Explotando el nuevo módulo OpenCL de Intel TBB

José Carlos Romero, Alejandro Villegas, Ángeles Navarro, Andrés Rodríguez, Rafael Asenjo[1]

Resumen— **Este artículo tiene como objetivo contribuir al desarrollo de la programación paralela trabajando en una de las herramientas desarrolladas por Intel: Intel Threading Building Blocks (Intel TBB). Hemos implementado una versión paralela de la aplicación ViVid, un algoritmo de detección de objetos, aprovechando la librería *Flow Graph* de TBB. En la última versión de esta librería se soporta un nuevo tipo de nodos que simplifica la ejecución de kernels OpenCL. Estos nodos OpenCL_node han simplificado la codificación de nuestro algoritmo para aprovechar al mismo tiempo tanto la CPU como la GPU.**

A la vista de los resultados obtenidos, podemos concluir que nuestra implementación heterogénea con Flow Graph y OpenCL_node mejora en gran medida a desarrollos anteriores, debido principalmente a explotar la técnica de zero-copy buffer entre GPU y CPU, en vez de lectura y escritura de buffers.

Palabras clave— **Intel Threading Building Blocks, Flow Graph, OpenCL, arquitectura heterogénea, programación paralela, computación de altas prestaciones.**

I. Introducción

La programación paralela representa el futuro de la computación y se ha convertido en el paradigma dominante en la arquitectura de computadores. Lleva años presente en la computación de altas prestaciones, sin embargo, con la aparición de los procesadores multi-core en los ordenadores personales y, más aún, con el surgimiento de los smartphones, la programación paralela se ha extendido a todas las áreas de la computación[1]. Los procesadores multi-core surgieron de la necesidad de solucionar el problema de las limitaciones físicas que se encontraron en el desarrollo de los procesadores single-core la pasada década. En aras de una mayor reducción del consumo energético, también se han popularizado recientemente arquitecturas heterogéneas on-chip, las cuales incorporan una GPU, o incluso una FPGA, junto con los cores.

El problema que plantean estas nuevas arquitecturas heterogéneas estriba principalmente en la dificultad de programación de aplicaciones que exploten todos los recursos computacionales. Como desarrolladores preocupados por el rendimiento, somos conscientes de la importancia de paralelizar las aplicaciones para no dejar cores de CPU ociosos. Usando el mismo razonamiento, en muchos casos tiene también sentido usar los cores de GPU al tiempo que los cores de CPU, de forma que no queden recursos desaprovechados en el chip. Evidentemente, los costes de desarrollo de aplicaciones heterogéneas aumentan significativamente en comparación con el desarrollo de aplicaciones serie. Nuestro objetivo a largo plazo es simplificar la implementación de aplicaciones heterogéneas, haciendo uso de características de alto nivel proporcionadas por C++11 y la librería Threading Building Blocks, TBB [2]. Más concretamente, evaluaremos el nuevo módulo que permite incluir código OpenCL [3] en algunos nodos de un Flow Graph de TBB, y así añadir la GPU como unidad computacional adicional [4].

Los algoritmos de percepción y visión por computador constituyen un objetivo claro en el desarrollo e implantación de la programación paralela, pues suelen tener un coste computacional muy alto y, típicamente, el tratamiento de datos se presta a la paralelización. Esto es, las estructuras de datos en algoritmos típicos de visión son matrices y arrays en los que se puede explotar paralelismo en mayor o menor medida.

El objetivo final de este artículo es, por tanto, implementar y evaluar una versión paralela de un algoritmo de detección de objetos, ViVid, apoyándonos en la librería *Intel TBB*, que desde hace poco incluye un nuevo módulo de OpenCL. Este nuevo módulo simplifica la tarea de programar un procesador Intel con GPU integrada, de forma que la aplicación explote al mismo tiempo los recursos computacionales de la CPU y la GPU. De esta forma se incrementa de forma notable el rendimiento de la aplicación al tiempo que se reduce el consumo energético.

El siguiente artículo tiene la estructura que se indica a continuación: comienza por la introducción en la que nos encontramos. A continuación se ponen de manifiesto los antecedentes así como se introduce el nodo OpenCL empleado. Se continúa explicando la implementación del grafo con el que se ha paralelizado el algoritmo, así como las características de los nodos que lo forman. A continuación se presentan los resultados experimentales obtenidos, así como los trabajos relacionados, para acabar con las conclusiones y lecciones aprendidas.

II. Antecentes

A. Algoritmo secuencial: ViVid

En este apartado abordamos la explicación de la aplicación paralelizada en este trabajo. Para desarrollar cualquier paralelización de un algoritmo, se debe partir de una aplicación secuencial ya funcional, la cual se paraleliza con una librería o modelo de programación concreto, para posteriormente realizar un análisis de rendimiento (speedup) con el objetivo de comprobar la mejora en tiempo de ejecución respecto

[1]Universidad de Málaga, Andalucía Tech, Dept. of Computer Architecture, Spain. e-mail: {jromero, avillegas, angeles, andres, asenjo}@ac.uma.es.

a la versión serie del programa.

Para este trabajo, se va a partir de una aplicación de procesamiento de imágenes, la cual es ideal para poder explotar las capacidades de procesamiento paralelo de las arquitecturas heterogéneas. Se trata de la aplicación ViVid [5] [6] (The Video Processing Library). Consiste en un algoritmo que emplea una técnica de "ventana deslizable" para la detección de objetos con alta probabilidad, realizado principalmente para investigadores en visión por computador, con el objetivo de servir de test para algoritmos de optimización y paralelización relacionados con el campo.

Como ya se ha comentado anteriormente, las aplicaciones de visión por computador y tratamiento de imágenes son ideales para el objetivo de este trabajo: explotar las capacidades de una arquitectura heterogénea. La alta intensidad computacional de la carga de trabajo, las fases de computación multietapa, la independencia de los frames y los elementos dentro de un frame, son algunas de las características que permite una flexibilidad para paralelizar sin tener que preocuparse por inconsistencias secuenciales.

Ya que la GPU está especializada en este tipo de procesamientos, realizar una paralelización de un código de este tipo que a la vez emplee la GPU y los cores de la CPU, permite conseguir una enorme mejora en el rendimiento, demostrando la importancia de programar adecuadamente para conseguir todo el rendimiento que ofrecen estas arquitecturas.

