UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE MÁLAGA

Graduado en Ingeniería Informática

Plantilla parallel_for heterogénea implementada en Intel oneAPI

Intel oneAPI implementation of a heterogeneous parallel_for template

Realizado por
Juan Pedro Domínguez Berdún

Tutorizado por
Rafael Asenjo Plaza

Departamento
ARQUITECTURA DE COMPUTADORES
UNIVERSIDAD DE MÁLAGA

MÁLAGA, Febrero 2021

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

GRADUADO EN INGENIERÍA INFORMÁTICA

PLANTILLA PARALLEL_FOR HETEROGÉNEA IMPLEMENTADA EN INTEL ONEAPI

INTEL ONEAPI IMPLEMENTATION OF A HETEROGENEOUS PARALLEL_FOR TEMPLATE

Realizado por

JUAN PEDRO DOMÍNGUEZ BERDÚN

Tutorizado por

RAFAEL ASENJO PLAZA

Departamento

ARQUITECTURA DE COMPUTADORES

UNIVERSIDAD DE MÁLAGA

MÁLAGA, JUNIO 2021

Fecha de la defensa: Febrero de 2021

A mi familia y a Sara por apoyarme todo el tiempo mientras hacía este proyecto

Resumen

Este trabajo consiste en el desarrollo e implementación de un *template* de alto nivel, basado en la librería oneTBB de Intel, que permita la ejecución de bucles paralelos de forma heterogénea (Heterogeneous Parallel For). El *template* se apoya en un planificador llamado LogFit que permite el reparto automático de la carga entre los cores de CPU y la GPU. Se proporciona una implementación basada en oneTBB 2020 que usa las clases Flow Graph con soporte de nodos OpenCL, y otra basada en oneAPI/SYCL y oneTBB 2021 que permite mantener en el mismo fuente el código de GPU y el de CPU. Este nuevo *template* permite la ejecución de forma fácil y eficiente de bucles paralelos sobre plataformas heterogéneas CPU-GPU, simplemente proporcionándole el rango de iteraciones del bucle, la función a ejecutar en CPU (en caso de ser este el dispositivo elegido para ejecutar un subrango de las mismas) y el kernel (en lenguaje OpenCL o en SYCL) a ejecutar en la GPU (en caso de ser esta la elegida).

Además de la implementación del *template* y la comprobación de su correcto funcionamiento, se ha realizado una comparación de rendimientos entre distintas implementaciones de dos *kernels* computacionales. Además del rendimiento se ha evaluado tanto la sobrecarga de la nueva abstracción y la reducción del esfuerzo de programación.

Palabras clave: **C++, TBB, OpenCL, Computación paralela, Computación heterogénea, GPU, oneAPI, DPC++, SYCL**

Abstract

This work consists in the development and implementation of a high level template based on the Intel oneTBB library, which allows the heterogeneous execution of parallel loops (Heterogeneous Parallel For) so that they can run on CPU+GPU platforms. This template leverages the LogFit scheduler that automatically distributes the computational load between the CPU cores and the GPU. We provide an implementation based on oneTBB 2020 that uses the Flow Graph classes with OpenCL nodes support, and a different one based on oneAPI/SYCL and oneTBB 2021 that takes advantage of the single-source programming paradigm offered by SYCL. This new template is intended to get easier and more efficient implementations of heterogeneous parallel loops that can run on heterogeneous architectures (CPU-GPU). To that end, the user only has to provide the loop range, the function that CPU will run (in case CPU is the chosen device that is going to run a subrange of iterations) and the kernel (written in OpenCL language or in SYCL), which is offloaded to the GPU (if it is the chosen device).

In addition to the template implementation and verification of its correct behavior, a performance evaluation among different kernel implementations has been carried out. Moreover, the abstraction overhead due to the high level template and the programming effort reduction have been also assessed.

Key words: C++, TBB, OpenCL, Parallel computing, Heterogeneous computing, GPU, oneAPI, DPC++, SYCL

Índice

Índice de figuras

Índice de Tablas

1 Introducción

1.1. Motivación

En muchos casos es importante explotar simultáneamente la potencia de las CPUs y la GPU para conseguir la eficiencia deseada (ya sea en cuanto a rendimiento o en cuanto a consumo de energía). Este reparto de trabajo óptimo entre los distintos dispositivos es todo un reto sobre todo en aplicaciones con una carga de trabajo irregular. En el grupo de investigación "Parallel Programming Models and Compilers" del Departamento de Arquitectura de Computadores de la Universidad de Málaga se ha abordado este problema desde hace algunos años. En este tiempo han propuesto distintas plantillas C++ basadas en Intel TBB [17], así como una variedad de planificadores que se encargan de distribuir la carga entre la CPU y la GPU (o más recientemente también la FPGA).

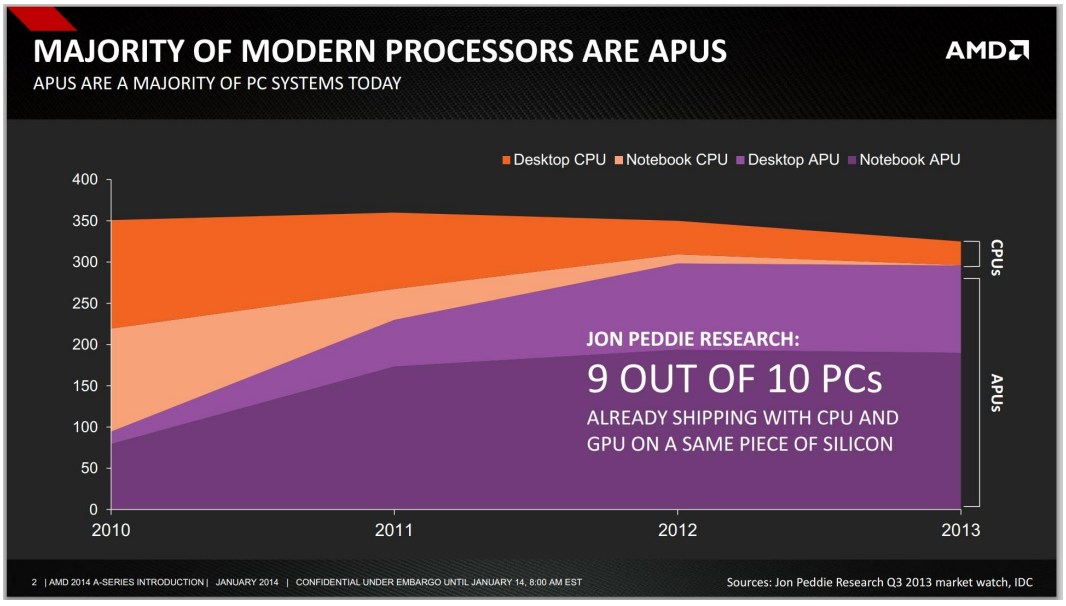

Figura 1: Porcentaje de dispositivos APU (GPU y CPU en un mismo chip) ya en el mercado.

Desde que se democratizaron las arquitecturas multicore a finales de la primera década de

este siglo, se puso de manifiesto la importancia de adaptar (paralelizar) las aplicaciones para que puedan sacar el máximo partido de estas arquitecturas. Más recientemente se han complicado aún más los procesadores de las plataformas de computación ya que han incorporado diversos aceleradores, más prominentemente, unidades gráficas o GPUs. Es en la implementación de aplicaciones para estas modernas arquitecturas heterogéneas (Figura 1) donde surge la necesidad de facilitar a los desarrolladores el aprovechamiento óptimo de los distintos recursos de procesamiento. El *framework* oneAPI[1] [9] de Intel surge como una solución que extiende el lenguaje C++ y que simplifica el mapeo de las distintas tareas en que se puede descomponer una aplicación sobre una arquitectura paralela heterogénea (con CPUs, GPUs y FPGAs).

Dentro de oneAPI encontramos la librería oneTBB[2] [10] de Intel, implementada sobre el lenguaje C++[3]. Dicha abstracción libera al programador de la necesidad de gestionar detalles de bajo nivel (gestión de threads, tareas, sincronización, etc.), permitiendo que se preocupe solamente de descomponer las funcionalidades de su aplicación en tareas sencillas que se ejecutarán de forma dinámica y balanceada, empleando el algoritmo de planificación subyacente *work-stealing*[4]. La librería oneTBB arranca un número configurable de hebras de ejecución, que por defecto será igual al número de núcleos de proceso de la unidad de computación, y que balanceará la carga de los mismos mediante el planificador de work-stealing integrado.

Aunque este modelo puede ser eficaz, frecuentemente debemos aprovechar no solo una unidad de computación, sino CPU y GPU de forma simultánea para poder alcanzar los niveles de consumo de energía o tiempo deseados. Dicho balanceo entre las unidades de proceso se complica en la medida que los distintos dispositivos exhiben diferentes capacidades computacionales y los algoritmos mapean de forma diferente en las distintas unidades de computación. En el grupo de investigación "Parallel Programming Models and Compilers" del departamento de Arquitectura de Computadores de la Universidad de Málaga (departamento donde se desarrolla este TFG), han propuesto un algoritmo de planificación y particionamiento adap-

[1] Próxima generación de las librerías de paralelización que soporta mejoras en legibilidad, eficiencia y simplicidad en la programación de aplicaciones heterogéneas

[2] TBB o Threading Building Blocks, ofrece clases de C++ moderno y programación genérica para explotar paralelismo de tareas en arquitecturas multicore de memoria compartida.

[3] C++ es uno de los lenguajes de programación más populares y sin duda el que mejor combina propiedades como prestaciones, abstracción a coste cero, composability y separación de intereses (separation of concerns).

[4] Work-stealing es un planificador que balancea dinámicamente la carga computacional entre distintos threads.

tativo de bucles paralelos que ha sido diseñado con el propósito de mejorar las posibilidades de ejecución en arquitecturas heterogéneas de aplicaciones como las mencionadas anteriormente, específicamente aquellas con acceso irregular a datos. Dicho algoritmo, denominado LogFit[5] [8], proporciona de forma dinámica un rango óptimo de iteraciones del bucle para ser ejecutado en GPU, así como para las CPU's, evitando que alguno de los procesadores quede sobrecargado o a la espera de datos.

Este algoritmo ha sido implementado en un *template*[6] de alto nivel (Heterogeneous Parallel_For) con el objetivo de simplificar el trabajo de los desarrolladores a la hora de implementar bucles paralelos, aumentando la productividad y eficiencia de las computaciones de dichos bucles en arquitecturas heterogéneas. Este Heterogeneous Parallel_For se ha implementado usando la librería TBB de Intel (concretamente usando el *template pipeline* que proporciona la misma) para la gestión de la ejecución paralela de tareas y OpenCL[7] [5] para la ejecución de código en la GPU. La utilidad de librería *parallel_for* da soporte a paralelización en CPU, así como el lenguaje OpenCL nos proporciona el soporte a la ejecución de código en GPU, los *kernels*[8].

Durante las ultimas versiones de la librería TBB, Intel ha proporcionado soporte a la ejecución de tareas programadas en lenguaje OpenCL sobre dispositivos GPU y CPU, siempre que el dispositivo sea compatible con el *driver*[9] de OpenCL instalado en el computador (por ejemplo, las GPUs de procesadores Intel, AMD o ARM). La librería oneTBB ha añadido dicho soporte mediante la característica llamada *opencl_node* [13], que simplifica el aprovechamiento de arquitecturas heterogéneas, especialmente aquellas con GPU integrada, como los procesadores Intel Core i7, los AMD APU o los Qualcomm Snapdragon. Sin embargo, esta característica

[5] El nombre hace referencia al "Logarithmic Fitting´´ que subyace y que determina el particionado del espacio de datos.

