established two independent parallel execution queues, due to the existence of two separate underlying architectures. The nodes of each queue are managed by the runtime through the DAG, and they can be in three different states: execution (blue), blocked waiting for resources (white) or finished (gray with a dashed line). The *Director* waits for events related to the DAG or performs independent tasks, such as resource management, receiving and sending notifications, status control or work reparation, some of which are essential within the *Scheduler*.

By switching to the stage 2, it can be distinguished how the Q2 is able to process nodes more efficiently, so the *Director* collects results of the write operation and enqueues new nodes of the DAG to the same queue, overlapping computation and communication. Collection operations are dependent on the memory model, the type of operations (explicit or implicit) and the amount of bytes used, thus they could be fast, as in unified memory, or slow, when using mixed models or while transferring large blocks. Finally, in the stage 3, the end of the Q1 is represented with the output data collection while in the Q2 a next writing task is added. This is linked to the branch created in stage 2, as soon as its computation task has started, distributing the DAG management among different time periods.

5.2.3 Load balancing algorithms

Coexecutor Runtime implements two kind of algorithms, static and dynamic, thus it is necessary to provide a runtime flexible enough to accommodate such modes of operation in an efficient manner. The experiments launched with EngineCL have shown the predominance of HGuided in most experiments. However, a novel, much more abstracted underlying technology is now being studied, with extended features and a constant evolution. For this reason, it is necessary to implement and validate the diverse load balancing algorithms

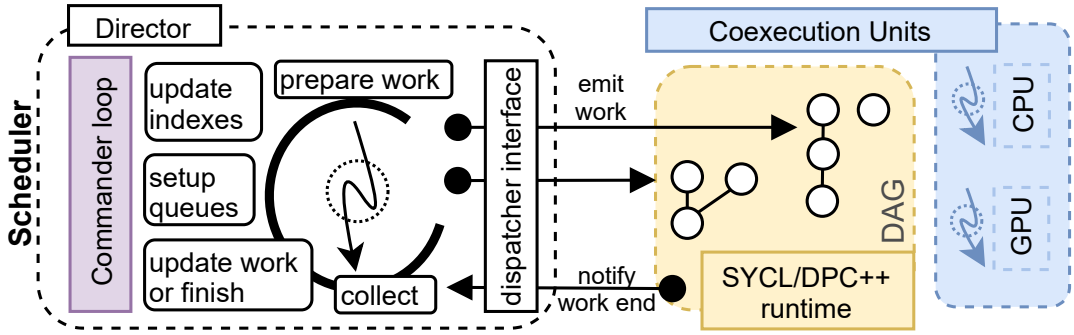


Figure 5-3: Commander's loop where the scheduling strategy is performed to coordinate the behaviors of the Coexecution Units.

presented in Section 2.2, analyzing and determining the behavior of *Coexecutor Runtime* and the underlying technology.

In order to generalize the architecture, the two previous approaches, synchronous static and asynchronous dynamic co-execution designs, are unified in favor of the latter. As no overheads have been found due to the additional management in the dynamic strategy with respect to the static one when using Intel oneAPI, it is decided to implement the static approach as a dynamic algorithm with just one package per device. Nevertheless, the pure static co-execution approach could be worthwhile in other architectures, less powerful host devices or SYCL implementations that offer disadvantages when exploiting decisions dynamically. This is because it requires fewer management structures, freeing up CPU and memory resources for proper load distribution.

In addition to the advantages described in the Section 5.2.2, the *Coexecutor Runtime* architecture offers an efficient scheduling module (*Scheduler*) that allows the dynamic strategies to be exploited easily, with a common internal scheduling interface and offering negligible synchronization overheads. However, it increases the internal complexity of the runtime, albeit it is hidden to the programmer, who receives a simple and straightforward API to operate with. Nevertheless, the dynamic co-execution design is sophisticated enough to use any static co-execution algorithm with no efficiency penalties, as it is shown in Section 5.5.

To enable dynamic policies to squeeze all the computing capacity out of the heterogeneous system, the *Scheduler* component is introduced, as it is shown in Figure 5-1(a). It configures the behavior of the load balancer, the distribution and division of the work packages (amount and region of data to be computed by each device), as well as the way to communicate with the different execution devices.

Figure 5-3 depicts the relationship of the *Coexecutor Runtime* with the runtime of oneAPI, all of it involved as part of the *Commander loop*. The *Coexecutor Runtime* internal communication is performed between the management threads, either those associated with the devices (right) or the global manager, usually associated with the *Director* (left). This view simplifies the runtime of oneAPI and its internal DAG management, being considered as a

single entity, part of the *Coexecution Units* (right). The *Director* performs a set of periodic actions, as a loop managing events and operations, among which are:

- Preparing and packing the next job to be issued.
- Collecting completed and updating pending jobs.
- Managing the end of a work block and its completion.
- Preparing and reusing the queue and command groups as well as other oneAPI primitives.
- Updating the indexes, ranges and offsets of memory entities, as well as variables and containers.

Every time a work package is prepared, the runtime adds a task in the DAG. Similarly, with the completion of a job, the *Commander* receives the notification to collect and merge the output data, if needed. This operation can be lightweight in case of using USM or using implicit operations, delegating more responsibility to oneAPI. The emission and reception of work is requested through a dispatch interface, as a way of unifying requests. Finally, when there are no more pending jobs, the *Commander* will notify the *Director* to close and destroy the primitives and management objects to return control to the application.

Based on the proposed architecture and its designed dynamic co-execution model, the three algorithms described in Section 2.2 of Chapter 2 are implemented in the *Scheduler*. The *Static* algorithm has a minimum management inside the *Scheduler* component, because it only runs as many iterations in the loop of events as devices are co-executing. Regarding the strictly dynamic strategies inside the *Scheduler*, it is not possible to know in advance the quantity of iterations, because it will depend on each execution parameters, as well as the number and type of devices. These operations increase the management overhead due to the operations related to the update of indexes and ranges, as well as the division of the problem into independent regions. Finally, concerning the differences in the operations carried out by *Commander*, *Dynamic* will simplify the number of instructions involved in the calculation of work packages compared to *HGuided*. This is explained since the latter performs a more sophisticated algorithm that takes into account certain conditions, including the computing power of each device. However, the calculation overheads of the latter are compensated by the efficiency of its workload distribution policy.

5.3 API Design

The *Coexecutor Runtime* has been designed to offer an API that is flexible as well as closely linked to the SYCL standard, favoring reuse of existing code and a slightly higher usability. Furthermore, it offers two modes of computation from the point of view of the programmers. The results of architectural and design decisions concerning expressiveness and usability enhancement are materialized in both modes, providing distinct usage facilities. The same SAXPY kernel is computed using both modes to compare its differences. Both code

```
1  coexecutor_runtime<hg> runtime;
2  runtime.config(CounitSet::CpuGpu, coexecutor_runtime::dist(0.35));
3  runtime.launch(data.size(), [&](coexecutor_unit *counit, package pkg) {
4    sycl::buffer<int, 1> buf_input(data.data() + pkg.offset,
5                                   sycl::range<1>(pkg.size));
6    counit->dispatch([&](sycl::handler &h) {
7      auto R = sycl::range<1>(pkg.size);
8      auto input = buf_input.get_access<sycl::access::mode::read_write>(h);
9      h.parallel_for(R, [=](sycl::item<1> it) {
10       auto tid = it.get_linear_id();
11       input[tid] = input[tid] * datav;
12     });
13   });
14 });
```

Listing 4: Coexecutor Runtime computing SAXPY with a dynamic algorithm using simultaneously CPU and GPU.

snippets show the runtime usage from the perspective of the programmer, thus omitting the initialization of the problem and its data, as well as the subsequent use of the results.

The simple mode it is shown in Listing 4, where an explicit embedded context (lambda function) is used to perform the computation in a few lines of code. It shows an example of use when computing the SAXPY problem simultaneously exploiting two different devices, being in this case the CPU and GPU. Line 1 instantiates the `coexecutor_runtime` prepared to compute a program using the HGuided balancing algorithm. In the next line, it is configured to use both the CPU and GPU, giving a hint of the computational power of 35% for the CPU compared to the GPU. This value will leverage the algorithm to further exploit co-execution efficiency. Next, the co-execution scope associated with the problem is provided (lines 3 to 14), where a lambda function captures the values used by reference. This scope is executed by each of the *Coexecution Units*, and therefore, they must establish independent memory reservations (or shared, if unified shared memory is exploited), using the values provided by the runtime itself through the `package` class. Line 6 opens an execution scope, associated to the kernel computation for each device. In lines 7 and 8 a read and write access is requested for the previous memory region (buffer *accessors*), indicating the execution space based on the given package size. Finally, lines 9 to 11 show the data-parallel execution, traversing the indicated execution space (`R`), using the `accessor` templates and variables needed (`datav`, `input`).

After executing the lines shown in Listing 4, the problem is computed simultaneously using both devices. The `launch` call blocks the program execution since it is a synchronous operation from the point of view of the programmer. Therefore, the data resulting from the computation is in the expected data structures and containers that the programmer uses in the C++ program (vector `input` in the example), without the need to create any new buffer exchange structures specific for the coexecution.

```
1  class SAXPY : public CommanderKernel {
2  public:
3   SAXPY(int *x_ptr, int *y_ptr, int *out_ptr, size_t data_l, float sc_fl)
4       : m_x_ptr(x_ptr), m_y_ptr(y_ptr), m_out_ptr(out_ptr),
5         m_sc(sc_fl), m_data_l(data_l){}
6
7   // pre-setup config: shared internally between Coexecution Units
8   void init(coexecutor_unit *counit){  // Director Scope
9     counit->add_buffer<int, 1>(0, m_x_ptr, sycl::range<1>(m_data_l));
10  }
11  // void init_completed() {} /* @callback */
12  program_size size() { return m_data_l; }
13  void compute(coexecutor_unit *counit, package pkg){
14    std::cout « "[" « counit->id() « "] computing...\n";
15    sycl::buffer<int, 1> buf_x =
16        *counit->get_buffer<int, 1>(0); // use all buffer
17    sycl::buffer<int, 1> buf_y(m_y_ptr+pkg.offset, sycl::range<1>(pkg.size));
18    sycl::buffer<int, 1> buf_out(m_out_ptr+pkg.offset,
19                                 sycl::range<1>(pkg.size));
20
21    // communicate with the Commander to dispatch a transaction
22    counit->dispatch([&](sycl::handler &h){ // Director-CoExecUnit comm.
23      auto R = sycl::range<1>(pkg.size);
24      auto x = buf_x.get_access<sycl::access::mode::read>(h);
25      auto y = buf_y.get_access<sycl::access::mode::read>(h);
26      auto out = buf_out.get_access<sycl::access::mode::discard_write>(h);
27      auto sc = (int)m_sc; // caching
28      h.parallel_for(R, [=](sycl::item<1> it){
29        auto tid = it.get_linear_id();
30        out[tid] = x[tid] * sc + y[tid];
31      });
32    });
33  }
34  /* @callback */
35  void compute_completed(coexecutor_unit *counit, package pkg){
36    std::cout « "[" « counit->id() « "] package " « pkg.id
37              « " computed with throughput " « pkg.throughput « "\n";
38    counit->dump_statistics(); // Stats per package - CoExecUnit related
39  }
40 private: // organized data:
41  int *m_x_ptr; int *m_y_ptr; int *m_out_ptr; size_t m_data_l; float m_sc;
42 };
```

Listing 5: SAXPY program definition using the `CommanderKernel` interface provided by Coexecutor Runtime to implement the kernel behavior as an independent unit.

On the other side, the extended computation mode offers more flexibility to the runtime and the kernel computation. It allows accessing extended methods and callback functions, based on the `CommanderKernel` interface provided by *Coexecutor Runtime*, as it is implemented in Listing 5 and instantiated and executed in Listing 6. The advantages of this computation mode are encapsulation of the program, increasing the maintainability, along with enhanced flexibility regarding runtime operations. Some of them allow the programmer to share buffers between devices (lines 8-10) and perform custom operations at specific

```
1  auto N = pow(10,8); std::vector<int> x; int* y; std::vector<int> out(N);
2
3  coexecutor_runtime<dyn> runtime;
4  runtime.config(CounitSet::CpuGpu, 128); // dynamic; 128 packages
5
6  x.assign(N, 1);
7  y = runtime.alloc<int>(N);
8  // more assigns: foreach, assign x[i], y[i] ...
9
10 SAXPY program{x.data(), y, out.data(), N, 3.14159};
11 runtime.run(program);
12 // from here on, we have the data in our C++ containers
13
14 runtime.free<int>(y);
```

Listing 6: Coexecutor Runtime using the CPU and GPU simultaneously to compute the SAXPY kernel definition of the Listing 5. This example shows the exploitation of the extended computation mode to enhance the flexibility of the runtime.

places during the *Coexecutor Runtime* execution process, such as, during the initialization and completion of the *Director* scope (lines 8 and 11), during the initialization or completion of the *Coexecution Units* or at the end of every work package completed (lines 36-40). The drawbacks are increased complexity and verbosity compared with the simple computation mode, but the latter is only recommended for smaller and easier kernel algorithms. Moreover, in this example the SAXPY computation is performed using a dynamic load balancing algorithm (`dyn`), splitting the workload in 128 packages, scheduled at runtime among the devices (lines 3-4 of Listing 6). Finally, *Coexecutor Runtime* allows using oneAPI features and extensions, integrated as part of the architecture of the runtime, presented in Section 5.2, or exposed via API calls to the programmer, such as using unified shared memory by calling the `alloc` and `free` methods of the runtime (shown in lines 7 and 14 of Listing 6). Taking into account the runtime design principles as outlined in these two examples, the *Coexecutor Runtime* hides all the implementation details, easing the use of its co-execution capabilities to exploit easily any oneAPI program.

Since the philosophy of Coexecutor Runtime is closer to oneAPI and its SYCL API, the level of abstraction is not so high as it is EngineCL compared with OpenCL. Here, it is designed to be located in the scope of the program to be computed, favoring composition and flexibility for all types of applications. Even so, more than 500 lines of SYCL code would be needed to implement the specific behavior seen in Listing 4, computing SAXPY, one of the simplest programs, and excluding other features such as support for shared memory, statistics and execution information, an optimized architecture, a dynamic scheduling system or the automatic incorporation of new devices, among others. Moreover, the more complex the problem to be computed or the more different devices are used, the easier it is to use CoexecutorRuntime with respect to a SYCL-based solution. This is because the *kernel* is a

region of code practically invariant, since it encapsulates the application behavior, but all the memory and execution operations, as well as devices, runtime and algorithmic management will require considerable non-scalable programming efforts.

5.4 Methodology

The experiments to validate the *Coexecutor Runtime [242, 392]*[1] have been carried out in two nodes, labelled *Desktop* and *DevCloud*, defined in Section 1.7.1.

To accomplish the validation, 6 benchmarks have been selected, which represent both regular and irregular behavior. These are Gaussian, MatMul, Taylor, Ray, Rap and Mandelbrot, all detailed in Section 1.7.2. Additionally, *NBody* is included as a special case, since it exposes a peculiar dynamic behavior to be exposed in Section 5.5.4. Table 5-1 presents the problem sizes and number of work-items launched to verify the behavior of the runtime, its co-execution capabilities and the validations performed with the chosen benchmarks. The property *data containers* refers to the C++structures and *buffers* (not to be confused with SYCL *Buffers*) that must be transferred to the devices, regardless of the memory transfer and abstraction strategy chosen.

The validation of the proposal is done by analyzing the co-execution when using four scheduling configurations in the heterogeneous system. As it is designed and summarized in Section 5.2.3, but highly detailed in Section 2.2, Static, Dynamic and HGuided algorithms are evaluated, labelled as St, Dyn and Hg, respectively. In addition, the dynamic scheduler is configured to run with 5 and 200 packages. Finally, the two different memory models presented in Section 2.1.3 have also been tested: Unified Shared Memory (USM) and SYCL Buffers (Buffers).

To guarantee integrity of the results, 50 executions are performed per case with an initial execution discarded to avoid warm-up penalties. Since the GPU is on-chip and there are hardware policies regarding frequency throttling due to temperature thresholds, some decisions have been applied to stabilize results. First, CPU turboboost is disabled and CPU governor is set to performance. CPU and GPU are set to 2400Mhz and 600Mhz fixed frequencies, respectively. Finally, every execution starts when the CPU socket temperature is under 38º. In DevCloud, where some of the previous conditions could not be applied, waiting times have been introduced between executions, increasing the duration of the experiments but stabilizing their measurements and results. The standard deviation is not shown because it is negligible in all cases.

To evaluate the performance of the *Co-executor Runtime* and its load balancing algorithms, the total response time, as well as the response time of each of the devices, are measured, including kernel computing and data transfer. Then, as it is exposed in Section 1.7.3, three metrics are calculated: balancing efficiency, speedup and heterogeneous efficiency. The

[1] https://github.com/oneAPI-scheduling/CoexecutorRuntime

Table 5-1: Memory usage and execution ranges for the benchmarks used to validate Coexecutor Runtime.

Property	Gaussian	MatMul	Taylor	Ray	Rap	Mandel	NBody
Read:Write data containers	2:1	2:1	3:2	1:1	2:1	0:1	2:2
Work-items ($N \times 10^5$)	262	237	10	94	5	703	4
Mem. usage (MiB)	195	264	46	35	6	1072	26

baseline is performed with respect to a pure oneAPI solution using its *host-device* programming model. Hence, *Coexecutor Runtime* is evaluated against the fastest device, which is the GPU for all the cases studied.

Furthermore, a scalability analysis of the co-execution with respect to the problem size is presented. To do this, the total execution time of the heterogeneous system using co-execution and of each of the individual devices has been measured, increasing the size of the problems.

Finally, two metrics have been used to assess the behavior of co-execution with respect to energy. The energy consumption of the whole system is measured using RALP counters via perf, providing measurements in Joules. On the other hand, the energy efficiency has been calculated using the Energy-Delay Product (EDP) metric, defined in Section 1.7.3. Since the values obtained have a very wide range, this metric is computed as a ratio with respect to the EDP of the only-GPU execution, as it is shown in Equation 5-1:

$$EDP_{ratio} = \frac{EDP_{GPU}}{EDP_{co-execution}} \tag{5-1}$$

5.5 Validation

This section presents the experimental results carried out to evaluate performance and energy of the Coexecutor Runtime, performing co-execution with three scheduling policies. Also, a scalability analysis is performed for CPU, GPU and the best scheduling algorithm. Finally, a special case is detailed when using dynamic approaches for the NBody benchmark.

5.5.1 Performance

The values measured in the experiments for the two different architectures evaluated, Desktop and DevCloud, are shown in Figures 5-4, 5-5 and 5-6, respectively. They are the balancing efficiency, speedup and heterogeneous efficiency achieved by co-execution in the CPU-GPU system, with respect to the only-GPU execution.

The abscissa axes show the benchmarks, each one with four scheduling policies and two

memory models, as defined in Section 5.4. Moreover, the geometric mean for each scheduling policy is shown on the right side (average).

The main conclusion that is important to highlight is that co-execution is always profitable from a performance point of view, as long as it is done with dynamic schedulers, and even more if using unified shared memory (*USM*, explained in Section 2.1.3), as the geometric mean summarizes for these benchmarks and scheduling configurations.

Regarding balancing efficiency, the optimal is 1.0, where both devices finish simultaneously without idle times. Any deviation from that value means more time to complete for one device compared with the other. Generally, the imbalance is below 1.0 due to the overheads introduced by the CPU because it has to process part of the workload as a device, but also to manage the *Coexecutor Runtime*, as the host. In Desktop, it rarely completes its computation workload before the GPU finishes, since the latter requires more resource management by the host, increasing the CPU load. As the number of cores in the system increases, the management cost of the runtime is reduced, benefiting the CPU and requiring more work than is allocated to it. This behavior can be appreciated when using the DevCloud node, with 12 logical cores, compared to 4 in Desktop.

Speedup allows assessing how much faster co-execution is compared to GPU-only execution, while heterogeneous efficiency helps understand how well the whole system is being utilized. Therefore, the latter metric allows comparing performance on both architectures, which cannot be done with speedup, as speedup in a heterogeneous system is always relative to the computational power of each device.

Analyzing the different load balancing algorithms, it can be seen that the Static offers the worst performance, even in regular applications where it should excel. This is because the initial communication overhead caused by sending a large work package, leads to a significant delay at the beginning of the execution, strongly penalizing the final performance. As can be seen later in the Section 5.5.3, even in a single, full kernel offload to the GPU, considerable CPU (host) management is required. Therefore, if CPU intensive is being performed without the possibility of alternating runtime management with kernel computation, as is the case with static co-execution, the whole system will generally be penalized. In theory, this algorithm should cause less communication and synchronization overhead. However, it fails to balance the workload properly, as can be seen in Figure 5-4, resulting in very low speedups, with averages below or equal to 1.0 in both architectures, which means that co-execution is not profitable.

Regarding dynamic algorithms, they provide good results in general, especially when the *USM* memory model is used. However, they have the drawback that the number of packages for each benchmark has to be carefully selected. A very small number of packages can lead higher imbalances causing a performance penalty, as can be seen in Gaussian, Mandelbrot or Ray, in the case of Dyn5. At the other extreme, a very large number of packages increases the communication overhead, impacting negatively on performance, as in Gaussian with

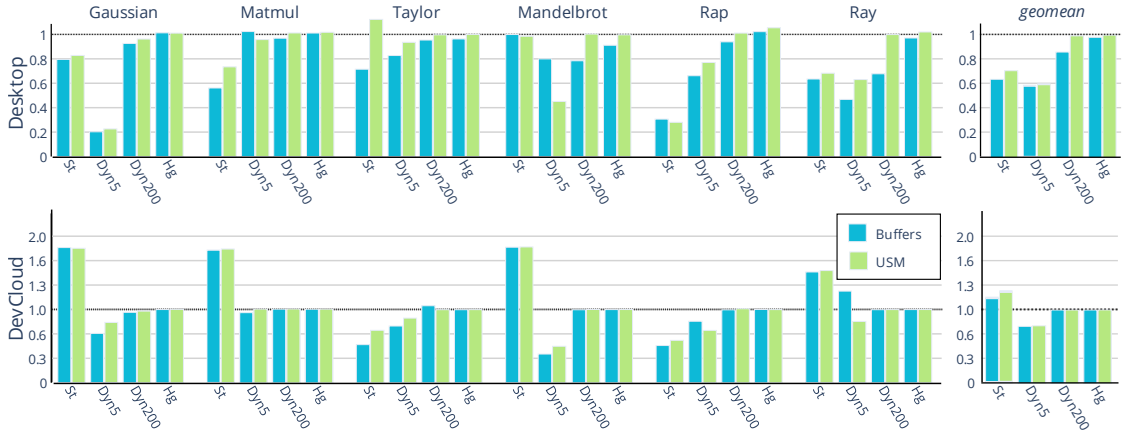


Figure 5-4: Balancing efficiency for a set of benchmarks when doing CPU-GPU co-execution in Desktop and DevCloud nodes.