Por tanto, en primer lugar se va a explicar el funcionamiento de este algoritmo serie, para posteriormente desarrollar la paralelización del mismo empleando Intel TBB y su módulo OpenCL para la parte GPU.

La librería de procesamiento de vídeo (ViVid) ha sido ideada como un framework para utilizar masivamente las capacidades de procesamiento paralelo de las GPU y para ayudar a los laboratorios de procesamiento y formación de imágenes. ViVid se ha organizado con la intención de que los investigadores en visión por computador sean los usuarios finales, especialmente los investigadores que se pueden beneficiar en el desarrollo de sus algoritmos de optimización y mejora de rendimiento, empleando ViVid como prueba. Este artículo es un ejemplo de ello.

Los cálculos de ViVid se hacen para determinar si el objeto buscado está en esa "ventana" (conjunto de píxeles) o no y entonces la "ventana" deslizará hasta la siguiente localización de la imagen, volviendo a repetirse los cálculos. Este proceso se repite hasta que la "ventana" haya recorrido toda la imagen. En la Figura 1 se puede ver la distribución de las 3 etapas en las que se divide el algoritmo, las cuales se van a describir a continuación:

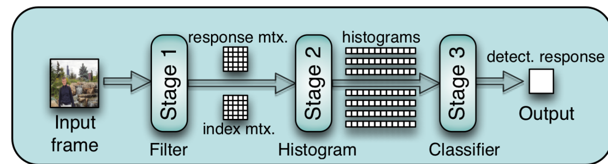

Fig. 1. Algoritmo ViVid.

- **Filter kernel:** Se aplica una operacion de convolución a cada pixel del frame, empleando un conjunto de filtros (100) en kernels de 3x3. Como resultado, se devuelve el índice del filtro con valor máximo, y dicho valor máximo, para cada pixel.
- **Histogram kernel:** Construye un histograma para los datos recogidos en el filtro anterior.
- **Classifier kernel:** Hace una comparación para comprobar si se ha encontrado el objeto buscado o no, mediante una operación similar a un producto de matrices.

B. *OpenCL_Node*

Además de la CPU, en este trabajo queremos explotar todas la capacidades de la arquitectura heterogénea completa, CPU y GPU, por lo que necesitamos conocer la manera de incorporar en nuestro programa el código que ejecuta la GPU.

OpenCL (Open Computing Language) [7] es el primer estándar de programación verdaderamente multiplataforma y libre para computación en arquitecturas heterogéneas. OpenCL es la alternativa libre a tecnologías propietarias como CUDA de Nvidia o STREAM de AMD. En el desarrollo de este trabajo, se ha empleado OpenCL para realizar la programación de la parte de código correspondiente a la GPU.

Programar en OpenCL tiene sus desventajas, ya que requiere de una inversión en programación de código bastante costosa, como por ejemplo: declaración y creación del entorno OpenCL, buffers de memoria para los datos, creación de los kernel (que es el equivalente a función en código tradicional CPU) a los que se les asocia un código OpenCL, así como las instrucciones para lanzar el código en la GPU u otro dispositivo OpenCL. Como se puede ver esto hace difícil para un programador el implementar programas que exploten a la vez la CPU y la GPU.

Gracias a Intel TBB [8] [9] la mayoría de estas dificultades desaparecen. Hace unos meses se añadió a la librería TBB un nuevo módulo para poder implementar grafos con códigos OpenCL[10]. Este nuevo módulo, es un nuevo tipo de nodo para *Flow Graph*, llamado OpenCL_node, el cual permite descargar código a la GPU de una forma mucho más amigable que haciendo uso del OpenCL estándar. El OpenCL_node align abstrae al programador de los detalles de implementación OpenCL de bajo nivel y que antes eran imprescindibles para poder lanzar un código OpenCL. De esta forma, con esta nueva funcionalidad de TBB, ejecutar código en la GPU se convirte prácticamente en una invocación a una función de CPU tradicional, con solo algunas consideraciones adicionales.

La curva de aprendizaje de *Flow Graph* es sin embargo un poco pendiente, pero una vez se supera, implementar códigos para arquitecturas heterogéneas

resulta realmente rápido y eficiente con este nuevo módulo. En este trabajo se quiere precisamente comprobar esta premisa. Hay que destacar las dificultades que conlleva familiarizarse y trabajar con una tecnología novedosa y que está aun en desarrollo.

III. Implementación del grafo paralelo

A continuación se va a explicar el funcionamiento del Grafo a través del cual se ha llevado a cabo la implementación de la versión paralelizada de ViVid. En la Figura 2 se puede apreciar la estructura general del grafo, del cual se explicará su funcionamiento, primero de forma general, para luego entrar en detalle en la programación de cada uno de los nodos que lo componen.

El grafo está diseñado como una estructura de control de uso de recursos basada en "tokens". Un "token" representa un recurso libre del sistema. De esta forma se va a poder controlar el paralelismo a nivel de CPU (con los distintos cores disponibles), así como la GPU. Por lo tanto, el programa comienza generando los "tokens" que se quiere tener disponibles (CPU y/o GPU), comenzando el grafo a través de su nodo fuente, el nodo *Input*.

El token esta formado por una "tupla" de datos, compuesta por el propio token, que es un entero que indica si esa imagen se va a ejecutar en la CPU o en la GPU, así como por todos los arrays y estructuras de datos necesarias para procesar la imagen. De esta forma se asegura, a través de esta tupla de datos compacta, que no hay interferencia de datos de resultados entre diferentes imágenes que están siendo procesadas en paralelo, y permanecen sin alterar los cálculos simultáneos sobre diversas imágenes.

Cabe destacar que este grafo permite, además de elegir mediante los token si se quiere procesar imágenes en la CPU o en la GPU, si se quiere lanzar una determinada etapa del algoritmo (que esta compuesto por 3 etapas como hemos explicado en la sección anterior) en la GPU. De esta forma se ofrece flexibilidad a la hora de configurar qué dispositivos se pueden usar en cada etapa del pipeline. Esto permitirá conseguir la mejor configuración posible para el algoritmo, como se verá en el apartado de resultados experimentales. Por tanto, al comienzo del programa se permite elegir además de si se activa la GPU, el número de cores CPU y cuales son las etapas del algoritmo en las que se permite descargar el trabajo de esa etapa a la GPU.

En la Figura 3 se puede ver las distintas configuraciones de etapas CPU-GPU que se pueden hacer, así como el código binario de tres dígitos que las identifica.

