Journal of Supercomputing manuscript No.
(will be inserted by the editor)

Cooperative CPU, GPU and FPGA Heterogeneous Execution with EngineCL

María Angélica Dávila Guzmán · Raúl Nozal · Rubén Gran Tejero · María Villarroya-Gaudó · Darío Suárez Gracia · Jose Luis Bosque

Received: date / Accepted: date

Abstract In heterogeneous systems, load balancing policies allow acceleration of tasks by distributing work among devices, thus delivering performance and energy efficiency. However, a key challenge that remains is programmability; specifically, releasing the programmer from the burden of managing data and devices with different architectures.

To this end, we extend EngineCL, a high-level framework built on top of OpenCL to support FPGA devices. Our proposal fully integrates FPGAs into the framework, enabling effective cooperation between CPU, GPU, and FPGA devices. With command overlapping and judicious data management, our work improves performance by up to 96% compared with single device execution and delivers energy-delay gains of up to 36%. Besides, adopting FPGAs does not require programmers to make big changes in their applications because the extensions do not modify the user-facing interface of EngineCL.

Keywords Heterogeneous scheduling · FPGA · Load Balancing · OpenCL

1 Introduction

Moore's Law and Dennard Scaling have driven the astonishing improvement of general purpose processors, CPUs, for the last decades. Their decline has helped the flourishing of heterogeneous systems, promising better performance and energy efficiency [10].

Such heterogeneous systems are often comprised of a CPU and an accelerator, with GPU being the most widely used accelerator. GPUs have delivered

M. A. Dávila Guzmán, M. Villarroya-Gaudó, R. Gran Tejero & D. Suárez Gracia
Universidad de Zaragoza, Spain
E-mail: {angelicadg,maria.villarroya,rgran,dario}@unizar.es

R. Nozal & J. L. Bosque
Universidad de Cantabria, Spain
E-mail: {raul.nozal, bosquejl}@unican.es

excellent performance for gamut of application domains including high performance computing. Further, a rich software ecosystem enables programmers to adopt them without suffering a high entry barrier. However, GPUs require substantial power dissipation that can be unaffordable in many environments. As a result, other devices such as FPGAs have emerged as complementary accelerators.

In comparison to other accelerators, FPGAs can provide better performance to power ratio. The down side, however, is that application development on FPGAs requires knowledge of digital design, which often is the main obstacle preventing their broad adoption by programmers. To mitigate this problem, high-Level synthesis frameworks with languages like C, C++, or OpenCL have emerged to improve programmer's productivity [19].

To program applications for heterogeneous systems, OpenCL provides an open standard supporting a general-purpose parallel programming model. It defines low-level primitives and core functionalities without masking the hardware architecture and allows portability across devices [28]. While OpenCL enables parallel execution between accelerator devices, it does not provide any support for load balancing between them. Load balancing is critical in order to minimize execution time in heterogeneous systems. This problem has been extensively studied, specifically in the context of two device systems; CPU coupled with either a GPU, a Xeon Phi, or an FPGA [7, 15–17, 22, 24].

EngineCL is a high-level framework that provides scheduling and data management primitives on top of OpenCL, easing the programmability of heterogeneous systems [21]. While EngineCL has been successful in systems coupling a CPU with either a GPU or Xeon Phi device, it does not yet provide support for FPGAs. As FPGAs have already been deployed in HPC systems, and furthermore, future systems will probably integrate more and more accelerators on a single die [9], it is going to be increasingly important for high level frameworks to provide efficient support for FPGAs.

This paper significantly extends EngineCL to provide FPGA support and load balance *parallel_for* constructs among both CPU, GPU, and FPGA, so that programmers can improve performance and energy efficiency without dealing the complexities of cooperative execution and device management. Such transparent cooperative execution has entailed a substantial number of modifications in the design and implementation of EngineCL. These include the communication mechanism between the host and the FPGA, how arguments are passed to the kernel, support for different kernels for each device, and the queuing system to overlap computation and communication.

Our experimental results show that for all the scenarios under consideration, heterogeneous systems deliver significant performance over using the fastest available device. This conclusion even holds when the heterogeneous system is comprised of unbalanced devices. This is crucial in presence of FPGAs for kernel performance can dramatically vary depending on the kernel implementation [31]. The average improvement using the best balancing algorithm is 58.1%. On the other hand, it should be noted that the improvements in performance are

not always followed by a reduction in energy consumption, for energy efficiency strongly depends on both the devices and the benchmark.

The main contributions of this paper are the following:

- We extend EngineCL to add FPGA support without changing user-facing APIs, so that FPGAs can effectively cooperate with other hardware accelerators in the co-execution of a single massive data-parallel kernel.
- We improve the implementation of EngineCL for FPGAs in several aspects via addition of: (a) host-device synchronization, (b) command queues management, (c) mechanisms to better overlap computation with communication, and (c) runtime workers to allow task kernels.
- We carry out an exhaustive evaluation of the both co-execution and the load-balancing algorithms on a 3-device heterogeneous platform. In these experiments three different metrics have been evaluated: performance, energy consumption, and energy-delay product.