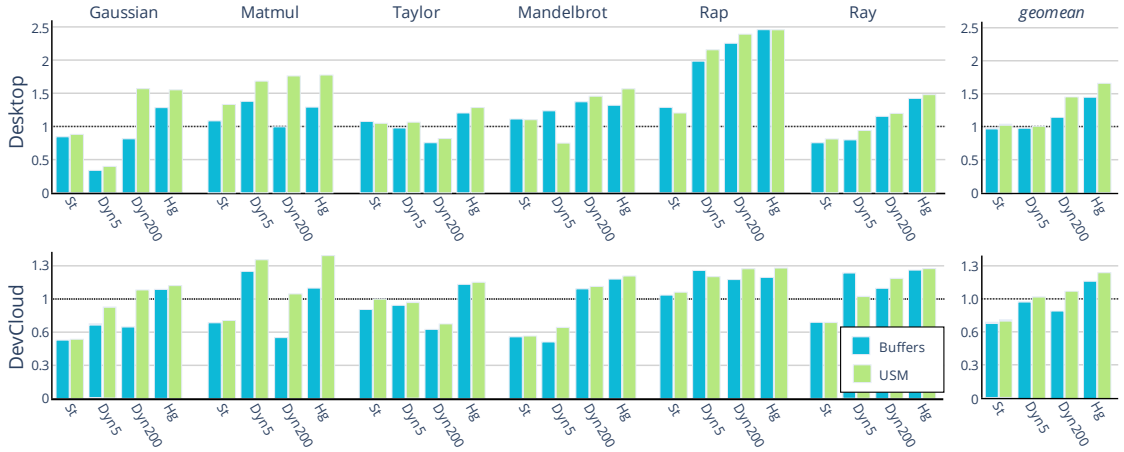


Figure 5-5: Speedups for a set of benchmarks when doing CPU-GPU co-execution in Desktop and DevCloud nodes.

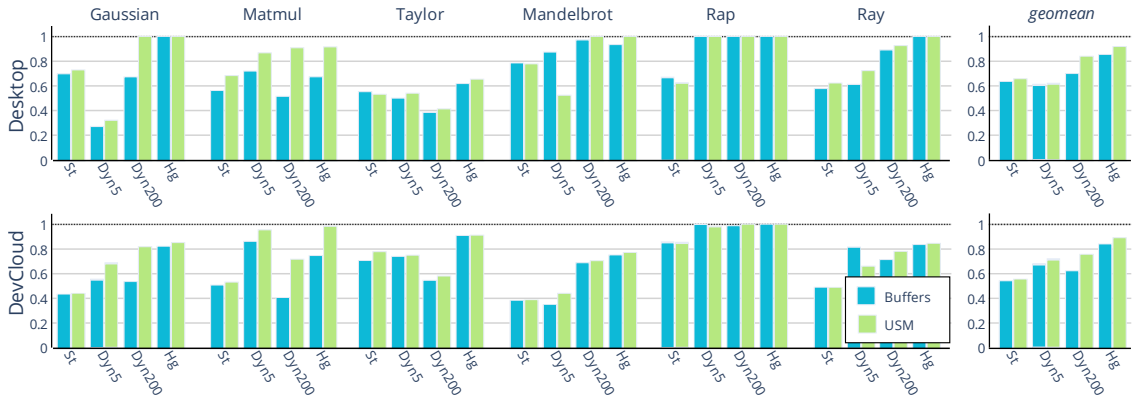


Figure 5-6: Efficiency for a set of benchmarks when doing CPU-GPU co-execution in Desktop and DevCloud nodes.

Buffers. In between, there is a tendency that the greater the number of packages, the better the balancing. Moreover, in such cases with good balancing efficiency, it achieves better performance, especially if USM is used. This is an expected behavior because the packages

are smaller and their computation is faster, giving less chance of imbalance in the completion of both devices. This is an interesting behavior since the *Coexecutor Runtime* is delivering high performance when using dynamic strategies due to the low overhead of the *Commander loop* when managing packages and events. This behavior is also found and detailed in the special case of NBody, explained later in Section 5.5.4.

The HGuided algorithm offers the best scheduling policy, both for regular and irregular kernels, and for the two architectures evaluated Desktop and DevCloud. This result is achieved thanks to a combination of two properties. On the one hand, its excellent balancing efficiency, that in average is very close to 1, as can be seen in Figure 5-4. On the other hand, the *Co-executor Runtime* does a great job in overlapping computation and communication, thus minimizing the impact of the synchronization and communication overhead inherent to dynamic algorithms. For example, HGuided scheduled, on average, 42 and 53 packages for Mandelbrot and Ray, while the best dynamic configuration (*Dyn200*) used 200 packages for both cases. Although the balancing efficiency is good in both scheduling configurations, the reduction of communications and synchronization mechanisms boosts the HGuided strategy compared with the Dynamic one. HGuided yields the best performance in all the analyzed benchmarks, with average speedup values of 1.65 and 1.26 in the Desktop and DevCloud architectures, respectively. Therefore, the co-execution is able to squeeze the maximum performance out of all the resources available in the system, such as the CPU and GPU shown in this validation, offering an efficiency of 0.92 on Desktop and 0.89 on DevCloud. Moreover, since it is a dynamic algorithm with high balancing efficiency, it does not require any a priori parameters, simplifying the programming effort.

These results show how the *Coexecutor Runtime* built on oneAPI technology is able to overlap computation and communications very well, which has two consequences. In the static algorithm, being limited by as many packets as devices, it cannot take advantage of any overlap, penalizing co-execution. On the other hand, in dynamic algorithms, it has a positive effect because the synchronization and communication overhead is greatly reduced, by overlapping them with the computation. Thus, there is a clear advantage of dynamic algorithms, which was not always the case with EngineCL and its OpenCL technology.

Considering the memory models, there is a general improvement in balancing and performance when using *USM* compared with *Buffers*. It can be observed than *USM* performs much better than *Buffers*, especially when dynamic strategies are being used and when large memory transfers interfere. This is highlighted in those benchmarks that manipulate multiple memory containers, either *Buffers* or *USM* regions, with large amounts of data, as it is detailed in Section 5.4. Splitting into many packages causes more management by the abstract memory containers, the SYCL *Buffers*, which is alleviated by using shared memory optimizations, such as in Gaussian, Matmul or Mandelbrot. However, the HGuided load balancer is so algorithmically efficient that it tends to alleviate the differences between memory container types.

Finally, it is important to note that there is generally a tendency for good balancing efficiency to be related with increased speedups. However, it is important to qualify this pattern, since the validation presents two different architectures and an interesting variety of benchmarks with distinct properties. For this reason, there are cases with imbalance issues, but the total work is completed in less time, because the device that receives more work executes faster. Thus, it achieves more throughput per packet, fewer synchronizations and communications, as well as less host management overheads, enhancing the total computation. This behavior is observed in MatMul, Ray and Taylor for dynamic algorithms, specially in configurations that include too many packages (*Dyn200*), where there is an increase in host-device communications. In such cases, the workload distribution and scheduling management can penalize performance, especially if Buffer-type structures are being used instead of shared memory optimizations (*USM*). This is also the main reason why HGuided excels since it solves these problems.

5.5.2 Scalability

This section presents a scalability analysis of system *Co-executor Runtime* with HGuided, the best load balancing algorithm determined in the previous section. To this aim, Figure 5-7 shows the evolution of the execution time of each benchmark with respect to the size of the problem, in different configurations: CPU-only, GPU-only and co-executing using HGuided scheduler.

The most important conclusion to be drawn is that, in all the cases studied, there is a turning point from which co-execution improves the performance of the fastest device. For very small problem sizes, the overhead introduced by the *Co-executor Runtime* cannot be compensated by the performance increase provided by the co-execution. These points are more noticeable in Gaussian, Mandelbrot, Ray and Taylor, because the differences in computing capacity between CPUs and GPUs are much more pronounced (13.5x, 4.8x, 4.6x and 3.2x, respectively).

Experiments have been done considering the memory models, but from a scalability and representation point of view, it is difficult to discern the differences between *USM* and *Buffers*. As reflected in Section 5.5.1, *USM* offers better efficiencies but both models show similar trends. For this reason, only *USM* is represented to clarify the representation of scalability and the impact of co-execution.

Matmul is a special case, since by increasing the size of the problem, a point is reached where co-execution obtains the same performance as the GPU-only. A detailed analysis of the hardware counters indicates how the LLC memory suffers constant invalidations between CPU and GPU. Temporary locality of the shared memory hierarchy is penalized when co-executing with very large matrices, because the GPU requests memory blocks aggressively. This does not occur in the other benchmarks, as there is no temporal locality. Since the data is only used once, there are no conflict misses.

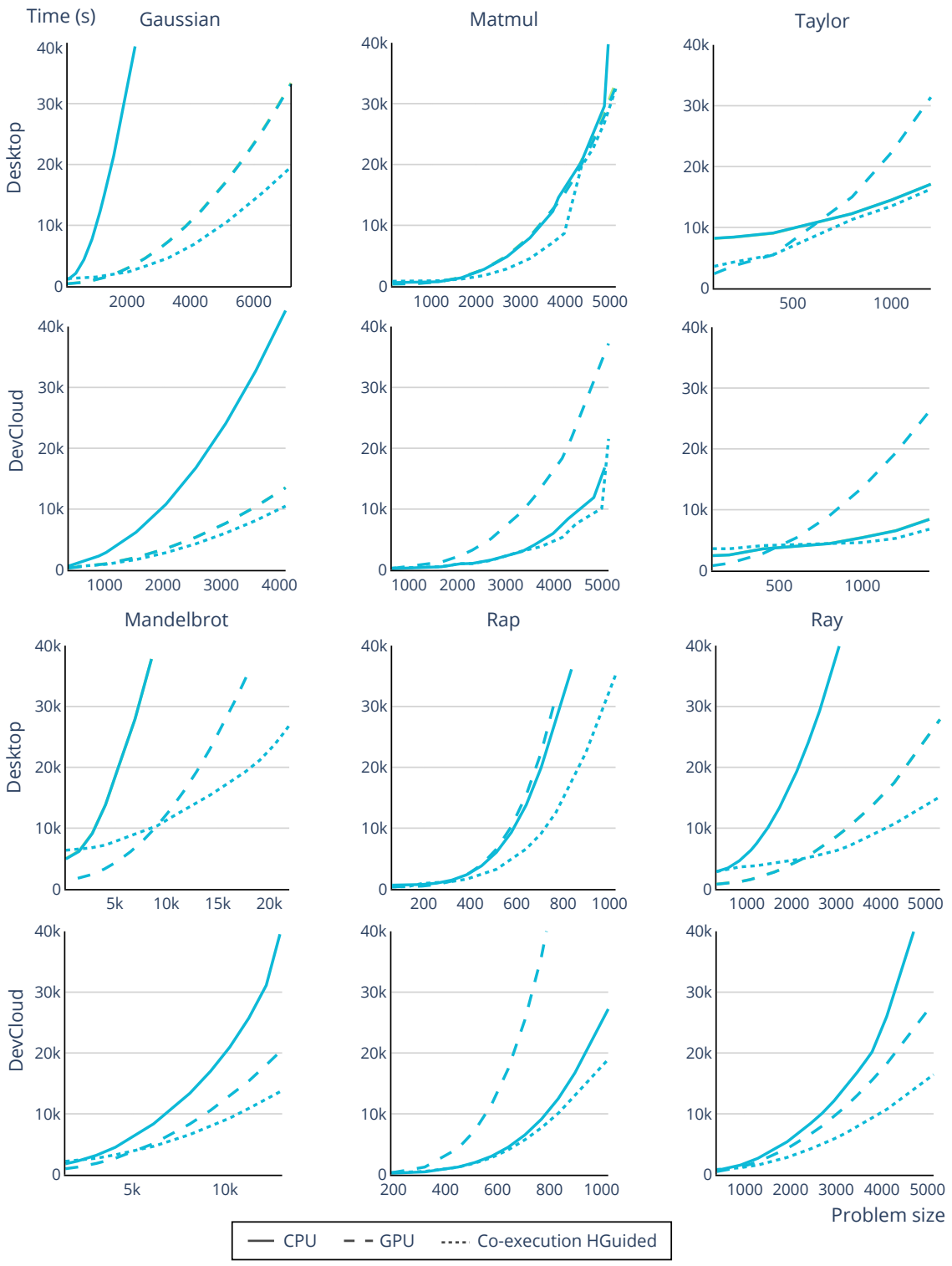


Figure 5-7: Scalability for CPU, GPU and CPU-GPU coexecution using the Coexecutor runtime with its HGuided scheduling policy and USM memory model for Desktop and DevCloud nodes.

5.5.3 Energy

This section presents the analysis of the energy consumption as well as the energy efficiency of the Desktop system, both when using only the CPU and the GPU, and with different

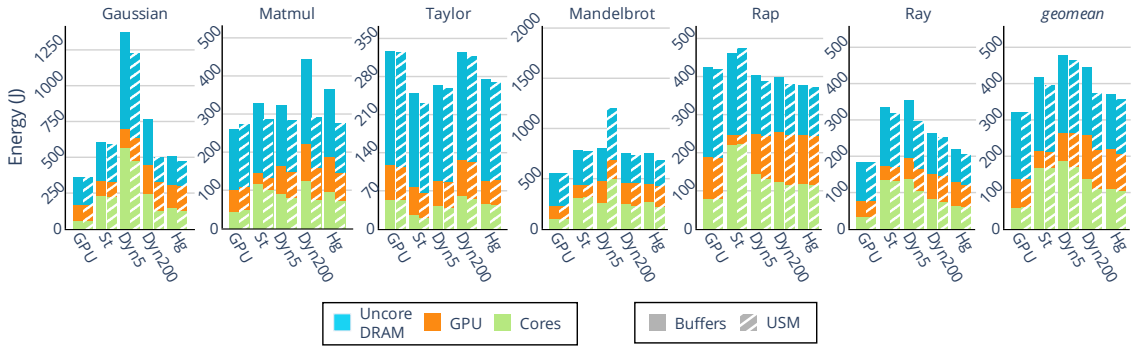


Figure 5-8: Energy consumption by cores, GPU and the other units of the package with the DRAM consumption for Desktop node.

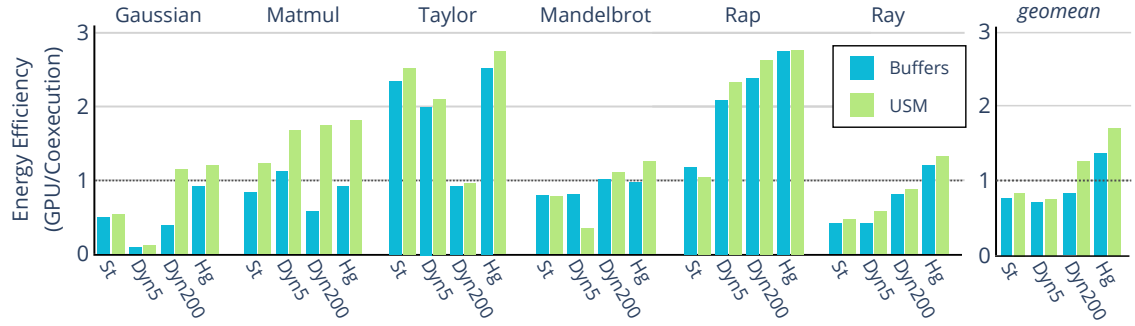


Figure 5-9: Energy Efficiency compared with GPU for Desktop node (more is better).

configurations of co-execution runtime. The DevCloud node does not offer any possibility to measure power consumption, such as RAPL counters, perf events or any Intel system tools. Therefore, this section focuses on analyzing the energy behavior of the Desktop node when using the oneAPI *Coexecutor Runtime*.

Figure 5-8 presents the energy consumption, with each bar composed of up to three regions representing the energy used by: the CPU cores, the GPU and the rest of the CPU package together with the DRAM (*uncore* + *dram*).

Considering the average energy consumption, using only the GPU is the safest option to ensure minimum energy consumption. This is because the energy savings achieved by the reduction in execution time thanks to co-execution, is not enough to counteract the increase in power consumption caused by the use of CPU cores. However, there are also benchmarks such as Taylor and Rap where co-executing does improve energy consumption over GPU, and others where co-execution and GPU-only have similar energy consumption, such as MatMul. This is because the computational powers between the two devices are considerably close, achieving very balanced distributions that result in efficient energy consumptions. Taylor has a computational power ratio of 0.44 : 0.56 (CPU and GPU), while Rap has a ratio of 0.39 : 0.61. The case of MatMul is particular as it is due to LLC sharing penalties, as it is explained in Section 5.5.2.

Regarding the schedulers, there is a clear correlation between performance and energy

consumption. Therefore, the algorithms that offer the best performance in co-execution are also the ones that consume the least energy. On the contrary, the schedulers that cause a lot of imbalance by giving more work to the CPU, spike the energy consumption, due to the higher usage of CPU cores, like Gaussian and Mandelbrot with Dyn5, and RAP with Static.

An interesting behavior can be observed in MatMul. The balancing efficiency using dynamic algorithms is very close to 1.0 in all cases. However, the performance obtained is very different using buffers than with USM. The explanation is found by analyzing the memory power consumption results shown in Figure 5-8. It can be seen that the memory power consumption (and therefore the memory usage) in the case of buffers is much higher than in the case of USM. The memory power consumption results correlate perfectly with the performance results. Also, Taylor presents a similar behavior.

Another very interesting metric is energy efficiency, which relates performance and energy consumption. In this case it is represented by the ratio of the Energy-Delay Product of the GPU with respect to the co-execution, presented in Figure 5-9. Therefore, values higher than 1.0 indicate that the co-execution is more energy efficient than the GPU.

Looking at the geometric mean, it can be concluded that co-execution is 72% more energy efficient than the GPU execution, using the HGuided scheduler and the USM memory model. Furthermore, this metric is indeed favorable to co-execution in all benchmarks studied, reaching improvements of up to 2.8x in Taylor and RAP. Thus, while co-execution consumes more energy in absolute terms on some benchmarks, the reduction in execution time compensates for this extra consumption, resulting in a better performance-energy trade-off.

5.5.4 NBody Benchmark

NBody presents an outstanding behavior, and a detailed study of the characteristics of the application, together with an evaluation of the throughputs, has highlighted the importance of dynamic strategies. Figure 5-10 shows the speedups of the *Coexecutor Runtime* with respect to the execution of a single package offloaded with the full problem size for a single device (*Single*). It depicts the results when using a single device, the CPU and GPU independently, for each supported memory model, when using different package distributions with the Dynamic scheduler. Each of the graphs shows 6 configurations of the scheduler, with 5, 50, 100, 400, 800 and 1200 packages of the same size, all dynamically scheduled. In addition, there are four rows of graphs, showing the results for 4 different number of molecules to compute in the NBody simulation, considering increasing problem sizes, from *x1* to *x4*.

It can be contrasted how NBody offers an extraordinary behavior when using the dynamic strategy provided by *Coexecutor Runtime*. It generally yields higher speedups as the number of packages and problem sizes increase, providing good scalability. However, when using a very high number of packages, a turning point is reached and the speedup starts to drop. This behavior occurs earlier when using *Buffers* than when using shared memory

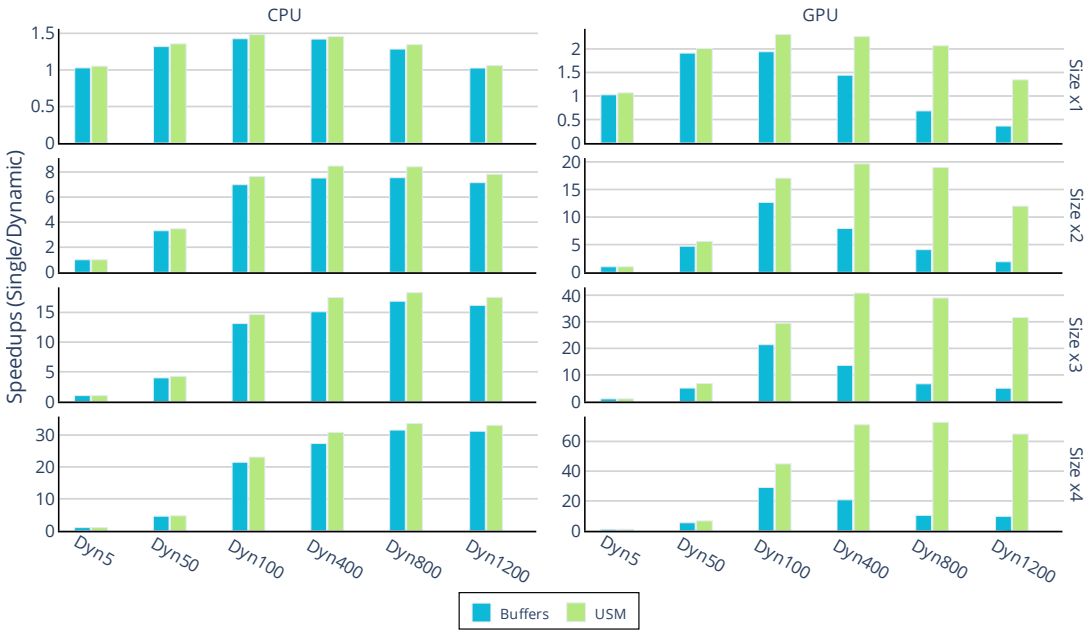


Figure 5-10: NBody speedups when using single-device dynamic policies for a set of increasing problem sizes. Baselines are single CPU or GPU execution per memory model.

USM. Additionally, the speedups obtained are not so good in the CPU compared with the GPU, but on the other hand the difference between memory models is not as important when using the CPU device. Moreover, in bigger problem sizes (*Size x3, x4*), larger package configurations, such as *Dyn400* or *Dyn800*, obtain even better speedups, especially on CPU. Therefore, for these cases and depending on the problem size, the inflection point is at a very large number of packages, with an amount in between *Dyn800* and *Dyn1200*. Beyond that point, many packages increase the communications penalization, degrading the overall performance. Furthermore, even if management and synchronization have been increased, the overhead is considerably reduced when using USM. The performance of NBody is excellent mainly due to three main factors: the characteristics of the kernel computation, the heterogeneous architecture, and the efficiency of the runtime. As a result of this combination, the cost of computation and communication produced by each package launched is very similar, so the computation-communication overlap helps to practically eliminate the communication cost. In other benchmarks the ratio of computation to communication is higher, so the overlap does not have as much impact. In addition, it is possible that the implementation of DPC++ and the oneAPI extensions recognize the independency of the data computation given by the multi-threaded architecture of *Coexecutor Runtime*, generating multiple command queues to transfer memory regions and perform parallel kernel executions, overlapping computation and communication, as has been seen in Chapter 3 and many papers [93, 125, 126, 128, 381].

The results presented in this section are from the Desktop node, but similar behaviors are found in the DevCloud server.

5.6 Conclusions

This Chapter details the *Coexecutor Runtime*, facilitating the exploitation of heterogeneous systems and providing co-execution to the novel oneAPI technology. As a result of this work, several conclusions can be obtained regarding its design and architecture, as well as behavioral aspects, thanks to the exhaustive experimental validation.