Se observan que las 3 etapas centrales del algoritmo ViVid se pueden ejecutar en GPU. Como CPU se ven 5 etapas: las 3 del algoritmo y las etapas de entrada y salida del mismo, ya que el grafo se ejecuta enteramente en la CPU. Por lo tanto, las etapas de entrada y salida se ejecutan siempre en CPU y las 3 etapas centrales, que contienen realmente el grueso de la carga computacional, son las que se pueden, o ejecutar en la CPU o descargarlas a la GPU. Por ejemplo si se elije la configuracion 111, significa que las 3 etapas del algoritmo están activadas para que se puedan ejecutar tanto en CPU como en GPU. Esto permite que, si llega un token de GPU a procesar un frame, éste haga las 3 etapas del algoritmo en la GPU. Si llega un token de CPU, ejecutará las 3 etapas por la CPU. De esta forma se aprovecha la heterogeneidad de la plataforma. Si por el contrario, se elije la configuracion 101, esto significa que solo las etapas 1 y 3 están activadas para ser ejecutadas por la GPU. Con esta configuración, cuando llegue un token de GPU a procesar un frame, va a poder descargar trabajo en la etapa 1 a la GPU, sin embargo la etapa 2 la ejecutará en CPU, volviendo a descargar trabajo a la GPU en la tercera etapa. Los token GPU son por tanto, unos token "híbridos" que pueden trabajar tanto en GPU como en CPU, y los token CPU van siempre por el camino de la CPU. Con esto, podemos ahora pasar a explicar el funcionamiento del grafo paralelizado, con la estructura de funcionamiento basado en tokens.

El grafo comienza, por tanto, a funcionar a través del nodo *Input*, el cual genera un identificador de imagen para poder tener ordenados los frames que se analizan, independientemente de si se procesa por GPU o por CPU. Se generan los token (compuestos por la tupla de datos antes descrita), a través del token_buffer, el cual se encarga de limitar el uso de recursos en el recorrido del grafo, permitiendo que sólo se procesen concurrentemente tantas imágenes como tokens disponibles se hayan deseado liberar.

El token se une al identificador de la imagen a través del *Join*, el cual une el camino del *Input* y del *Buffer*, y los pasa al nodo *Dispatch1*.

De aquí hasta la parte final del grafo, se sigue una estructura similar en cada una de las 3 etapas del algoritmo: Se comienza por un *dispatch*, el cual recibe el token con todos los datos y decide si se lanza dicho token a la CPU o la GPU en función de la configuración que se haya decidido y de si dicho token es de CPU o de GPU. Se puede dar el caso de que el token sea de GPU, pero como en dicha etapa no se ha activado la GPU, el token procesa por tanto esa etapa determinada del algoritmo en la CPU.

Se lanza el token correspondiente a la etapa, sea GPU o CPU y luego se devuelven los resultados al grafo, a través del *Indexer*, el cual hace de unión de la etapa GPU y la CPU. De esta forma la ramificación GPU-CPU que se había hecho se cierra y el grafo vuelve a tener un sólo camino. De este *Indexer* se pasa al *Dispatch* de la etapa siguiente. Este se encarga de recibir los datos, reordenarlos si son CPU o GPU, decidir en la siguiente etapa a qué camino se lanzan dichos datos, y proceder a descargar el trabajo a la etapa siguiente, sea CPU o GPU. Este proceso se vuelve a repetir hasta completar las 3 etapas del algoritmo.

Una vez analizada la imagen tanto en CPU como en GPU, se libera el "token" correspondiente, devolviéndolo al *Buffer*. De esta forma queda libre un

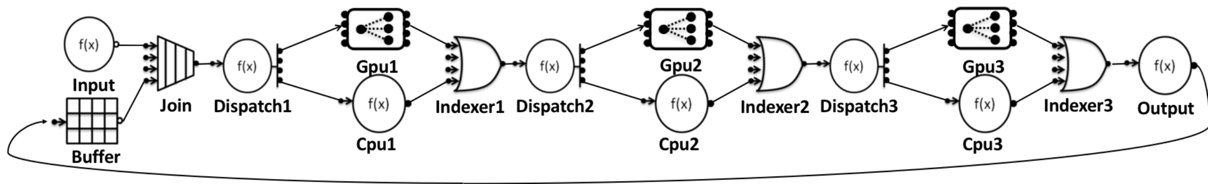

Fig. 2. Flow Graph de Vivid paralelizado.

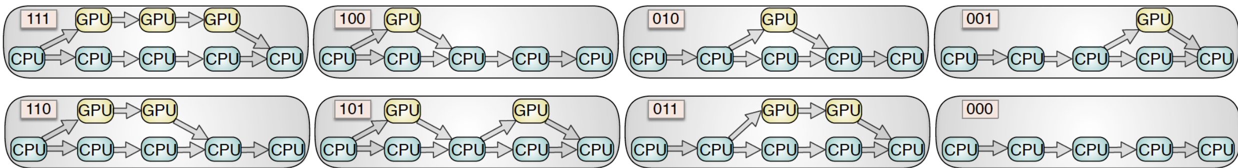

Fig. 3. Configuraciones ViVid.

recurso del sistema, listo para que pueda ocuparse de nuevo y procesar otra imagen. Mientras están los tokens disponibles "viajando" por el grafo, en el *Join* el grafo está bloqueado, de forma que no circula otra imagen hasta que no llegue un nuevo token por el *Buffer*. Hay ahí un flujo de datos que permanece cerrado y se abre cada vez que se ha liberado un nuevo token. De esta forma sólo circulan por el grafo tantos frames como tokens se han decidido crear. Cuando se analizan todos los frames disponibles, el *Input* se queda sin datos y es el que manda la señal de que el grafo deje de funcionar. Cuando el último frame acaba su procesamiento, el grafo termina y el programa finaliza.

Como se puede ver, esta implementación permite un uso circular y concurrente de los recursos del sistema, de forma que mientras queden imágenes que analizar, todos los recursos que se hayan querido emplear están en constante funcionamiento con la máxima eficiencia posible: cada thread (o recurso del sistema) analizando una imagen. Con esta implementación se consigue un alto grado de paralelismo. La implementación secuencial sólo permitía analizar una imagen al mismo tiempo, desde el comienzo hasta el final del algoritmo. Con la implementación basada en tokens y gracias a la potencia de *Flow Graph* se pueden estar ejecutando simultáneamente tantas imágenes como recursos disponibles tenga el sistema. Además, con el nuevo nodo de OpenCL, se permite añadir la enorme potencia que aporta la GPU al rendimiento de la aplicación, como se verá en el siguiente apartado de resultados.