[6] Abstracción sobre una librería que permite su ejecución sin tener en cuenta detalles sobre los tipos de datos con los que opera, tal como una cola no tiene en cuenta los datos de sus elementos para proporcionar funcionalidades como encolar o desencolar.

[7] OpenCL es tanto un lenguaje de programación como una API que permiten crear aplicaciones que aprovechen el potencial de paralelismo de los dispositivos gráficos y CPUs.

[8] Un kernel es una rutina que es ejecutada por un dispositivo acelerador, que bien puede tratarse de una GPU, la propia CPU o una FPGA.

[9] Es el software encargado de controlar un dispositivo hardware, necesario para interactuar desde otro dispositivo hardware o software.

está limitada al *template* de alto nivel *flow_graph*[10] de la librería. Por otro lado, usando el lenguaje DPC++ de oneAPI, podemos simplificar el desarrollo del código que se ejecuta en GPU. En esta segunda opción, el programador escribe directamente en C++, y en el mismo fichero que el código de CPU también podemos incluir el kernel que se lanza para ejecutar en GPU. Con todo esto, este trabajo pretende implementar el algoritmo de planificación LogFit en un *template parallel_for* de alto nivel utilizando ahora las nueva características de oneAPI disponibles: i) el *opencl_node* de *flow_graph*; y ii) el modelo de programación DPC++ que evita el uso de OpenCL.

Esta adaptación simplifica tanto conceptualmente como a nivel de usuario programador la implementación del mencionado Heterogeneous Parallel_For, así como la utilización de este *template* en futuras implementaciones software que puedan servirse de este procesado en paralelo de los datos. Si bien no siempre es posible paralelizar cualquier aplicación completamente, por la naturaleza intrínseca del problema a enfrentar, si es frecuente que parte de la misma sea susceptible de ser ejecutada en paralelo.

Comúnmente, el cómputo en paralelo requiere de creación de hebras, orquestación de los flujos de ejecución y protección ante manipulaciones no deseadas de la memoria compartida entre estas hebras y el proceso principal. Nuestra implementación no requiere manejadores para estas estructuras, pues la librería proporcionada por Intel es quien gestiona la concurrencia asociada al cómputo paralelo en arquitecturas multicore[11] heterogéneas (MPSoC). En el caso de oneTBB, solo debemos diseñar un grafo de alto nivel con nodos de ejecución de tareas especificas para CPU y GPU, así como el particionado del espacio de iteraciones a tratar por cada nodo o dispositivo. Dicho particionado es llevado a cabo por la solución ya implementada que es base de este TFG, el algoritmo Logfit, y dejando en manos del usuario del *template* la codificación de la rutina que se aplicará dentro del cuerpo del bucle, en el lenguaje C++ y en OpenCL o DPC++. De esta forma, con dos procedimientos definidos, se sirve de nuestro *template* para paralelizar fácilmente la ejecución del cuerpo del bucle sobre todo el espacio de iteraciones. Mientras que la implementación en DPC++ no requiere de OpenCL, al especifi-

[10] Se trata de un template que encapsula el flujo y encolado de tareas, datos y operaciones de memoria hacia una o varias unidades de procesamiento, dispositivos o hebras de ejecución.

[11] Que posee mas de una unidad de procesamiento.

carse el kernel en C++, pero sí de ambos procedimientos definidos (función en CPU y kernel en GPU).

1.2. Objetivos

El TFG consistirá en el desarrollo e implementación de un *template* de alto nivel empleando oneAPI de Intel que permita la ejecución de bucles paralelos de forma heterogénea (Heterogeneous Parallel_For) basado en el algoritmo LogFit y usando para ello, primero el nuevo *template flow_graph* con soporte de nodos OpenCL, y luego el lenguaje DPC++ para poder prescindir del lenguaje OpenCL y programar los dispositivos aceleradores en C++.

Este nuevo *template* implementado en este TFG permitirá la ejecución de forma fácil y eficiente de bucles paralelos sobre plataformas heterogéneas CPU-GPU, simplemente proporcionándole el rango de iteraciones del bucle; la función a ejecutar en CPU, cuando se trate este como el dispositivo elegido para ejecutar un subrango de iteraciones y el kernel en lenguaje OpenCL o C++ (según el caso tratado), a ejecutar en la GPU cuando sea esta la elegida por el algoritmo como dispositivo de ejecución de otro subrango.

En el caso de utilizar el *opencl_node*, se creará un *flow_graph* que reemplaza al *pipeline*[12] de la implementación actual, y todo el código de soporte para la ejecución de código OpenCL, por nodos OpenCL del *flow_graph*. En DPC++ se conservará el *pipeline*, pero se modificarán las estructuras de datos para utilizar punteros USM(*Unified Shared Memory*) y se adaptará el planificador para aceptar *kernels* DPC++.

Además de la implementación del *template* y la comprobación de su correcto funcionamiento, se hará una comparación de rendimiento entre la implementación secuencial y las basadas en *flow_graph* y DPC++. Asimismo se comparará frente a la implementación llevada a cabo con oneAPI, donde se ejecutarán una serie de pruebas de rendimiento y se comprobará la sobrecarga (si es que la hay) de la nueva abstracción introducida en oneAPI al incrementarse el nivel de abstracción sobre los dispositivos hardware.

Se revisará la bibliografía y documentación relativa a oneTBB, oneAPI y,en particular, al *template flow_graph* con soporte de nodos OpenCL y la extensión de C+++, DPC++. También analizaremos la implementación actual del planificador LogFit basado en el *pipeline*. Una vez

[12]Nombre que recibe la actual implementación heterogénea basada en oneTBB que no utiliza flow_graph.

re-escrito el planificador con el nuevo API[13], se experimentará en la plataforma de pruebas para realizar una comparación cuantitativa de la nueva implementación con respecto a las otras dos.

El proceso por el cual se llevarán a cabo los objetivos anteriormente descritos, será el siguiente:

- Revisión de manuales, tutoriales y libros de Intel oneTBB y oneAPI. Nos serviremos de la documentación publicada tanto por el equipo del Departamento de Arquitectura de Computadores, como de la oficial que proporciona Intel acerca del *template flow_graph* y oneAPI en sus enlaces web y repositorios de código libre.

- Estudio del planificador existente basado en *pipeline* sobre TBB. Analizaremos el modelo seguido para definir el orden y la forma en la que se ejecutan las tareas en el *pipeline* para tratar de llevar a cabo una función equivalente mediante el modelo de grafos de la implementación nueva y su adaptación a oneAPI.

- Re-implementación del planificador usando *flow_graph* con soporte de nodos OpenCL. Utilizaremos la librería oneTBB para, en conjunto con el actual algoritmo de particionado, llevar a cabo una posible mejora en el *pipeline*.

- Re-implementación del planificador usando DPC++. Utilizaremos la librería oneAPI para, en conjunto con el actual algoritmo de particionado, llevar a cabo una posible mejora en el *pipeline* eliminando la necesidad de OpenCL.

- Evaluación experimental en plataforma con procesador Intel y GPU integrada. Una vez el nuevo *template* este desarrollado, llevaremos a cabo diversas pruebas de ejecución con los diferentes modelos y sobre las mismas condiciones con el propósito de compararlos en rendimiento y simplicidad de uso.

- Extracción de resultados y conclusiones. Una vez finalizada la fase experimental, estará en nuestra mano decidir el *template* que mejor se adapte a las necesidades del problema, pues a pesar de cierta penalización en el rendimiento, podría resultar ventajoso el uso de uno u otro template bien por la naturaleza del problema o bien por la capacidad del

[13] Interfaz de programación de la aplicación.

mismo de expresarse de una forma u otra, lo que nos decantaría por una implementación concreta frente a sus rivales.

1.3. Resultados esperados

A priori, previo a la realización de los *templates*, así como a su posterior prueba en rendimiento, complejidad y utilización por parte de los usuarios, prevemos una pérdida de eficiencia en el rendimiento, como resultado de la capa de abstracción que Flow Graph y oneAPI aportan a la ejecución de *kernels* en la GPU.

A pesar de ello, también cabe esperar una reducción en la complejidad asociada al uso del *template* nuevo frente al Heterogeneous Parallel_For al no requerir de manejo de estructuras de datos complejas asociadas a OpenCL ni menos aún en oneAPI, donde prescindimos de OpenCL completamente.

Es en esta combinación entre rendimiento y usabilidad donde este proyecto arrojará luz sobre las mencionadas tecnologías e implementaciones. Por último deberemos tener en cuenta que el *pipeline* adapta un modelo genérico para CPU a ser utilizado conjuntamente con la GPU, mientras que las implementaciones nuevas son genéricas y contienen la funcionalidad que en el *pipeline* se ha desarrollado por si mismo, sin azucarillos sintácticos de ningún tipo.

Algunos de los motivos de dicho resultado esperado se resumen principalmente en el como la librería oneTBB y oneAPI deberá gestionar los recursos para que las estructuras propias de OpenCL, que son necesarias cuando ejecutamos código en GPU (aunque en DPC++ utilicemos *kernels* en C++, estas estructuras en lenguaje intermedio son necesarias), sean creadas de la forma más eficiente posible y permitan a los hilos de ejecución de la propia librería que introducen cierto retraso en la comunicación con la GPU sin llegar a penalizar el rendimiento del código ejecutado. En el caso de *flow_graph*, cuando profundizamos en la ejecución de estos nodos OpenCL, nos percatamos de que para cada una de sus invocaciones, requieren varias llamadas a funciones de librería como el selector de dispositivo de ejecución del kernel o la propia comunicación entre nodos adyacentes.

1.4. Estructura de la memoria

La presente memoria consta de 5 capítulos, incluyendo esta Introducción. A continuación, se detalla el contenido de cada capítulo:

1.4.1. Tecnologías

En este capítulo se realiza una visión general de las tecnologías que serán utilizadas, explicándolas brevemente, así como una pequeña introducción de la computación paralela.

1.4.2. Ejecución heterogénea de bucles paralelos

En este capítulo se profundiza en la librería desarrollada en el proyecto, realizando una comparación con el *pipeline* explicando el funcionamiento de algunos de sus mecanismos, así como mostrando la arquitectura que sigue el grafo implementado, su diseño a nivel de código y las partes fundamentales que deben ser implementadas por el usuario programador del *template*. También se desarrolla la extensión a oneAPI para cada uno de los planificadores, el basado *pipeline* y el basado en Flow Graph.

1.4.3. Resultados experimentales

En este capítulo se realiza una comparativa experimental del proyecto, discutiendo los resultados obtenidos en las diferentes implementaciones. Asimismo compararemos las complejidades y esfuerzo necesarios para poder utilizar y extender el *template*.

1.4.4. Conclusiones y líneas futuras

En este capítulo se mostraran las conclusiones obtenidas en el proyecto y las posibles líneas futuras para la continuación del mismo.

2

Tecnologías

2.1. Lenguaje de programación C++

Tanto para el desarrollo de la implementación actual como para el de la solución llevada a cabo en este TFG, se ha utilizado el lenguaje de programación C++ en sus ultimas versiones, C++14 empleada mayoritariamente por la propia librería oneTBB de Intel[11], pues el nuestra adaptación del Heterogeneous Parallel_For esta principalmente implementada en C++11 [6] (*flow_graph*) y C++17(DPC++[3]).

Comenzaremos explicando en que consiste y cuales son las principales características del lenguaje, así como las ventajas que aporta su uso en implementaciones de este estilo y las funcionalidades que presenta C++ moderno frente a las anteriores versiones del estándar, C++98 y C++03. El lenguaje de programación C++ es un lenguaje multipropósito, multiplataforma y enmarcado en el paradigma de los lenguajes imperativos y orientados a objetos. Nace como una extensión del lenguaje de programación de sistemas C, del que hereda su sintaxis y sistema de tipos, entre otros pilares fundamentales de C++, en la Figura 2 vemos una rápida comparación entre el mismo "Hola Mundo" escrito para cada lenguaje.