The rest of the paper is organized as follows. Section 2 presents previous work. After that, Section 3 introduces the EngineCL runtime. Section 4 describes our approach, while Section 5 presents the methodology that we use to obtain the results showed in Section 6. Finally, Section 7 concludes the work.

2 Related Work

High-Level Synthesis (HLS) has enabled to widen the programmers audience for FPGA and its inclusion in heterogeneous systems [14, 20].

Many different applications have benefited from heterogeneous execution in a plethora of systems; e.g., DNA/RNA alignment on a CPU+GPU system [8], graph analytics on a CPU+FPGA system [4]. Even a fully heterogeneous system, CPU+GPU+FPGA, has been proposed for accelerating a real-time location problem and a pipeline HPC application [5, 27].

Load balancing is a challenging aspect of heterogeneous computing that has been widely addressed. When benchmark behavior is defined and/or remains constant, static scheduling tuned for the application tends to provide the best results. Tsoi *et al.* divide the problem between devices with an analytic model [29]. However, static load balancers require an exploration phase, and they do not adapt to unexpected changes on application throughput, which can lead to load unbalance inefficiencies.

Dynamic balancers face the imbalance problem, but at the cost of potential penalties due to load balancer activity. Pandit and Govindarajan presented FluidiCL where a CPU and GPU work on a shared iteration space, and each device starts from the beginning and end of the iteration space, respectively [24]. In order to avoid load balancer penalties, Qilin, HDSS, and Concord propose to calculate the computational speed of each accelerator at runtime and then assign a single chunk of work to each accelerator [6, 12, 16]. While Qilin relies on a trained-database that provides execution-time projection for all the programs it has ever executed, HDSS and Concord rely on a brief exploration phase that

computes the relative computational speed of each device. The weakness of these proposals is that they cannot be adapted to irregular applications that are addressed by the adaptive schedulers like LogFit and H-guided [22, 23, 25, 30].

MKMD maps multiple kernel into multiple devices in a two-phase approach, the first phase assigns kernels to devices and the second enables work-group level partitioning to keep all devices busy [15]. Industry solutions include Intel TBB, supporting GPU offloading with OpenCL [13], or Qualcomm Heterogeneous Compute SDK, supporting GPU and DSP offloading [3]. For an ample overview of load balancing techniques, please refer to Mittal and Vetter [17].

In comparison to previous works, ours is the first proposal to face the load balancing problem for the *parallel_for* paradigm on a heterogeneous platform composed by three accelerators: CPU+GPU+FPGA.

3 EngineCL Description

EngineCL is a runtime acting as an OpenCL C++ wrapper to simplify the programming of heterogeneous devices and squeeze their performance out [21]. It is specially designed to be used in large data-parallel applications and provides three load balancing algorithms. EngineCL hides the underlying hardware details by considering a single virtual device to operate with and divides a single task among all the real devices based on the load balancing algorithm selected by the programmer.

EngineCL has been designed around three pillars: OpenCL, usability, and performance.While OpenCL allows code portability on different devices, the programmer is responsible for managing many concepts related to the architecture, such as platforms, devices, contexts, buffers, queues, kernels and arguments, data transfers and error control sections. As the number of devices and operations increases, the code grows quickly with OpenCL, decreasing the productivity and increasing the maintainability effort. EngineCL solves these issues by providing a runtime with a higher-level API that manages all the OpenCL resources of the underlying system independently.

The runtime follows Architectural Principles with well known Design Patterns to strengthen its flexibility. EngineCL is layered in three tiers (see Fig. 1): Tier-1 and 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. The Tier-2 should be accessed if the programmer wants to select a specific device and provide a specialized kernel or use more specific options. The Tier-3 consists of the hidden inner parts that allow a flexible system regarding memory management, pluggable schedulers, work distribution, high concurrency and OpenCL encapsulation.

EngineCL provides high external usability and internal adaptability to support new runtime features, such as new schedulers, device types or communication-computation overlapping strategies. This is accomplished through a layered architecture and a set of core modules well profiled and encapsulated.

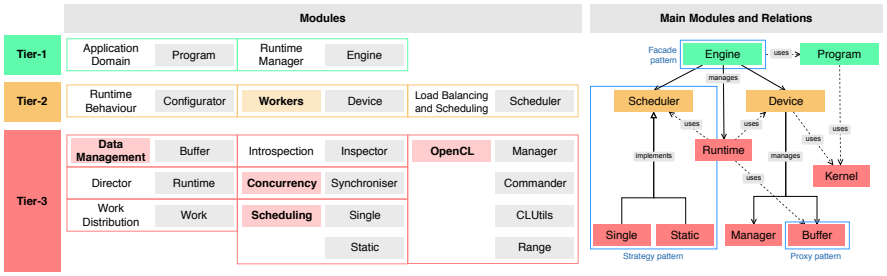

Fig. 1: EngineCL architecture: tiers, modules and applied patterns. The highlighted modules are extended to support FPGAs.

A set of well-known load balancing algorithms, described below, are provided [26]. The programmer should decide which one to use in each case, depending on the characteristics of the application and the architecture.