A. Nodos empleados

Una vez explicada la estructura general del Grafo, que consiste en una serie de nodos conectados entre sí que permite el intercambio de mensajes y datos entre ellos, cada uno con una funcionalidad determinada. Se van a explicar cuales han sido los nodos empleados, con la funcionalidad implementada en cada uno de ellos [2] [11] [12].

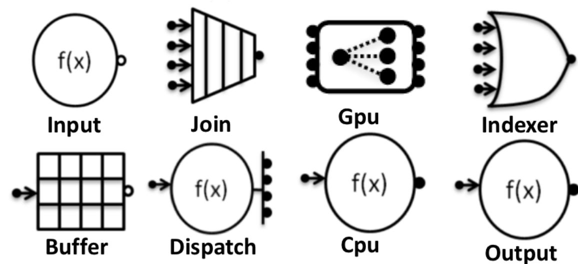

Fig. 4. Nodos empleados.

- **Input:** Nodo con el que comienza el programa. Es un nodo tipo *Source_node*. Es el que controla tanto el inicio como el final de la ejecución del grafo. Con la opción *input_node.activate()* se lanza la orden de que comience a generar mensajes de salida, y por tanto comienza el flujo de datos por el grafo. Este nodo sigue funcionando mientras tenga datos que lanzar al nodo con el que está conectado. Cuando se acaben sus datos, es decir, que ya se hayan lanzado todos los frames que se quieren procesar, el nodo dejará de emitir mensajes de salida. De esta forma, en cuanto los siguientes nodos acaben de procesar las últimas imágenes, el grafo se parará y acabará el programa. Es un nodo especial, pues normalmente los nodos reciben y devuelven mensajes, como si fueran argumentos de una función. Sin embargo, este nodo no recibe mensajes de entrada, sino que los genera él mismo, y pasa ese argumento como referencia al siguiente nodo. Realmente su salida es un bool, el cual mientras el nodo esté activo devuelve true, pasando a false cuando no tenga más que procesar. Este bool es el que controla cuando el grafo tiene que apagarse.
- **Buffer:** Nodo tipo *Buffer_node*. Se encarga de almacenar y trasladar los tokens desde su generación al comienzo del programa hasta el siguiente nodo, *Join*. Conecta con este nodo por un lado y por otro con el nodo de salida *Output*. De

esta forma se genera el grafo circular, cuando un token acaba una de las dos ejecuciones, vuelve al *Buffer*, para unirse a un nuevo identificador de imagen del nodo *Input* y comenzar de nuevo el procesamiento de un nuevo frame, mientras haya disponibles.

- **Join:** Nodo tipo *Join_node.* Se encarga de unir los datos procedentes del *Input* y del *Buffer*. Es decir, agrupa en una tupla de datos el identificador de imagen procedente del *Input* y el token procedente del *Buffer*. El camino que el identificador tomará a través de las etapas dependerá de si el token es de GPU o de CPU. Este nodo pasa dicha tupla al siguiente nodo, *Dispatch*. Este tipo de nodo se configura como "reserving". Esta particularidad significa que, el nodo no dará una salida hasta que haya una entra de cada tipo, sólo entonces agrupará todos los input en una tupla y generará una salida. La diferencia con la política predefinida de "queueing" es que ésta almacena los input en una cola, a la espera de que haya uno de cada tipo para generar una tupla. El modo "reserving" sólo los marca como disponibles, pero sin almacenarlos hasta que no haya para generar una tupla de salida. De esta manera sólo se consume un ítem de imagen cuando haya un token disponible para ella. Si se configurara como "queueing", el nodo *Input* mandaría rapidamente el bool de "false" para parar el grafo, pues hubiera pasado al *Join* todos sus datos sin ningún tipo de limitación.
- **Dispatch:** Nodo tipo *Dispatch_node.* Se encarga de recoger y analizar el token procedente, o bien del *Input* si esta en la primera etapa, o del *Indexer* correspondiente, si está en las etapas sucesivas. Si el token que ha recibido es un token GPU y está activada en esa etapa la GPU, envía el token a la ramificación GPU. Si por el contrario es un token CPU o de GPU sin tener activada la GPU en esa etapa, envía el token a la ramificación CPU. De esta forma se lanza el token a la ramificación GPU o CPU de la etapa correspondiente.
- **CPU:** Este nodo, es de tipo "function_node" y es donde se ha implementado cada una de las etapas del algoritmo para CPU. Este tipo de nodos se caracteriza por tener una sola entrada y una sola salida, a través de las cuales se reciben los argumentos, o datos, que se quieren procesar y se pasan al nodo siguiente. En cada una de las 3 etapas del algoritmo para CPU, este nodo recibe los datos procedentes del *Dispatch* correspondiente, que son los argumentos necesarios para procesar dicha etapa. A continuación ejecuta la etapa y recoge los resultados. Estos resultados se pasan por la salida del nodo al *Indexer* correspondiente, el cual se encargará de pasar dichos datos al proximo *Dispatch* y por tanto a la etapa siguiente, para que se complete el algoritmo.
- **Indexer:** Este nodo, tipo *Indexer_node*, es el que se encarga de unir la bifurcación CPU-GPU que se crea en cada etapa. Al contrario que el *Join*, este nodo no requiere de que haya un mensaje en cada una de las entradas, o conexiones, que tiene para lanzar los datos a la siguiente etapa. Si esto fuera así las etapas estarían limitadas a que se ejecute una etapa tanto en CPU como en GPU para poder continuar. Por el contrario este nodo permite que los mensajes pasen indistintamente nada más lleguen a su entrada, hacia la salida y vayan al nodo siguiente. Los mensajes se capturan con un identificador que le pone una etiqueta para saber de qué entrada viene, de forma que en el nodo con el que está conectado (el *Dispatch*), se puede hacer un análisis y determinar si ese mensaje ha venido desde la GPU o desde la CPU.
- **Output:** Nodo de salida del grafo, al que llegan los datos que ya han acabado de procesarse por todo el algoritmo a través de las 3 etapas de ViVid. En este nodo se "libera" los tokens y se devuelven al *Buffer*, para que se reciclen y vuelvan a procesar otro frame a través del grafo. Este nodo tiene la implementación adicional, que permite "deshacerse" de tokens de CPU, si en un determinado momento de la ejecución del programa se quiere liberar la CPU y que sólo trabaje la GPU cuando ya se han procesado un determinado porcentaje de los frames disponibles. Esto sirve para que en la parte final del programa, cuando van quedando pocos frames por analizar, la GPU no se quede ociosa y pueda seguir sacando todo el partido a su capacidad de procesamiento paralelo.
- **GPU:** Este nodo es un poco especial al resto, y por tanto se ha dejado para el final y así explicarlo en más detalle. Es el motivo del desarrollo de este trabajo, ya que este nodo (los 3 que componen las etapas GPU) son de tipo *OpenCL_node*, el nuevo nodo que ha desarrollado Intel para esta herramienta *Flow Graph* y que nos va a permitir descargar trabajo a la GPU de una forma muy cómoda para el programador.