Aporta, en sus comienzos, el concepto de clase y orientación a objetos, frente a C. Esta orientación a objetos se refleja introduciendo el concepto de clase como forma de tipar y agrupar conceptualmente el concepto de función[14] o método[15] y el struct[16] con el que los datos y

[14]En programación, definimos una función como una sección de código que para determinados parámetros de entrada, produce una determinada salida. Un ejemplo de función simple seria cualquier operador aritmético como la suma de enteros

[15]En programación, definimos un método como una sección de código aislado que no produce una determinada salida, sino que altera comportamientos de otras entidades como objetos, estas entidades pueden ser argumentos de entrada o de carácter global

[16]En programación en C y C++, se trata de una unidad lógica que relaciona variables entre si, de forma que su asignación de valores en la memoria es colindante

```
#include <iostream>
int main() {
    std::cout << "Hola Mundo." << std::endl;
    return 0;
}

#include <stdio.h>
int main() {
    printf("Hola Mundo.\n");
    return 0;
}
```

Figura 2: Comparación del programa Hola Mundo entre C++ (arriba) y C (abajo)

las rutinas ahora se aglutinan en un mismo lugar, el objeto. Los objetos en C++ se caracterizan por tener variables miembro (propiedades) y funciones miembro (métodos), pero además, adquieren estado con lo que la evolución del programa se define por como manipulamos e interactuamos con estos objetos y ellos entre si.

Como mencionamos anteriormente, C++ es un lenguaje de propósito general que además es capaz de ser compilado y ejecutado en prácticamente todas las arquitecturas modernas (x86, x64, ARM, etc.) así como en los principales sistemas operativos (Linux, Windows, MacOS, etc), en nuestras pruebas experimentales hemos optado por la utilización del SO Linux sobre una arquitectura x86 debido a alta disponibilidad de esta plataforma, la posibilidad de monitorizar el uso de recursos y a que Intel proporciona el código fuente de oneTBB para ser compilado de forma óptima en su arquitectura x86. Es esta capacidad lo que otorga su potencia como lenguaje para gran parte de las aplicaciones que no estén fuertemente ligadas al sistema operativo que las ejecuta.

Nuestra implementación se ha basado en C++11 que aporta gran cantidad de ventajas frente a la versión anterior, C++03, entre ellas mejoras y mayor preponderancia en el uso de los *templates*, donde ya no es necesario especificar que el tipo esperado sea una clase determinada, extendiéndose a un tipo genérico, que podría ser un struct, class o tipo base del lenguaje (ver ejemplo en Figura 3).

C++11 incluye expresiones lambdas ya incluidas en la gran mayoría de los lenguajes mo-

```
#include <iostream>

template<typename TipoNumero>
struct Punto {
    TipoNumero x, y;
};
struct PintarPunto {
    template<typename TipoNumero>
    void operator()(Punto<TipoNumero> pto) {
        auto x{pto.x};
        TipoNumero y{pto.y};
        std::cout << "Coordenada X:" << x << std::endl << "Coordenada Y:" << y << std::endl;
    }
};
int main() {
    Punto<int> puntoEntero{-1, 1};
    Punto<double> puntoReal{-0.5, 0.5};
    PintarPunto pintarPunto;
    pintarPunto(puntoEntero);
    pintarPunto<double>(puntoReal);
    return 0;
}
```

Figura 3: Demostración del uso de un *Template* con funtores de C++.

dernos (Figura 4) y tipado inteligente mediante el uso de la palabra `auto` en lugar de especificar el tipo al declarar una variable en aquellos casos en los que el compilador es capaz de inferir el tipo basándose en la parte derecha de la asignación. Además unifica la inicialización empleando llaves y aporta la palabra reservada `nullptr` como constante de valor de cualquier puntero a *null*[17]. Estas son algunas de las características relevantes de C++11 que hemos empleado en nuestra implementación.

Como vemos en la Figura 3, el compilador es capaz tanto de inferir el tipo de una variable `auto` como el tipo del *template* que estamos utilizando, de forma que no es necesario repetir el mismo tipo constantemente, facilitando su lectura y mantenimiento. Hemos optado por emplear tanto el tipado explícito como inferido para mostrar ambos usos.

[17]Constante de programación que implementa el valor abstracto de *la nada*, el conjunto vacío

```
#include <iostream>

struct Vector {
    double x, y;
};
int main() {
  Vector unitario{1, 0};
  Vector ejemplo{2, -3};
  //Lambda que devuelve el modulo de un Vector
  auto modulo = [](Vector v) {
    return (v.x * v.x) + (v.y * v.y);
  };
  //Otra lambda que usa la lambda anterior
  auto unitarioDe = [modulo](Vector v){
    Vector unitario{v.x / modulo(v), v.y/modulo(v)};
    return unitario;
  };
  std::cout
      << "Vector Ejemplo: {" << ejemplo.x << ", " << ejemplo.y << " }" << std::endl
      << "Modulo:" << modulo(ejemplo) << std::endl
      << "Vector Unitario: {"
      << (unitarioDe(ejemplo)).x << ", " << (unitarioDe(ejemplo)).y << " }"
      << std::endl;
  return 0;
}
```

Figura 4: Demostración del uso de una función *lambda* en C++.

2.2. Computación en GPU con OpenCL

Para llevar a cabo computación heterogénea, necesitamos al menos de dos tipos distintos de sistemas de computación. En este trabajo, nuestra solución requiere de un procesador común, una CPU y de un procesador gráfico o GPU. Esta variedad de dispositivos de ejecución nos obliga a tener un medio de comunicación entre los distintos dispositivos. La librería OpenCL hace las veces de esa interfaz común mediante un paradigma maestro-trabajador, si bien no es frecuente su uso para ejecución en CPU, en GPU es ampliamente utilizado por su clara especificación y por ser compatible con prácticamente cualquier acelerador gráfico en alguna de sus versiones.

Un programa OpenCL consta de cuatro partes fundamentales que en conjunto permiten la

ejecución de código de forma paralela en GPU, éstas son:

- La plataforma OpenCL, que modela y permite una abstracción a los dispositivos que ejecutarán el programa. Ofrece una interfaz común a todos los dispositivos en ella. Un ejemplo de plataforma es el conjunto de dispositivos que soporta un determinado *driver* de ejecución.
- El modelo de ejecución, define el como la concurrencia se traslada al hardware físico.
- El kernel de ejecución, es la propia rutina que será ejecutada de forma paralela en el dispositivo OpenCL.
- El modelo de memoria que proporciona una abstracción de las distintas modalidades de memoria que existen en un dispositivo gráfico (global, local, privada).

2.2.1. Plataforma y dispositivo

OpenCL proporciona una abstracción del hardware determinada por el modelo de plataforma y dispositivo. Este modelo consiste en un anfitrión conectado a una serie de dispositivos y define los roles entre ellos. Pueden coexistir más de una plataforma en un mismo computador, pues dependerán del *driver* que utilicen los dispositivos conectados al computador.

Un dispositivo está a su vez compuesto por unidades de cómputo y estas por elementos de procesamiento. En el caso de las GPU's, como la de la Figura 5, se ve claramente como tienen más de un procesador y este a su vez, decenas de núcleos de procesamiento.

El modelo de plataforma nos permite codificar nuestros programas de OpenCL ligados a una o más plataformas, con lo que el hardware real que subyace al dispositivo de la API nos es irrelevante, siempre que tenga el *driver* adecuado a nuestro programa. La elección de plataforma y dispositivo queda relegada a tiempo de ejecución, en el cual deberemos consultar mediante varias llamadas a la API las plataformas, así como sus dispositivos disponibles para decidir dónde ejecutar nuestro programa OpenCL.

Como vemos en el ejemplo 6, debemos seleccionar tanto la plataforma como el dispositivo OpenCL, esta selección requiere de de una llamada a la API con un puntero que recibirá el número de plataformas disponibles, alojar tanta memoria en un puntero como plataformas se quieren obtener (siempre menor o igual al número total devuelto en la primera llamada) y una

Figura 5: Esquema de una GPU Pascal GP100 de Nvidia.

segunda llamada con este número y el puntero que apuntará a las plataformas encontradas. De igual modo y proporcionando además el id de la plataforma y el tipo de los dispositivos (CPU, GPU o ambos) debemos seleccionarlos. Para ello realizamos la primera llamada con el id y el tipo, reservamos memoria y realizamos una segunda llamada que ahora sí, nos devuelve los dispositivos OpenCL.

```
cl_int clGetPlatformIDs(
    cl_uint num_entries, cl_platform_id *platforms, cl_uint *num_platforms
);

cl_int clGetDeviceIDs(
    cl_platform_id platform, cl_device:type device:type, cl_uint num_entries,
    cl_device:id *devices, cl_uint *num_devices
);
```

Figura 6: Interfaz de selección de plataforma y dispositivo en OpenCL.

2.2.2. Modelo de ejecución y programación del kernel

Una vez hayamos seleccionado la plataforma y el dispositivo de ejecución, debemos ejecutar nuestro código. Es aquí donde OpenCL ha definido el cómo se llevará a cabo esta comunicación entre el anfitrión y los dispositivos objetivo (Figura 7).

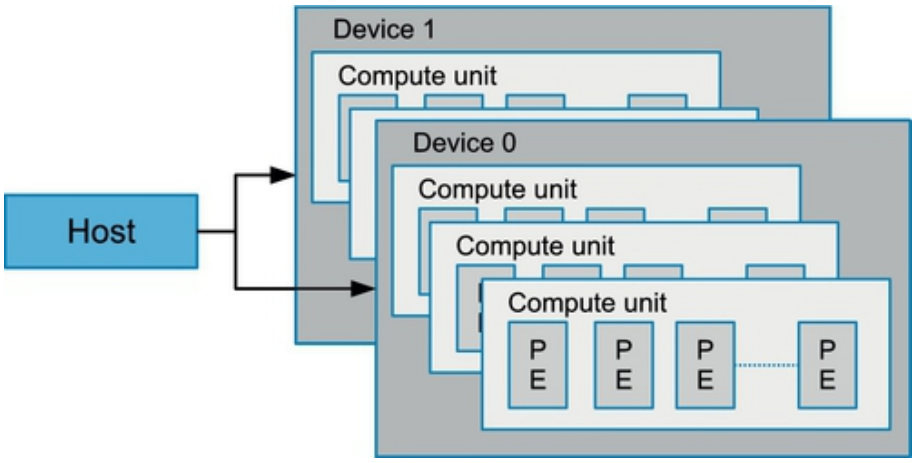

Figura 7: Esquema de la API de OpenCL y su modelo anfitrión-dispositivo

La abstracción que nos facilita ejecutar programas es el contexto, esto es, un entorno que posee mecanismos de coordinación entre el anfitrión y los dispositivos, manejo de las estructuras de memoria y qué dispositivo está ejecutando qué kernel.

Para ejecutar un kernel una vez creado el contexto, OpenCL emplea una cola de comandos en la que el anfitrión envía acciones que deberán realizar los dispositivos. Debe existir al menos una cola por dispositivo del contexto. Estas acciones podrían ser, por ejemplo, ejecutar un kernel determinado o devolver cierto valor al anfitrión. Dichos comandos se encolan de forma que el dispositivo puede lanzar eventos de vuelta, como errores, notificaciones de finalización o estados propios del comando encolado o en ejecución.

Por otra parte, si bien la ejecución de un programa en OpenCL requiere de estas estructuras, lo único que realmente ejecuta el hardware real (dispositivo OpenCL) es el contenido del kernel, una función sintácticamente parecida a C con algunos añadidos para determinar en qué núcleo esta siendo ejecutada. Hablaremos a continuación de cómo una tarea paralelizable es llevada a cabo en un kernel de GPU.