- **Static** This algorithm splits the data-set in as many packages as devices are in the system, proportionally to their computing capabilities. This division relies on knowing the percentage of workload assigned to each device in advance, and therefore the execution time between the devices is equalized. It minimizes the number of synchronization points, therefore, it performs well when facing regular loads. However, it is not adaptable, so its performance might not be as good with irregular loads.
- **Dynamic** It divides the data-set in packages of equal size, much more than the number of devices. A master thread in the host assigns packages to the different devices, including the CPU. This algorithm adapts to the irregular behavior of some applications. However, each package represents a synchronization point between the device and the host, where data are exchanged and a new package is launched.
- **HGuided** The Heterogeneous Guided algorithm is an attempt to reduce the synchronization points of the Dynamic, while retaining its adaptiveness. It makes larger packages at the beginning and reduces the size of the subsequent ones as the execution progresses, until the minimum package size, given as a parameter, is reached. Furthermore, the size of the packet is weighted by the computing power of each device, defined as the amount of work that this device can complete in a time span. This adjusts the number of packets to achieve a more accurate load balancing than with all other algorithms.

  The size of the package for device i is calculated as follows:

$$packet\_size\_H = min\Big(Min\_pacakage\_size, \lfloor \frac{G_r P_i}{k \sum_{j=1}^{n} P_j} \rfloor\Big) \qquad (1)$$

  Where G_r is the number of pending work-groups in each launch, P_i is the computing power of the device i. Finally, k is a constant, between 2 and 3, and the smaller k, the faster decreases the packet size. This avoids too big package sizes when there are few devices.

4 Coupling FPGA to EngineCL

Compared to other accelerators, FPGAs require a special work-flow for its integration into EngineCL. For running into a FPGA, an OpenCL kernel has to be synthesized off-line with a High Level Synthesis tool, such as Xilinx SDSoC or Intel/Altera OpenCL SDK [1, 2]. HLS tools appeared to help the adoption of FPGAs because they release programmers from learning complex hardware description languages, such as Verilog or VHDL. In fact, many of the HLS tools translate a high level programming language to a hardware description language, and then synthesize the generated code.

By using OpenCL, most of the boilerplate code (platform, device, context, buffers) can be shared among accelerators, which simplifies the FPGA integration into EngineCL. The main difference between FPGAs and the rest of accelerators (GPU, Xeon Phi, ...) is the compilation time. For FPGAs, kernels have to be compiled off-line, and the process can take up to several hours.

OpenCL kernels for FPGAs are written in C99-like code that contains the kernel's functional description and that is portable across devices; however, performance is not so portable because OpenCL hides the underlying architecture. Unlike a GPU that makes use of a myriad of cores to process many threads in parallel, an FPGA exposes a pipeline parallelism where multiple threads run in parallel but at different stages of the processing pipeline. The pipeline structure is defined by the HLS tools at compilation. Therefore, programmers have always to keep in mind this pipeline model to ensure good performance.

In terms of parallelism, the deeper the pipeline is, the greater the number of threads that can simultaneously advance. As a result, the FPGA performance mainly depends on two factors: initiation interval and frequency. Initiation interval is the number of clock cycles between two consecutive kernel threads (work-items) start, and frequency is the cycle time of the largest pipeline stage. Discussing the programming patterns that influence these two factors is out of scope of this work and has been previously discussed [18, 31].

EngineCL provides support for heterogeneous environments composed by CPU, GPU, and Xeon-Phi devices. The inclusion of FPGAs in EngineCL is not straightforward and requires modifications in several key points of the runtime: host device synchronization, kernel arguments management, and command queues management. Besides, to launch a kernel, EngineCL has to load the bit stream from a file to the FPGA.

Originally, EngineCL relied on asynchronous callbacks to notify kernel completion to the scheduler. Unfortunately, Intel's FPGA OpenCL runtime v17.1 requires a subsequent OpenCL function invocation to evaluate event status and call pending registered callbacks, which could not always be guaranteed in EngineCL. To operate with FPGA, callbacks are replaced with synchronous OpenCL commands managed by multiple host threads and command queues for each device.

To use the schedulers, we extend the EngineCL arguments in *workers* of Tier-2 and change data management of Tier-3, Fig. 1. In *workers* of Tier-2, we add iterations argument, which are the amount of work, because task-based

kernels have statically defined just one work-item per execution. Originally, EngineCL copied every input data into memory of all devices, limiting the maximum problem size to the size of the smallest memory device, in our case the FPGA. To support any problem size, we have performed two modifications: 1) replacement of the offset argument with two item range (begin, end) arguments at each kernel invocation, 2) support for sending input data as required. Since each device has its own memory, the runtime keeps track of each size and sends chunks small enough to fit into the device memory.

Finally, to improve performance, we add a second command queue and two output device buffers (A and B in Fig. 2) to overlap memory read and computation commands as depicted in Fig. 2. Each kernel invocation alternates output buffers, A and B, to avoid write-after-write hazard. Since the FPGA driver only allows one transaction over PCI-e at a time, there is no opportunity to overlap read and write commands. To ensure overlapping with the single transaction requirement, we enqueue the write and compute of chunk i on queue 0 before enqueue the read command of chunk $i-1$ on queue 1, and then wait for both to complete with a *finish barrier* and a blocking read command, respectively. This change improves performance up to 30% (PCIe write speed is lower than the read one) for communication-bound problems and has little effect in performance-bound ones.