Intel oneAPI and its unified programming model, based on SYCL, is a powerful approach to facilitate the programming of heterogeneous systems. Although initially limited to Intel devices, it is gaining strong adoption among other manufacturers and devices. The extensions provided, such as the possibility of exploiting unified shared memory (USM), give the technology greater capabilities, enhancing its versatility. However, this technology does not allow leveraging the devices of the heterogeneous system efficiently. For this reason, CoexecutorRuntime is designed and implemented as an abstraction over the oneAPI technology, with the objective of inheriting its advantageous features, and solving its drawbacks. In addition to enabling co-execution, a fundamental point is its API similar to SYCL. Thus, the system is abstracted while facilitating its programmability in a style defined by the standard, while providing different levels of detail to define the co-execution of the problem. This enables the reuse of programs and greater extensibility in the kernel code, improving the expressiveness when implementing applications. Another relevant point is that the multi-threaded architecture has been designed with asynchrony and dynamic mechanisms in mind, since the objective is to exploit the heterogeneous system in the best possible way. Thanks to this strategies, it has been possible to implement load balancing algorithms efficiently. Finally, the third factor to consider is that the runtime has mechanisms to guarantee its extensibility without penalizing its API design, optimization efforts and the evolution of the oneAPI capabilities, making it easy to adapt new features and extensions.

The most important conclusion drawn from the results is that co-execution is worthwhile when using dynamic algorithms. The underlying technology is able to offer exceptional performance by exploiting dynamic mechanisms, favoring a high computation-communication overlap, with extreme cases such as the one shown in the NBody study. This conclusion is further evidenced when using the HGuided algorithm, which offers the best dynamic behavior for *Coexecutor Runtime*, and the unified shared memory strategy. However, even if USM is the most efficient strategy, the abstraction of the memory model with respect to the scheduling system allows to exploit also SYCL Buffers with good co-execution results, promoting compatibility with devices that do not support USM. It is important to note how the benchmark is performed against the GPU, the fastest device, and yet it is worthwhile to use the runtime and perform co-execution beyond an inflection point. This is generally located in small problem sizes, as it is highlighted by the scalability analysis.

Considering the benchmarks studied, *Coexecutor Runtime* achieves an efficiency of 0.92 in Desktop and 0.89 in DevCloud, in addition to achieving 72% more energy efficiency than

when using only the GPU. All these results are achieved due to efficient synchronization, architecture design decisions, computation and communication overlap, and the underlying oneAPI technology and its DPC++ compiler and runtime. Finally, co-execution has been validated with CPUs and integrated GPUs, but *Coexecutor Runtime* is also able of effortlessly exploit other types of architectures that will be incorporated into oneAPI.

Conclusions & Future Work

CHAPTER 6

Conclusions & Future Work

Trends in recent years are demonstrating how the ubiquity of heterogeneous systems is key to exploiting all kinds of computational problems, from the most ambitious ones with large data volumes and in HPC environments, through specific accelerations in embedded devices, to commercial applications and cloud service servers. However, current programming models require portability and integration efforts, penalizing usability, all while not squeezing the full efficiency of these massively parallel systems. This dissertation proposes the development of heterogeneous runtimes with a simplified API to facilitate programmability and improve the performance and energy efficiency of these systems. For that purpose, it draws on current trends and de facto standards, performing abstractions and optimizations on these technologies. The two proposals made, EngineCL and CoexecutorRuntime, take different approaches in the resolution of the objectives. The first one offers a flexible engine with maximum compatibility between architectures, centered on OpenCL and with an API providing a very high level of abstraction, offering a runtime decoupled from the application domain. The second focuses on offering a solution as close as possible to the computational problem, providing an architecture close to C++ and SYCL, with co-execution capabilities through an API compatible with oneAPI technology. Finally, two integrations made with one of the designed engines are adapted and evaluated by applying and extending it to two situations for which it was not originally designed. This chapter highlights the most important conclusions of the work done in this PhD, synthesized and grouped in the three previous chapters. In addition, multiple lines of future work are outlined, both general and specific, as a result of the research and contributions presented.

Chapter contents

6.1 Conclusions

The popularity of massively parallel systems, and especially heterogeneous computing, is driven by their excellent capabilities, particularly their performance and energy efficiency. The diversity of computationally intensive and data-intensive problems continues to grow, so a future with even greater heterogeneity is inevitable. As a result of this constant demand, more and more specialized and innovative architectures are emerging, including all kinds of on-chip units and external accelerators. However, this brings a multitude of challenges from the software perspective, as programming becomes radically more complicated. The host-device accelerator offloading model is limited by both low architectural scalability and usability, as it under-utilizes existing devices and wastes power, especially on the host. Execution models focus on task-based parallelism to make it easier for programmers to use accelerators, since they assign a coarse-grained task per accelerator. However, they require manual adaptation efforts and do not fully exploit system resources, as it is often data parallelism which facilitates the exploitation of these devices. Co-execution strategies are useful to squeeze the potential of heterogeneous systems, but are not supported by existing frameworks, requiring the programmer to perform a manual transformation of the applications, incurring in high engineering efforts that may lead to errors or low optimization.

This dissertation presents several contributions focused on enabling and optimizing co-execution in heterogeneous systems, relieving the programmer and promoting high usability. The main approach to solving these problems is through the conception of runtimes that abstract the underlying systems, orchestrate all operations with existing devices and facilitate performance portability. Furthermore, scheduling strategies and load balancing algorithms are incorporated, evaluated and optimized to efficiently execute the problems to be solved. The approach of the proposals is always multi-objective, trying to squeeze the maximum performance and energy efficiency not only avoiding penalizing the usability, but improving it. The programmer will be able to generate maintainable, extensible and portable code, making implementations built independently of the heterogeneous systems used. The purpose is to take advantage of all available computing devices to solve tasks in a cooperative way. Moreover, the efforts elaborated here have contributed to any further evolution and maintainability, with special emphasis on runtime extensibility. The incorporation of future accelerators and even variations of their low-level APIs should be a simple task thanks to the modular design, flexible architecture and encapsulation principles, as demonstrated by the various integrations and other external work. The aim is to make good use of existing architectures and programming models, but also to prepare for future technological changes. Therefore, this dissertation addresses these objectives by making contributions grouped in three blocks.

First, the EngineCL runtime is proposed as a flexible and portable solution for heterogeneous systems, focusing on improving the programmability of OpenCL technology. The

design principles underlying the resulting architecture and API are usability and performance, thereby also improving its energy efficiency. Thanks to the practices carried out on top of the OpenCL technology, it is possible to comfortably exploit all the devices of a heterogeneous system, thanks to the compatibility not only between different manufacturers but also between existing driver versions and architectures. Since it has a layered design based on software architecture practices, its maintainability and extensibility excels, as it has been verified both in the software engineering metrics and in the integrations presented in Chapter 3. One of the fundamental advantages of the runtime is its abstraction, encapsulating the execution core and decoupling the problem domain, thus allowing the isolation of the algorithmic and hardware optimization fundamentals of the problem to be solved. Moreover, this modularity has been especially exploited by incorporating a pluggable scheduling system that has made it possible to squeeze the performance of the heterogeneous system with various load balancing algorithms. Validation has been exhaustive, considering two fronts, programmability and performance. EngineCL has been compared with OpenCL in a counterproductive situation for the former proposal, since only a single device offloading is used for the comparison, without exploiting all the existing capabilities. Even so, regarding the worst cases found per device for all benchmarks studied, the average overhead is around 1.3%, with the maximum found being 2.8%. The runtime scaled well as the problem size and execution duration increased, with a decreasing tendency towards negligible overheads. As for usability, it improves significantly in all the metrics analyzed, reaching extreme cases of up to 21 and 8.5 times better on average, considering error control or operational complexity, respectively. Maintainability improves even more as the number and type of devices in the system increased, standing out over a pure execution based only on OpenCL. Furthermore, the validation on two heterogeneous systems, one HPC and one commodity, with 6 different architectures, obtained remarkable results with average efficiencies of 0.89 and 0.82, respectively. Finally, considering the energy validation, EngineCL reached improvements of up to 1.6 times better than the most energy efficient device. The co-execution achieved, on average, 1.36 and 1.39 times better energy efficiency than the GPU for regular and irregular problems, respectively. In short, the runtime is able to take advantage of all existing architectures without effort for the programmer, while achieving high performance, good energy efficiencies and low overhead compared with OpenCL.

Secondly, EngineCL is extended for two different application scenarios, making integrations for time-constrained executions and also in the exploitation of a molecular dynamics simulator where the code was hand-optimized for CPUs and OpenCL cannot be competitive against it. One of the clearest conclusions is the versatility of the designed runtime, and how it can be adapted comfortably to situations for which it was not initially designed.

The integration performed in a commodity node as a service server performs optimizations to be competitive in the execution of kernels of short duration. The architectural approach focuses on optimizing the initial stages of the runtime, parallelizing the configura-

tion stages, reusing OpenCL primitives and optimizing data management, so as to lighten the overhead of using a management infrastructure. On the other hand, the exploration and tuning of parameter values of the best used load balancing algorithm has been performed.The optimization of the runtime together with parameter tuning results in a scheduling combination that is always the most efficient, obtaining averages of 0.84 for the applications studied. This contribution reduces the inflection point from which it is advantageous to perform co-execution, bringing heterogeneous systems closer to these initially counterproductive computations.

On the other hand, the second integration solves a performance degradation encountered when using OpenCL technology in the computation of molecular dynamics kernels, as part of the ls1-MarDyn simulator. The runtime architecture is extended, incorporating a new CPU-native execution core and hybrid co-execution capabilities, allowing to efficiently exploit the heterogeneous system. This contribution delves into the exploration of combinations of heterogeneous programming models, amplifying their potential for exploitation and preventing the programmer from being aware of the internal management complexity. The validation and experimental results highlight several conclusions, such as the importance of having diverse scheduling algorithms to better squeeze the variety of problems, the importance of a lightweight but flexible arquitecture with efficient mechanisms, and how the new native execution kernel achieves comparable performance to the highly optimized version developed and used for years. Furthermore, considering the complete heterogeneous system, better performance and energy efficiency are achieved than with the current mechanism, with improvement averages of 1.38x and 1.60x, respectively.

Finally, thanks to the new Intel oneAPI technology, CoexecutorRuntime is conceived and built to facilitate the exploitation of heterogeneous architectures from a modern programming perspective based on C++ and SYCL. The proposal provides a design close to the problem domain and with an oneAPI-compatible programming interface, combining ease of use with the extensions of this technology. The principle of a single source code and the versatility of C++ in the definition of operations and its architecture are preserved, thus facilitating its modification and extension. The limitation of oneAPI with respect to co-execution has been overcome thanks to the incorporation of a multi-threaded asynchronous architecture that exploits the simultaneous execution of devices, benefiting from the potential of dynamic strategies and implementing various load balancing algorithms. The proposal has been validated in terms of performance, energy efficiency and scalability, to verify the behavior in the face of regular and irregular problems, since the possibility of using dynamic scheduling mechanisms were previously non-existent. Experimental results show how the runtime is especially advantageous when using a dynamic load balancing algorithm and exploiting shared memory optimization. The validation on two heterogeneous machines, a commodity node and an HPC node, reached average efficiencies of 0.92 and 0.89, respectively. Moreover, the more balanced the computational powers of the devices used are, the faster they take advan-

tage of co-execution, scaling appropriately as the problem size increases and achieving better energy efficiencies. The exhaustive study and evaluation of the oneAPI technology and its DPC++ compiler during the conception of the runtime has led to an efficient synchronization, allowing computation-communication overlap and highlighting the good behavior of CoexecutorRuntime when using dynamic algorithms.

The contributions presented herein led to the conclusion that the initial hypothesis has been verified. The runtimes developed effectively improve initially competing objectives. On the one hand, enhancing those related to the efficiency of heterogeneous systems, such as performance, scalability and energy efficiency. On the other hand, alleviating the challenges regarding compatibility, programmability and maintainability of these architectures and systems, simplifying the effort required to exploit them properly.

A fundamental factor of these proposals is the interest aroused in the community, as well as their applicability and effective extensibility, materializing in other external works and integrations. Both EngineCL and CoexecutorRuntime have been extended by another research group, adapting the runtimes to leverage architectures such as FPGAs, further improving the exploitation of such heterogeneous systems effortlessly.

6.2 Future Work

This dissertation has managed to advance in the proposed objectives, obtaining tangible results both in terms of programmability and performance and energy efficiency, but the exploratory path continues. It is an ambitious problem on which a multitude of scientists and professionals, from academia to industry, have been generating proposals and contributions for years. There are many approaches to this problem, including architectural solutions, programming models and abstraction frameworks, but ultimately, they all nourish each other, converging and building progressively better solutions.

Three main research fronts can be distinguished on the basis of this thesis:

- *Combination of runtime systems, technologies and models.* Considering the two engines designed, a combined solution between both proposals could be conceived and exploited, with the best of each of them: the compatibility and usability provided by EngineCL together with the flexibility and easy adaptability of CoexecutorRuntime. This front is wide because the diversity is huge, but ambitious explorations can result in really good approaches. It is important to explore new proposals and extensions, both of previous runtimes and new technologies, introducing the most relevant ones in order to build integral solutions as versatile as possible. As it has been seen in Chapter 5 with the case of oneAPI, the initial limitation in architectures of a single vendor is soon overcome, becoming a new heterogeneous computing framework applicable to many architectures. Without sufficient hatching and combination of possibilities,

it does not generate traction and adoption, generating new innovations. Hybrid programming models are the prelude to new languages and the incorporation of functionalities in existing runtimes.

- *Scheduling algorithms.* Just as the designed runtimes have achieved very high efficiencies, they still have room for improvement. Architectural and software optimization solutions have been key to exploit the variety of architectures, systems and problems shown, but there is still a niche for enhancement, especially as the types of devices and their specialized capabilities continue to grow. On the one hand, contrasted with the evaluations performed on the different proposed runtimes, load balancing algorithms are not decoupled from the implementations. Thus, it is important to elaborate proposals considering their runtimes and execution technologies from the very conception, empirically validated and not being considered as purely theoretical algorithms. Moreover, algorithms have traditionally been focused on performance maximization, but other objectives could be considered, such as lower energy consumption or higher energy efficiency. It is also possible to optimize the best utilization of specialized devices and their available units, avoiding hogging all their resources if they are not going to be exploited. Additionally, some proposals can include greater efficiency in obtaining suitable results, something of vital importance in current computations in fields such as deep learning, visual or signal processing, where precision may not be as important as a compromise between accuracy-error and throughput.

- *Heterogeneous inter- and intra-node computing.* A field with great impact is to leverage the proposals built by amplifying their scope of application, raising the potential of exploitation by being extended to heterogeneous clusters. The integration performed between hybrid programming models and the extension of execution cores outline the first steps in this path, albeit limited to a single node. The runtime should be extended to enable replication between nodes, increasing communication and load distribution capabilities, while exploiting hybrid computing patterns with well-established distributed computing technologies, such as MPI or even PGAS. This end-to-end execution solution could be approached by considering internally all technologies, frameworks and languages involved, managed by itself as encapsulated parts. In such case, it facilitates establishing high level optimizations and patterns, relieving the programmer from complex decisions. Additionally, two types of scheduling algorithms could be considered, those that distribute the work between nodes, and those that do it within the node, each one attending to different considerations to squeeze the maximum out of the set of distributed nodes.

In view of the proposals presented, the details of their design and the experimental results, more specific future works are emphasized:

- *Support for problems based on other execution types.* Provide both EngineCL and CoexecutorRuntime with support for problems that require execution modes other than those based on data-parallel iterations, such as multi-kernel executions, streaming processes or iterative problems. The execution of applications with different computational functions can be found in deep learning training, bioinformatics or in the application of filters and pattern recognition, among others. Stream-based computing, usually binary data flows, generally involves the entire spectrum of stream, signal, video, audio and communications processing. Finally, iterative computations are typical in all types of simulators, where each epoch solves a series of stages, drawing on information solved in previous epochs.

- *Expand the type of devices evaluated and exploited.* The diversity of existing devices is immense, and for this reason, there is a great deal of scope for study and experimentation. On the one hand, there are specific accelerators for neural, visual, signal or even pattern processing. These devices can be integrated into the runtimes, either by means of an appropriate driver (OpenCL), or by encapsulation and native incorporation, as has been done in the Chapter 4 on hybrid paradigm integrations. On the other hand, CoexecutorRuntime has been evaluated only with Intel architectures, due to its current support with oneAPI technology. However, there are devices from other manufacturers that could be integrated and evaluated. For instance, evaluation of neural and visual acceleration devices such as Intel Compute Stick, AMD Navi RDNA and Intel Xe discrete GPUs, as well as A311D heterogeneous SoCs are some of the architectures that are beginning to be explored, in addition to ongoing work with FPGAs.

- *Broaden heterogeneous computing platforms and nodes.* Commodity and HPC nodes have been the focus so far, but embedded platforms and SoCs have yet to be addressed. There are low-power computers that offer processors with special neural and vector processing units, GPUs integrated on-chip and digital and visual signal coprocessors, as well as NVME technology and access to ports where specific accelerator sticks can be attached. This heterogeneous amalgam enables a myriad of possibilities, where the challenges seen in the integration Chapter for time-limited scenarios will be emphasized. However, low-power units that are efficiently exploited and take advantage of all specialized resources represent a breakthrough for energy saving, distributed edge computing and IoT, such as autonomous vehicles and drones, forced to do energy-efficient in-situ computations. Nevertheless, EngineCL has already been integrated and has support for exploiting both embedded mobile SoCs, running Android and Linux, and heterogeneous commodity nodes running Microsoft Windows. It currently has multi-platform support, being adapted to work with .NET and NDK, enabling the use of Microsoft Visual C++ runtime and Java Native Interface (JNI). How-

ever, it is still necessary to evaluate performance and exploit real applications used in such systems, from 3D graphic modeling and rendering, to trading algorithms and crypto-mining, as well as mobile IoT processing and inference acceleration tools.

- *Support for distributed computing, both clusters and service servers.* Both EngineCL and CoexecutorRuntime can be encapsulated and used as load balancing management runtimes within the node, once distributed by consolidated technologies. These technologies can be either MPI, focusing on the message-passing paradigm, PGAS languages for the distribution of data structures and objects, or virtual machine runtimes specialized in service orchestration, such as the Erlang BEAM. However, this exploratory space not only contemplates the optimization of inter-process communications and memory sharing, but also the possibility of incorporating these technologies as an integral solution, so that EngineCL or CoexecutorRuntime are the orchestrators of the entire heterogeneous cluster.

- *Dynamic scheduling algorithms specifically designed for CoexecutorRuntime.* The excellent capabilities of this runtime and its asynchronous architecture have demonstrated a very efficient behavior for dynamic mechanisms, discovering exceptional peculiarities when faced with some kind of patterns. A dynamic algorithm expressly created for this runtime could benefit from such behaviors, provided that these cases are analyzed and generalized to be properly exploited.

- *Dynamic scheduling algorithms that take into account the characteristics of architectures.* EngineCL provides a unique architecture that allows to comfortably extend scheduling algorithms, in addition to offering a very large compatibility by supporting hybrid computing thanks to its native execution cores and OpenCL. Load balancing algorithms could be created to exploit all this diversity, considering different optimization criteria and the types of architectures present. EngineCL is a runtime that has the advantage of being able to validate itself against a multitude of architectures, and for this reason, it is a good framework to exploit different lines of optimization. Load balancing algorithms can consider not only performance and energy efficiency, but also the correct use of the resources present and the occupancy of their units. For example, after a learning phase, divide workload types based on architectures with more vector units, those with neural processors and those with more versatility in the divergence of conditions, such as architectures exploiting branch predictors and multiple layers in the memory hierarchy. In short, facilitating the workload to be distributed in the best way among the existing devices.

- *Exploitation of other accelerators in the ls1-MarDyn simulator.* Modern and powerful multi-cores and other architectures with multiple vector units can be interesting in processing problems where CPUs have excelled with respect to other accelerators, as

seen in the integration in the ls1-MarDyn simulator. For instance, it will be interesting evaluating the NEC SX-Aurora TSUBASA vector engine coprocessor with up to 10 cores per processor, the next Intel Shappire Rapids with up to 56 cores, as well as the AMD Epyc Genoa and Ryzen Threadripper CPUs, having up to 96 cores. With the extension of the runtime and its new processing modes, it is now possible to exploit hybrid co-execution with diverse technologies.

- *Runtime integration strategies.* The incorporation of runtimes and heterogeneous programming models in real applications is far from trivial. For this reason, techniques can be devised to address the study of bottlenecks, technology porting, architectural transformation and incorporation of proposals that improve usability and performance. Besides, as a result of these studies, innovative solutions can be developed that exploit techniques to improve throughput or device utilization, with the consequent improvement in performance. Some of these techniques could involve data compaction and serialization to objects that exploit texture units, device code portability tools with support for pointer indirections in data structures or the adaptation of kernels to higher level programming models (OpenCL C++) with support for features used in real-world applications.

- *Exploiting heterogeneity in modern languages.* This dissertation has approached the problem from an industrial perspective, where traditionally most applications that exploit massively parallel computing are built in C++, with its advantages and disadvantages. However, there are multiple languages among which Rust or Julia stand out, which are increasingly adopted and may become a significant industrial alternative in a few years. These languages provide the software with other guarantees, both in terms of security and determinism, as well as maintainability and performance. Some of them even offer abstractions to use technologies such as OpenCL, but they are still far from being complete solutions that enable effortless co-execution. For this reason, the goal could be both to provide wrappers and interfaces to reuse the runtimes proposed here, as well as to enable these languages with these capabilities without incurring in foreign-function interfaces (FFI), bindings and external languages. Nevertheless, the impact of accelerating bottleneck regions in languages that are not as computationally efficient, but which are much more popular in all kinds of domains, should not be underestimated. Service servers with millions of monthly requests worldwide can benefit from the exploitation of the technologies and runtimes designed through the use of connectors, such as Erlang Ports, PHP extensions or FFIs for Node.js and Python, among others.

- *Include new execution cores in EngineCL.* There are other technologies with great popularity in the market, such as CUDA for Nvidia devices, and some more restricted ones, such as ROCm for some of the latest AMD generations or HiP for several models

of AMD and Nvidia GPUs. However, their usefulness and efficiency is demonstrated by the number of jobs and industries using these technologies. Hence, as it has been carried out in the extension for hybrid models, a new execution core can be included to support them, extending the portability of existing programs and obtaining an increase in efficiency by extending the hybrid co-execution.

- *Extension of real-world applications and benchmarks*. The diversity of characteristics of the benchmarks used is substantial, but new applications comprising other patterns, primitives or extensions of the technologies explored could continue to be incorporated. This is important to validate both runtimes, algorithmic proposals and future integral solutions, so it is important to have a sufficient repertoire. Furthermore, the work done on the ls1-MarDyn simulator shows the inherent complexity of using real applications and integrative efforts, but it further consolidates the proposals generated. For this reason, it is important to expand the programs used by incorporating real-world softwares, presenting a clear benefit to industrial or scientific applications that are currently in use. Using this approach, HPC applications used in industry and research centers can be addressed, from astrophysics and molecular dynamics to fire prediction, as well as general applications used by hundreds of thousands of users worldwide, including geographic information systems procedures, rendering engines, or multimedia processors and converters, among others.