La carga de trabajo enviada a un dispositivo OpenCL se divide en grupos de trabajo (*workgroups*) y estos en elementos de trabajo (*workitems*). Nos referiremos a ellos por sus siglas en

inglés, WG y WI. Un kernel nos permite generar tareas pequeñas como sumas, restas o desplazamientos lógicos en forma de WI que se ejecutan de forma independiente. Un ejemplo es la suma de dos vectores, donde a cada componente del primero debe sumarse el correspondiente del segundo y el resultado almacenarse en un tercero. Esta suma que podría realizarse mediante un bucle, podemos paralelizarla generando tantos WI como elementos tengan los vectores y, a partir de OpenCL 2.0, generar automáticamente el WG con el número de WI óptimo para aprovechar todas las unidades de procesamiento de nuestro dispositivo (Figura 8). La expresividad y potencia de la paralización con OpenCL reside en este mapeo entre elementos de procesamiento (abstracción de los núcleos físicos) y WI .

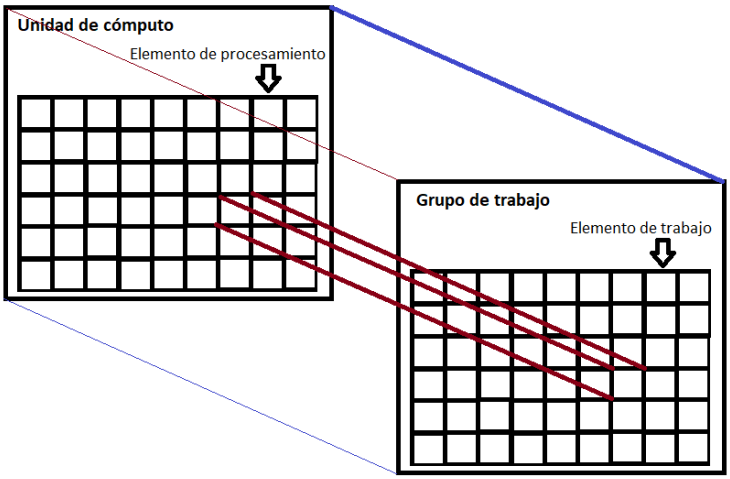

Figura 8: Mapeo entre hardware y modelo de ejecución OpenCL

Ahora bien, para generar estos WI, debemos especificar el rango de índices de nuestro programa OpenCL, denominado *NDRange*, del inglés *N-Dimensional Range*, o traducido, Rango N-Dimensional, que determinará el número y las dimensiones de los WI. También deberemos tener en cuenta que WI pertenecientes al mismo WG poseen elementos de sincronización y espacio de memoria compartido.

2.2.3. Implicaciones del *NDRange*

Cuando definimos el *NDRange* para la ejecución de un programa OpenCL, determinamos no solo el número de WI que queremos, sino además, las dimensiones del problema; en el

ejemplo anterior, al sumar un vector unidimensional, nuestro *NDRange* sería de la forma {size_vector, 1, 1} mientras que si hubiéramos hecho una suma de matrices, habríamos empleado la segunda dimensión también como parte del rango, p.e. {size_dim1, size_dim2, 1}. OpenCL nos permite hasta tres dimensiones y éstas deben estar reflejadas en el tamaño del WG, pues contendrá los WI que debemos procesar. Como mencionamos anteriormente, OpenCL 2.0 nos da la posibilidad de no definir el tamaño del WG, esto es empleado por Intel oneTBB al requerir solo el *NDRange* en la creación del *opencl_node*.

2.2.4. Modelo de memoria

OpenCL define una abstracción de la memoria que nos garantiza determinadas operaciones independientemente del hardware real, mientras que proporciona a los fabricantes un modelo en el que mapear la memoria y operaciones de su hardware real.

Es esta abstracción la que nos otorga comandos para transferencia de datos entre la memoria principal y las distintas memorias que conforman las GPUs modernas. Asimismo, ofrece estructuras de datos equivalentes a los vectores de C, el *buffer*[12] y las imágenes, estas últimas con mayor grado de información y complejidad respecto a los datos que albergan. También provee de una implementación similar a una cola, el *pipe*, que permite entrada y acceso a datos en orden FIFO[18].

2.3. Librería Intel one *Threading Building Blocks*

Nuestra solución, así como la anteriormente implementada, están fuertemente ligadas a la librería Intel oneTBB que nos provee de *templates* para el cómputo paralelo en CPU y, en la versión utilizada para la adaptación a Flow Graph al Heterogeneous Parallel-For da soporte a la ejecución de código OpenCL en las GPU compatibles, muchas de ellas integradas en los procesadores de los computadores comunes.

Esta librería desarrollada en C++ y basada en el modelo de tareas, provee de funciones y plantillas de alto nivel para la construcción de aplicaciones que aprovechan la paralelización posible en CPUs multinúcleo. Una de las ventajas de esta librería es que es multiplataforma, pues no está ligada exclusivamente a procesadores Intel, ya que puede ejecutarse sobre chips

[18]FIFO proviene del inglés First In, First Out, primero en llegar, primero en salir

de otras compañías o arquitecturas.

La librería oneTBB facilita el desarrollo de las aplicaciones donde la carga computacional se mapea en multitud de tareas que pueden ser ejecutadas en paralelo en lugar de mapearla en distintos hilos de ejecución. Este modelo de diseño permite escalabilidad y portabilidad al no depender de llamadas al sistema para administrar los hilos y la concurrencia. oneTBB nos proporciona interesantes ventajas, entre ellas caben destacar las siguientes:

- Al promover tareas en lugar de hebras, los desarrolladores no necesitan administrar los hilos de ejecución, solo definir las rutinas de ejecución de las tareas, volviendo el código agnóstico de la plataforma y sistema operativo de ejecución.

- Incorpora un planificador en tiempo de ejecución basado en *work-stealing* (robo de tareas) que consigue balancear dinámicamente la carga de trabajo. Es decir, los threads que procesan las tareas (*workers*) roban tareas a otros threads en caso de no tener ninguna en su cola de tareas pendientes.

- Compatibilidad plena e interoperabilidad con otras librerías de paralelización.

- Enfatiza la paralelización a nivel de datos, esto es por ejemplo, definir una tarea tal que si el rango de la entrada forma parte de otro mayor, distintas porciones del rango completo puedan computarse sin que se vean afectadas. Esta propiedad permite que el programa escale en base al número de procesadores, sin necesidad de recompilar el código.

- El desarrollo de oneTBB está diseñado para ser lo más genérico posible, promoviendo interfaces basadas en plantillas que puedan adaptarse a las necesidades de cada desarrollo o aplicación.

En la versión empleada de la librería, Intel ha incluido un modelo de flujo de los datos en el programa que sigue la estructura de un grafo donde los datos pueden ser tratados en tareas diferentes según convenga, así como la inclusión de un determinado tipo de nodo especial que es el *opencl_node* al cual se le proporciona un kernel escrito en OpenCL y una tupla con los argumentos de entrada, luego solo se debe esperar a la terminación del mismo para obtener el resultado del cómputo. Es esta la novedad que inspira este trabajo, ya que nos da un paso más de abstracción con respecto al modelo de tareas, permitiéndonos olvidarnos sobre como

compilar, reservar memoria y recibir el evento de finalización desde el dispositivo, en inglés *callback*, al ejecutar un kernel en OpenCL desde nuestro programa usuario del *template*.

Intel oneTBB encapsula en las clases de Flow Graph la funcionalidad que nos permite modelar el flujo de los datos de un programa mediante un grafo, donde cada nodo realiza una tarea (*functional_node*), construye o divide una tupla (*join_node* y *split_node*, respectivamente) y propaga el resultado en su arco o arcos de salida o genera datos de forma ininterrumpida (*source_node*), entre otros tipos. Uno de estos tipos especiales de nodo es el *streaming_node* que está diseñado para ser ejecutado en un lugar distinto a la CPU. En la versión utilizada de TBB, el *streaming_node* solo puede instanciarse como *opencl_node* que nos permite ejecutar un *kernel*, en este caso, codificado en OpenCL.

El nodo OpenCL requiere de un procesador compatible con el *driver* empleado por oneTBB para enviar código a la GPU. Para su construcción debe recibir un fichero que contenga el kernel a ejecutar, la referencia al dispositivo que ejecutará la tarea, los parámetros de entrada a la función del kernel y el *NDRange* de OpenCL, estos se pasarán como argumentos de entrada. Al tratarse de una funcionalidad aún en etapa de pre-lanzamiento esta limitada a la primera plataforma existente en el dispositivo, esto es, nuestro sistema operativo puede tener solo una CPU y su GPU compatibles, pero detectarse en mas de una plataforma, con lo que si quisiéramos ejecutar en GPU y esta se encontrase en la segunda tendremos que modificar la actual implementación de oneTBB para que encontrase nuestra GPU. Definido lo anterior, explicaremos como hemos aplicado este modelo de grafos a nuestro problema y el papel fundamental que ejerce el *opencl_node*.

2.3.1. Flow Graph al detalle

Para poder empezar a utilizar *flow_graph*, deberemos importar *flow_graph.h* o como en nuestro caso, *flow_graph_opencl_node.h*. Una vez tenemos importada la librería, para poder utilizar un grafo como herramienta de paralelización, debemos instanciar la clase *graph*, crear nodos que realicen las tareas de nuestro programa y conectar estos nodos entre si mediante arcos dirigidos entre sus puertos de salida y entrada. Una vez hemos diseñado nuestro grafo, implementado los nodos de ejecución y conectamos estos entre si para generar el flujo de datos. Debemos iniciar el primer nodo que actuará como detonante de los sucesores, como fichas de dominó. Para que nuestro flujo de ejecución a lo largo del grafo finalice deberemos emplear

el método *wait_for_all()* del objeto *graph* para que nuestro programa espere la finalización de los cálculos y llamadas entre nodos.

A continuación, hemos implementado un grafo sencillo que haría las veces de "Hola Mundo" en *flow_graph*.

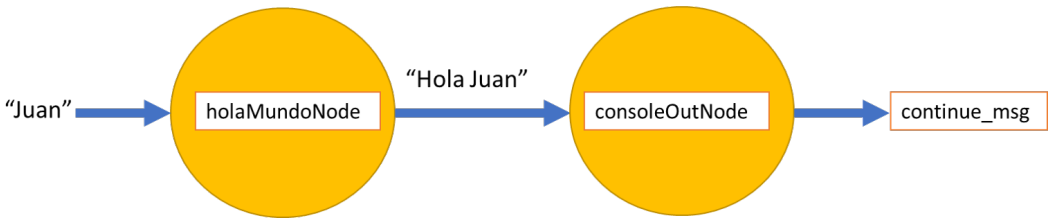

Figura 9: Representación del grafo implementado en la figura 10

El grafo tendría la forma que vemos en la Figura 9, donde ``Juan'' sería detonante de la ejecución en el momento en el que se llama al método *try_put()* del `holaMundoNode`. Esta cadena es procesada por el cuerpo del nodo, definido en la función lambda de su inicialización, que como vemos, solo añade un ``Hola '' por delante. A continuación, conecta con el `consoleOutNode` que recibe un texto y lo muestra por consola. En este caso, el nodo función que hemos declarado no tiene valor de retorno. oneTBB permite tanto emplear una arquitectura de paso de mensajes que comunican la salida de un nodo con la entra de otro, o comunicar eventos cuando en los nodos función no hay argumentos de entrada o salida definidos, en cuyo caso se envía un *continue_msg* que hace saber al sucesor que la tarea ha sido ejecutada. En la Figura 10 detallamos la implementación del grafo anteriormente explicado.