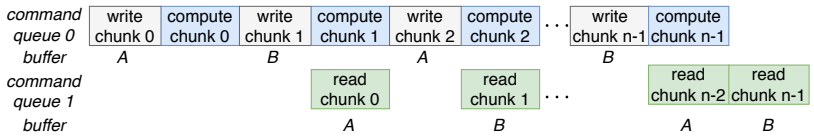

Fig. 2: Overview of command queue overlapping with the two command queues and the two buffer sets (A and B).

Please note that all these changes do not entail any modification in the apps built on top of EngineCL because user-facing APIs remain the same.

5 Experimental Methodology

The experiments have been conducted in a heterogeneous system composed by an Intel core i7-6700k CPU (64 GB of RAM), a NVIDIA GeForce GTX TITAN X GPU (12GB of RAM), and an Altera DE5NET Stratix V GX FPGA (4GB of RAM); each device runs OpenCL version 2.0 (LINUX), 1.2 (CUDA 9.1.83), and 1.0 (Intel SDK v17.1) for the CPU, GPU, and FGPA, respectively.

Six benchmarks from different domains have been considered: Matrix Multiplication, Mersenne Twister, and Sobel Filter from the Intel Altera OpenCL repository, Watermarking and AES decrypt from the Xilinx SDAccel repository, and Nearest Neighbor from Rodinia optimized for FPGA [32]. To improve application performance, we implemented different kernel versions tuned for

each device. For example, FPGA devices work better with task-based kernels and very long shift register loops, while CPU and GPU perform better with NDRange kernels and smaller shift registers. But there are two exceptions, Matrix Multiplication and Nearest Neighbor that obtain the best results with an NDRange-based kernel on the FPGA. Table 1 shows the benchmark size (measured in work-items), main parameters, and FPGA resource utilization. In general, lower FPGA resource utilization translates into higher frequency.

Since the initialization latency of the OpenCL FPGA runtime is much higher than that of the other two devices, we launch a work-item kernel command before starting the heterogeneous execution. Otherwise, the scheduler penalizes the FPGA for splitting the work.

Table 1: Work-items and FPGA characteristics: Clock frequency(CF), initialization interval (II), and FPGA kernel resources: adaptive logic module (AL), logic registers (LR), Memory blocks (MB), and DSP.

Benchmark	Work items	CF (MHz)	II	AL %	LR %	MB %	DSP %
Matrix Multiplication	16×10^3	238.9	n/a	79	28	47	100
Mersenne Twister	22×10^7	274.4	~ 1	40	5	19	88
Watermarking	11×10^8	226.8	~ 1	16	10	15	3
Sobel filter	12×10^9	295.3	~ 1	13	8	18	0
AES	11×10^8	299.9	2	20	9	18	0
Nearest-Neighbor	40×10^8	210.8	n/a	54	19	31	94

In order to evaluate energy consumption, we rely on reading hardware counters with Intel RALP and NVIDIA system manager for the CPU and the GPU, respectively. Since the FPGA does not provide power counters, we measure its power with the Newtons4th PPA520 power analyzer, sampling at 10^6 samples per second, and a PCI riser card [11]. FPGA power is the sum of the power drained from the PCIe edge connector and the auxiliary 6-pin Molex connector.

6 Results

This section analyzes the inclusion of the FPGA inside EngineCL by exploring its 3 load balancing algorithms on a CPU+GPU+FPGA heterogeneous system. First of all, it explains how the parameters of each algorithm have been optimized. Then, the results obtained in terms of both performance and energy are analyzed.

Static With static scheduling, the user has to choose the amount of work each device performs before starting execution. This distribution should be tuned for achieving good load balancing and unfortunately, it is required an exhaustive exploration to find out the fine tuned distribution.To make matters

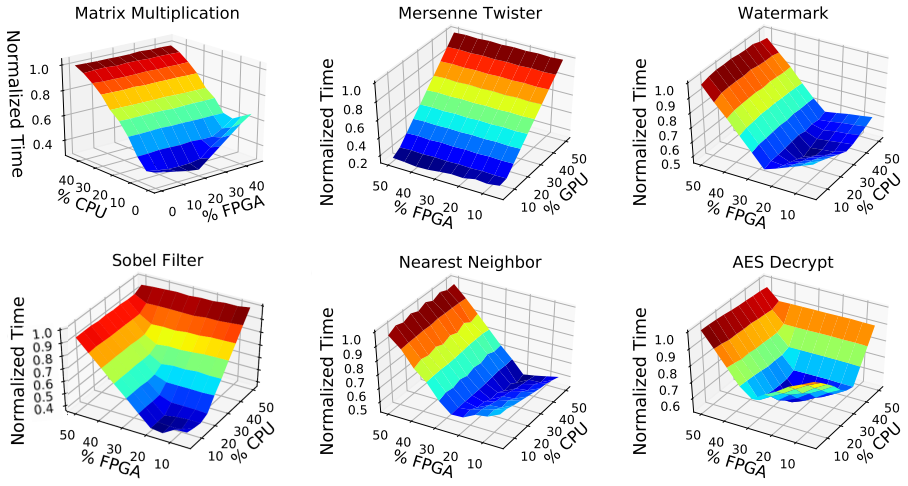

Fig. 3: Normalized execution time to the worst for the Static scheduler. Work proportions go up to 50% for two devices, and the third device performs the remainder work.

worse, this distribution is specific for each problem, input data and, compute device. For instance, in this paper 100 executions per benchmark have been performed to achieve these values. Fig. 3 shows the normalized execution time to the CPU with a percentage sweep in steps of 10% for two devices, while the third device takes the remainder work $(100 - (percentage_1 + percentage_2))$. Overall, setting the optimal percentages improves performance, with gains ranging between 16 and 79%.