In conclusion, the optimization of performance and energy efficiency in massively parallel systems is an open problem, as it is a relevant and impactful, but complex and multi-objective issue, necessarily having to be approached from multiple perspectives.

Publications and Contributions

The research presented in this PhD thesis is supported or highly related with the following publications, all obtained during the quest to accomplish the PhD.

Journals:

- R. Nozal and J. L. Bosque, "Straightforward Heterogeneous Computing with the oneAPI Coexecutor Runtime", *Electronics*, vol. 10, no. 19:2386. Sep. 2021. **[JCR Q2; 125/266]**.
- R. Nozal, J. L. Bosque, and R. Beivide, "EngineCL: Usability and Performance in Heterogeneous Computing", *Future Generation Computer Systems*, vol. 107, no. C, pp. 522–537, Jun. 2020. **[JCR Q1; 7/110]**.
- M. A. Dávila Guzmán, R. Nozal, R. Gran Tejero, M. Villarroya-Gaudó, D. Suárez Gracia, and J. L. Bosque, "Cooperative CPU, GPU, and FPGA heterogeneous execution with EngineCL", *The Journal of Supercomputing*, vol. 75, no. 3, pp. 1732–1746, Mar. 2019. **[JCR Q2; 24/53]**.
- R. Nozal, B. Perez, J. L. Bosque, and R. Beivide, "Load Balancing in a Heterogeneous World: CPU-Xeon Phi co-execution of data-parallel kernels", *The Journal of Supercomputing*, vol. 75, no. 3, pp. 1123–1136, Feb. 2019. **[JCR Q2; 24/53]**.

Peer-reviewed conferences and workshops:

- R. Nozal and J. L. Bosque, "Exploiting Co-execution with oneAPI: Heterogeneity from a Modern Perspective", *27th International European Conference on Parallel and Distributed Computing (Euro-Par 2021: Parallel Processing)*, L. Sousa, N. Roma, and P. Tomás, Eds., Cham: Springer International Publishing, pp. 501–516. Lisbon, Portugal. Apr. 2021. Presented Sep. 2021. **[GII CoRE A- (Idx 2018)]**.
- R. Nozal, C. Niethammer, J. Gracia, and J. L. Bosque, "Feasibility study of Molecular Dynamics kernels exploitation using EngineCL", *19th International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms (HeteroPar'2021)*, *Euro-Par 2021: Parallel Processing Workshops*. Lisbon, Portugal. Apr. 2021. Presented Aug. 2021.

- R. Nozal, J. L. Bosque, and R. Beivide, "Towards Co-execution on Commodity Heterogeneous Systems: Optimizations for time-constrained scenarios", *2019 International Conference on High Performance Computing & Simulation (HPCS)*, IEEE, pp. 628–635, Dublin, Ireland. May 2019. Presented Jul. 2019. **[GII CoRE B (Idx 2018)]**.

- R. Nozal and J. L. Bosque, "EngineCL as a high performance runtime system for heterogeneous computing", *Programmability and Architectures for Heterogeneous Multicores, Int. Workshop MULTIPROG-2019. European Network of Excellence on High Performance and Empedded Architecture and Compilation (HIPEAC 2019)*. Valencia, Spain. Dec 2018. Presented Jan. 2019.

- M. A. Dávila-Guzmán, R. Nozal, R. Gran, M. Villaroya-Gaudó, D. Suárez, and J. L. Bosque, "First steps towards CPU, GPU, and FPGA parallel execution with EngineCL", *Proc. 18th Int. Conf. Comput. Math. Method Sci. Eng. C. (CMMSE)*. Cádiz, Spain. May 2018.

- R. Nozal, B. Pérez, and J. L. Bosque, "Towards co-execution of massive data-parallel opencl kernels on CPU and Intel Xeon Phi", *Proc. 17th Int. Conf. Comput. Math. Methods Sci. Eng. (CMMSE)*, pp. 1561–1572. Cádiz, Spain. May 2017. Presented Jul. 2017.

Other presentations at conferences, workshops and industrial events:

- R. Nozal and J. L. Bosque, "Towards an efficient adaptive load balancing algorithm for heterogeneous computing", *Int. Conf. Comput. Math. Methods Sci. Eng. and Conf. on High Performance Computing (CMMSE and CHPC 2021)*. Cádiz, Spain. Jul. 2021.

- R. Nozal and J. L. Bosque, "Towards high-level heterogeneous co-execution via oneAPI *(Hacia la co-ejecución heterogénea de alto nivel con oneAPI)*", *XXXI Jornadas de Paralelismo (JP2021), Sociedad de Arquitectura y Tecnología de Computadores (SARTECO)*. Málaga, Spain. Sep. 2021.

- R. Nozal and J. L. Bosque, "EngineCL: Usability and Performance in Heterogeneous Computing *(EngineCL: usabilidad y rendimiento en computación heterogénea)*", *XXX Jornadas de Paralelismo (JP2019), Sociedad de Arquitectura y Tecnología de Computadores (SARTECO)*. Cáceres, Spain. Sep. 2019.

Open source software:

- R. Nozal, "Coexecutor Runtime", C++/SYCL Heterogeneous Runtime using the oneAPI technology, promoting a compatible and extensible API and exploiting dynamic policies. Author. Free open-source repository at https://github.com/oneAPI-scheduling/CoexecutorRuntime.

- R. Nozal, "EngineCL", C++ Heterogeneous Runtime focusing on usability and performance, powered mainly by the OpenCL technology, offering a high level API with a flexible and efficient architecture. Author. Free open-source repository at https://github.com/EngineCL/EngineCL.

- E. Stafford, B. Pérez and R. Nozal, "Sauna", C tool to measure energy consumption of CPUs, GPUs and XeonPhi. Contributor (XeonPhi, AMD GPUs, sysfs devices). Free open-source repository at https://github.com/esteban-stafford/sauna.

The following publication is out of the scope of this dissertation but was performed during the same period:

- J. Larruskain, D. Celorrio, I. Barrio, A. Odriozola, S. M. Gil, J. R. Fernandez-Lopez, R. Nozal, I. Ortuzar, J. A. Lekue, and J. M. Aznar, "Genetic variants and hamstring injury in soccer: An association and validation study", *Medicine and science in sports and exercise*, vol. 50, no. 2, pp. 361–368, 2018. **[JCR Q1]**

Bibliography

[1] Gordon E Moore et al. *Cramming more components onto integrated circuits*. 1965 (see p. 5)

[2] Robert H Dennard, Fritz H Gaensslen, Hwa-Nien Yu, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. "Design of ion-implanted MOSFET's with very small physical dimensions." In: *IEEE Journal of Solid-State Circuits* 9.5 (1974), pp. 256–268 (see p. 5)

[3] Tse-Yu Yeh and Yale N Patt. "A comprehensive instruction fetch mechanism for a processor supporting speculative execution." In: *ACM SIGMICRO Newsletter* 23.1-2 (1992), pp. 129–139 (see p. 5)

[4] Harry Dwyer and Hwa C Torng. "An out-of-order superscalar processor with speculative execution and fast, precise interrupts." In: *ACM SIGMICRO Newsletter* 23.1-2 (1992), pp. 272–281 (see p. 5)

[5] Pradeep K Dubey and Michael J Flynn. "Optimal pipelining." In: *Journal of Parallel and Distributed Computing* 8.1 (1990), pp. 10–19 (see p. 5)

[6] Kazuaki Murakami, Naohiko Irie, and Shinji Tomita. "SIMP (Single Instruction stream/-Multiple instruction Pipelining): A novel high-speed single-processor architecture." In: *ACM SIGARCH Computer Architecture News* 17.3 (1989), pp. 78–85 (see p. 5)

[7] Mike Johnson. *Superscalar multiprocessor design*. Prentice-Hall, Inc., 1991 (see p. 5)

[8] Norman P Jouppi and David W Wall. "Available instruction-level parallelism for superscalar and superpipelined machines." In: *ACM SIGARCH Computer Architecture News* 17.2 (1989), pp. 272–282 (see p. 5)

[9] Dean M Tullsen, Susan J Eggers, and Henry M Levy. "Simultaneous multithreading: Maximizing on-chip parallelism." In: *Proceedings of the 22nd annual international symposium on Computer architecture*. 1995, pp. 392–403 (see p. 5)

[10] Jack L Lo, Joel S Emer, Henry M Levy, Rebecca L Stamm, Dean M Tullsen, and Susan J Eggers. "Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading." In: *ACM Transactions on Computer Systems (TOCS)* 15.3 (1997), pp. 322–354 (see p. 5)

[11] Herb Sutter et al. "The free lunch is over: A fundamental turn toward concurrency in software." In: *Dr. Dobb's journal* 30.3 (2005), pp. 202–210 (see p. 5)

[12] Nian Liu, Jinyu Gu, Dahai Tang, Kenli Li, Binyu Zang, and Haibo Chen. "Asymmetry-aware Scalable Locking." In: *arXiv preprint arXiv:2108.03355* (2021) (see p. 5)

[13] Amit Kumar Singh, Alok Prakash, Karunakar Reddy Basireddy, Geoff V Merrett, and Bashir M Al-Hashimi. "Energy-efficient run-time mapping and thread partitioning of concurrent OpenCL applications on CPU-GPU MPSoCs." In: *ACM Transactions on Embedded Computing Systems (TECS)* 16.5s (2017), pp. 1–22 (see p. 5)

[14] Ananya Muddukrishna, Peter A Jonsson, and Mats Brorsson. "Locality-aware task scheduling and data distribution for OpenMP programs on NUMA systems and manycore processors." In: *Scientific Programming* 2015 (2015) (see p. 5)

[15] Robert Schöne, Thomas Ilsche, Mario Bielert, Andreas Gocht, and Daniel Hackenberg. "Energy efficiency features of the intel skylake-sp processor and their impact on performance." In: *2019 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE. 2019, pp. 399–406 (see p. 5)

[16] José L Bosque, Oscar D Robles, Luis Pastor, and Angel Rodríguez. "Parallel CBIR implementations with load balancing algorithms." In: *Journal of parallel and distributed computing* 66.8 (2006), pp. 1062–1075 (see p. 6)

[17] Emilio Castillo, Cristóbal Camarero, Ana Borrego, and Jose Luis Bosque. "Financial applications on multi-CPU and multi-GPU architectures." In: *The Journal of Supercomputing* 71.2 (2015), pp. 729–739 (see p. 6)

[18] G Ortega, Julia Lobera, MP Arroyo, Inmaculada García, and Ester M Garzón. "High performance computing for optical diffraction tomography." In: *2012 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE. 2012, pp. 195–201 (see p. 6)

[19] Jose A Piedra-Fernandez, Gloria Ortega, James Z Wang, and Manuel Canton-Garbin. "Fuzzy content-based image retrieval for oceanic remote sensing." In: *IEEE Transactions on Geoscience and Remote Sensing* 52.9 (2013), pp. 5422–5431 (see p. 6)

[20] Alejandro Gutierrez-Alcoba, Gloria Ortega, Eligius MT Hendrix, and Inmaculada García. "Accelerating an algorithm for perishable inventory control on heterogeneous platforms." In: *Journal of Parallel and Distributed Computing* 104 (2017), pp. 12–18 (see p. 6)

[21] Pablo Toharia, Oscar D Robles, Ricardo SuáRez, Jose Luis Bosque, and Luis Pastor. "Shot boundary detection using Zernike moments in multi-GPU multi-CPU architectures." In: *Journal of Parallel and Distributed Computing* 72.9 (2012), pp. 1127–1133 (see p. 6)

[22] F Orts, G Ortega, AM Puertas, Inmaculada García, and Ester M Garzón. "On solving the unrelated parallel machine scheduling problem: active microrheology as a case study." In: *The Journal of Supercomputing* 76.11 (2020), pp. 8494–8509 (see p. 6)

[23] James Jeffers and James Reinders. *Intel Xeon Phi coprocessor high performance programming*. Newnes, 2013 (see pp. 6, 9, 37, 41)

[24] David Kirk and W Hwu Wen-Mei. *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann, 2016 (see pp. 6, 9, 35, 41)

[25] Sparsh Mittal. "A survey on evaluating and optimizing performance of Intel Xeon Phi." In: *Concurrency and Computation: Practice and Experience* 32.19 (2020), e5742 (see p. 6)

[26] Chao-Tung Yang, Jung-Chun Liu, Yu-Wei Chan, Endah Kristiani, and Chan-Fu Kuo. "Performance benchmarking of deep learning framework on Intel Xeon Phi." In: *The Journal of Supercomputing* 77.3 (2021), pp. 2486–2510 (see p. 6)

[27] Ji-Hoon Kang, Jinyul Hwang, Hyung Jin Sung, and Hoon Ryu. "High-performance simulations of turbulent boundary layer flow using Intel Xeon Phi many-core processors." In: *The Journal of Supercomputing* (2021), pp. 1–18 (see p. 6)

[28] Victoria Sanz, Adrián Pousa, Marcelo Naiouf, and Armando De Giusti. "Accelerating Pattern Matching on Intel Xeon Phi Processors." In: *International Conference on Algorithms and Architectures for Parallel Processing*. Springer. 2020, pp. 262–274 (see p. 6)

[29] Fang Huang, Hao Yang, Jian Tao, Jian Wang, and Xicheng Tan. "Preliminary study on the automatic parallelism optimization model for image enhancement algorithms based on Intel's® Xeon Phi." In: *Concurrency and Computation: Practice and Experience* (2021), e6260 (see p. 6)

[30] Kengo Nakajima, Balazs Gerofi, Yutaka Ishikawa, and Masashi Horikoshi. "Efficient Parallel Multigrid Method on Intel Xeon Phi Clusters." In: *The International Conference on High Performance Computing in Asia-Pacific Region Companion*. 2021, pp. 46–49 (see p. 6)

[31] Simon J Pennycook, Chris J Hughes, Mikhail Smelyanskiy, and Stephen A Jarvis. "Exploring simd for molecular dynamics, using intel® xeon® processors and intel® xeon phi coprocessors." In: *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE. 2013, pp. 1085–1097 (see p. 6)

[32] Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. "Knights landing: Second-generation intel xeon phi product." In: *Ieee micro* 36.2 (2016), pp. 34–46 (see p. 6)

[33] Mitsuo Yokokawa, Ayano Nakai, Kazuhiko Komatsu, Yuta Watanabe, Yasuhisa Masaoka, Yoko Isobe, and Hiroaki Kobayashi. "I/o performance of the sx-aurora tsubasa." In: *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. 2020, pp. 27–35 (see p. 6)

[34] Matthias Noack, Erich Focht, and Thomas Steinke. "Heterogeneous active messages for offloading on the NEC SX-Aurora TSUBASA." In: *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. 2019, pp. 26–35 (see p. 6)

[35] Hiroyuki Takizawa, Shinji Shiotsuki, Naoki Ebata, and Ryusuke Egawa. "An OpenCL-Like Offload Programming Framework for SX-Aurora TSUBASA." In: *2019 20th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*. IEEE. 2019, pp. 282–288 (see p. 6)

[36] Kazuhiko Komatsu, Shintaro Momose, Yoko Isobe, Osamu Watanabe, Akihiro Musa, Mitsuo Yokokawa, Toshikazu Aoyama, Masayuki Sato, and Hiroaki Kobayashi. "Performance evaluation of a vector supercomputer SX-Aurora TSUBASA." In: *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2018, pp. 685–696 (see p. 6)

[37] Sergio Rivas-Gomez, Antonio J Pena, David Moloney, Erwin Laure, and Stefano Markidis. "Exploring the vision processing unit as co-processor for inference." In: *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. 2018, pp. 589–598 (see pp. 6, 7)

[38] Gerard JM Smit, André BJ Kokkeler, Gerard K Rauwerda, and Jan WM Jacobs. "Reconfigurable Multicore Architectures for Streaming Applications." In: *Model-Based Design for Embedded Systems*. CRC Press, 2018, pp. 347–374 (see p. 6)

[39] Imad Al Assir, Mohamad El Iskandarani, Hadi Rayan Al Sandid, and Mazen AR Saghir. "Arrow: A RISC-V Vector Accelerator for Machine Learning Inference." In: *arXiv preprint arXiv:2107.07169* (2021) (see p. 6)

[40] Christophe Clienti, Serge Beucher, and Michel Bilodeau. "A system on chip dedicated to pipeline neighborhood processing for mathematical morphology." In: *2008 16th European Signal Processing Conference*. IEEE. 2008, pp. 1–5 (see p. 6)

[41] Yeyong Pang, Shaojun Wang, Yu Peng, and Xiyuan Peng. "Fully Pipelined Soft Vector Processor as a CPU Accelerator." In: *Chinese Journal of Electronics* 26.6 (2017), pp. 1198–1205 (see p. 6)

[42] Tassadaq Hussain, Oscar Palomar, Adrian Cristal, Eduard Ayguade, and Amna Haider. "Mvpa: An fpga based multi-vector processor architecture." In: *2016 13th International Bhurban Conference on Applied Sciences and Technology (IBCAST)*. IEEE. 2016, pp. 213–218 (see p. 6)

[43] Marziyeh Nourian, Mostafa Eghbali Zarch, and Michela Becchi. "Optimizing Complex OpenCL Code for FPGA: A Case Study on Finite Automata Traversal." In: *2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE. 2020, pp. 518–527 (see p. 7)

[44] Iman Firmansyah and Yoshiki Yamaguchi. "OpenCL implementation of FPGA-based signal generation and measurement." In: *IEEE Access* 7 (2019), pp. 48849–48859 (see p. 7)

[45] Maria A. Dávila-Guzmán, Rubén Gran Tejero, María Villarroya-Gaudó, and Darío Suárez Gracia. "An Analytical Model of Memory-Bound Applications Compiled with High Level Synthesis." In: *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2020, pp. 218–218 (see p. 7)

[46] Maria Angélica Dávila-Guzmán, Rubén Gran Tejero, María Villarroya-Gaudó, Darío Suárez Gracia, Lester Kalms, and Diana Göhringer. "A Cross-Platform OpenVX Library for FPGA Accelerators." In: *2021 29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. 2021, pp. 75–83 (see p. 7)

[47] Maria A. Dávila-Guzmán, Rubén Gran Tejero, María Villarroya-Gaudó, and Darío Suárez Gracia. "Analytical Model for Memory-Centric High Level Synthesis-Generated Applications." In: *IEEE Transactions on Computers* (2021) (see p. 7)

[48] Maria A Dávila-Guzmán, Raúl Nozal, R Gran, M Villaroya-Gaudó, D Suárez, and Jose Luis Bosque. "First Steps Towards CPU, GPU, and FPGA Parallel Execution with EngineCL." In: *Proc. 18th Int. Conf. Comput. Math. Method Sci. Eng. C*. 2018 (see pp. 7, 13, 62, 132)

[49] Ian Kuon and Jonathan Rose. "Measuring the gap between FPGAs and ASICs." In: *IEEE Transactions on computer-aided design of integrated circuits and systems* 26.2 (2007), pp. 203–215 (see p. 7)

[50] Eriko Nurvitadhi, David Sheffield, Jaewoong Sim, Asit Mishra, Ganesh Venkatesh, and Debbie Marr. "Accelerating binarized neural networks: Comparison of FPGA, CPU, GPU, and ASIC." In: *2016 International Conference on Field-Programmable Technology (FPT)*. IEEE. 2016, pp. 77–84 (see p. 7)

[51] Eriko Nurvitadhi, Dongup Kwon, Ali Jafari, Andrew Boutros, Jaewoong Sim, Phillip Tomson, Huseyin Sumbul, Gregory Chen, Phil Knag, Raghavan Kumar, et al. "Why compete when you can work together: Fpga-asic integration for persistent rnns." In: *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE. 2019, pp. 199–207 (see p. 7)

[52] Sang Yoon Park and Pramod Kumar Meher. "Efficient FPGA and ASIC realizations of a DA-based reconfigurable FIR digital filter." In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 61.7 (2014), pp. 511–515 (see p. 7)

[53] Amara Amara, Frederic Amiel, and Thomas Ea. "FPGA vs. ASIC for low power applications." In: *Microelectronics journal* 37.8 (2006), pp. 669–677 (see p. 7)

[54] Arun Subramaniyan and Reetuparna Das. "Parallel automata processor." In: *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2017, pp. 600–612 (see p. 7)

[55] Ke Wang, Elaheh Sadredini, and Kevin Skadron. "Sequential pattern mining with the micron automata processor." In: *Proceedings of the ACM International Conference on Computing Frontiers*. 2016, pp. 135–144 (see p. 7)

[56] Keira Zhou, Jack Wadden, Jeffrey J Fox, Ke Wang, Donald E Brown, and Kevin Skadron. "Regular expression acceleration on the micron automata processor: Brill tagging as a case study." In: *2015 IEEE International Conference on Big Data (Big Data)*. IEEE. 2015, pp. 355–360 (see p. 7)

[57] Gene Frantz. "Digital signal processor trends." In: *IEEE micro* 20.6 (2000), pp. 52–59 (see p. 7)

[58] John A Stratton, Jyothi Krishna Viswakaran Sreelatha, Rajiv Ravindran, Sachin Sudhakar Dake, and Jeevitha Palanisamy. "Optimizing Halide for Digital Signal Processors." In: *2020 IEEE Workshop on Signal Processing Systems (SiPS)*. IEEE. 2020, pp. 1–6 (see pp. 7, 41)

[59] Jia-Jhe Li, Chi-Bang Kuan, Tung-Yu Wu, and Jenq Kuen Lee. "Enabling an opencl compiler for embedded multicore dsp systems." In: *2012 41st International Conference on Parallel Processing Workshops*. IEEE. 2012, pp. 545–552 (see pp. 7, 41)

[60] Pramesh Pandey, Prabal Basu, Koushik Chakraborty, and Sanghamitra Roy. "GreenTPU: Improving timing error resilience of a near-threshold tensor processing unit." In: *2019 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE. 2019, pp. 1–6 (see p. 7)

[61] Youngeun Kwon and Minsoo Rhu. "A disaggregated memory system for deep learning." In: *IEEE Micro* 39.5 (2019), pp. 82–90 (see p. 7)

[62] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. "In-datacenter performance analysis of a tensor processing unit." In: *Proceedings of the 44th annual international symposium on computer architecture*. 2017, pp. 1–12 (see p. 7)

[63] Norman Jouppi, Cliff Young, Nishant Patil, and David Patterson. "Motivation for and evaluation of the first tensor processing unit." In: *IEEE Micro* 38.3 (2018), pp. 10–19 (see p. 7)

[64] Kuan-Chieh Hsu and Hung-Wei Tseng. "Accelerating applications using edge tensor processing units." In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2021, pp. 1–14 (see p. 7)

[65] Savvas Sioutas, Sander Stuijk, Twan Basten, Lou Somers, and Henk Corporaal. "Programming tensor cores from an image processing DSL." In: *Proceedings of the 23th International Workshop on Software and Compilers for Embedded Systems*. 2020, pp. 36–41 (see p. 7)

[66] Minsoo Rhu. "Accelerator-centric deep learning systems for enhanced scalability, energy-efficiency, and programmability." In: *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE. 2018, pp. 527–533 (see p. 7)