Cabe mencionar que oneTBB no crea hilos de ejecución para nuestros nodos, en su lugar, ofrece una capa de abstracción de forma que los nodos son tareas que van siendo recogidas y procesadas por las hebras de ejecución que han sido previamente creadas al inicializarse la librería. Dependiendo de la concurrencia seleccionada para nuestros nodos, podríamos llegar incluso a tener hebras en espera si nuestros nodos están definidos con concurrencia de tipo serial, es decir, las llamadas al nodo se procesarán una a una, impidiendo que existan dos tareas al mismo tiempo en ejecución para ese nodo.

Como hemos explicado anteriormente, el componente principal de nuestra adaptación del Heterogeneous Parallel-For, el nodo OpenCL, requiere de una mínima interacción por parte del usuario de la plantilla. A grandes ragos su uso consiste en la creación de la tupla de entrada al nodo con la restricción de los índices de principio y final en una estructura en la primera

```
#define TBB_PREVIEW_FLOW_GRAPH_NODES 1
#define TBB_PREVIEW_FLOW_GRAPH_FEATURES 1

#include <tbb/flow_graph.h>
#include <iostream>
#include <string>

using namespace std;
using namespace tbb;

int main() {
    flow::graph grafo();
    flow::function_node<string, string> holaMundoNode(
        graph, flow::unlimited, [](string nombre) -> string { return "Hola " + nombre; });
     flow::function_node<string> consoleOutNode(
        graph, flow::unlimited, [](string texto){ cout << texto << endl; });
    flow::make_edge(holaMundoNode, consoleOutNode);
    holaMundoNode.try_put("Juan");
    grafo.wait_for_all();
    return 0;
}
```

Figura 10: Ejemplo de un Hola mundo empleando TBB.

posición de la misma. Por tanto, ¿cómo funciona *opencl_node* y que nos facilita oneTBB para su uso? Procederemos a explicarlo en la siguiente sección.

2.3.2. *Opencl_node* al detalle

El nodo OpenCL hereda del ya mencionado *streaming_node*, pero además, para poder ser usado correctamente, debemos tener en consideración clases y funciones que modifican el comportamiento y la ejecución del kernel enviado al nodo. Entre estas clases y plantillas encontramos los funtores `DeviceSelector`, `DeviceFilter` y `Factory`:

- `DeviceFilter`: encargado de generar una lista de dispositivos compatibles con OpenCL que se utiliza en la inicialización del `Factory`. Esta limitado a filtrar dispositivos de una plataforma única, que como anticipábamos, es la primera que proporciona el sistema.
- `Factory`: es el *template* encargado de proporcionar a nuestro *opencl_node* la lista de dispositivos donde puede ser ejecutado, se inicializa con un `DeviceFilter`.

- `DeviceSelector`: es el funtor encargado de determinar por cada ejecución del *opencl_node* que dispositivo elegir para ejecutar el kernel.

```
struct GpuDeviceSelector {
private:
  bool firstTime = true;
  tbb::flow::opencl_device device;
public:
  template<typename DeviceFilter>
  operator()(tbb::flow::opencl_factory<DeviceFilter>& f) {
    if(firstTime){
      firstTime = false;
      auto deviceIterator =
        std::find_if(f.devices().cbegin(), f.devices().cend(),
            [](const tbb::flow::opencl_device& d) { return d.type() == CL_DEVICE_TYPE_GPU; }
        );
      device = deviceIterator == f.devices().cend()? *f.devices().cbegin() : *deviceIterator;
    }
    return device;
  }
}
```

Figura 11: DeviceSelector con caché de dispositivo.

Si no se define alguno de los anteriores en el momento de instanciar un nodo OpenCL, oneTBB utilizará la implementación por defecto que podríamos resumir en un `DeviceFilter` que selecciona todos los dispositivos de la primera plataforma, un `Factory` que devuelve todos los dispositivos del filtro utilizado y un `DeviceSelector` que devuelve el primer dispositivo del `Factory` asignado. En nuestra implementación hemos decidido utilizar nuestro propio `DeviceSelector` en lugar del disponible por defecto, basándonos en el ejemplo propuesto por el capítulo 19 de Pro TBB [17]. La implementación se detalla en la Figura 11 donde cacheamos el dispositivo de la GPU para acelerar la ejecución del nodo OpenCL, de forma que devuelve la primera GPU disponible en el `Factory` por defecto, que a su vez utiliza el filtro por defecto. Para llevar a cabo la caché, hemos añadido un pequeño atributo al selector, esta optimización es recomendable ya que su ejecución añade una sobrecarga a nuestras mediciones de tiempo respecto al rendimiento real de las tareas.

2.4. Interfaz oneAPI

Intel oneAPI [14] ha sido desarrollada como un paso más allá de la librería oneTBB, evolucionando desde el *template* intermediario hasta una interfaz que permite utilizar un único fichero fuente que albergue el código de tanto el dispositivo anfitrión que ejecuta código secuencial como los métodos que se utilizarán para construir los *kernels* que serán ejecutados por los dispositivos aceleradores. Reside aquí una de las grandes ventajas frente a oneTBB, donde el kernel debe ser especificado en un fichero OpenCL por separado.

Para utilizar este nuevo modelo de paralelización ha sido determinante la inclusión de DPC++[19] y SYCL[20][16]. Es en SYCL donde encontramos las nuevas funcionalidades frente a OpenCL, ya que permite la codificación de *kernels* en C++ y con las ventajas de C++11 como las funciones lambda. Con estas tecnologías, Intel oneAPI construye así un modelo que reduce aún mas la complejidad de oneTBB y su utilización para codificar programas paralelos.

2.4.1. El compilador DPC++ y estándar SYCL

Intel oneAPI se basa en los lenguajes C++ y el estándar de SYCL, es aquí donde entraremos en detalle sobre qué beneficios nos aportan frente al tradicional C++ y OpenCL, tanto por separado como cuando se usan combinados, como en las implementaciones que presentamos en este TFG, basadas en el *template pipeline* y el *template flow_graph* que usan la librería oneTBB.

2.4.2. SYCL

El modelo de programación SYCL, pronunciado «sikle», es un lenguaje de programación de uso gratuito y de dominio específico, con énfasis en el paradigma de fuente única (tanto el código del anfitrión como el del acelerador puede estar codificados en el mismo fichero fuente o unidad de traducción). Esto significa que el modelo de SYCL pretende abstraernos de la implementación de *kernels* en OpenCL para realizar la transformación por parte del propio compilador quien tomará como base funciones y librerías de un subconjunto de C++.

[19] *Data-Parallel C++* que aporta funcionalidad adicional al compilador de C++ para orientarlo a la paralelización a nivel de datos

[20] Modelo de programación para aceleradores que aporta ventajas y soporte a un subconjunto del lenguaje C++ como código fuente de kernels

El paradigma de fichero fuente único[15] permite la codificación de las funciones del procesador anfitrión y los *kernels* a ejecutar en los dispositivos, tal como podemos ver en la Figura 12, implementamos fácilmente los métodos que deseamos ejecutar en nuestro dispositivo anfitrión, así como acelerador.

```
constexpr int NUM=16;
#include <CL/sycl.hpp>
#include <iostream>
using namespace cl::sycl;
int main() {
    int data[NUM];
    { // Create queue on chosen device
        queue myCPUQueue{cpu_selector{}};
        queue myGPUQueue{gpu_selector{}};

        // Create buffer using host allocated "data" array
        buffer<int, 1> buf { data, range<1> { NUM } };

        // In this case we use GPU queue only
        myGPUQueue.submit([&](handler& h) {
            auto writeResult = buf.get_access<access::mode::discard_write>(h);
            h.parallel_for<class simple_test>(range<1> { NUM }, [=](id<1> idx) {
                writeResult[idx] = idx[0];
            });
        });
    } // Buffer goes out of scope. Blocks until kernel output
  // available in "data" array.
    for (int i = 0; i < num; i++)
        std::cout << "data[" << i << "] = " << data[i] << "\n";
    return 0;
}
```

Figura 12: Ejemplo de programa simple en SYCL. (2020) Data Parallel C++. Adaptado

Empleando el código de la Figura 12 como hoja de ruta, procederemos a explicar cada una de las partes que componen SYCL. La novedad que salta a la vista frente a un programa que utiliza OpenCL, es que por ahora[21] SYCL sigue formando parte de la librería de C++ de OpenCL, por lo que pasaría a sustituir la forma en que compilamos, ejecutamos y trabajamos con *kernels* de GPU. Asimismo, en este pequeño ejemplo ya apreciamos las ventajas del paradigma de

[21]La especificación de SYCL2020 se desmarca de OpenCL y tiene su propio dominio de nombres.

fuente única, pues en lugar de generar estructuras de datos especificas de GPU, inicializarlas con valores de las estructuras del dispositivo anfitrión y coordinar su copiado, empleamos un *array* de C++, sobre el que construimos un *buffer* que es la estructura que maneja internamente la gestión de copiado y compartición de memoria entre el anfitrión y el acelerador.

Esta simplicidad en el manejo de datos, produce una buena sinergia con disponer de una lambda, funtor o rutina que hace las veces de kernel, pero localizada junto con las llamadas al acelerador. Como veremos en el Capítulo 3, nuestra implementación en *flow_graph* requiere un fichero con el kernel en OpenCL y la definición redundante de las estructuras de datos para poder ejecutar código en GPU. En SYCL, este esfuerzo adicional no es necesario, pues el compilador, interpretará este método y generará el kernel adecuado y optimizado para los posibles aceleradores que existan y seleccionemos en tiempo de ejecución o compilación. En este caso elegimos el *gpu_selector*.

Ahora bien, ¿en qué consisten estos *cpu_selector* y *gpu_selector*? Para responder a esto, debemos recordar los mencionados anteriormente modelos de anfitrión y dispositivo de OpenCL y los `DeviceSelector` y `DeviceFilter` de oneTBB, pues al fin y al cabo son todos abstracciones de cómo se implementa en OpenCL qué procesador ejecuta y qué código. SYCL proporciona interfaces para seleccionar la unidad de proceso que ejecutará el kernel, esto dota al compilador mayor capacidad a la hora de llevar a cabo optimizaciones, pues saber de antemano el tipo de dispositivo que ejecutará el código le permite aplicar las pasadas de optimización más oportunas.

SYCL incorpora el paralelismo mediante los métodos disponibles en el *handler*[22], como vemos en la Tabla 1, que recibe de argumento la función que encolamos a un dispositivo determinado. Como vemos en el ejemplo 12, el *handler* tiene un método que aplica el principio de oneTBB *parallel_for* sobre el que se basa el *template pipeline* del Heterogeneous Parallel_For. Es esta cola de eventos uno de los pilares de SYCL, pues cada cola esta ligada con cardinalidad uno a uno con un dispositivo acelerador, anfitrión o CPU, mientras que un dispositivo puede no tener ninguna, tener una o varias colas asignadas, una cola debe asociarse con exactamente un dispositivo para eliminar posibles indeterminaciones sobre donde se ejecutará el kernel.

[22]El handler es el tipo que define la entrada de un evento en el dispositivo acelerador.