Dynamic With this scheduler, each device fetches and executes chunks of work (equally-sized for all devices) until there is no work left. Fig. 4 shows how chunk size has a large impact on throughput, measured in output gigabytes per second, and, hence, on performance. In general, all benchmarks benefit from larger chunks except Matrix Multiplication. In Matrix Multiplication, the number of iterations to split is smaller than in other benchmarks, and the computational intensity is higher, so that the lower runtime overhead of smaller chunks does not pay off for the imbalance increment.

HGuided Starts with large chunks that are automatically reduced. Two parameters tune the chunk size: computing power and minimum packet size. On this paper, we have used the computing power ratios computed for the static scheduler, but the sensitivity of HGuided to this parameter is small because when the minimum packet size is large enough, overall throughput remains high. This condition is relatively easy to fulfill because most benchmarks reach good performance with small chunks, as shown in Fig. 4.

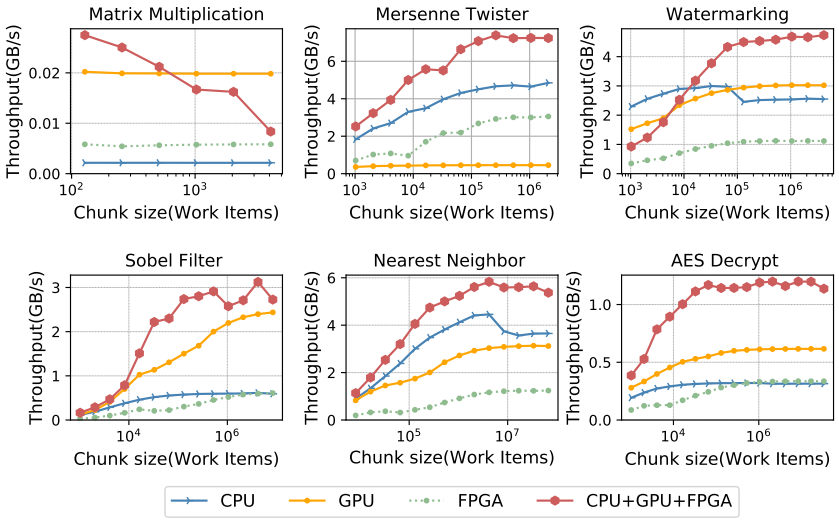

Fig. 4: Dynamic scheduler Throughput (GBs) of CPU, GPU and FPGA with chunk size variation.

Performance For the sake of brevity, from here on, all results use the best found parameters for all 3 schedulers. First, Table 2 compares the performance of the 3 schedulers, including the percentage of improvement of the heterogeneous system with respect to the best single device (PI). To measure the effectiveness of load balancing, it also presents the imbalance percentage (IM), as $\frac{T_{LD}-T_{FD}}{T_{LD}} \cdot 100$, where T_{FD} and T_{LD} are the execution time of the device that finished at first and last, respectively. Finally, the number of chunks is also shown (#C). On the other hand, Fig. 5 shows the normalized execution time to the CPU of the three single devices and all 3 devices in cooperation.

Analyzing these results, it should first be highlighted that in all cases a substantial improvement is achieved by using heterogeneous systems, compared to using a single device (the best one for each benchmark). Overall, gains range between 17.8 and 96.6% for Sobel Filter and AES, respectively. The static achieves the best results in 3 of the 6 benchmarks (Matrix Multiplication, Watermarking and Nearest Neighbor). The advantages of adaptability do not outweigh the increase in overload imposed by dynamic algorithms. This is so, thanks to the great effort optimizating parameters made off-line. The dynamic algorithm only yields the best result in Mersenne Twister with 87.9%, and HGuided gets the best in Sobel Filter and AES. In average, the best algorithm is Dynamic with an improvement of 58.0%, followed by static (50.4%) and HGuided (50.0%).