[67] Bin Zhang, Chen Zhao, Kuizhi Mei, Jizhong Zhao, and Nanning Zheng. "Hierarchical and parallel pipelined heterogeneous soc for embedded vision processing." In: *IEEE Transactions on Circuits and Systems for Video Technology* 28.6 (2017), pp. 1434–1444 (see p. 7)

[68] Lorenzo Petrosino, Giulio Iannello, Mario Merone, and Luca Vollero. "Image sensors and VPU acceleration for data analysis and classification." In: *2021 IEEE International Workshop on Metrology for Industry 4.0 & IoT (MetroInd4. 0&IoT)*. IEEE. 2021, pp. 392–396 (see p. 7)

[69] Suyash Bakshi and Lennart Johnsson. "Analysis of Factors Affecting Power Consumption and Energy Efficiency of SGEMM on the Low-Power Myriad-2 VPU." In: *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE. 2021, pp. 76–78 (see p. 7)

[70] Hongqiang Wang, Jay Yun, and Alex Bourd. "Opencl optimization and best practices for qualcomm adreno gpus." In: *Proceedings of the International Workshop on OpenCL*. 2018, pp. 1–8 (see p. 7)

[71] David Baldo, Alessandro Mecocci, Stefano Parrino, Giacomo Peruzzi, and Alessandro Pozzebon. "A Multi-Layer LoRaWAN Infrastructure for Smart Waste Management." In: *Sensors* 21.8 (2021), p. 2600 (see p. 7)

[72] Chenhao Xie, Jieyang Chen, Jesun Firoz, Jiajia Li, Shuaiwen Leon Song, Kevin Barker, Mark Raugas, and Ang Li. "Fast and scalable sparse triangular solver for multi-gpu based hpc architectures." In: *50th International Conference on Parallel Processing*. 2021, pp. 1–11 (see p. 7)

[73] Nitin A Gawande, Jeff A Daily, Charles Siegel, Nathan R Tallent, and Abhinav Vishnu. "Scaling deep learning workloads: Nvidia dgx-1/pascal and intel knights landing." In: *Future Generation Computer Systems* 108 (2020), pp. 1162–1172 (see p. 7)

[74] Amir Yazdanbakhsh, Jongse Park, Hardik Sharma, Pejman Lotfi-Kamran, and Hadi Esmaeilzadeh. "Neural acceleration for gpu throughput processors." In: *Proceedings of the 48th international symposium on microarchitecture*. 2015, pp. 482–493 (see p. 7)

[75] Sumin Kim, Seunghwan Oh, and Youngmin Yi. "Minimizing GPU Kernel Launch Overhead in Deep Learning Inference on Mobile GPUs." In: *Proceedings of the 22nd International Workshop on Mobile Computing Systems and Applications*. 2021, pp. 57–63 (see p. 7)

[76] Roberto Giorgi, Farnam Khalili, and Marco Procaccini. "Energy efficiency exploration on the zynq ultrascale+." In: *2018 30th International Conference on Microelectronics (ICM)*. IEEE. 2018, pp. 48–54 (see p. 8)

[77] Panagiotis Mousouliotis, Stavros Zogas, Panagiotis Christakos, Georzios Keramidas, Nikos Petrellis, Christos Antonopoulos, and Nikolaos Voros. "Exploiting Vitis Framework for Accelerating Sobel Algorithm." In: *2021 10th Mediterranean Conference on Embedded Computing (MECO)*. IEEE. 2021, pp. 1–5 (see p. 8)

[78] Khoa Dang Pham, Anuj Vaishnav, Malte Vesper, and Dirk Koch. "ZUCL: a ZYNQ Ultrascale+ framework for OpenCL HLS applications." In: *FSP Workshop 2018; Fifth International Workshop on FPGAs for Software Programmers*. VDE. 2018, pp. 1–9 (see p. 8)

[79] Balint Barna Kövari and Emad Ebeid. "MPDrone: FPGA-based Platform for Intelligent Real-time Autonomous Drone Operations." In: *2021 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)*. IEEE. 2021, pp. 71–76 (see p. 8)

[80] Juan Carlos Castillo, Francisco Almeida, Vicente Blanco, and M Carmen Ramírez. "Web services based platform for the cell counting problem." In: *European Conference on Parallel Processing*. Springer. 2014, pp. 83–92 (see pp. 8, 99)

[81] Vincent Chau, Xiaowen Chu, Hai Liu, and Yiu-Wing Leung. "Energy efficient job scheduling with DVFS for CPU-GPU heterogeneous systems." In: *Proceedings of the Eighth International Conference on Future Energy Systems*. 2017, pp. 1–11 (see p. 8)

[82] Borja Pérez, Esteban Stafford, José Luis Bosque, and Ramón Beivide. "Energy efficiency of load balancing for data-parallel applications in heterogeneous systems." In: *The Journal of Supercomputing* 73.1 (2017), pp. 330–342 (see pp. 8, 48, 72)

[83] Sunbal Cheema and Gul N Khan. "Power and Performance Analysis of Deep Neural Networks for Energy-aware Heterogeneous Systems." In: *2020 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. IEEE. 2020, pp. 2184–2189 (see p. 8)

[84] Yuxiang Gao, Saeed Iqbal, Peng Zhang, and Meikang Qiu. "Performance and power analysis of high-density multi-GPGPU architectures: A preliminary case study." In: *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*. IEEE. 2015, pp. 66–71 (see p. 8)

[85] Qiang Liu and Wayne Luk. "Heterogeneous systems for energy efficient scientific computing." In: *International Symposium on Applied Reconfigurable Computing*. Springer. 2012, pp. 64–75 (see p. 8)

[86] Mohamed Zahran. "Heterogeneous computing: Here to stay." In: *Communications of the ACM* 60.3 (2017), pp. 42–45 (see pp. 8, 52)

[87] Keith A Britt and Travis S Humble. "High-performance computing with quantum processing units." In: *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 13.3 (2017), pp. 1–13 (see p. 8)

[88] Keith A Britt, Fahd A Mohiyaddin, and Travis S Humble. "Quantum accelerators for high-performance computing systems." In: *2017 IEEE International Conference on Rebooting Computing (ICRC)*. IEEE. 2017, pp. 1–7 (see p. 8)

[89] Travis S Humble, Ronald J Sadlier, and Keith A Britt. "Simulated execution of hybrid quantum computing systems." In: *Quantum Information Science, Sensing, and Computation X*. Vol. 10660. International Society for Optics and Photonics. 2018, p. 1066002 (see p. 8)

[90] Tom Deakin and Simon McIntosh-Smith. "Evaluating the performance of HPC-style SYCL applications." In: *Proceedings of the International Workshop on OpenCL*. 2020, pp. 1–11 (see pp. 9, 38)

[91] Wei-Chen Lin, Tom Deakin, and Simon McIntosh-Smith. "On measuring the maturity of SYCL implementations by tracking historical performance improvements." In: *International Workshop on OpenCL*. 2021, pp. 1–13 (see pp. 9, 38)

[92] Beau Johnston, Jeffrey S Vetter, and Josh Milthorpe. "Evaluating the Performance and Portability of Contemporary SYCL Implementations." In: *2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE. 2020, pp. 45–56 (see pp. 9, 38)

[93] R. Nozal, J. L. Bosque, and R. Beivide. "EngineCL: Usability and Performance in Heterogeneous Computing." In: *Future Generation Computer Systems* 107.C (June 2020), pp. 522–537 (see pp. 9, 12, 72, 141, 159)

[94] David R Kaeli, Perhaad Mistry, Dana Schaa, and Dong Ping Zhang. *Heterogeneous computing with OpenCL 2.0*. Morgan Kaufmann, 2015 (see pp. 9, 40, 41, 43)

[95] Sunita Chandrasekaran and Guido Juckeland. *OpenACC for Programmers: Concepts and Strategies*. Addison-Wesley Professional, 2017 (see pp. 9, 36)

[96] Aaftab Munshi. "The opencl specification." In: *2009 IEEE Hot Chips 21 Symposium (HCS)*. IEEE. 2009, pp. 1–314 (see p. 9)

[97] John E Stone, David Gohara, and Guochun Shi. "OpenCL: A parallel programming standard for heterogeneous computing systems." In: *Computing in science & engineering* 12.3 (2010), p. 66 (see p. 9)

[98] James Reinders, Ben Ashbaugh, James Brodman, Michael Kinsner, John Pennycook, and Xinmin Tian. *Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL*. Springer Nature, 2021 (see pp. 10, 38, 46)

[99] Kui Wang, Jari Nurmi, and Tapani Ahonen. "Accelerating Computation on an Android Phone with OpenCL Parallelism and Optimizing Workload Distribution between a Phone and a Cloud Service." In: *2016 Intl IEEE Conferences on Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld)*. IEEE. 2016, pp. 636–642 (see p. 10)

[100] Onur Atan, Yiannis Andreopoulos, Cem Tekin, and Mihaela van der Schaar. "Bandit framework for systematic learning in wireless video-based face recognition." In: *IEEE Journal of Selected Topics in Signal Processing* 9.1 (2014), pp. 180–194 (see p. 10)

[101] Roberto Di Lauro, Flora Giannone, Luigia Ambrosio, and Raffaele Montella. "Virtualizing general purpose GPUs for high performance cloud computing: an application to a fluid simulator." In: *2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications*. IEEE. 2012, pp. 863–864 (see p. 10)

[102] Federico Simmross-Wattenberg, Manuel Rodríguez-Cayetano, Javier Royuela-del-Val, Elena Martin-Gonzalez, Elisa Moya-Sáez, Marcos Martín-Fernández, and Carlos Alberola-López. "OpenCLIPER: an OpenCL-based C++ Framework for overhead-reduced medical image processing and reconstruction on heterogeneous devices." In: *IEEE journal of biomedical and health informatics* 23.4 (2018), pp. 1702–1709 (see pp. 10, 15)

[103] Ibrahim Savran, Elif Aras, Gökhan Uzer, and Shaafici Abdi. "Accelerating gene identification in DNA sequences with CUDA and OpenCL." In: *2018 26th Signal Processing and Communications Applications Conference (SIU)*. IEEE. 2018, pp. 1–4 (see p. 10)

[104] Adrian Odriozola, José A Riancho, Raul Nozal, Arancha Bermúdez, Ana Santurtún, Jana Arozamena, and María Teresa Zarrabeitia. "Chimerism analysis in transplant patients: A hypothesis-free approach in the absence of reference genotypes." In: *Clinica Chimica Acta* 414 (2012), pp. 85–90 (see p. 10)

[105] Ummu Habibe Unal and Ibrahim Savran. "Accelerating next generation sequencing read errors correction with CUDA." In: *2017 25th Signal Processing and Communications Applications Conference (SIU)*. IEEE. 2017, pp. 1–4 (see p. 10)

[106] Alexey Cheptsov, Stefan Wesner, and Bastian Koller. "Service-Oriented Development of Workflow-Based Semantic Reasoning Applications." In: *International Journal of Distributed Systems and Technologies (IJDST)* 5.1 (2014), pp. 40–53 (see p. 10)

[107] Nauman Ahmed, Hamid Mushtaq, Koen Bertels, and Zaid Al-Ars. "GPU accelerated API for alignment of genomics sequencing data." In: *2017 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*. IEEE. 2017, pp. 510–515 (see p. 10)

[108] Mitsuhiro Okada, Takayuki Suzuki, Naoya Nishio, Hasitha Waidyasooriya, and Masanori Hariyama. "FPGA-accelerated Searchable Encrypted Database Management Systems for Cloud Services." In: *IEEE Transactions on Cloud Computing* (2020) (see p. 10)

[109] Mohamed G Malhat and Ashraf B El-Sisi. "Parallel ward clustering for chemical compounds using opencl." In: *2015 Tenth International Conference on Computer Engineering & Systems (ICCES)*. IEEE. 2015, pp. 23–27 (see p. 10)

[110] Yanpeng Cao, Feng Yu, and Yongming Tang. "A digital watermarking encryption technique based on FPGA cloud accelerator." In: *IEEE Access* 8 (2020), pp. 11800–11814 (see p. 10)

[111] Yoji Yamato, Tatsuya Demizu, Hirofumi Noguchi, and Misao Kataoka. "Proposal of Automatic GPU Offloading Technology on Open IoT Environment." In: *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*. Vol. 2. IEEE. 2018, pp. 634–639 (see p. 10)

[112] Chi-Sheng Shih, Yu-Kai Chen, Joen Chen, and Norman Chang. "Virtual cloud core: Opencl workload sharing framework for connected devices." In: *2013 IEEE Seventh International Symposium on Service-Oriented System Engineering*. IEEE. 2013, pp. 486–493 (see p. 10)

[113] Chin-Chen Chang, Wai-Kong Lee, Yanjun Liu, Bok-Min Goi, and Raphael C-W Phan. "Signature gateway: Offloading signature generation to IoT gateway accelerated by GPU." In: *IEEE Internet of Things Journal* 6.3 (2018), pp. 4448–4461 (see p. 10)

[114] Akhila Prabhakaran and J Lakshmi. "Cost-benefit Analysis of Public Clouds for offloading in-house HPC Jobs." In: *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE. 2018, pp. 57–64 (see p. 10)

[115] Shuo Wang, Yun Liang, and Wei Zhang. "Poly: Efficient heterogeneous system and application management for interactive applications." In: *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2019, pp. 199–210 (see p. 10)

[116] Christoph Niethammer, Stefan Becker, Martin Bernreuther, Martin Buchholz, Wolfgang Eckhardt, Alexander Heinecke, Stephan Werth, Hans-Joachim Bungartz, Colin W Glass, Hans Hasse, et al. "ls1 mardyn: The massively parallel molecular dynamics code for large systems." In: *Journal of chemical theory and computation* 10.10 (2014), pp. 4455–4464 (see pp. 11, 115)

[117] Nikola Tchipev, Steffen Seckler, Matthias Heinen, Jadran Vrabec, Fabio Gratl, Martin Horsch, Martin Bernreuther, Colin W Glass, Christoph Niethammer, Nicolay Hammer, et al. "TweTriS: Twenty trillion-atom simulation." In: *The International Journal of High Performance Computing Applications* 33.5 (2019), pp. 838–854 (see p. 11)

[118] Iuliana Marin, Nicolae Goga, and Maria Goga. "Benchmarking MD systems simulations on the graphics processing unit and multi-core systems." In: *2016 IEEE International Symposium on Systems Engineering (ISSE)*. IEEE. 2016, pp. 1–5 (see p. 11)

[119] Hasitha Muthumala Waidyasooriya, Masanori Hariyama, and Kota Kasahara. "Architecture of an FPGA accelerator for molecular dynamics simulation using OpenCL." In: *2016 IEEE/ACIS 15th International Conference on Computer and Information Science (ICIS)*. IEEE. 2016, pp. 1–5 (see p. 11)

[120] István Lőrentz, Răzvan Andonie, and Levente Fabry-Asztalos. "Accelerating Molecular Structure Determination Based on Inter-Atomic Distances Using OpenCL." In: *IEEE Transactions on Parallel and Distributed Systems* 26.12 (2014), pp. 3250–3263 (see p. 11)

[121] Alexander S Minkin, Anton B Teslyuk, Andrey A Knizhnik, and Boris V Potapkin. "GPGPU performance evaluation of some basic molecular dynamics algorithms." In: *2015 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE. 2015, pp. 629–634 (see p. 11)

[122] Ziming Zhong, Vladimir Rychkov, and Alexey Lastovetsky. "Data partitioning on multicore and multi-GPU platforms using functional performance models." In: *IEEE Transactions on Computers* 64.9 (2014), pp. 2506–2518 (see pp. 11, 15, 55)

[123] Janghaeng Lee, Mehrzad Samadi, Yongjun Park, and Scott Mahlke. "Transparent CPU-GPU collaboration for data-parallel kernels on heterogeneous systems." In: *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*. IEEE. 2013, pp. 245–255 (see pp. 11, 15, 54, 55, 61)

[124] Janghaeng Lee, Mehrzad Samadi, Yongjun Park, and Scott Mahlke. "Skmd: Single kernel on multiple devices for transparent cpu-gpu collaboration." In: *ACM Transactions on Computer Systems (TOCS)* 33.3 (2015), pp. 1–27 (see pp. 11, 15, 54, 55, 61)

[125] M. A. Dávila Guzmán, R. Nozal, R. Gran Tejero, M. Villarroya-Gaudó, D. Suárez Gracia, and J. L. Bosque. "Cooperative CPU, GPU, and FPGA heterogeneous execution with EngineCL." In: *The Journal of Supercomputing* 75.3 (Mar. 2019), pp. 1732–1746 (see pp. 12, 13, 15, 41, 61, 62, 70, 72, 132, 141, 159)

[126] R. Nozal, J. L. Bosque, and R. Beivide. "Towards Co-execution on Commodity Heterogeneous Systems: Optimizations for Time-Constrained Scenarios." In: *2019 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE. 2019, pp. 628–635 (see pp. 12, 48, 61, 141, 159)

[127] Borja Pérez, E Stafford, JL Bosque, and R Beivide. "Sigmoid: an auto-tuned load balancing algorithm for heterogeneous systems." In: *Journal of Parallel and Distributed Computing* (2021) (see pp. 12, 25, 55)

[128] R. Nozal, B. Perez, J. L. Bosque, and R Beivide. "Load balancing in a heterogeneous world: CPU-Xeon Phi co-execution of data-parallel kernels." In: *The Journal of Supercomputing* 75.3 (2019), pp. 1123–1136 (see pp. 12, 13, 15, 41, 61, 63, 141, 159)

[129] Feng Zhang, Jidong Zhai, Bingsheng He, Shuhao Zhang, and Wenguang Chen. "Understanding co-running behaviors on integrated CPU/GPU architectures." In: *IEEE Transactions on Parallel and Distributed Systems* 28.3 (2016), pp. 905–918 (see pp. 12, 16, 54)

[130] Jie Shen, Ana Lucia Varbanescu, Yutong Lu, Peng Zou, and Henk Sips. "Workload partitioning for accelerating applications on heterogeneous platforms." In: *IEEE Transactions on Parallel and Distributed Systems* 27.9 (2015), pp. 2766–2780 (see p. 12)

[131] Lukasz Szustak, Roman Wyrzykowski, Lukasz Kuczynski, and Tomasz Olas. "Architectural Adaptation and Performance-Energy Optimization for CFD Application on AMD EPYC Rome." In: *IEEE Transactions on Parallel and Distributed Systems* 32.12 (2021), pp. 2852–2866 (see p. 12)

[132] Cristian Constantinescu. "AMD EPYC™ 7002 Series–A Processor with Improved Soft Error Resilience." In: *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks-Supplemental Volume (DSN-S)*. IEEE. 2021, pp. 33–36 (see p. 12)

[133] Samuel Naffziger, Noah Beck, Thomas Burd, Kevin Lepak, Gabriel H Loh, Mahesh Subramony, and Sean White. "Pioneering Chiplet Technology and Design for the AMD EPYC™ and Ryzen™ Processor Families: Industrial Product." In: *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2021, pp. 57–70 (see p. 12)

[134] Evgenii P. Vasiliev, Valentina D. Kustikova, Valentin D. Volokitin, Evgeny A. Kozinov, and Iosif B. Meyerov. "Performance Analysis of Deep Learning Inference in Convolutional Neural Networks on Intel Cascade Lake CPUs." In: *Mathematical Modeling and Supercomputer Technologies*. Ed. by Dmitry Balandin, Konstantin Barkalov, Victor Gergel, and Iosif Meyerov. Cham: Springer International Publishing, 2021, pp. 346–360 (see p. 12)

[135] Maxim A Krivov, Nikita G Iroshnikov, Andrey A Butylin, Anna E Filippova, and Pavel S Ivanov. "Comparison of AMD Zen 2 and Intel Cascade Lake on the Task of Modeling the Mammalian Cell Division." In: *International Conference on Mathematical Modeling and Supercomputer Technologies*. Springer. 2020, pp. 320–333 (see p. 12)

[136] Douglas Doerfler, Farzad Fatollahi-Fard, Colin MacLean, Tan Nguyen, Samuel Williams, Nicholas Wright, and Marco Siracusa. "Experiences Porting the SU3_Bench Microbenchmark to the Intel Arria 10 and Xilinx Alveo U280 FPGAs." In: *International Workshop on OpenCL*. 2021, pp. 1–9 (see p. 13)

[137] Yong Wang, Yongfa Zhou, Qi Scott Wang, Yang Wang, Qing Xu, Chen Wang, Bo Peng, Zhaojun Zhu, Katayama Takuya, and Dylan Wang. "Developing medical ultrasound beamforming application on GPU and FPGA using oneAPI." In: *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. 2021, pp. 360–370 (see p. 13)

[138] Srihari Cadambi, Giuseppe Coviello, Cheng-Hong Li, Rajat Phull, Kunal Rao, Murugan Sankaradass, and Srimat Chakradhar. "COSMIC: middleware for high performance and reliable multiprocessing on xeon phi coprocessors." In: *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*. 2013, pp. 215–226 (see p. 13)

[139] Raúl Nozal, Borja Pérez, and Jose Luis Bosque. "Towards co-execution of massive data-parallel OpenCL kernels on CPU and Intel Xeon Phi." In: *Proc. 17th Int. Conf. Comput. Math. Methods Sci. Eng.(CMMSE)*. 2017, pp. 1561–1572 (see pp. 13, 63)

[140] Noah Wolfe, Tianyu Liu, Christopher Carothers, and Xie George Xu. "Heterogeneous concurrent execution of Monte Carlo photon transport on CPU, GPU and MIC." In: *Proceedings of the 4th Workshop on Irregular Applications: Architectures and Algorithms*. 2014, pp. 49–52 (see p. 13)

[141] Jiri Filipovic and Siegfried Benkner. "OpenCL kernel fusion for GPU, Xeon Phi and CPU." In: *2015 27th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE. 2015, pp. 98–105 (see p. 13)

[142] Jaume Bosch, Antonio Filgueras, Miquel Vidal, Daniel Jimenez-Gonzalez, Carlos Alvarez, and Xavier Martorell. "Exploiting parallelism on GPUs and FPGAs with OmpSs." In: *Proceedings of the 1st Workshop on AutotuniNg and aDaptivity AppRoaches for Energy efficient HPC Systems*. 2017, pp. 1–5 (see p. 13)

[143] Enrico Calore, Alessandro Gabbana, Sebastiano Fabio Schifano, and Raffaele Tripiccione. "Optimization of lattice Boltzmann simulations on heterogeneous computers." In: *The International Journal of High Performance Computing Applications* 33.1 (2019), pp. 124–139 (see p. 13)

[144] Marco Minutoli, Maurizio Drocco, Mahantesh Halappanavar, Antonino Tumeo, and Ananth Kalyanaraman. "cuRipples: Influence maximization on multi-GPU systems." In: *Proceedings of the 34th ACM International Conference on Supercomputing*. 2020, pp. 1–11 (see p. 13)

[145] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. "Groute: An asynchronous multi-GPU programming model for irregular computations." In: *ACM SIGPLAN Notices* 52.8 (2017), pp. 235–248 (see p. 13)

[146] Trong-Tuan Vu and Bilel Derbel. "Parallel Branch-and-Bound in multi-core multi-CPU multi-GPU heterogeneous environments." In: *Future Generation Computer Systems* 56 (2016), pp. 95–109 (see p. 13)