Work Type	Handler class method	Summary
Device code execution	*single_task*	Excecute a single instance of a device function.
	parallel_for	Multiple overloads are available to launch device code with different combinations of work sizes
	parallel_for_work_group	Launch a kernel using hierarchical parallelism
Explicit memory operation	*copy*	Copy data between locations specified by accessor, pointer, and/or shared_ptr. Then copy occurs as part of the DAG, including dependency tracking.
	update_host	Trigger update of host data backing of a buffer object.
	fill	Initialize data in a buffer to a specified value

Tabla 1: Funciones de la clase handler. (2020) Data Parallel C++. Adaptado

2.4.3. DPC++

DPC++ es una extensión de C++ y SYCL y está incorporado en oneAPI de Intel para proporcionar mayor funcionalidad que los estándares sobre los que se basa, ISO C++ y Khronos Group SYCL. Aporta sobre SYCL un modelo único de memoria, mediante el que no es necesario reescribir nuestros códigos en C++ para incorporar el *buffer* descrito anteriormente, sino que mediante el manejo de punteros construye un modelo de memoria compartida donde el código preexistente en C++ que es compatible con punteros sigue funcionando sin cambiar su implementación. También aporta mayor manejo de las dependencias de datos que existen entre tareas, utilizando colas con orden y definiendo funciones y librería para el manejo de los WG y los WI de OpenCL que mencionamos al comienzo de este capítulo. Así pues, el compilador de DPC++ de Intel nos permite no solo seleccionar el modelo de memoria único, sino elegir

explícitamente donde queremos localizar nuestros datos, en el dispositivo o en el anfitrión junto con la posibilidad de especificar el orden de ejecución de las tareas que poseen dependencias entre sí, tanto explícitamente declaradas como implícitas mediante compartición de punteros.

La compartición de punteros del anfitrión o de la memoria compartida con el dispositivo es inmediata de forma que es directamente visible por el kernel, en el caso de los *buffers*, es necesario utilizar un objeto *accessor* que es provisto por el *buffer* de datos. Este objeto puede ser de varios tipos, lectura, escritura, ambos, con descarte de la información previa o no.

2.4.4. oneAPI al detalle

Intel define oneAPI como el modelo de programación unificado para arquitecturas heterogéneas. Dentro de esta interfaz, encontramos librerías como oneTBB, objeto de este TFG, y lenguajes y compiladores como DPC++, explicado anteriormente.

oneAPI cubre de esta forma las dos partes del desarrollo software de aplicaciones paralelas: la programación de aplicaciones paralelas que se sirven de una librería de paralelización y la programación directa de algoritmos de paralelización que requieren de un lenguaje especifico que a su vez, serán API y librería de otras aplicaciones. Es en este segundo marco donde DPC++ es crítico, ya que es el lenguaje utilizado para llevar a cabo estos algoritmos, mientras que oneTBB encajaría en el marco de una API al no requerir de compilador propio y ser posible utilizarla desde un programa en C++ común. Algunas de las partes que conforman oneAPI son:

- `DPC++`: lenguaje base de oneAPI para programar aceleradores y multiprocesadores. DPC++ permite a los desarrolladores reutilizar código frente a hardware cambiante (CPUs, GPUs y FPGAs), así como afinar su rendimiento.
- `oneDPL`: Subconjunto de DPC++ para Data Parallel Library, que permite emplear los algoritmos del Standar Template Library de C++11, STL, y de sus funcionalidades en aceleradores sin apenas modificar el código fuente.
- `Level Zero`: Proporciona la interfaz de mas bajo nivel de la API que permite interactuar directamente con el hardware subyacente mientras ofrece abstracciones sobre las operaciones que se realizan, como bien puede ser una cola de comandos o un evento de sincronización.

- oneTBB: La librería de Intel que da soporte a ejecución paralela en sistemas multiprocesador de forma simplificada y eficiente, frente al control de hebras y sincronización que se requiere en un programa C++ corriente.

3

Ejecución heterogénea de bucles paralelos

3.1. Especificación del problema

Debemos afrontar la optimización de la ejecución y creación de aplicaciones que aprovechen las arquitecturas heterogéneas que surgen en los computadores modernos. Para limitar el problema, nos centraremos en la ejecución heterogénea, CPU+GPU, de bucles paralelos, parallel_for. El objetivo es diseñar un planificador oculto en un template parallel_for de alto nivel, el cual sea capaz de repartir las iteraciones en bloques o chunks que serán procesados en CPU y en GPU. Uno de los objetivos de diseño debe ser mantener una elevada productividad del programador, para lo que debemos ofrecer una API con alta programabilidad que oculte los detalles de bajo nivel. Idealmente, el programador sólo deberá especificar el código que ejecuta un chunk de iteraciones en la CPU y el correspondiente que hace lo mismo en la GPU. El planificador se encargará de calcular el tamaño de los chunks para maximizar el rendimiento de CPU y GPU, así como de asignar cada chunk a cada dispositivo de forma que estén siempre ocupados y la carga esté balanceada.

Para ello, nuestra solución se basa como hemos mencionado, en la nueva funcionalidad de TBB llamada Flow Graph así comen en la reciente librería de Intel oneAPI. A continuación procederemos a detallar las distintas implementación que dan solución a este problema.

3.2. Heterogeneous Parallel_For

En la implementación de partida, basada en el *template pipeline* de oneTBB, la solución propuesta requiere del trabajo y conocimiento del usuario para compilar el *kernel* a ejecutar en GPU, así como para definir las estructuras de datos que albergarán los argumentos del mismo; a esto se debe sumar la sobrecarga de medidas de tiempo y rendimiento que, al no poder implementarse en otra rutina distinta a la que realiza y recibe la llamada al *kernel*, tampoco es implementable por la plantilla del Heterogeneous Parallel_For. Su funcionamiento es el siguiente:

- El *pipeline* recibe por parte del usuario el rango de iteraciones, un objeto que contiene los métodos que se van a ejecutar en CPU con oneTBB y el método que será el encargado de compilar y ejecutar el *kernel* de OpenCL con unos argumentos definidos de forma análoga al procedimiento de CPU.

- Cuando se ejecutará el kernel de GPU o el método de CPU depende tanto de la capacidad física del computador, como el número de CPU's, o así como del resultado del algoritmo de particionado (LogFit). Esta elección, una vez tomada, se implementa mediante un *token* que pasa por las etapas del pipeline haciendo las veces de controlador del nivel de concurrencia y de la disponibilidad de las unidades de computación. Es decir, si nuestro computador tiene cuatro CPU's, el *template* generará cuatro *tokens* de CPU que se consumirán cuando se llame al *operatorCPU* y serán repuestos cuando el procedimiento concluya, de forma que no existan tareas de CPU que no se ejecuten por falta de recursos. También existe un *token* de GPU que se consume cuando la GPU está procesando un bloque de iteraciones y se libera cuando termina.

- Es a lo largo de estas ejecuciones y en base al rendimiento por unidad de tiempo, que denominaremos *throughput*, obtenido de cada una, donde el algoritmo de particionado decide el tamaño de la la porción del espacio de iteraciones (en adelante llamado *chunk*) que recibirá el dispositivo al que pertenezca el *token* asociado en la medida.

El *template* nos proporciona funciones para obtener estructuras de memoria que sean accesibles tanto por oneTBB como por nuestro *kernel* de forma que simplifica el manejo de los datos y su transferencia al dispositivo correspondiente. También provee de una clase base con la que

construir el cuerpo de ejecución del problema a resolver, estas clases son `HBuffer` y `HTask` respectivamente. Es en la implementación del `HTask` donde el usuario del *template* debe construir el modo de ejecución del *kernel* OpenCL que se invocará en la función `operatorGPU`, mientras que el manejo del *command_queue* o el identificador del *kernel* son delegados a la librería, puede verse un esquema de implementación de un `Body` en la Figura 13.

```
#include "HTask.h"
#include "HBuffer.h"

using namespace hbb;

class Body : public HTask {
  HBuffer<int> *buf_a;
public:
  Body(KernelInfo k, HBuffer * buf_a) : HTask(k) {...}
  void operatorCPU(int begin, int end) {
    int *a = buf_a->getHostPtr(BUFF_RW);
    for(int i = begin; i < end, i++) a[i] = a[i] + 1;
  }
  void operatorGPU(int begin, int end) {
    setKernelArg(0,sizeof(int), &begin);
    setKernelArg(1,sizeof(int), &end);
    setKernelArg(2,sizeof(BUFF), buf_a->getDevicePtr(BUFF_RW));
    launchKernel(begin, end);
  }
}
```

Figura 13: Ejemplo de un Hola mundo empleando TBB.

Por otra parte, el algoritmo de particionado del espacio de datos actual, es reutilizable en su totalidad, requiriendo únicamente de su importación en cualquier posible implementación que así lo requiera, con lo que en la implementación basada en *flow_graph* hemos utilizado el mismo sin modificaciones. Trataremos detalladamente dicho algoritmo en la sección 3.2.1.

La implementación existente ha sido desarrollada con el objetivo de ser multiplataforma y así debe continuar, ya que el propósito de la plantilla es permitir aprovechar al máximo las prestaciones de los procesadores modernos que incluyen una GPU integrada. Muchos de estos procesadores no son solo para ordenadores de escritorio, también se encuentran en servidores multipropósito y en dispositivos bajo arquitectura ARM. En otras palabras, cualquier platafor-

ma móvil o tablet tiene el potencial de ejecutar código paralelo heterogéneo al incluir un chip gráfico junto con el procesador.

La librería actual ha sido desarrollada en C++ al ser la tecnología elegida por Intel para el desarrollo de su oneTBB, en parte por la eficiencia de los programas desarrollados en este lenguaje y en parte por su versatilidad, como comentamos anteriormente. Al pretender un mayor público, el lenguaje debe ser lo más extendido y utilizable posible. Gracias a un lenguaje capaz de ser compilado para distintas arquitecturas con mínimos cambios, más allá de determinadas cabeceras para OpenCL, el Heterogeneus Parallel_For no requiere de complejas modificaciones para ejecutarse sobre MacOS, Linux, Windows o incluso Linux-ARM.

3.2.1. LogFit: Algoritmo de particionado

Describiremos el algoritmo de particionado del espacio de iteraciones de la implementación de partida, así como nuestra propuesta utilizada para encontrar un *chunk* tal que su tamaño maximice el rendimiento de cada procesador disponible. Posteriormente este tamaño se irá ajustando en base a irregularidades y cargas desiguales entre los procesadores, derivadas del propio uso del procesador por otros procesos o del sistema operativo.

El algoritmo ha sido diseñado en tres fases: fase de exploración, fase estable y fase final (Figura 14). En la fase de exploración el algoritmo va muestreando entre diferentes tamaños de *chunks* para maximizar el *throughput* de la CPU y GPU, hasta alcanzar el cumplimiento de la condición de estabilización. Esta condición se ve representada en la Figura 15, donde la subfigura a ilustra la fase de exploración y la b la fase estable o de explotación. La condición de estabilización, asegura que las variaciones del *chunk* apenas interfieren en el rendimiento. El objetivo principal es optimizar el rendimiento de la GPU y balancear la carga de trabajo entre los *cores* de CPU y la GPU. Una vez se ha llegado a la estabilidad, el algoritmo sigue funcionando para adaptarse a las variaciones de *throughput* que tienen lugar principalemente en códigos irregulares durante la ejecución del *kernel*. La fase final presta mayor atención al numero de iteraciones restantes donde el balanceo entre CPU y GPU adquiere mayor importancia frente al rendimiento.

En todas las fases, la estimación del *chunk* viene dada por una aproximación del *throughput* de GPU a la función logarítmica, de ahí el nombre del algoritmo. Además de las fases, el algoritmo requiere una condición de parada con la que distinguir la fase final de las anteriores.