However, dynamic algorithms better balance the workloads, except in the case of Matrix Multiplication and Mersenne Twister. Actually, they achieve a perfect balance in Watermarking (HGuided), Solbel Filter (Dynamic) and AES (both). Therefore, it can be concluded that the performance improvements are

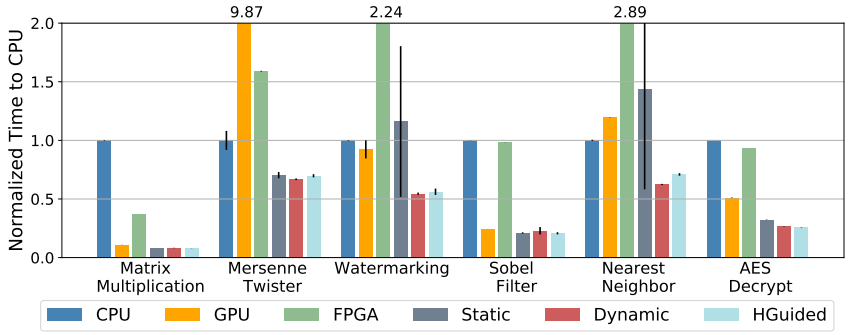

Fig. 5: Overall performance of cooperative execution CPU+GPU+FPGA with three load balancers for each benchmark. Times are normalized to CPU in every benchmark.

Table 2: Performance Improvement (PI) relative to the fastest single device execution, imbalance (IM), and average number of chunks (#C) for Static, Dynamic and HGuided policies and all benchmarks.

Benchmark	Static			Dynamic			HGuided		
	PI %	IM %	#C	PI %	IM %	#C	PI %	IM %	#C
Matrix Multiplication	37.3	4.2	3	32.0	5.5	128	37.0	39.1	18.0
Mersenne Twister	42.2	2.0	20	87.9	6.4	60	43.3	8.8	25.0
Watermarking	79.5	1.4	28	69.2	0.8	280	64.3	0.0	36.4
Sobel Filter	16.2	1.1	23	6.7	0.0	11930	17.8	0.6	118.0
Nearest Neighbor	69.4	2.7	13	60.3	0.3	960	41.0	9.2	16.0
AES decrypt	58.1	4.1	23	92.4	0.0	280	96.6	0.0	94.6
Mean	50.4	2.6		58.1	2.2		50.0	9.6	

not only due to load balancing, but also to the application's ability to take advantage of the architecture features of the different devices.

Finally, Fig. 5, shows that, despite the great difference in devices architecture and performance, EngineCL always achieves a performance improvement compared to using a single device. Moreover, given the diversity of benchmarks' behavior, selecting the best device in advance is almost impossible, but with this proposal, we bring out the best in all of them.

Energy consumption. Table 3 shows the average power for single devices running all benchmarks. We discern 3 different power metrics: idle (I), which corresponds to the device sitting idle; programmed (P), which corresponds to the power of the FPGA after it has been programmed, so P does not apply for CPU and GPU devices; device and host running (DR and HR), which corresponds to the power when a kernel is running split between device and host, when possible (GPU and FPGA). Therefore, HR represents the power dissipated by EngineCL, and the OpenCL runtime and driver.

Comparing the values, idle power keeps on the same range for the 3 devices, and both the CPU and GPU reach much higher DR values, 80.2 and 132.1W for the CPU and GPU, respectively. On the contrary, the FPGA has a much lower DR of 28.8 W, but a high programming power (P), from 20.8 to 25.1W, suggesting that once programmed, running at least a small proportion of compute in FPGA could be beneficial compared to "waste" the device in programmed state. The only caveat is that HR impacts on FPGA energy efficiency because its value is on par with DR.

Table 3: Average Power (W) for single device configurations. I, P, DR, HR represents idle, programmed, device running, and host running power, respectively.

	Average Power (W)								
	CPU		GPU			FPGA			
	I	DR+HR	I	DR	HR	I	P	DR	HR
Matrix Mult.	13.6	77.7	15.8	132.1	29.2	14.3	25.1	28.8	27.7
Mer. Twister		33.0		88.5	25.8		23.3	23.6	17.2
Watermarking		45.2		45.2	29.6		20.8	21.2	19.7
Sobel Filter		76.2		87.4	31.1		20.8	21.4	14.4
Near. Neighbor		32.1		82.3	32.1		23.3	23.6	22.5
AES Decrypt		80.2		112.1	27.3		21.2	21.7	30.2

Fig.6 shows the normalized total energy compared to CPU-only energy. While in terms of performance, all three schedulers improve execution time compared to the best single-device, for energy they do not. In all but Matrix Multiplication and Sobel, heterogeneous execution degrades energy.

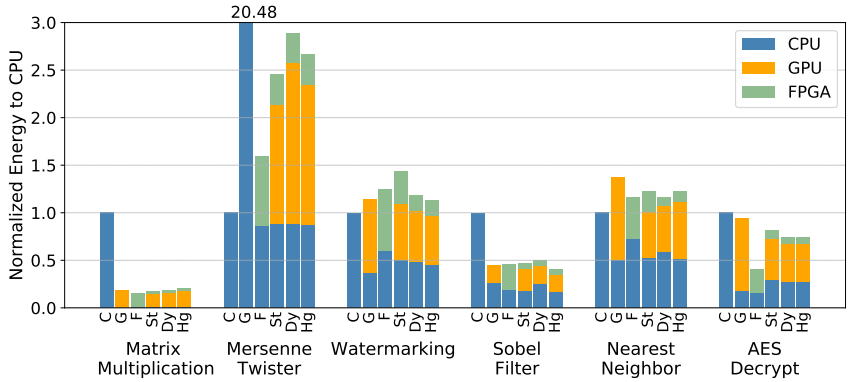

Fig. 6: Overall energy of cooperative execution CPU(C)+GPU(G)+FPGA(F) with Static(St), Dynamic (Dy) and HGuided (HG) load balancers for each benchmark. Energy are normalized to the CPU device in every benchmark.