[147] Paulo Ferrão, Hélder Marques, and Hervé Paulino. "Stream Processing on Hybrid CPU/Intel® Xeon Phi™ Systems." In: *European Conference on Parallel Processing*. Springer. 2018, pp. 796–810 (see p. 13)

[148] Nilanjan Goswami, Amer Qouneh, Chao Li, and Tao Li. "An Empirical-cum-Statistical Approach to Power-Performance Characterization of Concurrent GPU Kernels." In: *arXiv preprint arXiv:2011.02368* (2020) (see p. 13)

[149] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures." In: *Concurrency and Computation: Practice and Experience* 23.2 (2011), pp. 187–198 (see p. 14)

[150] Alejandro Duran, Eduard Ayguadé, Rosa M Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. "Ompss: a proposal for programming heterogeneous multi-core architectures." In: *Parallel processing letters* 21.02 (2011), pp. 173–193 (see pp. 14, 38)

[151] Josep M Perez, Rosa M Badia, and Jesus Labarta. "A dependency-aware task-based programming environment for multi-core architectures." In: *2008 IEEE international conference on cluster computing*. IEEE. 2008, pp. 142–151 (see p. 14)

[152] Harri Renney, Benedict R Gaster, and Tom Mitchell. "OpenCL vs: Accelerated finite-difference digital synthesis." In: *Proceedings of the International Workshop on OpenCL*. 2019, pp. 1–11 (see p. 15)

[153] Chakib Mustapha Anouar Zouaoui and Nasreddine Taleb. "CL_ARRAY: A new generic library of multidimensional containers for c++ compilers with extension for OpenCL framework." In: *Computer Languages, Systems & Structures* 50 (2017), pp. 53–81 (see p. 15)

[154] Jung-Hyun Hong and Ki-Seok Chung. "Parallel LDPC decoding on a GPU using OpenCL and global memory for accelerators." In: *2015 IEEE International Conference on Networking, Architecture and Storage (NAS)*. IEEE. 2015, pp. 353–354 (see p. 15)

[155] Rashid Kaleem, Rajkishore Barik, Tatiana Shpeisman, Chunling Hu, Brian T Lewis, and Keshav Pingali. "Adaptive heterogeneous scheduling for integrated GPUs." In: *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*. IEEE. 2014, pp. 151–162 (see pp. 15, 55)

[156] Esteban Stafford, Borja Pérez, Jose Luis Bosque, Ramón Beivide, and Mateo Valero. "To distribute or not to distribute: the question of load balancing for performance or energy." In: *European Conference on Parallel Processing*. Springer. 2017, pp. 710–722 (see pp. 15, 27, 48)

[157] Michael Boyer, Kevin Skadron, Shuai Che, and Nuwan Jayasena. "Load balancing in a changing world: dealing with heterogeneity and performance variability." In: *Proceedings of the ACM International Conference on Computing Frontiers*. 2013, pp. 1–10 (see pp. 15, 55)

[158] Mehmet E. Belviranli, Laxmi N. Bhuyan, and Rajiv Gupta. "A Dynamic Self-scheduling Scheme for Heterogeneous Multiprocessor Architectures." In: *ACM Trans. Archit. Code Optim.* 9.4 (Jan. 2013), 57:1–57:20 (see pp. 16, 55)

[159] Antonio Vilches, Rafael Asenjo, Angeles Navarro, Francisco Corbera, Rubén Gran, and María Garzarán. "Adaptive partitioning for irregular applications on heterogeneous CPU-GPU chips." In: *Procedia Computer Science* 51 (2015), pp. 140–149 (see pp. 16, 55)

[160] Jan Lemeire, Jan G Cornelis, and Laurent Segers. "Microbenchmarks for gpu characteristics: The occupancy roofline and the pipeline model." In: *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*. IEEE. 2016, pp. 456–463 (see p. 16)

[161] Thanh Tuan Dao and Jaejin Lee. "An auto-tuner for OpenCL work-group size on GPUs." In: *IEEE Transactions on Parallel and Distributed Systems* 29.2 (2017), pp. 283–296 (see p. 16)

[162] Patrik Goorts, Sammy Rogmans, Steven Vanden Eynde, and Philippe Bekaert. "Practical examples of gpu computing optimization principles." In: *2010 International Conference on Signal Processing and Multimedia Applications (SIGMAP)*. IEEE. 2010, pp. 46–49 (see p. 16)

[163] Ghassan Shobaki, Austin Kerbow, and Stanislav Mekhanoshin. "Optimizing occupancy and ILP on the GPU using a combinatorial approach." In: *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*. 2020, pp. 133–144 (see p. 16)

[164] Yuval Eshkol and Intel. *Debugging and Optimizing OpenCL Applications*. Last accessed November 2021. 2016. URL: https://www.iwocl.org/wp-content/uploads/iwocl-2016-best-practices-to-debug-opencl.pdf (see p. 24)

[165] Intel. *OpenCL™ Developer Guide for Intel® Processor Graphics*. Last accessed November 2021. 2016. URL: https://software.intel.com/content/www/us/en/develop/documentation/iocl-opg/top.html (see p. 24)

[166] Albert Navarro Torrentó. "Optimization of OpenCL applications on FPGA." MA thesis. Universitat Politècnica de Catalunya, 2018 (see p. 24)

[167] Intel. *oneAPI GPU Optimization Guide*. Last accessed November 2021. 2021. URL: https://software.intel.com/content/www/us/en/develop/documentation/oneapi-gpu-optimization-guide/top.html (see p. 24)

[168] Intel. *Intel® oneAPI DPC++ FPGA Optimization Guide*. Last accessed November 2021. 2021. URL: https://software.intel.com/content/www/us/en/develop/documentation/oneapi-fpga-optimization-guide/top.html (see p. 24)

[169] Intel. *Intel® oneAPI samples*. Last accessed November 2021. 2020. URL: https://github.com/oneapi-src/oneAPI-samples (see p. 24)

[170] John E. Stone, David Gohara, and Guochun Shi. "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems." In: *IEEE Des. Test* 12.3 (May 2010), pp. 66–73 (see p. 24)

[171] *AMD Accelerated Parallel Processing (APP) Software Development Kit (SDK) V3*. Last accessed December 2017. December 2017. URL: http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/ (see p. 24)

[172] Jose Luis Bosque and Luis Pastor Perez. "Theoretical scalability analysis for heterogeneous clusters." In: *IEEE International Symposium on Cluster Computing and the Grid, 2004. CCGrid 2004.* IEEE. 2004, pp. 285–292 (see p. 25)

[173] Luis Pastor. "An efficiency and scalability model for heterogeneous clusters." In: *Cluster*. Vol. 1. 2001, p. 427 (see p. 25)

[174] Jose L Bosque, Oscar D Robles, Pablo Toharia, and Luis Pastor. "Evaluating scalability in heterogeneous systems." In: *The Journal of Supercomputing* 58.3 (2011), pp. 367–375 (see p. 25)

[175] Emilio Castillo, Cristóbal Camarero, Ana Borrego, and Jose Luis Bosque. "Financial Applications on multi-CPU and multi-GPU Architectures." In: *J. Supercomput.* 71.2 (Feb. 2015), pp. 729–739 (see p. 26)

[176] Emilio Castillo, Miquel Moreto, Marc Casas, Lluc Alvarez, Enrique Vallejo, Kallia Chronaki, Rosa Badia, Jose Luis Bosque, Ramon Beivide, Eduard Ayguade, et al. "CATA: criticality aware task acceleration for multicore processors." In: *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2016, pp. 413–422 (see p. 26)

[177] Francisco Almeida and Jesús Vigo-Aguiar. "High performance computing tools in science and engineering." In: *The Journal of Supercomputing* 65.3 (2013), pp. 997–998 (see p. 26)

[178] Advanced Micro Devices Inc. *AMD CodeXL: a comprehensive tool suite that enables developers to harness the benefits of CPUs, GPUs and APUs.* Last accessed November 2021. 2016. URL: `https://github.com/GPUOpen-Tools/CodeXL` (see p. 26)

[179] Intel. *Intel VTune Profiler: tool to optimize application performance, system performance, and system configuration for HPC, cloud, IoT, media and storage.* Last accessed November 2021. 2021. URL: `https://software.intel.com/content/www/us/en/develop/tools/vtune-profiler.html` (see p. 26)

[180] Intel. *Intel Intercept Layer for OpenCL*. Last accessed November 2021. 2018. URL: `https://github.com/intel/opencl-intercept-layer` (see p. 27)

[181] Intel. *Profiling Tools Interfaces for GPU - OpenCL Tracer*. Last accessed November 2021. 2021. URL: `https://github.com/intel/pti-gpu` (see p. 27)

[182] Evgeny Peshkov. *CLTracer - cross-platform cross-vendor OpenCL profiler*. Last accessed November 2021. 2020. URL: `https://www.cltracer.com` (see p. 27)

[183] Ben Juurlink, Jan Lucas, Nadjib Mammeri, Martyn Bliss, Georgios Keramidas, Chrysa Kokkala, and Andrew Richards. "The LPGPU2 Project: Low-Power Parallel Computing on GPUs." In: *Proceedings of the 20th International Workshop on Software and Compilers for Embedded Systems*. 2017, pp. 76–80 (see p. 27)

[184] Esteban Stafford, Borja Pérez, and Raúl Nozal. *Sauna: C tool to measure energy consumption of CPUs, GPUs and XeonPhi.* Last accessed November 2021. 2017. URL: `https://github.com/esteban-stafford/sauna` (see p. 27)

[185] Efraim Rotem, Alon Naveh, Doron Rajwan, Avinash Ananthakrishnan, and Eli Weissmann. "Power Management Architecture of the 2nd Generation Intel Core Microarchitecture, Formerly Codenamed Sandy Bridge." In: *IEEE Int. HotChips Symp. on High-Perf. Chips (HotChips~2011)*. 2011 (see p. 27)

[186] NVIDIA. *NVIDIA Management Library (NVML)*. Last accessed April 2019. 2018. URL: `https://developer.nvidia.com/nvidia-management-library-nvml` (see p. 27)

[187] John Cheng, Max Grossman, and Ty McKercher. *Professional CUDA c programming*. John Wiley & Sons, 2014 (see p. 35)

[188] Jason Sanders and Edward Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010 (see p. 35)

[189] Bohan Wang and Jernej Barbic. "CUDA Deformers for Model Reduction." In: *Motion, Interaction and Games*. 2020, pp. 1–10 (see p. 35)

[190] Chin-Pin Ko, Praveen Kumar Chittem, Chiang-An Hsu, Mohammad Alkhaleefah, Min-Jui Huang, and Yang-Lang Chang. "CUDA-enabled Programming for Accelerating Flood Simulation." In: *2021 the 5th International Conference on Graphics and Signal Processing*. 2021, pp. 72–75 (see p. 35)

[191] Yu Ma, Liguo Zhang, and Xia Zhang. "Target tracking algorithm based on cuda parallel acceleration." In: *Proceedings of the 2019 3rd International Conference on Computer Science and Artificial Intelligence*. 2019, pp. 333–337 (see p. 35)

[192] Ban Quy Tran, Thai Van Nguyen, Dat Duy Tran, Anh Duy Tran, and Hoang Van Nguyen. "Accelerating Exemplar-based Image Inpainting with GPU and CUDA." In: *2021 10th International Conference on Software and Computer Applications*. 2021, pp. 173–179 (see p. 35)

[193] Xuechao Li and Po-Chou Shih. "Performance comparison of cuda and openacc based on optimizations." In: *Proceedings of the 2018 2nd High Performance Computing and Cluster Technologies Conference*. 2018, pp. 53–57 (see pp. 35, 36)

[194] Alejandro Acosta, Robert Corujo, Vicente Blanco, and Francisco Almeida. "Dynamic load balancing on heterogeneous multicore/multiGPU systems." In: *HPCS*. Ed. by Waleed W. Smari and John P. McIntire. IEEE, 2010, pp. 467–476 (see p. 35)

[195] JJ Moreno, G Ortega, Ernestas Filatovas, José A Martínez, and Ester M Garzón. "Improving the Energy Efficiency of Evolutionary Multi-objective Algorithms." In: *International Conference on Algorithms and Architectures for Parallel Processing*. Springer. 2016, pp. 62–75 (see p. 35)

[196] JJ Moreno, G Ortega, Ernestas Filatovas, José A Martínez, and Ester M Garzón. "Improving the performance and energy of non-dominated sorting for evolutionary multiobjective optimization on GPU/CPU platforms." In: *Journal of Global Optimization* 71.3 (2018), pp. 631–649 (see p. 35)

[197] JJ Moreno, G Ortega, Ernestas Filatovas, José A Martínez, and Ester M Garzón. "Using low-power platforms for evolutionary multi-objective optimization algorithms." In: *The Journal of Supercomputing* 73.1 (2017), pp. 302–315 (see p. 35)

[198] Barbara Chapman, Gabriele Jost, and Ruud Van Der Pas. *Using OpenMP: portable shared memory parallel programming*. Vol. 10. MIT press, 2008 (see p. 36)

[199] Ruud Van der Pas, Eric Stotzer, and Christian Terboven. *Using OpenMP# The Next Step: Affinity, Accelerators, Tasking, and SIMD*. MIT press, 2017 (see p. 36)

[200] Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Education Group, 2003 (see p. 36)

[201] Jacob Lambert, Seyong Lee, Jeffrey S Vetter, and Allen D Malony. "CCAMP: an integrated translation and optimization framework for OpenACC and OpenMP." In: *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2020, pp. 1–14 (see p. 36)

[202] Michael Klemm, Eduardo Quiñones, Tucker Taft, Dirk Ziegenbein, and Sara Royuela. "The OpenMP API for High Integrity Systems: Moving Responsibility from Users to Vendors." In: *ACM SIGAda Ada Letters* 40.2 (2021), pp. 48–50 (see p. 36)

[203] Swapnil Gaikwad, Andy Nisbet, and Mikel Luján. "Hosting OpenMP programs on Java virtual machines." In: *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*. 2019, pp. 63–71 (see p. 36)

[204] Hervé Yviquel, Lauro Cruz, and Guido Araujo. "Cluster programming using the openmp accelerator model." In: *ACM Transactions on Architecture and Code Optimization (TACO)* 15.3 (2018), pp. 1–23 (see p. 36)

[205] Huan Zhou, José Gracia, and Ralf Schneider. "MPI collectives for multi-core clusters: Optimized performance of the hybrid MPI+ MPI parallel codes." In: *Proceedings of the 48th International Conference on Parallel Processing: Workshops*. 2019, pp. 1–10 (see p. 36)

[206] Joseph Schuchart, Mathias Nachtmann, and José Gracia. "Patterns for OpenMP task data dependency overhead measurements." In: *International Workshop on OpenMP*. Springer. 2017, pp. 156–168 (see p. 36)

[207] Christoph Niethammer, José Gracia, T Hilbrich, A Knupfer, MM Resch, and WE Nagel. "Tools for High Performance Computing." In: *Switzerland: Springer International Publishing, AG* (2017) (see p. 36)

[208] Andreas Knüpfer, José Gracia, Wolfgang E Nagel, and Michael M Resch. *Tools for High Performance Computing 2013: Proceedings of the 7th International Workshop on Parallel Tools for High Performance Computing, September 2013, ZIH, Dresden, Germany*. Springer, 2014 (see p. 36)

[209] Masahiro Nakao, Hitoshi Murai, and Mitsuhisa Sato. "Multi-accelerator extension in OpenMP based on PGAS model." In: *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*. 2019, pp. 18–25 (see p. 36)

[210] Andreas Kurth, Koen Wolters, Björn Forsberg, Alessandro Capotondi, Andrea Marongiu, Tobias Grosser, and Luca Benini. "Mixed-data-model heterogeneous compilation and OpenMP offloading." In: *Proceedings of the 29th International Conference on Compiler Construction*. 2020, pp. 119–131 (see p. 36)

[211] Rob Farber. *Parallel programming with OpenACC*. Newnes, 2016 (see p. 36)

[212] Kohei Fujita, Takuma Yamaguchi, Tsuyoshi Ichimura, Muneo Hori, and Lalith Maddegedara. "Acceleration of element-by-element kernel in unstructured implicit low-order finite-element earthquake simulation using openacc on pascal gpus." In: *2016 Third Workshop on Accelerator Programming Using Directives (WACCPD)*. IEEE. 2016, pp. 1–12 (see p. 36)

[213] Marco Kupiainen, Jing Gong, Lilit Axner, Erwin Laure, and Jan Nordström. "GPU-acceleration of A High Order Finite Difference Code Using Curvilinear Coordinates." In: *Proceedings of the 2020 International Conference on Computing, Networks and Internet of Things*. 2020, pp. 41–47 (see p. 36)

[214] Akihiro Tabuchi, Masahiro Nakao, Hitoshi Murai, Taisuke Boku, and Mitsuhisa Sato. "Performance evaluation for a hydrodynamics application in XcalableACC PGAS language for accelerated clusters." In: *Proceedings of Workshops of HPC Asia*. 2018, pp. 1–10 (see p. 36)

[215] Ahmad Lashgar and Amirali Baniasadi. "Openacc cache directive: Opportunities and optimizations." In: *2016 Third Workshop on Accelerator Programming Using Directives (WACCPD)*. IEEE. 2016, pp. 46–56 (see p. 36)

[216] W Hwu Wen-mei. *Heterogeneous System Architecture: A new compute platform infrastructure*. Morgan Kaufmann, 2015 (see p. 37)

[217] Mohamed Zahran. *Heterogeneous computing: Hardware and software perspectives*. Morgan & Claypool, 2019 (see pp. 37, 52)

[218] Philipp Holzinger, Marc Reichenbach, and Dietmar Fey. "A new generic HLS approach for heterogeneous computing: on the feasibility of high-level synthesis in HSA-compatible systems." In: *Proceedings of the 18th International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*. 2018, pp. 18–27 (see p. 37)

[219] Michael .O Agbaje, Adekunle Bammeke, and Onome Blaise Ohwo. "Heterogeneous System Architecture (HSA)." In: *International Journal of Scientific Research in Computer Science, Engineering and Information Technology (IJSRCSEIT)*. Vol. 3. 3. 2018, pp. 539–546 (see p. 37)

[220] Nandinbaatar Tsog, Marielle Gallardo, Sweta Chakraborty, Torbjörn Martinson, Alexandra Hengl, Magnus Moberg, Adem Sen, Mobyen Uddin Ahmed, Shahina Begum, Moris Behnam, et al. "Supporting Autonomous Vehicle Applications on the Heterogeneous System Architecture." In: *7th Conference on the Engineering of Computer Based Systems*. 2021, pp. 1–8 (see p. 37)

[221] Hao-Che Hsu, Chih-Wei Yeh, Shih-Hao Hung, Wei-Chung Hsu, Chung-Ta King, and Yeh-Ching Chung. "Hsaemu 2.0: full system emulation for hsa platforms with soft-mmu." In: *Proceedings of the international conference on research in adaptive and convergent systems*. 2016, pp. 230–235 (see p. 37)

[222] Yuan-Ming Chang, Shao-Chung Wang, Chun-Chieh Yang, Yuan-Shin Hwang, and Jenq-Kuen Lee. "Enabling PoCL-based runtime frameworks on the HSA for OpenCL 2.0 support." In: *Journal of Systems Architecture* 81 (2017), pp. 71–82 (see p. 37)

[223] James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. " O'Reilly Media, Inc.", 2007 (see p. 37)

[224] Michael Voss, Rafael Asenjo, and James Reinders. *Pro TBB: C++ parallel programming with threading building blocks*. Apress, 2019 (see p. 37)

[225] Abhishek Bhattacharjee, Gilberto Contreras, and Margaret Martonosi. "Parallelization libraries: Characterizing and reducing overheads." In: *ACM Transactions on Architecture and Code Optimization (TACO)* 8.1 (2011), pp. 1–29 (see p. 37)

[226] Aditya Prakash and Parag Chaudhuri. "Comparing performance of parallelizing frameworks for grid-based fluid simulation on the cpu." In: *Proceedings of the 8th Annual ACM India Conference*. 2015, pp. 1–7 (see p. 37)

[227] Ammar Ahmad Awan, Hari Subramoni, and Dhabaleswar K Panda. "An in-depth performance characterization of CPU-and GPU-based DNN training on modern architectures." In: *Proceedings of the Machine Learning on HPC Environments*. 2017, pp. 1–8 (see p. 37)

[228] Wenli Xu, Hao Cha, and Mo Zhou. "Research and Realization of Software Radar Signal Processing Based on Intel MKL." In: *2011 International Conference on Computer and Management (CAMAN)*. IEEE. 2011, pp. 1–6 (see p. 37)

[229] Yujia Zhai, Elisabeth Giem, Quan Fan, Kai Zhao, Jinyang Liu, and Zizhong Chen. "FT-BLAS: a high performance BLAS implementation with online fault tolerance." In: *Proceedings of the ACM International Conference on Supercomputing*. 2021, pp. 127–138 (see p. 37)

[230] Wu Zheng, An Hong, Jin Xu, Chi Mengxian, Lü Guofeng, Wen Ke, and Zhou Xin. "Research and Optimization of Fast Convolution Algorithm Winograd on Intel Platform." In: *Journal of Computer Research and Development* 56.4 (2019), p. 825 (see p. 37)

[231] Denisa-Andreea Constantinescu, Angeles Navarro, Francisco Corbera, Juan-Antonio Fernández-Madrigal, and Rafael Asenjo. "Efficiency and productivity for decision making on low-power heterogeneous CPU+ GPU SoCs." In: *Journal of Supercomputing* 77.1 (2021) (see p. 37)

[232] Peter Pirkelbauer, Amalee Wilson, Christina Peterson, and Damian Dechev. "Blaze-tasks: A framework for computing parallel reductions over tasks." In: *ACM Transactions on Architecture and Code Optimization (TACO)* 15.4 (2019), pp. 1–25 (see p. 37)

[233] Tobias Baumann, Matthias Noack, and Thomas Steinke. "Performance Evaluation and Improvements of the PoCL Open-Source OpenCL Implementation on Intel CPUs." In: *International Workshop on OpenCL*. 2021, pp. 1–12 (see pp. 37, 41)

[234] Adarsh Yoga and Santosh Nagarakatte. "A fast causal profiler for task parallel programs." In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 2017, pp. 15–26 (see p. 37)

[235] Intel. *oneAPI Level Zero*. Last accessed Nov 2021. URL: `https://spec.oneapi.io/level-zero/latest/index.html` (see p. 38)

[236] Yinan Ke, Mulya Agung, and Hiroyuki Takizawa. "neoSYCL: a SYCL implementation for SX-Aurora TSUBASA." In: *The International Conference on High Performance Computing in Asia-Pacific Region*. 2021, pp. 50–57 (see p. 38)

[237] Peter Thoman, Daniel Gogl, and Thomas Fahringer. "Sylkan: Towards a Vulkan Compute Target Platform for SYCL." In: *International Workshop on OpenCL*. 2021, pp. 1–12 (see p. 38)

[238] Gábor Dániel Balogh and István Reguly. "Automatic Parallelisation of Sturctured Mesh Computations with SYCL." In: *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE. 2021, pp. 821–822 (see p. 38)