  Este nodo se activa si ha recibido un token correspondiente desde el *Dispatch*. Una vez activado, se encarga de comenzar el camino para analizar una imagen en la GPU. Para poder usar variables tipo array en los nodos de OpenCL, se debe usar un tipo de dato especial: "opencl_buffer". Este tipo de dato, creado para OpenCL y su implementación en los *Flow Graph*, se encarga de abstraer al programador de toda la ardua tarea de crear el entorno necesario para poder transferir variables de la CPU a la GPU y procesarlas en un kernel OpenCL. Esta era una de las complicaciones existentes en la anterior implementación de ViVid, que ahora se ve significativamente simplificada gracias al nodo OpenCL_node.

  Las variables se crean en el main (para que

```
//Graph and OpenCL nodes definition
    opencl_graph g;
    opencl_program<> program(g, "Filtros_GPU_copy.cl");
    opencl_node<buffer_datosGPU1> GPU_F1(g, program.get_kernel("blockwise_distance_kernel"))
        ;
    opencl_node<buffer_datosGPU2> GPU_F2(g, program.get_kernel("cellHistogramKernel3"));
    opencl_node<buffer_datosGPU3> GPU_F3(g, program.get_kernel("pairwiseDistanceKernel"));
    std::array<int, 2> range1{height -2,width -2};
    std::array<int, 2> range2{n_parts_y_gpu, n_parts_x_gpu};
    std::array<int, 2> range3{n_total_coeff/dict_size, n_total_coeff/dict_size};
    GPU_F1.set_range(range1);
    GPU_F2.set_range(range2);
    GPU_F3.set_range(range3);
```

Fig. 5. Declaración nodos OpenCL.

```
    make_edge(input_node,input_port<0>(join_input));
    make_edge(token_buffer,input_port<1>(join_input));
    make_edge(join_input,dispatch_node);
    //
    make_edge(output_port<0>(dispatch_node), cpu_node);
    make_edge(output_port<1>(dispatch_node), input_port<0>(gpu_node));
    make_edge(output_port<2>(dispatch_node), input_port<1>(gpu_node));
    make_edge(output_port<3>(dispatch_node), input_port<2>(gpu_node));
    make_edge(output_port<4>(dispatch_node), input_port<3>(gpu_node));
    make_edge(output_port<5>(dispatch_node), input_port<4>(gpu_node));
    make_edge(output_port<6>(dispatch_node), input_port<5>(gpu_node));
    make_edge(output_port<7>(dispatch_node), input_port<6>(gpu_node));
    make_edge(output_port<8>(dispatch_node), input_port<7>(gpu_node));
    //
    make_edge(cpu_node,input_port<0>(node_indexer_1));
    make_edge(output_port<0>(gpu_node),input_port<0>(join_input2));
    make_edge(output_port<1>(gpu_node),input_port<1>(join_input2));
    make_edge(output_port<2>(gpu_node),input_port<2>(join_input2));
    make_edge(output_port<3>(gpu_node),input_port<3>(join_input2));
    make_edge(output_port<4>(gpu_node),input_port<4>(join_input2));
    make_edge(output_port<5>(gpu_node),input_port<5>(join_input2));
    make_edge(output_port<6>(gpu_node),input_port<6>(join_input2));
    make_edge(output_port<7>(gpu_node),input_port<7>(join_input2));
```

Fig. 6. Declaración make_edge.

sean accesibles por el *OpenCL_node*). Una vez creadas las variables necesarias para realizar el procesamiento en las distintas etapas GPU, se lanzan hacia el primer filtro y así comienza a ejecutarse el análisis de la imagen por parte de la GPU, repitiéndose el mismo proceso para las 2 etapas siguientes.
Se puede ver, en la Figura 5, los códigos de la creación y definición de los 3 nodos OpenCL, la carga de los códigos correspondientes a cada etapa, y la definición de los rangos de cálculo, para optimizar el paralelismo en GPU.

Por último, una vez explicados todos los nodos empleados, queda mostrar como se han realizado las conexiones entre ellos. Para unir dos nodos, en *Flow Graph* se emplea el método *"make_edge(X,Y)"* siendo X e Y los dos nodos que se quieren unir. De esta forma, los mensajes que salgan del nodo X van al nodo Y directamente, creándose así el grafo de nodos enlazados. Los nodos que tienen una única entrada y salida se conectan directamente a través de los *make_edge*. Sin embargo, si se tienen múltiples entradas y salidas, se deben unir cada una de éstas identificando si es entrada o salida mediante *input_port* u *output_port* respectivamente (y el número de la entrada o salida del nodo que ocupen), para cada uno de los elementos que entren o salgan del nodo. En la Figura 6 se puede ver un ejemplo de algunas de las conexiones que tiene el grafo.

IV. Resultados experimentales

En este apartado se van a mostrar los resultados conseguidos una vez realizadas las distintas pruebas de speedup y análisis de tiempos para distintas configuraciones paralelas del programa.