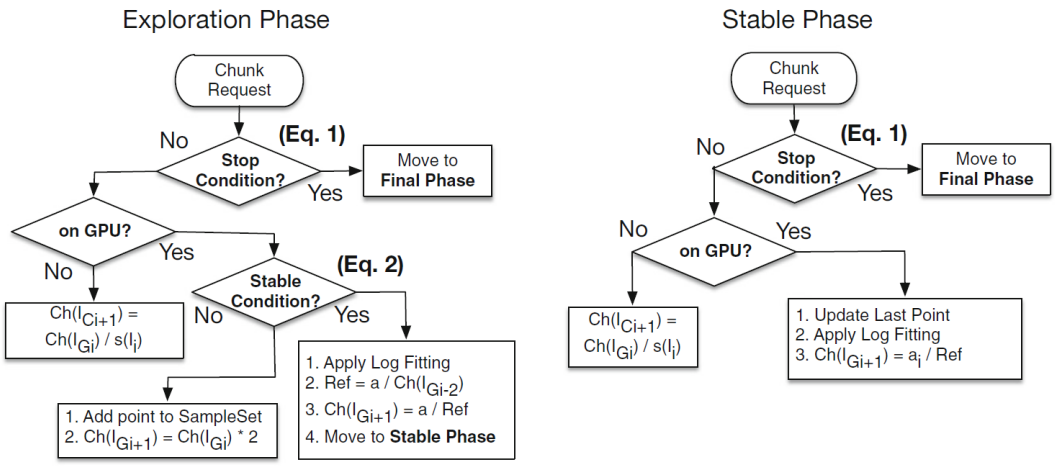

Figura 14: Diagrama de estados y decisión del algoritmo.

Entramos en esta fase final cuando no existen iteraciones restantes suficientes para la creación de un *chunk* por dispositivo. En esta situación, se debe decidir donde ejecutar las pocas iteraciones restantes para minimizar el tiempo de ejecución, mantener todos las unidades de computación balanceadas y finalizar lo antes posible.

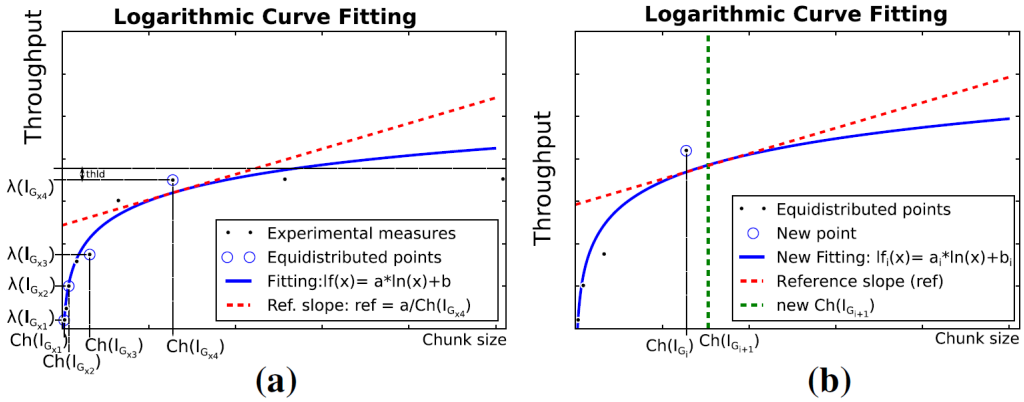

Figura 15: Representación del tamaño del chunk y el rendimiento del procesador (CPU o GPU).

La fase de exploración comienza con la GPU en espera de un *chunk*, es entonces donde LogFit debe determinar si ha de enviar un *chunk* del mismo tamaño que el anterior o una nueva muestra con la que extraer mas información. Para dicha decisión, debe evaluarse la condición de estabilización, que calcula cuánto mejora el *throughput* con respecto a los dos *chunks* anteriores, dando como válida la condición cuando el valor es similar en los tres últimos muestreos, según un umbral predeterminado en el logaritmo. Una vez aceptada la condición,

se toman cuatro tamaños de chunk de entre los muestreados, eligiendo los que esté más o menos equiespaciados. Con estos cuatro puntos se ajusta una función logarítmica. Una vez hemos determinado el modelo, ajustamos una recta tangente que será empleada en la fase estable para determinar el tamaño óptimo del *chunk* y modificarlo, si fuese necesario.

En la fase estable, se sigue muestreando el *throughput* obtenido durante la ejecución de cada *chunk* de iteraciones en la GPU. Este nuevo punto substituye al cuarto punto de la regresión logarítmica, con lo que ésta varía y su nueva recta referencia nos proporciona un tamaño de *chunk* distinto. La fase estable continua hasta satisfacer la condición de parada mencionada anteriormente, dando paso a la fase final.

La fase final, como comentamos, debe determinar el o los dispositivos donde la ejecución finalice lo antes posible, así que se elegirá el menor tiempo entre computar solo en GPU, CPU o repartir las iteraciones restantes entre CPU y GPU.

Para una explicación más pormenorizada del algoritmo de planificación LogFit, recomendamos al lector interesado que se dirija a [8].

3.3. Adaptación a Flow Graph

En lugar de usar el *template* de *pipeline* de la implementación original, hemos diseñado nuestra adaptación del Heterogeneous Parallel_For basándonos ahora en la funcionalidad del Flow Graph de TBB, utilizando para ello cuatro nodos. Ahora los argumentos de entrada para el grafo son el número de CPUs, el rango completo de iteraciones y el equivalente al `HTask` del *parallel_for* original. Detallaremos a continuación su uso e implementación.

3.3.1. Uso y funcionamiento

Hemos implementado nuestra solución teniendo en cuenta la simplificación en su uso, de forma que el usuario de la librería no tenga que tener conocimientos sobre como opera oneTBB ni el grafo subyacente al modelo de ejecución paralela. Para acercarnos a este objetivo hemos definido el menor número de funciones y operaciones que el usuario debe realizar para que el sistema opere.

Nuestra solución requiere de la implementación de la clase abstracta `IBody`, donde se define principalmente la función `operatorCPU`, de carácter análogo a la del `HTask` y en lugar de

`operatorGPU`, hemos adaptado el paso de argumentos al formato que emplea TBB mediante una tupla de C++. El usuario debe implementar el método `GetGPUArgs` que devuelve una tupla que contiene los argumentos de entrada del kernel OpenCL que ejecuta la rutina en GPU. Este método debe ser personalizado por el usuario que es quien conoce en tiempo de compilación de su aplicación los tipos de los argumentos de entrada del kernel.

Una vez definido el cuerpo de nuestro programa, debemos instanciar la clase `GraphScheduler` con los parámetros correspondientes: un objeto de tipo `Params`, que alberga el número de CPU's y GPU's que se utilizarán, la ruta al fichero con el *kernel* OpenCL, el nombre del *kernel* dentro del fichero, la ruta al fichero de entrada de datos que serán parte de la ejecución y un nombre para la ejecución. También debemos proporcionar el objeto de la clase personalizada que hereda de `IBody` con las funciones anteriormente mencionadas, así como una instanciación del algoritmo de particionado que se desee emplear. Éste debe heredar de la clase abstracta `IEngine` que en la implementación aquí defendida es LogFit, pero que puede ser cualquier otro que satisfaga las restricciones de la herencia anterior.

De esta forma, el código mínimo se puede formar con los ejemplos mostrados en las Figuras 16, 17, 18 y 19 que detallaremos una a una a continuación.

3.3.2. Ejemplo de uso

A continuación iremos detallando como utilizaríamos el *template* en una paralelización de un programa que realiza una suma de dos vectores.

```
#include "ProvidedDataStructures.h"

using t_index = struct _t_index{int begin, end;};
using buffer_f = tbb::flow::opencl_buffer<cl_float>;
using dim_range = std::vector<size_t>;

using type_gpu_node = tbb::flow::tuple<t_index, buffer_f, buffer_f, buffer_f>;
```

Figura 16: Ejemplo de implementación de las estructuras de datos requeridas para el uso del template

- En la Figura 16 mostramos cómo el usuario programador debe crear un fichero que con-

tenga los tipos y estructuras de datos que requiere su problema en particular. Debemos definir obligatoriamente para usar el *template* los tipos siguientes: `t_index`, `dim_range` y `type_gpu_node` y, adicionalmente el tipo `buffer_t` que es empleado por el kernel como entrada en nuestro ejemplo de suma de vectores. Para completar las estructuras necesarias, se requiere además enlazar con las proporcionadas por nuestra implementación para el manejo de los "*includes*".

```
class TestExecutionBody :
    IBody<dim_range, t_index, buffer_f, buffer_f, buffer_f> {
public:
    void OperatorCPU(int begin, int end) {
        for (int i = begin; i < end; i++) {
            Chost[i] = Ahost[i] + Bhost[i];
        }
    }
    void ShowCallback() {
        for (int i = 0; i < vsize; i++) {
            std::cout << i << ": "<<Ahost[i]<<" + "<<Bhost[i]<<" = "<<Chost[i]<<'\n';
        }
    }
    type_gpu_node GetGPUArgs(t_index indexes) {
        return std::make_tuple(indexes, Adevice, Bdevice, Cdevice);
    };
    size_t GetVsize() {return ndRange.size;}
    dim_range GetNDRange() {return ndRange;}
};
```

Figura 17: Ejemplo del Body para la suma de dos vectores en flow_graph

- Una vez definidas las estructuras de datos del problema, debemos implementar en nuestra clase `Body` (ver Figura 17) las funciones mencionadas anteriormente. Es aquí donde `GetGPUArgs` debe ser implementado pues la tupla de entrada del kernel ha de ser definida en tiempo de compilación y solo el usuario conoce los argumentos del kernel. Este tipo ha de ser una `tbb::flow::tuple` que se construya a partir de los tipos de entrada del kernel que utilizará TBB en el *opencl_node* y en el mismo orden. Si bien `GetGPUArgs` siempre es de la forma: `std::make_tuple<t_index, Targ1, Targ2, ...>(indexes, arg1, arg2, ...)`, donde `TargX` es el tipo de su correspondiente estructura de datos

`argX`, debe ser implementado por el usuario.

Asimismo esta clase define los métodos que serán computados cuando la CPU sea el dispositivo de ejecución elegido y los que son necesarios para definir el `NDRange` de la ejecución en GPU. En nuestro caso este rango es de una dimensión por tratarse de la suma de vectores, pero podría ser de dos si multiplicásemos matrices. `GetVsize` proporciona el total de iteraciones que debemos computar y es requerido por LogFit para poder distribuir la carga de trabajo, así como concluir la ejecución una vez finalizado el espacio de iteraciones.

```
__kernel void sum(t_index indexes, __global float* Adevice,
                  __global float* Bdevice, __global float* Cdevice) {
    int idx = get_global_id(0) + indexes.begin;
    if (idx >= indexes.end) return;
    Cdevice[idx] = Adevice[idx] + Bdevice[idx];
}
```

Figura 18: Ejemplo de kernel invocado en el opencl_node de flow_graph

- Para poder ejecutar código en GPU, oneTBB requiere un fichero fuente adicional que contenga el método a ejecutar por el dispositivo, el *kernel*, implementado en la Figura 18. El *kernel* OpenCL implementa una única operación de suma ya que al computarse en GPU, el código en si ya esta siendo paralelizado a nivel de WG y WI, simplificando al máximo su uso (Código 18).

- Para finalizar nuestro ejemplo, hemos implementado una función `main` (ver Figura 19) mínima que recibe como argumentos de entrada los parámetros necesarios como el nombre del `kernel`, el fichero que lo contiene y el numero de hebras de CPU que se desean lanzar en paralelo.

3.3.3. Implementación

A continuación, explicaremos la otra cara del *template*, su implementación interna y como nos servimos del modelo *flow_graph* y del algoritmo LogFit. Comenzaremos desde la visión general e iremos aproximándonos al código a medida que entremos en detalle.

```
#include <cstdlib>
#include <iostream>
#include <tbb/task_scheduler_init.h>
#include "Implementations/Bodies/TestExecutionBody.h"
#include "Implementations/Engines/LogFitEngine.h"
#include "Implementations/Scheduler/GraphScheduler.h"

using namespace std;
using namespace tbb;

int main(int argc, char** argv){

  Params p = ConsoleUtils::parseArgs(argc, argv);
    auto logFitEngine{new LogFitEngine(p.numcpus, p.numgpus, 1, 1)};
    auto body{new TestExecutionBody()};

    GraphScheduler <TestExecutionBody, LogFitEngine, t_index, buffer_f, buffer_f, buffer_f>
        logFitGraphScheduler(p, *body, *logFitEngine);

    logFitGraphScheduler.StartParallelExecution();
    delete(body);
    delete(logFitEngine);
  return EXIT_SUCCESS;
}
```

Figura 19: Ejemplo de implementación del main para la suma de dos vectores

3.3.4. Diseño del grafo.

Nuestro grafo que modela la ejecución paralela en CPU-GPU consta de cuatro nodos (Figura 20), organizados en forma de grafo cíclico dirigido. El fin de la ejecución se da cuando el `dispatcher_node` encargado de activar un nodo CPU o GPU, según el tipo del *token* que recibe como entrada, deja de recibir *tokens*.