[239] Vladyslav Kucher, Florian Fey, and Sergei Gorlatch. "Unified Cross-Platform Profiling of Parallel C++ Applications." In: *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE. 2018, pp. 57–62 (see p. 38)

[240] Biagio Peccerillo and Sandro Bartolini. "PHAST-A portable high-level modern C++ programming library for GPUs and multi-cores." In: *IEEE Transactions on Parallel and Distributed Systems* 30.1 (2018), pp. 174–189 (see p. 38)

[241] Grigore Lupescu and Nicolae Ţăpuş. "Design of hashtable for heterogeneous architectures." In: *2021 23rd International Conference on Control Systems and Computer Science (CSCS)*. IEEE. 2021, pp. 172–177 (see p. 38)

[242] Raúl Nozal and Jose Luis Bosque. "Exploiting Co-execution with OneAPI: Heterogeneity from a Modern Perspective." In: *Euro-Par 2021: Parallel Processing*. Ed. by Leonel Sousa, Nuno Roma, and Pedro Tomás. Cham: Springer International Publishing, 2021, pp. 501–516 (see pp. 38, 150)

[243] Ben Ashbaugh, Alexey Bader, James Brodman, Jeff Hammond, Michael Kinsner, John Pennycook, Roland Schulz, and Jason Sewall. "Data Parallel C++ Enhancing SYCL Through Extensions for Productivity and Performance." In: *Proceedings of the International Workshop on OpenCL*. 2020, pp. 1–2 (see p. 38)

[244] Kate Gregory and Ade Miller. *C++ AMP: accelerated massive parallelism with Microsoft Visual C++*. Microsoft Press, 2012 (see p. 38)

[245] K Shyamala, K Raj Kiran, and D Rajeshwari. "Design and implementation of GPU-based matrix chain multiplication using C++ AMP." In: *2017 Second International Conference on Electrical, Computer and Communication Technologies (ICECCT)*. IEEE. 2017, pp. 1–6 (see p. 38)

[246] Lingze Zhang, Yongxing Du, and Daocheng Wu. "GPU-Accelerated FDTD simulation of human tissue using C++ AMP." In: *2015 31st International Review of Progress in Applied Computational Electromagnetics (ACES)*. IEEE. 2015, pp. 1–2 (see p. 38)

[247] Ru Zhu. "Speedup of Micromagnetic Simulations with C++ AMP On Graphics Processing Units." In: *Computing in Science & Engineering* 18.4 (2015), pp. 53–59 (see p. 38)

[248] Erik Wynters. "C++ amp makes it easy to explore parallel processing on GPUs in a college course or research project." In: *Journal of Computing Sciences in Colleges* 33.6 (2018), pp. 197–199 (see p. 38)

[249] M Graham Lopez, Christopher Bergstrom, Ying Wai Li, Wael Elwasif, and Oscar Hernandez. "Using C++ AMP to Accelerate HPC Applications on Multiple Platforms." In: *International Conference on High Performance Computing*. Springer. 2016, pp. 563–576 (see p. 38)

[250] Stefan Mocanu, Gabriel Munteanu, and Daniela Saru. "GPGPU optimized parallel implementation of AES using C++ AMP." In: *Journal of Control Engineering and Applied Informatics* 17.2 (2015), pp. 73–81 (see p. 38)

[251] Florentino Sainz, Sergi Mateo, Vicenç Beltran, Jose L Bosque, Xavier Martorell, and Eduard Ayguadé. "Leveraging ompss to exploit hardware accelerators." In: *2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing*. IEEE. 2014, pp. 112–119 (see p. 38)

[252] Borja Pérez, Esteban Stafford, Jose Luis Bosque, Ramon Beivide, Sergi Mateo, Xavier Teruel, Xavier Martorell, and Eduard Ayguadé. "Extending OmpSs for OpenCL kernel co-execution in heterogeneous systems." In: *2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE. 2017, pp. 1–8 (see pp. 38, 39, 72)

[253] Alejandro Fernández, Vicenç Beltran, Xavier Martorell, Rosa M Badia, Eduard Ayguadé, and Jesus Labarta. "Task-based programming with ompss and its application." In: *European Conference on Parallel Processing*. Springer. 2014, pp. 601–612 (see p. 39)

[254] Joseph Schuchart, Christoph Niethammer, and José Gracia. "Fibers are not (P) Threads: The Case for Loose Coupling of Asynchronous Programming Models and MPI Through Continuations." In: *27th European MPI Users' Group Meeting*. 2020, pp. 39–50 (see p. 39)

[255] Robin Kumar Sharma and Marc Casas. "Wavefront parallelization of recurrent neural networks on multi-core architectures." In: *Proceedings of the 34th ACM International Conference on Supercomputing*. 2020, pp. 1–12 (see p. 39)

[256] Antoni Navarro Muñoz, Arthur F. Lorenzon, Eduard Ayguadé Parra, and Vicenç Beltran Querol. "Combining Dynamic Concurrency Throttling with Voltage and Frequency Scaling on Task-based Programming Models." In: *50th International Conference on Parallel Processing*. 2021, pp. 1–11 (see p. 39)

[257] Evgeny Kuznetsov and Vladimir Stegailov. "Porting CUDA-based molecular dynamics algorithms to AMD ROCm platform using hip framework: performance analysis." In: *Russian Supercomputing Days*. Springer. 2019, pp. 121–130 (see p. 39)

[258] Kawthar Shafie Khorassani, Jahanzeb Hashmi, Ching-Hsiang Chu, Chen-Chun Chen, Hari Subramoni, and Dhabaleswar K Panda. "Designing a ROCm-Aware MPI Library for AMD GPUs: Early Experiences." In: *International Conference on High Performance Computing*. Springer. 2021, pp. 118–136 (see p. 39)

[259] Nathan Otterness and James H Anderson. "Amd gpus as an alternative to nvidia for supporting real-time workloads." In: *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2020 (see p. 39)

[260] Nikolay Kondratyuk, Vsevolod Nikolskiy, Daniil Pavlov, and Vladimir Stegailov. "GPU-accelerated molecular dynamics: State-of-art software performance and porting from Nvidia CUDA to AMD HIP." In: *The International Journal of High Performance Computing Applications* 35.4 (2021), pp. 312–324 (see p. 39)

[261] Jianbin Fang, Chun Huang, Tao Tang, and Zheng Wang. "Parallel programming models for heterogeneous many-cores: a comprehensive survey." In: *CCF Transactions on High Performance Computing* 2.4 (2020), pp. 382–400 (see pp. 39, 52)

[262] Yifan Sun, Saoni Mukherjee, Trinayan Baruah, Shi Dong, Julian Gutierrez, Prannoy Mohan, and David Kaeli. "Evaluating performance tradeoffs on the radeon open compute platform." In: *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE. 2018, pp. 209–218 (see p. 39)

[263] Tianji Wu, Bo Wang, Yi Shan, Feng Yan, Yu Wang, and Ningyi Xu. "Efficient pagerank and spmv computation on amd gpus." In: *2010 39th International Conference on Parallel Processing*. IEEE. 2010, pp. 81–89 (see p. 39)

[264] Takuma Nomizu, Daisuke Takahashi, Jinpil Lee, Taisuke Boku, and Mitsuhisa Sato. "Implementation of xcalablemp device acceleration extention with opencl." In: *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. IEEE. 2012, pp. 2394–2403 (see p. 39)

[265] B Neelima and Prakash S Raghavendra. "Recent trends in software and hardware for GPGPU computing: a comprehensive survey." In: *2010 5th International Conference on Industrial and Information Systems*. IEEE. 2010, pp. 319–324 (see pp. 39, 52)

[266] Ben Sander, Greg Stoner, Siu-chi Chan, WH Chung, and Robin Maffeo. "HCC: A C++ Compiler For Heterogeneous Computing." In: *HSA Foundation, Tech. Rep.* (2015) (see p. 39)

[267] Thomas Heller, Hartmut Kaiser, Patrick Diehl, Dietmar Fey, and Marc Alexander Schweitzer. "Closing the performance gap with modern c++." In: *International Conference on High Performance Computing*. Springer. 2016, pp. 18–31 (see p. 39)

[268] Dave Turner, Dan Andresen, Kyle Hutson, and Adam Tygart. "Application performance on the newest processors and GPUs." In: *Proceedings of the Practice and Experience on Advanced Research Computing*. 2018, pp. 1–7 (see p. 39)

[269] Michael Wong and Hal Finkel. "Distributed & Heterogeneous Programming in C++ for HPC at SC17." In: *Proceedings of the International Workshop on OpenCL*. 2018, pp. 1–7 (see p. 39)

[270] Michal Babej and Pekka Jääskeläinen. "HIPCL: Tool for Porting CUDA Applications to Advanced OpenCL Platforms Through HIP." In: *Proceedings of the International Workshop on OpenCL*. 2020, pp. 1–3 (see p. 39)

[271] Alexey Borisov and Evgeny Myasnikov. "Implementation of" Magma" and" Kuznyechik" ciphers using HIP." In: *2020 International Conference on Information Technology and Nanotechnology (ITNT)*. IEEE. 2020, pp. 1–5 (see p. 39)

[272] Seyyed Salar Latifi Oskouei, Hossein Golestani, Matin Hashemi, and Soheil Ghiasi. "Cndroid: Gpu-accelerated execution of trained deep convolutional neural networks on android." In: *Proceedings of the 24th ACM international conference on Multimedia*. 2016, pp. 1201–1205 (see p. 40)

[273] Hervé Guihot. "RenderScript." In: *Pro Android Apps Performance Optimization*. Springer, 2012, pp. 231–263 (see p. 40)

[274] Alejandro Acosta and Francisco Almeida. "Towards the optimal execution of Renderscript applications in Android devices." In: *Simulation Modelling Practice and Theory* 58 (2015), pp. 55–64 (see p. 40)

[275] Sebastian Kuckuk, Tobias Preclik, and Harald Köstler. "Interactive particle dynamics using opencl and kinect." In: *International Journal of Parallel, Emergent and Distributed Systems* 28.6 (2013), pp. 519–536 (see p. 40)

[276] Arturo Garcia, Jose Omar Alvizo Flores, Ulises Olivares Pinto, and Felix Ramos. "Fast Data Parallel Radix Sort Implementation in DirectX 11 Compute Shader to Accelerate Ray Tracing Algorithms." In: *EURASIA GRAPHICS: International Conference on Computer Graphics, Animation and Gaming Technologies*. 2014, pp. 27–36 (see p. 40)

[277] Gregory Gutmann, Daisuke Inoue, Akira Kakugo, and Akihiko Konagaya. "Parallel interaction detection algorithms for a particle-based live controlled real-time microtubule gliding simulation system accelerated by GPGPU." In: *New Generation Computing* 35.2 (2017), pp. 157–180 (see p. 40)

[278] Nicola Capodieci, Roberto Cavicchioli, and Andrea Marongiu. "A Taxonomy of Modern GPGPU Programming Methods: On the Benefits of a Unified Specification." In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2021) (see pp. 40, 52)

[279] Benedict Gaster, Lee Howes, David R Kaeli, Perhaad Mistry, and Dana Schaa. *Heterogeneous computing with openCL: revised openCL 1*. Newnes, 2012 (see p. 40)

[280] Sangmin Seo, Jun Lee, Gangwon Jo, and Jaejin Lee. "Automatic OpenCL work-group size selection for multicore CPUs." In: *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*. IEEE. 2013, pp. 387–397 (see p. 41)

[281] Thanh Tuan Dao, Jungwon Kim, Sangmin Seo, Bernhard Egger, and Jaejin Lee. "A performance model for GPUs with caches." In: *IEEE Transactions on Parallel and Distributed Systems* 26.7 (2014), pp. 1800–1813 (see p. 41)

[282] Andrew SD Lee and Tarek S Abdelrahman. "Launch-time optimization of OpenCL GPU kernels." In: *Proceedings of the General Purpose GPUs*. 2017, pp. 32–41 (see p. 41)

[283] Zheng Wang, Dominik Grewe, and Michael FP O'boyle. "Automatic and portable mapping of data parallel programs to opencl for gpu-based heterogeneous systems." In: *ACM Transactions on Architecture and Code Optimization (TACO)* 11.4 (2014), pp. 1–26 (see p. 41)

[284] Paul Harvey, Saji Hameed, and Wim Vanderbauwhede. "Accelerating Lagrangian particle dispersion in the atmosphere with OpenCL across multiple platforms." In: *Proceedings of the International Workshop on OpenCL 2013 & 2014*. 2014, pp. 1–8 (see p. 41)

[285] Naveen Suda, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-sun Seo, and Yu Cao. "Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks." In: *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2016, pp. 16–25 (see p. 41)

[286] Tanner Young-Schultz, Lothar Lilge, Stephen Brown, and Vaughn Betz. "Using OpenCL to enable software-like development of an FPGA-accelerated biophotonic cancer treatment simulator." In: *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2020, pp. 86–96 (see p. 41)

[287] Nagendra Gulur and Narayanan L Suriya. "Understanding the Performance Benefit of Asynchronous Data Transfers in OpenCL Programs Executing on Media Processors." In: *2015 IEEE 22nd International Conference on High Performance Computing (HiPC)*. IEEE. 2015, pp. 135–144 (see p. 41)

[288] Heikki Kultala, Timo Viitanen, Heikki Berg, Pekka Jääskeläinen, Joonas Multanen, Mikko Kokkonen, Kalle Raiskila, Tommi Zetterman, and Jarmo Takala. "LordCore: Energy-efficient OpenCL-programmable software-defined radio coprocessor." In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.5 (2019), pp. 1029–1042 (see p. 41)

[289] Saoni Mukherjee, Yifan Sun, Paul Blinzer, Amir Kavyan Ziabari, and David Kaeli. "A comprehensive performance analysis of HSA and OpenCL 2.0." In: *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE. 2016, pp. 183–193 (see p. 43)

[290] Benedict R Gaster and Lee Howes. "OpenCL C++." In: *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*. 2013, pp. 86–95 (see p. 43)

[291] Borja Pérez, José Luis Bosque, and Ramón Beivide. "Simplifying programming and load balancing of data parallel applications on heterogeneous systems." In: *Proceedings of the 9th Annual Workshop on General Purpose Processing using Graphics Processing Unit*. 2016, pp. 42–51 (see pp. 48, 54, 61–63, 72)

[292] Javier Diaz, Camelia Munoz-Caro, and Alfonso Nino. "A survey of parallel programming models and tools in the multi and many-core era." In: *IEEE Transactions on parallel and distributed systems* 23.8 (2012), pp. 1369–1386 (see p. 52)

[293] Sparsh Mittal and Jeffrey S Vetter. "A survey of CPU-GPU heterogeneous computing techniques." In: *ACM Computing Surveys (CSUR)* 47.4 (2015), pp. 1–35 (see p. 52)

[294] Branimir Pervan and Josip Knezović. "A Survey on Parallel Architectures and Programming Models." In: *2020 43rd International Convention on Information, Communication and Electronic Technology (MIPRO)*. IEEE. 2020, pp. 999–1005 (see p. 52)

[295] Roger D Chamberlain. "Architecturally truly diverse systems: A review." In: *Future Generation Computer Systems* 110 (2020), pp. 33–44 (see p. 52)

[296] Yuxiang Li and Zhiyong Zhang. "Parallel computing: review and perspective." In: *2018 5th International Conference on Information Science and Control Engineering (ICISCE)*. IEEE. 2018, pp. 365–369 (see p. 52)

[297] Nachiket Kapre and Samuel Bayliss. "Survey of domain-specific languages for FPGA computing." In: *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE. 2016, pp. 1–12 (see p. 52)

[298] Chengbin Fan, Hui Deng, Feng Wang, Shoulin Wei, Wei Dai, and Bo Liang. "A survey on task scheduling method in heterogeneous computing system." In: *2015 8th International Conference on Intelligent Networks and Intelligent Systems (ICINIS)*. IEEE. 2015, pp. 90–93 (see p. 52)

[299] Christoforos Kachris and Dimitrios Soudris. "A survey on reconfigurable accelerators for cloud computing." In: *2016 26th International conference on field programmable logic and applications (FPL)*. IEEE. 2016, pp. 1–10 (see p. 52)

[300] Michael Haidl, Michel Steuwer, Tim Humernbrum, and Sergei Gorlatch. "Multi-stage programming for GPUs in C++ using PACXX." In: *Proceedings of the 9th Annual Workshop on General Purpose Processing using Graphics Processing Unit*. 2016, pp. 32–41 (see p. 53)

[301] Michael Haidl and Sergei Gorlatch. "High-level programming for many-cores using C++ 14 and the STL." In: *International Journal of Parallel Programming* 46.1 (2018), pp. 23–41 (see p. 53)

[302] J. Szuppe. "Boost.Compute: A Parallel Computing Library for C++ Based on OpenCL." In: *Int. Workshop on OpenCL*. IWOCL '16. Vienna, Austria: ACM, 2016 (see p. 53)

[303] Moisés Viñas, Basilio B Fraguela, Diego Andrade, and Ramón Doallo. "Heterogeneous distributed computing based on high-level abstractions." In: *Concurrency and Computation: Practice and Experience* 30.17 (2018), e4664 (see p. 53)

[304] Johannes de Fine Licht, Michaela Blott, and Torsten Hoefler. "Designing scalable FPGA architectures using high-level synthesis." In: *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2018, pp. 403–404 (see p. 53)

[305] Erik Zenker, Benjamin Worpitz, René Widera, Axel Huebl, Guido Juckeland, Andreas Knüpfer, Wolfgang E Nagel, and Michael Bussmann. "Alpaka–An Abstraction Library for Parallel Kernel Acceleration." In: *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. 2016, pp. 631–640 (see p. 53)

[306] Eric Holk, Milinda Pathirage, Arun Chauhan, Andrew Lumsdaine, and Nicholas D Matsakis. "GPU programming in rust: Implementing high-level abstractions in a systems-level language." In: *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*. IEEE. 2013, pp. 315–324 (see p. 53)

[307] Kazuaki Ishizaki, Akihiro Hayashi, Gita Koblents, and Vivek Sarkar. "Compiling and Optimizing Java 8 Programs for GPU Execution." In: *2015 International Conference on Parallel Architecture and Compilation (PACT)*. 2015, pp. 419–431 (see p. 53)

[308] Michail Papadimitriou, Eleni Markou, Juan Fumero, Athanasios Stratikopoulos, Florin Blanaru, and Christos Kotselidis. "Multiple-tasks on multiple-devices (MTMD): exploiting concurrency in heterogeneous managed runtimes." In: *Proceedings of the 17th ACM SIGPLAN/SIGOPS international conference on virtual execution environments*. 2021, pp. 125–138 (see p. 53)

[309] Ioana Dogaru and Radu Dogaru. "Using Python and Julia for Efficient Implementation of Natural Computing and Complexity Related Algorithms." In: *2015 20th International Conference on Control Systems and Computer Science*. 2015, pp. 599–604 (see p. 53)

[310] Jae-Ho Lee, Hyun-Woo Cho, Chang-Hoon Jung, Dong-Hyun Kim, and Cheol-Hoon Lee. "WebCL prototype for high performance browser running on Android-powered mobile device." In: *2016 International Conference on Information and Communication Technology Convergence (ICTC)*. 2016, pp. 1039–1041 (see p. 53)

[311] Håvard H Holm, André R Brodtkorb, and Martin L Sætra. "GPU computing with Python: Performance, energy efficiency and usability." In: *Computation* 8.1 (2020), p. 4 (see p. 53)

[312] Martín Abadi. "TensorFlow: learning functions at scale." In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. 2016, pp. 1–1 (see p. 53)

[313] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. "Pytorch: An imperative style, high-performance deep learning library." In: *Advances in neural information processing systems* 32 (2019), pp. 8026–8037 (see p. 53)

[314] Thomas Heller, Patrick Diehl, Polytechnique Montreal Montreal, Canada Zachary Byerly, John Biddiscombe, Hartmut Kaiser, Zachary Byerly, and Hart-Mut Kaiser. "HPX – An open source C++ Standard Library for Parallelism and Concurrency." In: *2017 Workshop on Open Source Supercomputing* (2017) (see pp. 53, 54)

[315] Marcin Copik and Hartmut Kaiser. "Using sycl as an implementation framework for hpx. compute." In: *Proceedings of the 5th International Workshop on OpenCL*. 2017, pp. 1–7 (see pp. 53, 54)

[316] Patrick Daleiden, Andreas Stefik, and Philip Merlin Uesbeck. "GPU programming productivity in different abstraction paradigms: a randomized controlled trial comparing CUDA and thrust." In: *ACM Transactions on Computing Education (TOCE)* 20.4 (2020), pp. 1–27 (see p. 53)

[317] H Carter Edwards, Christian R Trott, and Daniel Sunderland. "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns." In: *Journal of parallel and distributed computing* 74.12 (2014), pp. 3202–3216 (see p. 53)

[318] David Beckingsale, Richard Hornung, Tom Scogland, and Arturo Vargas. "Performance portable C++ programming with RAJA." In: *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. 2019, pp. 455–456 (see p. 53)

[319] J. Enmyren and C. W. Kessler. "SkePU: A multi-backend skeleton programming library for multi-gpu systems." In: *Proc. 4th Int. Workshop on High-Level Parallel Programming and Applications* (Sept. 2010) (see pp. 53, 54)

[320] August Ernstsson, Lu Li, and Christoph Kessler. "SkePU 2: Flexible and type-safe skeleton programming for heterogeneous parallel systems." In: *International Journal of Parallel Programming* 46.1 (2018), pp. 62–80 (see pp. 53, 54)

[321] M. Steuwer, P. Kegel, and S. Gorlatch. "SkelCL - A portable skeleton library for high-level GPU programming." In: *IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum* May 2011 (2011), pp. 1176–1182 (see pp. 53, 54)

[322] Guilherme Andrade, Wilson de Carvalho, Renato Utsch, Pedro Caldeira, Alberto Alburquerque, Fabricio Ferracioli, Leonardo Rocha, Michael Frank, Dorgival Guedes, and Renato Ferreira. "ParallelME: A parallel mobile engine to explore heterogeneity in mobile computing architectures." In: *European Conference on Parallel Processing*. Springer. 2016, pp. 447–459 (see p. 53)

[323] Wilson de Carvalho, Guilherme Andrade, Pedro Caldeira, Renato Utsch, Renato Antônio Celso Ferreira, Leonardo Rocha, Renan Carvalho, and Millas Násser. "Exploring heterogeneous mobile architectures with a high-level programming model." In: *2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE. 2017, pp. 25–32 (see pp. 53, 54)

[324] Alejandro Duran, Eduard Ayguadé, Rosa M. Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. "OmpSs: A proposal for programming heterogeneous multi-core architectures." In: *Parallel Processing Letters* 21.02 (2011), pp. 173–193 (see p. 53)

[325] Akihiro Tabuchi, Hitoshi Murai, Masahiro Nakao, Tetsuya Odajima, and Taisuke Boku. "XcalableACC: An Integration of XcalableMP and OpenACC." In: *XcalableMP PGAS Programming Language*. Springer, Singapore, 2021, pp. 123–146 (see p. 53)

[326] Didem Unat, Xing Cai, and Scott B Baden. "Mint: realizing CUDA performance in 3D stencil methods with annotated C." In: *Proceedings of the international conference on Supercomputing*. 2011, pp. 214–224 (see p. 53)