En primer lugar, se va a mostrar el hardware sobre el que se han realizado las pruebas de rendimientos que se van a describir. En particular, se ha utilizado un procesador heterogéneo de Intel, con GPU integrada con las características siguientes:

- CPU: Intel(R) Core(TM) i7-5775C CPU @ 3.30GHz con 6MB de cache.
- GPU: Intel(R) Iris Pro Graphics 6200 @ 300MHz y 128MB de memoria eDRAM.
- El sistema cuenta con 32GB de RAM.
- La maquina funciona sobre un sistema operativo CentOS Linux, versión 7.2.1511

Hay que recordar que el grafo tiene distintas configuraciones, en función de como se quiera descargar el trabajo a la GPU en las distintas etapas, como se vio en la Figura 3. Probar todas las configuraciones no sería recomendable ni eficiente. La forma de saber, por tanto, cual es la configuración óptima

es analizar los tiempos de procesamiento de cada una de las etapas, tanto en CPU como en GPU. En función de los análisis de tiempos en la versión serie del programa, se podrá ver en qué etapas es mejor lanzar a la GPU y cuales mantener en la CPU. Aunque, en un principio, lo lógico sería pensar que la configuración más eficiente va a ser la 111 (es decir, todas las etapas por la GPU), hay que analizar los tiempos de cada etapa para validarlo.

En la Figura 7 pueden observarse los resultados tras un análisis de tiempo por las 3 etapas, tanto por CPU como por GPU. El gráfico muestra que claramente la carga computacional está repartida entre las etapas 1 y 3, quedándose la etapa 2 con muy poca carga computacional. Es una etapa de reordenación de los resultados mediante un histograma, más que de realizar cálculos intensivos. A primera vista parece, por tanto, que la etapa 2 no va a aprovechar los recursos proporcionados por la GPU. Para asegurarse correctamente, se ha realizado un análisis de tiempos comparando un análisis de una imagen SD (414x600) frente a una imagen HD (1080x1980). En la Figura 8 se puede ver el mismo análisis de tiempo pero para la imagen en HD. En este caso se puede ver como, tanto para el procesamiento en CPU como GPU, la etapa 1 tiene más porcentaje de peso en el cómputo total que cuando se procesaba la imagen SD.

En la Figura 9 se puede ver los resultados numéricos del análisis de imagen SD vs HD. Como se puede volver a comprobar, la etapa 2 apenas consume tiempo de cálculo, en relación a las etapas 1 y 3. Además se puede comprobar como, la etapa 2 se ejecuta incluso antes en CPU que en GPU, al menos en la imagen SD. Esto es debido a que, ya que tiene tan poco tiempo de cálculo, el procesador pierde más tiempo en trasladar la información a la GPU y recoger los resultados que lo que tarda la CPU en procesar su etapa. Con esta información, es fácil deducir que la configuración más eficiente va a ser la 101, ya que en la etapa 2 no se va a perder tiempo en descargar el poco trabajo a la GPU y va a realizarse directamente en CPU.

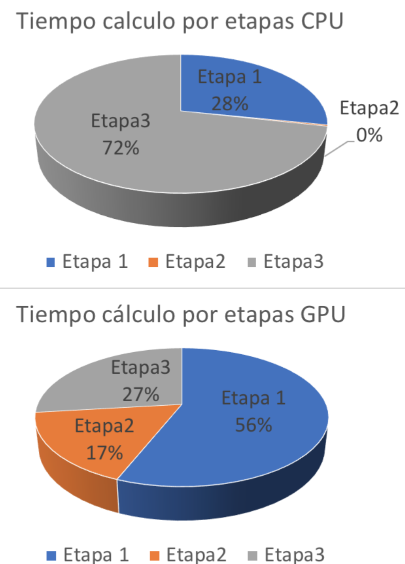

Fig. 7. Tiempo de cálculo por etapas CPU y GPU. SD.

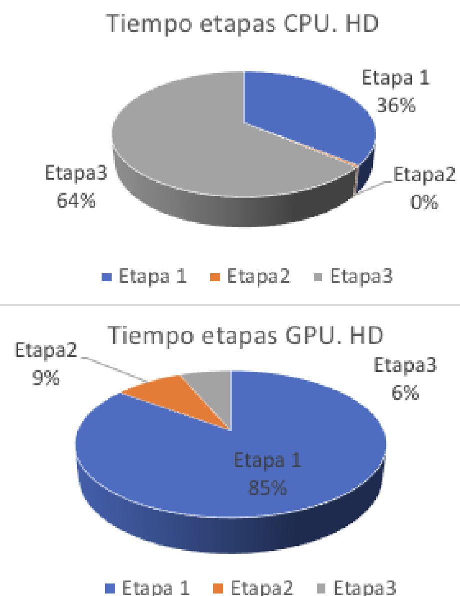

Fig. 8. Tiempo de cálculo por etapas CPU y GPU. HD.

A. Análisis de Speedup

Por tanto, se decide realizar una prueba de rendimiento para la configuración 101. Para ello se va a calcular el speedup con las distintas configuraciones de que se dispone: 1CPU, 2CPU, 3CPU, 4CPU, 1CPUGPU, 2CPUGPU, 3CPUGPU y 4 CPUGPU. Para cada una de estas configuraciones se va a emplear un tamaño de problema distinto: 100, 500, 1000 y 2000 frames. Para un análisis más completo, se va a realizar varios análisis de speedup, con estas configuraciones descritas, sobre distintos problemas:

- Speedup configuración 101 a Imagen SD: 416x600.
- Speedup configuración 101 a Imagen HD: 1080x1920.
- Speedup de la implementación de referencia basada en el template pipeline de TBB en lugar de FlowGraph con OpenCL_node.

Análisis de cada uno de los speedup:

- **Imagen SD**.
  En la Figura 10 se puede ver el speedup obtenido al procesar imágenes de baja resolución, SD.
  Se puede comprobar como hay dos zonas claramente diferenciadas:
  - La zona homogénea donde trabaja sólo la CPU. En ella se encuentran las distintas configuraciones desde la versión secuencial 1CPU, aumentando hasta que funciona la CPU con sus 4 cores (4CPU). En este caso se ve un speedup lineal, que va aumentando casi idealmente al aumentar el número de cores disponibles.

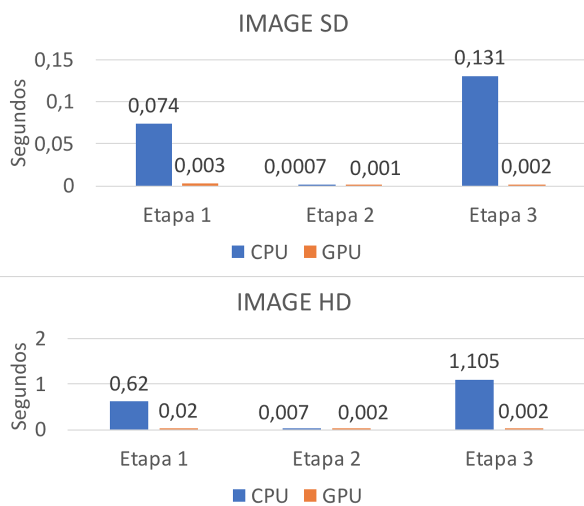

Fig. 9. Tiempo por etapas en SD vs HD.