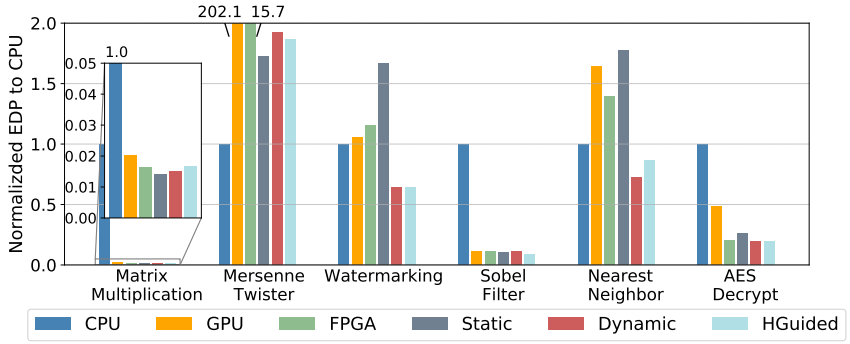

Fig. 7: Overall normalized energy-delay product of cooperative execution CPU+GPU+FPGA with three load balancers for each benchmark. The lower the better, and Matrix Multiplication results have been zoomed for clarity.

Within the groups of benchmarks than improve/degrade energy, behavior is similar, so, for the sake of brevity, we only comment on a representative benchmark per group: Mersenne Twister and Sobel Filter. The former experiences an energy degradation around 2.5× for the heterogeneous configurations with regards to CPU because of the high GPU consumption; e.g., for the dynamic scheduler, GPU only processes 6% of the work-items and consumes 70W DR [1]. In terms of energy efficiency (work-items/joule), the GPU is around 23× worst than the CPU and the FPGA. The later, Sobel Filter, presents an opposite behavior compared with Mersenne Twister. In HGuided, normalized energy improves by 60 and 11% compared with CPU and GPU only, respectively. In this case, the GPU and FPGA DR consume 80 and 21.4W and computes 65 and 17% of the work-items, respectively. For both devices, their energy-efficiency is 3× better than CPU's.

Since we are comparing different device architectures, we compute the energy-delay product, EDP, as shown in Fig. 7. The cooperative execution in Matrix Multiplication, Watermark, Sobel Filter, Nearest Neighbor and AES Decrypt improves EDP over the best single device, with a maximum improvement of 36% for HGuided in Watermarking compared to CPU. In those cases, the performance gains of multiple devices pays off for the extra energy consumption.

7 Conclusions and Future Work

FPGAs can provide excellent performance with limited energy consumption, presenting an improvement opportunity for supercomputing systems. Nevertheless, FPGA programming with hardware description languages requires

[1] This value corresponds to the cooperative execution and is lower than GPU-only DR, see Table 3, because there is less continuous work on the device.

more expertise than programming other accelerators such as GPUs. Therefore, FPGA adoption requires high-level programming tools to facilitate this task. EngineCL is an OpenCL-based framework allowing the automatic heterogeneous execution of parallel loops in multiple devices thanks to its load balancing algorithms. This article proposes an EngineCL extension to support FPGAs, so that users can cooperatively execute parallel loops in CPU+GPU+FGPA systems. To boost performance, the extension overlaps data transfer and compute operations by implementing multiple command queues and allows to execute per-device tuned kernels. To ease the adoption of the FPGAs, the extension does not change any user-facing APIs of EngineCL.

The results show that EngineCL provides performance improvements ranging between 16 and 96% on a system with computationally unbalanced devices. Load balancing policies do not manage to do so well in the case of total energy consumed and energy efficiency. In case of energy consumption, the cooperative approach never beats the best single device in any benchmark. On the other hand, the cooperative approach is more energy efficient in 5 out of 6 benchmarks. These results indicate that it would be interesting study energy-aware load balancing policies.

Besides, future work could assess which is the best composition for a supercomputer node, whether to provide all nodes with different accelerators such as GPU and FPGAs, or have nodes with a single type of accelerator.

Acknowledgements

The authors would like to thank the anonymous reviewers, Shaizeen Aga for their suggestions, and Luis Piñuel Moreno for his help measuring FPGA power. This work was supported in part by grants TIN2016-76635-C2 (AEI/FEDER, UE), gaZ: T48 research group (Aragón Gov. and European ESF), the University of Zaragoza (JIUZ-2017-TEC-09), HiPEAC4 (European H2020/687698), the Spanish Ministry of Education (FPU16/03299) and the CAPAP-H Network grant TIN2016-81840-REDT. M. A. Dávila-Guzmán is supported by a Universidad de Zaragoza-Banco Santander PhD scholarship.