[327] Jiang-Zhou He, Wen-Guang Chen, Guang-Ri Chen, Wei-Min Zheng, Zhi-Zhong Tang, and Han-Dong Ye. "OpenMDSP: Extending OpenMP to Program Multi-Core DSPs." In: *Journal of Computer Science and Technology* 29.2 (2014), pp. 316–331 (see p. 53)

[328] Satya P Jammy, Gihan R Mudalige, Istvan Z Reguly, Neil D Sandham, and Mike Giles. "Block-structured compressible Navier–Stokes solution using the OPS high-level abstraction." In: *International Journal of Computational Fluid Dynamics* 30.6 (2016), pp. 450–454 (see p. 53)

[329] Denis Demidov. "AMGCL: An efficient, flexible, and extensible algebraic multigrid implementation." In: *Lobachevskii Journal of Mathematics* 40.5 (2019), pp. 535–546 (see p. 53)

[330] Patricia Suriana, Andrew Adams, and Shoaib Kamil. "Parallel associative reductions in halide." In: *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2017, pp. 281–291 (see p. 53)

[331] Bradford L Chamberlain, Steve Deitz, Mary Beth Hribar, and Wayne Wong. "Chapel." In: *Programming Models for Parallel Computing* (2015), pp. 129–159 (see p. 53)

[332] Anchu Rajendran and V Krishna Nandivada. "DisGCo: A Compiler for Distributed Graph Analytics." In: *ACM Transactions on Architecture and Code Optimization (TACO)* 17.4 (2020), pp. 1–26 (see p. 53)

[333] Richard Membarth, Oliver Reiche, Frank Hannig, Jürgen Teich, Mario Körner, and Wieland Eckert. "Hipa cc: A domain-specific language and compiler for image processing." In: *IEEE Transactions on Parallel and Distributed Systems* 27.1 (2015), pp. 210–224 (see p. 54)

[334] Joshua Auerbach, David F Bacon, Perry Cheng, and Rodric Rabbah. "Lime: a java-compatible and synthesizable language for heterogeneous architectures." In: *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*. 2010, pp. 89–108 (see p. 54)

[335] Christophe Dubach, Perry Cheng, Rodric Rabbah, David F Bacon, and Stephen J Fink. "Compiling a high-level language for GPUs: (via language support for architectures and compilers)." In: *ACM SIGPLAN Notices* 47.6 (2012), pp. 1–12 (see p. 54)

[336] Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo, and Jaejin Lee. "SnuCL: an OpenCL framework for heterogeneous CPU/GPU clusters." In: *Proceedings of the 26th ACM international conference on Supercomputing*. 2012, pp. 341–352 (see p. 54)

[337] Yi-Ping You, Hen-Jung Wu, Yeh-Ning Tsai, and Yen-Ting Chao. "VirtCL: a framework for OpenCL device abstraction and management." In: *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2015, pp. 161–172 (see pp. 54, 55)

[338] Jaehoon Jung, Daeyoung Park, Gangwon Jo, Jungho Park, and Jaejin Lee. "SnuRHAC: A Runtime for Heterogeneous Accelerator Clusters with CUDA Unified Memory." In: *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*. 2021, pp. 107–120 (see p. 54)

[339] Ashwin M Aji, Antonio J Peña, Pavan Balaji, and Wu-chun Feng. "MultiCL: Enabling automatic scheduling for task-parallel workloads in OpenCL." In: *Parallel Computing* 58 (2016), pp. 37–55 (see pp. 54, 55)

[340] Pierre Huchant. "Static Analysis and Dynamic Adaptation of Parallelism." PhD thesis. Université de Bordeaux, 2019 (see p. 54)

[341] Sylvain Henry, Alexandre Denis, Denis Barthou, Marie-Christine Counilh, and Raymond Namyst. "Toward OpenCL automatic multi-device support." In: *European Conference on Parallel Processing*. Springer. 2014, pp. 776–787 (see p. 54)

[342] Prasanna Pandit and R Govindarajan. "Fluidic kernels: Cooperative execution of opencl programs on multiple heterogeneous devices." In: *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. 2014, pp. 273–283 (see pp. 54, 55, 61)

[343] Azzam Haidar, Chongxiao Cao, Asim Yarkhan, Piotr Luszczek, Stanimire Tomov, Khairul Kabir, and Jack Dongarra. "Unified development for mixed multi-gpu and multi-coprocessor environments using a lightweight runtime environment." In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE. 2014, pp. 491–500 (see pp. 54, 55)

[344] Lanjun Wan, Weihua Zheng, and Xinpan Yuan. "HCE: A Runtime System for Efficiently Supporting Heterogeneous Cooperative Execution." In: *IEEE Access* 9 (2021), pp. 147264–147279 (see p. 54)

[345] Bryan Donyanavard, Tiago Mück, Santanu Sarma, and Nikil Dutt. "SPARTA: Runtime task allocation for energy efficient heterogeneous manycores." In: *2016 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*. IEEE. 2016, pp. 1–10 (see p. 54)

[346] Tarun Beri, Sorav Bansal, and Subodh Kumar. "The Unicorn Runtime: Efficient distributed shared memory programming for hybrid CPU-GPU clusters." In: *IEEE Transactions on Parallel and Distributed Systems* 28.5 (2016), pp. 1518–1534 (see pp. 54, 55)

[347] Thierry Gautier, Joao VF Lima, Nicolas Maillard, and Bruno Raffin. "Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures." In: *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE. 2013, pp. 1299–1308 (see pp. 54, 55)

[348] Janghaeng Lee, Mehrzad Samadi, and Scott Mahlke. "Orchestrating multiple data-parallel kernels on multiple devices." In: *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE. 2015, pp. 355–366 (see pp. 54, 55)

[349] Jianlong Zhong and Bingsheng He. "Kernelet: High-throughput GPU kernel executions with dynamic slicing and scheduling." In: *IEEE Transactions on Parallel and Distributed Systems* 25.6 (2013), pp. 1522–1532 (see pp. 54, 55)

[350] Javier Cabezas, Isaac Gelado, John E Stone, Nacho Navarro, David B Kirk, and Wen-mei Hwu. "Runtime and architecture support for efficient data exchange in multi-accelerator applications." In: *IEEE Transactions on Parallel and Distributed Systems* 26.5 (2014), pp. 1405–1418 (see p. 54)

[351] Jungwon Kim, Honggyu Kim, Joo Hwan Lee, and Jaejin Lee. "Achieving a single compute device image in OpenCL for multiple GPUs." In: *ACM Sigplan Notices* 46.8 (2011), pp. 277–288 (see pp. 54, 55, 61)

[352] Pierre Huchant, Marie-Christine Counilh, and Denis Barthou. "Automatic opencl task adaptation for heterogeneous architectures." In: *European conference on parallel processing*. Springer. 2016, pp. 684–696 (see pp. 54, 55, 61)

[353] Deives Kist, Bruno Pinto, Rodrigo Bazo, Andre Rauber Du Bois, and Gerson Geraldo H Cavalheiro. "Kanga: a skeleton-based generic interface for parallel programming." In: *2015 International Symposium on Computer Architecture and High Performance Computing Workshop (SBAC-PADW)*. IEEE. 2015, pp. 68–72 (see p. 54)

[354] Luciano Baresi, Giovanni Quattrocchi, and Nicholas Rasi. "Resource Management for TensorFlow Inference." In: *International Conference on Service-Oriented Computing*. Springer. 2021, pp. 238–253 (see p. 54)

[355] Pekka Jääskeläinen, John Glossner, Martin Jambor, Aleksi Tervo, and Matti Rintala. "Offloading C++ 17 Parallel STL on System Shared Virtual Memory Platforms." In: *International Conference on High Performance Computing*. Springer. 2018, pp. 637–647 (see p. 54)

[356] Jakub Szuppe. "Boost. Compute: A parallel computing library for C++ based on OpenCL." In: *Proceedings of the 4th International Workshop on OpenCL*. 2016, pp. 1–39 (see p. 54)

[357] Ajai V George, Sankar Manoj, Sanket R Gupte, Sayantan Mitra, and Santonu Sarkar. "Thrust++: Extending thrust framework for better abstraction and performance." In: *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*. IEEE. 2017, pp. 368–377 (see p. 54)

[358] Jeff R Hammond, Michael Kinsner, and James Brodman. "A comparative analysis of Kokkos and SYCL as heterogeneous, parallel programming models for C++ applications." In: *Proceedings of the International Workshop on OpenCL*. 2019, pp. 1–2 (see p. 54)

[359] Marcin Copik and Hartmut Kaiser. "Using SYCL As an Implementation Framework for HPX.Compute." In: *Proceedings of the 5th International Workshop on OpenCL*. IWOCL 2017. New York, NY, USA: ACM, 2017, 30:1–30:7 (see p. 54)

[360] David del Rio Astorga, Manuel F Dolz, Javier Fernández, and J Daniel García. "A generic parallel pattern interface for stream and data processing." In: *Concurrency and Computation: Practice and Experience* 29.24 (2017), e4175 (see p. 54)

[361] Michel Steuwer, Malte Friese, Sebastian Albers, and Sergei Gorlatch. "Introducing and implementing the allpairs skeleton for programming multi-GPU systems." In: *International Journal of Parallel Programming* 42.4 (2014), pp. 601–618 (see p. 54)

[362] Ana Moreton-Fernandez, Arturo Gonzalez-Escribano, and Diego R Llanos. "Multi-device controllers: a library to simplify parallel heterogeneous programming." In: *International Journal of Parallel Programming* 47.1 (2019), pp. 94–113 (see p. 54)

[363] Gabriel Rodriguez-Canal, Yuri Torres, Francisco J Andújar, and Arturo Gonzalez-Escribano. "Efficient heterogeneous programming with FPGAs using the Controller model." In: *The Journal of Supercomputing* (2021), pp. 1–16 (see p. 54)

[364] Kai Ma, Xue Li, Wei Chen, Chi Zhang, and Xiaorui Wang. "Greengpu: A holistic approach to energy efficiency in gpu-cpu heterogeneous architectures." In: *2012 41st international conference on parallel processing*. IEEE. 2012, pp. 48–57 (see p. 54)

[365] Guibin Wang and Xiaoguang Ren. "Power-efficient work distribution method for cpu-gpu heterogeneous system." In: *International symposium on parallel and distributed processing with applications*. IEEE. 2010, pp. 122–129 (see p. 54)

[366] Lanjun Wan, Weihua Zheng, and Xinpan Yuan. "Efficient inter-device task scheduling schemes for multi-device co-processing of data-parallel kernels on heterogeneous systems." In: *IEEE Access* 9 (2021), pp. 59968–59978 (see p. 55)

[367] Jiajian Xiao, Philipp Andelfinger, Wentong Cai, Paul Richmond, Alois Knoll, and David Eckhoff. "OpenABLext: An automatic code generation framework for agent-based simulations on CPU-GPU-FPGA heterogeneous platforms." In: *Concurrency and Computation: Practice and Experience* 32.21 (2020), e5807 (see p. 55)

[368] Mr Yasir Noman Khalid. "Load-Balanced Multi-Job Scheduling For Heterogeneous CPU-GPU Systems." PhD thesis. CAPITAL UNIVERSITY, 2020 (see p. 55)

[369] Anes Abdennebi, Anıl Elakaş, Fatih Taşyaran, Erdinç Öztürk, Kamer Kaya, and Sinan Yıldırım. "Machine learning-based load distribution and balancing in heterogeneous database management systems." In: *Concurrency and Computation: Practice and Experience* (2021), e6641 (see p. 55)

[370] Asad Hayat. "A Load-Balanced Task Scheduler for Heterogeneous Systems based on Machine Learning." PhD thesis. CAPITAL UNIVERSITY, 2021 (see p. 55)

[371] Anuj Vaishnav. *Modular FPGA Systems with Support for Dynamic Workloads and Virtualisation*. The University of Manchester (United Kingdom), 2020 (see p. 55)

[372] Anuj Vaishnav, Khoa Dang Pham, and Dirk Koch. "Heterogeneous resource-elastic scheduling for CPU+ FPGA architectures." In: *Proceedings of the 10th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*. 2019, pp. 1–6 (see p. 55)

[373] S Carlos, Pablo Toharia, Jose Luis Bosque, and Oscar D Robles. "Static multi-device load balancing for opencl." In: *2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications*. IEEE. 2012, pp. 675–682 (see p. 55)

[374] Klaus Kofler, Ivan Grasso, Biagio Cosenza, and Thomas Fahringer. "An automatic input-sensitive approach for heterogeneous task partitioning." In: *Proceedings of the 27th international ACM conference on International conference on supercomputing*. 2013, pp. 149–160 (see p. 55)

[375] Kyle Spafford, Jeremy Meredith, and Jeffrey Vetter. "Maestro: data orchestration and tuning for opencl devices." In: *European Conference on Parallel Processing*. Springer. 2010, pp. 275–286 (see p. 55)

[376] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. "Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping." In: *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2009, pp. 45–55 (see p. 55)

[377] Siham Tabik, G Ortega, Ester M Garzón, and D Suárez. "A data partitioning model for highly heterogeneous systems." In: *European Conference on Parallel Processing*. Springer. 2016, pp. 468–479 (see p. 55)

[378] Angeles Navarro, Antonio Vilches, Francisco Corbera, and Rafael Asenjo. "Strategies for maximizing utilization on multi-CPU and multi-GPU heterogeneous architectures." In: *The Journal of Supercomputing* 70.2 (2014), pp. 756–771 (see p. 55)

[379] Alberto Cabrera, Alejandro Acosta, Francisco Almeida, and Vicente Blanco. "A heuristic technique to improve energy efficiency with dynamic load balancing." In: *The Journal of Supercomputing* 75.3 (2019), pp. 1610–1624 (see p. 55)

[380] Erich Gamma, Ralph Johnson, Richard Helm, Ralph E Johnson, John Vlissides, et al. *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH, 1995 (see pp. 66, 104, 120, 123)

[381] Maria Angelica Davila Guzman, Ruben Gran Tejero, Maria Villarroya Gaudo, and Dario Suarez Gracia. "Towards the inclusion of FPGAs on commodity heterogeneous systems." In: *2018 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE. 2018, pp. 554–556 (see pp. 70, 159)

[382] Cleidson R.B. De Souza and David L.M. Bentolila. "Automatic evaluation of API usability using complexity metrics and visualizations." In: *2009 31st International Conference on Software Engineering - Companion Volume, ICSE 2009* (2009), pp. 299–302 (see p. 79)

[383] Rajendra K. Bandi, Vijay K. Vaishnavi, and Daniel E. Turk. "Predicting maintenance performance using object-oriented design complexity metrics." In: *IEEE Transactions on Software Engineering* 29.1 (2003), pp. 77–87 (see p. 79)

[384] Girish Maskeri Rama and Avinash Kak. "Some structural measures of API usability." In: *Software - Practice and Experience* (2013) (see p. 79)

[385] Thomas Scheller and Eva Kühn. "Automated measurement of API usability: The API Concepts Framework." In: *Information and Software Technology* 61 (2015), pp. 145–162 (see p. 79)

[386] Alejandro Acosta and Francisco Almeida. "Paralldroid: Performance analysis of gpu executions." In: *European Conference on Parallel Processing*. Springer. 2014, pp. 387–399 (see p. 131)

[387] Alejandro Acosta and Francisco Almeida. "Android TM development and performance analysis." In: *The Journal of Supercomputing* 70.2 (2014), pp. 649–659 (see p. 131)

[388] Sergio Afonso, Alejandro Acosta, and Francisco Almeida. "Automatic Generation of OpenCL Code for ARM Architectures." In: *European Conference on Parallel Processing*. Springer. 2016, pp. 96–107 (see p. 131)

[389] Daniel Stadelmann, M. Teßmann, and Ch. Schiedermeier. "Entwicklung einer OpenCL-Implementierung für die VideoCore IV GPU des Raspberry Pi." ("Development of an OpenCL implementation for the VideoCore IV GPU of the Raspberry Pi") Bachelor's Thesis. TH Nürnberg Georg Simon Ohm, Nürnberg, Germany, 2017 (see p. 131)

[390] Diego García Cosío, Jose Luis Bosque, and Raúl Nozal. "Herramienta para el procesamiento de audio en tiempo real mediante OpenCL." ("Tool for real-time audio processing using OpenCL") Bachelor's Thesis. Universidad de Cantabria, Santander, Spain, 2020 (see p. 132)

[391] Daniel Torre Miguel, Raúl Nozal, and Jose Luis Bosque. "Procesamiento de vídeo en dispositivos embebidos mediante OpenCL." ("Video processing on embedded devices using OpenCL") Bachelor's Thesis. Universidad de Cantabria, Santander, Spain, 2020 (see p. 132)

[392] Raúl Nozal and Jose Luis Bosque. "Straightforward Heterogeneous Computing with the oneAPI Coexecutor Runtime." In: *Electronics* 10.19 (2021) (see p. 150)

List of top citations

List of the most influential journals and conferences in this dissertation, based on the number of distinct papers cited throughout the document. Only those containing at least three relevant citations are shown.

[16] The Journal of Supercomputing (JoS)

[16] International Workshop on OpenCL and SYCL Conference (IWOCL & SYCLcon)

[12] IEEE Transactions on Parallel & Distributed Systems (TPDS)

[12] European Conference on Parallel Processing (Euro-Par)

[8] IEEE International Parallel & Distributed Processing Symposium Workshops (IPDPSW)

[7] ACM Transactions on Architecture & Code Optimization (TACO)

[6] IEEE/ACM International Symposium on Microarchitecture (MICRO)

[6] Journal of Parallel & Distributed Computing (JPDC)

[6] International Conference on Parallel Processing (ICPP)

[6] IEEE International Conference on High Performance Computing (HiPC)

[5] International Conference on Parallel Architecture & Compilation Techniques (PACT)

[5] ACM International Conference on Supercomputing (ICS)

[4] ACM/IEEE International Symposium on Computer Architecture (ISCA)

[4] IEEE International Parallel & Distributed Processing Symposium (IPDPS)

[4] IEEE Int. Symposium on Computer Architecture & High Performance Computing (SBAC-PAD)

[4] Concurrency and Computation: Practice and Experience (CCPE)

[4] International Journal of Parallel Programming (IJPP)

[3] Future Generation Computer Systems (FGCS)

[3] International Conference on High Performance Computing & Simulation (HPCS)

[3] IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)

[3] ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP)

[3] International Conference on High Performance Computing in Asia-Pacific Region (HPC Asia)

[3] ACM SIGPLAN Notices

[3] International Conference for High Performance Computing, Networking, Storage & Analysis (SC)

Optimizing Performance and Energy Efficiency in Massively Parallel Systems

by **Raúl Nozal González**

KEYWORDS

Heterogeneous Computing · Co-execution · HPC · Parallel Programming · Performance Portability · Accelerators · Runtime Systems · Usability · Maintainability · Load Balancing · Scheduling · C++ · OpenCL · Intel oneAPI · SYCL · API Design · Software Architecture · Programming Languages

COLOPHON

This document is typeset using Donald E. Knuth's TEX typesetting system, through `LuaTex`[1] engine by Taco Hoekwater *et al.* implemented in Christian Schenk's `MiKTeX`[2]. Its template is developed over `KOMA-Script srcbook`[3] document class maintained by Markus Kohm. The bibliography is organized with Oliver Kopp's `JabRef`[4] and processed by `BibLaTeX`[5] using `Biber`[6] as its backend, which are mainly maintained by Philip Kime and François Charette respectively. The typeface used for regular text is Robert Slimbach's Minion Pro.[7] Sans-serif text is written in Slimbach and Carol Twombly's Myriad Pro.[8] Jim Lyles's `Vera Mono`[9] and Claudio Beccari's *Asana Math*[10] fonts are used for `monospaced` and *mathematical* text respectively.

All the graphics and schemes are self-made, designed using `Inkscape`[11], `Draw.io`[12] and `Gimp`[13]. Charts are created with a custom self-made low-level library on top of `d3`[14], although manual post-processing is applied to improve the overall quality and comprehensibility.

[1] `http://www.luatex.org`
[2] `https://miktex.org`
[3] `https://www.ctan.org/pkg/koma-script`
[4] `https://www.jabref.org`
[5] `https://www.ctan.org/pkg/biblatex`
[6] `https://www.ctan.org/pkg/biber`
[7] `https://fonts.adobe.com/fonts/minion`
[8] `https://fonts.adobe.com/fonts/myriad`
[9] `https://www.dafont.com/es/bitstream-vera-mono.font`
[10] `https://www.ctan.org/tex-archive/fonts/Asana-Math`
[11] `https://inkscape.org`
[12] `https://draw.io`; `https://app.diagrams.net`
[13] `https://www.gimp.org`
[14] `https://d3js.org`

FUNDING ACKNOWLEDGEMENTS

This PhD has been supported by the Spanish Ministry of Education (FPU16/03299 grant), the Spanish Science and Technology Commission under contracts TIN2016-76635-C2-2-R and PID2019-105660RB-C22.

This work has also been partially supported by the *Mont-Blanc 3: European Scalable and Power Efficient HPC Platform based on Low-Power Embedded Technology* project (G.A. No. 671697) from the European Union's Horizon 2020 Research and Innovation Programme (H2020 Programme). Some activities have also been funded by the Spanish Science and Technology Commission under contract TIN2016-81840-REDT (CAPAP-H6 network).

The *Integration II: Hybrid programming models* of Chapter 4 has been partially performed under the Project HPC-EUROPA3 (INFRAIA-2016-1-730897), with the support of the EC Research Innovation Action under the H2020 Programme. In particular, the author gratefully acknowledges the support of the SPMT Department of the High Performance Computing Center Stuttgart (HLRS).

Raúl Nozal is a Computer Engineer and Topography Engineer, expert in Software Architecture. He obtained these degrees by three different universities, *Universidad de Cantabria* (*UNICAN*), *Universidad del País Vasco* (*UPV/EHU*) and *Universidad a Distancia de Madrid* (*UDIMA*), achieving extraordinary awards for the best record in all of them. His professional experience ranges from Senior Developer to Software Architect and CTO, as a result of working in the industry for more than 9 years. He has mentored various business projects and R&D software, being co-founder of three innovative and technology companies, currently acting as advisor for some of them. For the last 5-years he has been working on his PhD with the Computer Architecture and Technology Group of the Department of Computer Engineering and Electronics, part of the University of Cantabria. He enjoys guiding projects and master theses, as well as teaching parallel programming to computer science students. His research has a multi-objective approach, combining optimization and performance with software engineering. His main research interests include programming languages, heterogeneous systems, high performance computing, web technologies and software architecture.

A tribute to Joe Armstrong

In memory of one of the main designers of the Erlang programming language and its Erlang/OTP runtime system.
His ambitious PhD has fostered one of the most important platforms to develop scalable, concurrent and fault-tolerant distributed systems.

UNIVERSITY OF CANTABRIA

Doctoral Thesis

Optimizing Performance and Energy Efficiency in Massively Parallel Systems

Raúl Nozal

supervised by

José Luis Bosque

Doctor of Philosophy

Computer Architecture and Technology Group

Department of Computer Engineering and Electronics

Santander

January 2017 – November 2021