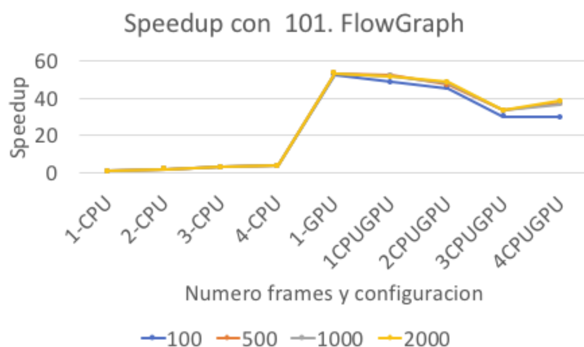

Fig. 10. speedup configuracion 101 FlowGraph.

– La zona heterogénea donde se añade la GPU a los cores CPU. Ahí se produce un gran aumento de speedup, al añadirle la enorme potencia de procesamiento paralelo que tiene la GPU. De hecho el speedup de la parte homogénea es algo pobre al lado del que se consigue al activar la GPU. De un speedup de 3,9 que se consigue con la configuracion 4CPU se pasa a un speedup de 50 al activar sólo la GPU y desactivar la CPU para el procesamiento del algoritmo.
Sin embargo cuando se va añadiendo los cores CPU para que funcione completamente el sistema heterogéneo se ve en el speedup como disminuye ligeramente, no pudiendo alcanzar el pico de rendimiento que se consiguió con la configuración GPU. Esto es debido a que la GPU que se utiliza es una GPU integrada en el mismo chip que la CPU, por tanto comparten la asignación o budget de potencia. La limitación en el consumo de potencia del chip hace que, cuando intenta funcionar a pleno rendimiento, no hay potencia suficiente para que todos sus componentes (4CPU y GPU) funcionen a toda su potencia. Esto provoca que el chip baje la frecuencia de la GPU y que por tanto procese imágenes con menor throughput que cuando estaba sola. No se alcanza el mismo rendimiento que solo con la GPU porque la transmisión de datos por el chip entre CPU y GPU consume ancho de banda de memoria y añade un overhead de sincronización entre dispositivos extra. Esto resulta en que las configuraciones heterogéneas CPU+GPU no lleguen a superar a la configuración con sólo GPU.

- **Imagen HD**.
En una segunda batería de pruebas, se ejecuta el algoritmo con una mayor carga computacional. Para ello usamos imágenes de mayor resolución que incluyen mayor cantidad de datos a procesar (más píxeles). Se espera, por tanto, que el speedup que se consiga con la paralelización sea mayor. Además, al estar la GPU especializada en el tipo de datos que se manejan en este trabajo, píxeles de imágenes, el paralelismo que se va a lograr con la GPU va a ser aun mayor. En la Figura 11 se puede ver el speedup que se consigue con este tamaño de datos, donde se puede observar como la parte CPU es practicamente igual el speedup, mientras que con la GPU se consigue alcanzar un speedup entorno a 100x (el doble que cuando se procesan imágenes SD).

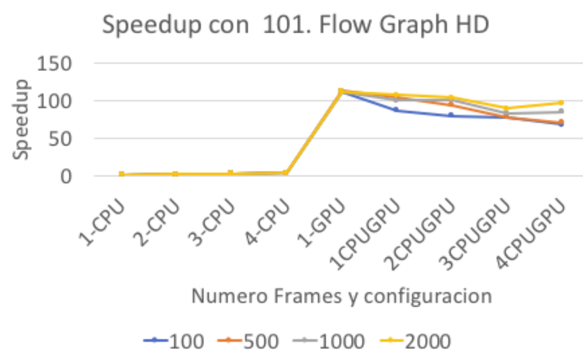

Fig. 11. speedup configuracion 101 FlowGraph. HD.

Por último se ha realizado una comparación con la anterior implementación que había de este algoritmo: una implementación basada en pipeline [13]. Esta implementación mantiene el mismo rendimiento si se utiliza una configuración homogénea de solo CPU. Sin embargo si se añade la componente heterogénea (GPU), aunque el rendimiento aumenta mucho (como era de esperar), no llega a alcanzar el pico de rendimiento que se ha descrito en la implementación de *Flow Graph*. Esto es debido a que la implementación de pipeline no explota la técnica de zero-copy buffer. Es decir, esta implementación previa se basa en las rutinas de OpenCL para implementar las operaciones de device-to-host y host-to-device explícitas. Esto acarraea un consumo extra de memoria y de tiempo perdido en movimiento de datos, que acaba limitando el rendimiento global. La implementación de *Flow Graph* donde se utiliza el nuevo nodo OpenCL, emplea sin embargo "zero-copy-buffer" para los datos que se quieren procesar

en la GPU. Esta técnica evita la copia y creación de nuevos buffers por lo que se ahorra bastante procesamiento y tiempo de copia de datos, resultando el speedup de la Figura 10.

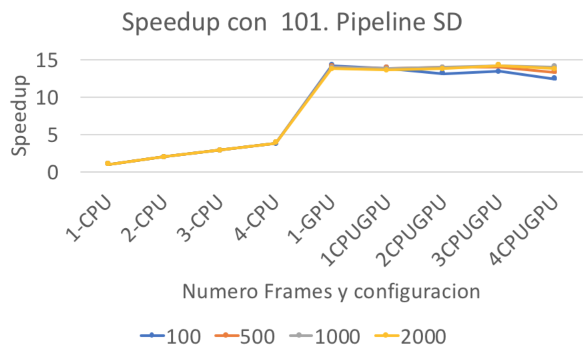

Fig. 12. speedup configuracion 101 Pipeline SD.

V. Trabajos relacionados

Una aproximación para programar aplicaciones de Streaming, es usar un lenguaje de programación que soporte Streams, por ejemplo SreamIt [14]. Pero actualmente estos enfoques no proveen soporte para ejecuciones heterogéneas CPU-GPU.