References

1. Altera SDK for OpenCL Programming Guide. URL http://www.altera.co.uk/literature/hb/opencl-sdk/aocl_programming_guide.pdf
2. SDSoC environment user guide. URL www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/ug1027-sdsoc-user-guide.pdf
3. Qualcomm Snapdragon Heterogeneous Compute SDK (2018). URL https://developer.qualcomm.com/software/heterogeneous-compute-sdk
4. Zhou, S. et al.: Accelerating graph analytics on cpu-fpga heterogeneous platform. In: SBAC-PAD, pp. 137–144 (2017)
5. Alawieh, M. et al: A high performance FPGA-GPU-CPU platform for a real-time locating system. In: EUSIPCO, pp. 1576–1580 (2015)
6. Belviranli, M. E. et al.: A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures. ACM Trans. Archit. Code Optim. **9**(4), 57:1–57:20 (2013)

7. Binotto, Al.P.D. et al.: Towards dynamic reconfigurable load-balancing for hybrid desktop platforms. IPDPSW (2010)
8. Chen X. et al.: CMSA: a heterogeneous CPU/GPU computing system for multiple similar RNA/DNA sequence alignment. In: BMC Bioinformatics (2017)
9. Chung, E. S. et al.: Single-chip heterogeneous computing: Does the future include custom logic, FPGAs, and GPGPUs? In: Proc. of the 43rd Ann. Int. Symp. on Microarchitecture, MICRO '43, pp. 225–236. IEEE Computer Society, Washington, DC, USA (2010)
10. Horowitz, M.: 1.1 computing's energy problem (and what we can do about it). In: ISSCC, pp. 10–14 (2014)
11. Igual, F.D., Jara, L.M., Pérez, J.I.G., Piñuel, L., Prieto-Matías, M.: A power measurement environment for PCIe accelerators. Computer Science - R&D **30**(2), 115–124 (2015)
12. Kaleem,R. et al.: Adaptive heterogeneous scheduling for integrated GPUs. In: PACT, pp. 151–162. ACM, New York, NY, USA (2014)
13. Katranovet, A. et al.: Intel threading building block (TBB) flow graph as a software infrastructure layer for OpenCL-based computations. In: ACM IWOCL, pp. 9:1–9:3 (2016)
14. Koch, D., et al. (eds.): FPGAs for Software Programmers. Springer, Cham (2016)
15. Lee, J., et al.: Orchestrating Multiple Data-Parallel Kernels on Multiple Devices. In: Intl. Conf. on Parallel Architectures and Compilation Techniques, pp. 355–366 (2016)
16. Luk, C.-K. et al.: Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping. IEEE/ACM Micro-42 p. 45 (2009)
17. Mittal, S.a.: A Survey of CPU-GPU Heterogeneous Computing Techniques. ACM Computing Surveys **47**(4), 1–35 (2015)
18. Momeni, A. et al.: Hardware thread reordering to boost OpenCL throughput on FPGAs. In: ICCD, pp. 257–264 (2016)
19. Muslim, F. B. et al.: Efficient fpga implementation of opencl high-performance computing applications via high-level synthesis. IEEE Access **5** (2017)
20. Nane, R, et al.: A survey and evaluation of FPGA high-level synthesis tools. IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems **35**(10), 1591–1604 (2016)
21. Nozal, R. et al.: EngineCL: Usability and Performance in Heterogeneous Computing. arXiv (2018). URL `https://arxiv.org/abs/1805.02755`
22. Nozal, R. et al.: Load balancing in a heterogeneous world: Cpu-Xeon Phi co-execution of data-parallel kernels. The Journal of Supercomputing (2018)
23. Nunez-Yanez, J.e.a.: Simultaneous multiprocessing in a software-defined heterogeneous fpga. The Journal of Supercomputing (2018)
24. Pandit, P. et al.: Fluidic kernels: Cooperative execution of OpenCL programs on multiple heterogeneous devices. Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (2014)
25. Pérez, B.e.a.: Energy efficiency of load balancing for data-parallel applications in heterogeneous systems. The Journal of Supercomputing **73**(1), 330–342 (2017)
26. Pérez, B. et al.: Simplifying programming and load balancing of data parallel applications on heterogeneous systems. In: GPGPU, pp. 42–51. ACM, New York, NY, USA (2016)
27. Rethinagiri, S. K. et al.: Trigeneous Platforms for Energy Efficient Computing of HPC Applications. In: Intl. Conf. on High Performance Computing Trigeneous. IEEE (2015)
28. Stone, B.J.E. et al.: OpenCL: A Parallel Programming Standard For Heterogeneous Computing Systems (2010)
29. Tsoi, K. H. et al.: Axel: A heterogeneous cluster with FPGAs and GPUs. In: ACM/SIGDA FPGA, pp. 115–124. ACM, New York, NY, USA (2010)
30. Vilches, A. et al.: Adaptive partitioning for irregular applications on heterogeneous CPU-GPU chips. Procedia Computer Science, ICCS **51**, 140 – 149 (2015)
31. Wang, Z. et al.: A performance analysis framework for optimizing OpenCL applications on FPGAs. In: Proc. of HPCA), pp. 114–125 (2016)
32. Zohouri, H.R. et al.: Evaluating and optimizing OpenCL kernels for high performance computing with FPGAs. In: SC, pp. 35:1–35:12. IEEE Press, Piscataway, NJ, USA (2016)