Mediante el uso de cores CPU y GPU, el cómputo simultáneo en plataformas heterogéneas ofrece un mayor rendimiento que solo CPU o solo GPU [15]. Sin embargo, los frameworks de programación que proporcionan soporte para la computación en arquitecturas heterogéneas como [16], [17], [18], [19] solo consideran el tiempo de ejecución al decidir la distribución de tareas entre los núcleos de CPU y GPU.

La diferencia entre estos trabajos relacionados y el nuestro radica en que se centran en los patrones de datos paralelos, mientras que nosotros nos centramos en aplicaciones de Streaming. El trabajo [13], con una estructura pipeline para realizar la distribución de trabajo en la arquitectura heterogénea CPU-GPU, es quizás el más cercano al nuestro. La diferencia con respecto al trabajo actual es que nosotros actualizamos la implementación mediante el uso de las clases TBB *Flow Graph*, en vez de una estructura pipeline. Además, gracias al uso de los nodos OpenCL conseguimos mejores rendimientos en GPU al explotar de forma transparente al programador la optimización de zero-copy-buffer.

VI. Conclusiones

Las principales aportaciones de este trabajo se listan a continuación:

- Se ha estudiado en profundidad la librería Intel TBB y su nuevo módulo OpenCL: Intel proporciona con TBB un modelo de programación que facilita la implementación de aplicaciones paralelas. Este nuevo módulo OpenCL está disponible desde fechas recientes, por lo que aún está en proceso de desarrollo. Precisamente con este trabajo queremos contribuir a la evaluación y posible mejora del mismo. Por lo pronto, hemos comprobado que efectivamente facilita la programación de código heterogéneo que descarga trabajo a GPUs compatibles con OpenCL.
- Se ha paralelizado con éxito la aplicación ViVid mediante un *Flow Graph* de Intel TBB y nodos OpenCL. De esta forma hemos conseguido además la ejecución heterogénea (CPU + GPU) de nuestra aplicación.
- Se ha realizado un estudio de escalabilidad del algoritmo paralelizado: Hemos realizado un estudio de escalabilidad mostrando gráficas de speedup de las versiones paralelas con respecto a la secuencial. De esta forma comprobamos la ganancia que podemos obtener usando la GPU en este tipo de aplicaciones de visión. Además se ha mejorado el rendimiento respecto a la anterior implementación basada en un template TBB pipeline. Esto es debido a que las clases OpenCL de FlowGraph emplean la técnica zero-copy-buffer, optimización no implementada en la versión basada en pipeline.
- Finalmente, se ha experimentado con el potencial de las arquitecturas heterogéneas: las arquitecturas heterogéneas llevan ya un tiempo entre los usuarios, sin embargo, debido a las dificultades que encontramos para programar este tipo de plataformas, no es frecuente que los desarrolladores exploten todo el potencial de las mismas. Creemos que TBB y el nuevo módulo OpenCL facilitan el desarrollo de aplicaciones heterogéneas y permiten de forma productiva exprimir el rendimiento de las plataformas con CPU+GPU.

Agradecimientos

El presente trabajo ha sido financiado mediante el proyecto TIN2016-80920-R por parte del Ministerio de Economía, Industria y Competitividad, perteneciente al Plan Nacional de Investigación destinado al estudio de arquitecturas de altas prestaciones para aplicaciones intensivas de datos, y por la Universidad de Málaga (Campus de Excelencia Internacional Andalucía Tech).

Referencias

[1] Rafael Asenjo, "Why multicores and how to exploit them. apuntes de programacion en multiprocesadores," *Master en Ingenieria Mecatronica*, 2017.
[2] "Intel TBB Developer Guide," https://software.intel.com/en-us/node/506045, Accessed: 2018-05-26.
[3] James Reinders, "The parallel universe," 2017.
[4] Mike Voss, Pablo Reble, and Jackson Marusarz, "CPUs,GPUs,FPGAs: Managing the alphabet soup with Intel Threading Building Blocks," 2017.
[5] Mert Dikmen and Huang Thomas S., "Dissertation of visual detection and recognition using local features," *University of Illinois*, 2012.
[6] Mert Dikmen, Josep Torrellas, and Sayan Mitra, "Dissertation of visual detection and recognition using local features," *University of Illinois*, 2014.
[7] "OpenCL Description," http://www.amd.com/es-xl/solutions/professional/hpc/opencl, Accessed: 2018-05-27.
[8] "Tutorial Intel TBB," http://www.inf.ed.ac.uk/teaching/courses/ppls/TBBtutorial.pdf, Accessed: 2018-05-26.

[9] "Tutorial Intel TBB Web," https://www.threadingbuildingblocks.org/intel-tbb-tutorial, Accessed: 2018-05-26.

[10] Alexei Katranov and Alexey Kukanov, "Intel threading building block (intel tbb) flow graph as a software infrastructure layer for opencl-based computations," pp. 1–3, 04 2016.

[11] "Intel TBB Developer Reference," https://software.intel.com/en-us/node/506130, Accessed: 2018-05-26.

[12] "FlowGraph Node-Summary," https://nlguillemot.wordpress.com/2016/10/24/tbb-flow-graph-nodes-summary, Accessed: 2018-05-26.

[13] A. Vilches, A. Navarro, R. Asenjo, F. Corbera, R. Gran, and M. J. Garzarán, "Mapping streaming applications on commodity multi-cpu and gpu on-chip processors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 4, pp. 1099–1115, April 2016.

[14] Robert Soule, Michael I. Gordon, Saman Amarasinghe, Robert Grimm, and Martin Hirzel, "Dynamic expressivity with static optimization for streaming languages," in *Intl Conf on Distributed Event-Based Systems*, Jun. 2013.

[15] Canqun Yang et al., "Adaptive optimization for petascale heterogeneous CPU/GPU computing," in *CLUSTER*, 2010, pp. 19–28.

[16] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim, "Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *MICRO 42*. 2009, pp. 45–55, ACM.

[17] J. Planas et al., "Self-adaptive OmpSs tasks in heterogeneous environments," in *Proc. of IPDPS*, 2013.

[18] T. Gautier, J.V.F. Lima, N. Maillard, and B. Raffin, "XKaapi: A runtime system for data-flow task programming on heterogeneous architectures," in *IPDPS 2013*, 2013, pp. 1299–1308.

[19] C. Augonnet et al., "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurr. Comput. : Pract. Exper.*, vol. 23, no. 2, Feb. 2011.

[20] Rafael Asenjo and Angeles Navarro, "Intel TBB tutorial," 2017.