A diferencia del resto del grafo, el `dispatcher_node` no tiene ninguna arista de salida ya que realmente queda implementado como un nodo función que hace un `try_put` al nodo función de CPU o al nodo OpenCL de la GPU, en lugar de propagar una salida, por lo que el grafo no conectaría este nodo con sus dos "sucesores". A continuación, están los nodos de cómputo en CPU y GPU. Las entradas de estos nodos "sucesores" vienen determinadas por el tipo de ejecución que realizan, recibiendo el nodo CPU el rango de índices correspondiente

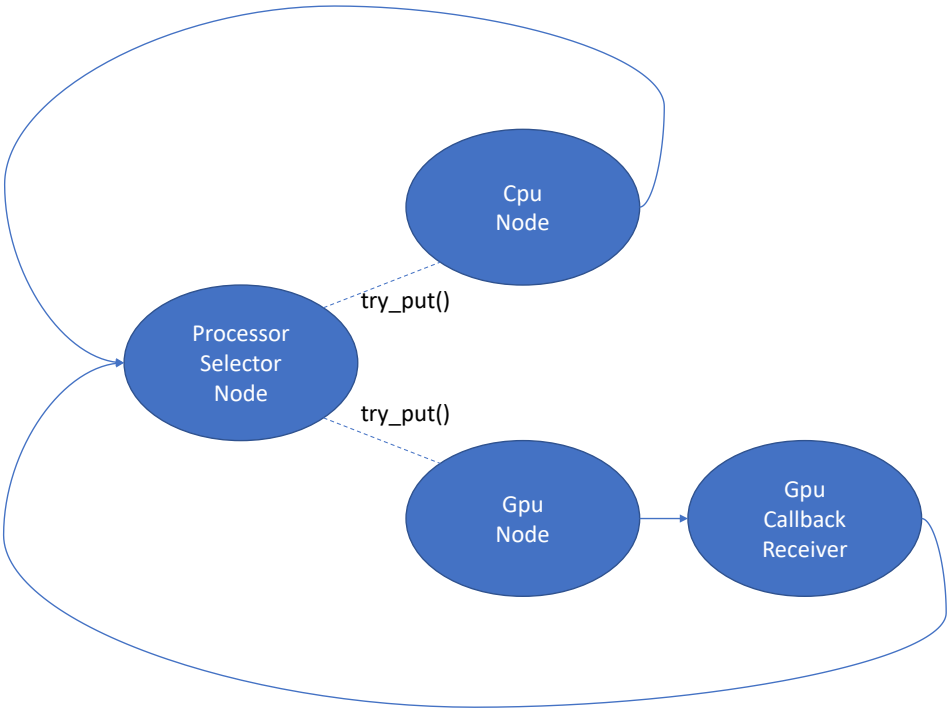

Figura 20: Esquema de la arquitectura de la solución basada en flow_graph

a su *chunk* y llamando a la rutina implementada en el objeto Body del usuario. Es antes y después de la ejecución del operatorCPU cuando el nodo CPU toma medidas del tiempo y *throughput* de la CPU, dichos datos son transmitidos al algoritmo LogFit para su procesado y optimización del tamaño del *chunk* en la siguiente iteración.

Por su parte, el nodo OpenCL requiere de todos los argumentos del *kernel* para ser ejecutado. Es aquí donde el método del usuario `GetGPUArgs` es invocado y su tupla resultado procesada por el dispositivo OpenCL. Como hemos mencionado en el capítulo segundo, el *opencl_node* permite la especificación de un `DeviceSelector` que nos garantice el dispositivo de ejecución de nuestro kernel. Nuestra implementación emplea una versión inspirada en el usado en el libro Pro TBB [10] (11) mediante la cual cacheamos el dispositivo correspondiente a la GPU para mayor eficiencia y certidumbre en la ejecución y medidas del rendimiento.

Conectado a este nodo encontramos su sucesor, el `gpuJoiner`, encargado de recibir el mensaje de compleción del *opencl_node* y disparar el evento de GPU inactiva en forma de *token* GPU que es directamente propagado al `dispatcher_node` que volverá a calcular un *chunk* de GPU y repetir el ciclo de computación.

Este nodo intermedio que conecta con el *opencl_node* no es requerido por el nodo de CPU,

ya que este ha sido implementado como nodo función de oneTBB y con conectarlo de vuelta al `dispatcher_node` pasando un token CPU como valor de retorno, es suficiente para indicar la finalización de un ciclo de ejecución en CPU. De esta forma no es necesario propagar un objeto concurrente que nos determine el numero de dispositivos en espera, simplificando la gestión de los mismos.

Entrando en detalle sobre el `gpu_node`, al realizar parte de la ejecución en GPU, debemos construir una tupla con los argumentos del *kernel*, responsabilidad del usuario del *template*. En dicha tupla, es obligatorio por motivos de implementación del nodo `gpuJoiner` que los índices de ejecución del kernel sean de un tipo nombrado `t_index` para poder crear un arco entre ellos como explicaremos más adelante. Para la creación de esta tupla, se requiere del procedimiento `GetGPUArgs`, el único proceso que debe conocer internamente el usuario para usar nuestro *template*.

Como adelantamos, el primer argumento tanto del *kernel* como de la tupla esta reservado para una estructura que albergue los índices primero y último. Esta restricción de diseño es necesaria porque al ser código de oneTBB quien maneja el envío y la captura de los datos hacia la GPU mediante el *opencl_node*, no podemos controlar cuando termina la ejecución de la función de GPU salvo que lo conectemos a un nodo sucesor, y por tanto, nos sería imposible tomar mediciones de tiempo. Para salvar esta "caja negra", se ha conectado el nodo GPU a un sucesor en su puerto cero, la estructura que aglutina los índices, que es la restricción mencionada. Recordemos que el *opencl_node* es heredero de la clase *streaming_node* que se define con tantos puertos de salida como argumentos de entrada tiene y cuyo sucesor será activado una vez el dispositivo haya finalizado la ejecución. Con esta única restricción en el código usuario, podemos construir un sucesor conectado al puerto de salida cero del tipo nodo función que recibe un `t_index` y devuelve un *token* GPU al `dispatcher_node` que como en el caso de la CPU, actúa de mensaje indicando que la GPU esta lista para seguir procesando *chunks*.

3.3.5. Implementación del GraphScheduler

Una vez definido el flujo de datos y de tareas de computación que lleva a cabo nuestro planificador, procederemos a detallar como construimos cada uno de los nodos, su funcionamiento y que ventajas aporta frente a la implementación original basada en *pipeline*.

Con el objetivo de comparar las diversas implementaciones (Heterogeneous Parallel_For,

```
#include "tbb/tick_count.h"
#include "../../Utils/ConsoleUtils.h"

class IScheduler {
protected:
    Params parameters;
    tick_count startBench, stopBench, startCpu, stopCpu, startGpu, stopGpu;
    double runtime;
    IScheduler(Params p) : parameters{p} {}

public:
    virtual void StartParallelExecution() = 0;

    void startTimeAndEnergy(){
        startBench = tick_count::now();
    }
    void endTimeAndEnergy(){
        stopBench = tick_count::now();
        runtime = (stopBench-startBench).seconds()*1000;
    }
    void saveResultsForBench() {
        ConsoleUtils::saveResultsForBench(parameters, runtime);
    }
};
```

Figura 21: Implementación de la clase abstracta IScheduler

Flow Graph y oneAPI) hemos extraído la implementación del algoritmo de la implementación del planificador. Para ello empleamos tanto métodos de tipo virtual[23] como clases abstractas[24] que definen las tareas propias del planificador y del algoritmo. En el Código 21 se ve qué métodos proveemos como estándar y cual debe implementar el planificador elegido, `StartParallelExecution`. En el desarrollo de este TFG, nuestro `GraphScheduler` implementa ahí la ejecución del *flow_graph*. Futuros desarrollos sobre nuevos planificadores podrían reescribir ese método y decidir de forma diferente como ejecutar el código paralelo sin necesidad de reescribir el algoritmo de selección de *chunks*. Esta ventaja no solo es de cara a ampliar el

[23]Un método que permite redefinir su implementación a las clases herederas, prevaleciendo esta sobre la implementación de la clase padre.

[24]Implementadas en C++ con al menos uno de sus métodos de tipo virtual igual a cero, por lo que no es posible instanciarlas directamente.

estudio con distintos planificadores como el mencionado *pipeline*, sino que ofrece modularidad para alternar entre algoritmos agnósticos de la planificación de tareas y ejecución de *kernels* en aceleradoras.

```
class IEngine {
public:
    virtual unsigned int getGPUChunk(unsigned int begin, unsigned int end) = 0;
    virtual unsigned int getCPUChunk(unsigned int begin, unsigned int end) = 0;
    virtual void recordGPUTh(unsigned int chunk, float time) = 0;
    virtual void recordCPUTh(unsigned int chunk, float time) = 0;
    virtual void reStart() = 0;
};
```

Figura 22: Implementación de la clase abstracta del algoritmo de particionado

Nuestro `GraphScheduler` (Figura 28) es así un *template* que acepta objetos herederos de `IEngine` (Figura 22), como `LogFitEngine` en nuestro caso. Asimismo requiere de un objeto que implemente `IBody` implementado por el usuario y una especificación de los argumentos de entrada del *kernel*, del mismo modo y orden que se definen para el método `GetGPUArgs`. Esto es debido a que el *opencl_node* requiere de esta especificación en tiempo de compilación y que no es posible descomponer la tupla en sus tipos sin conocer cuales son. Es una concesión que debemos pagar por utilizar tuplas, más adelante hablaremos de como iteramos sobre la tupla de entrada del nodo GPU y los retos a los que nos enfrentamos.

Por tanto, la inicialización de nuestro `GraphScheduler` queda como vimos en el ejemplo 19:

```
GraphScheduler <IBody, LogfitEngine, t_index, [typenames for GPU params...]>
  logFitGraphScheduler(parameters, *body, *logFitEngine)
```

Como hemos resaltado, el tipo `IBody` debe reemplazarse por el tipo heredero implementado para resolver el problema del usuario y los `typenames` deben ser de la forma mencionada en el ejemplo de suma de vectores.

Profundizando en `GraphScheduler`, su constructor es el encargado de inicializar las estructuras de datos, nodos y conexiones, para ello detallaremos uno a uno el proceso de construcción de los nodos del grafo:

- El cpuNode (Figura 23): La implementación del cpuNode se realiza sobre el nodo función de oneTBB, este recibe una cierta entrada y devuelve un valor u objeto de salida. Nuestro CPU ejecutará el operatorCPU entre las medidas de tiempo que usa el algoritmo LogFit. Hemos decidido utilizar un nodo con concurrencia ilimitada debido a que puede ocurrir que mas de un token de CPU sea encolado mientras se ejecuta una tarea. Esta concurrencia es clave para permitir el cómputo paralelo en CPU, evitando usar el *parallel_for* sobre el que trabaja el pipeline.

```
cpuNode {
    graph, flow::unlimited,
    [&](t_index indexes) -> ProcessorUnit {
        startCpu = tick_count::now();
        body.OperatorCPU(indexes.begin, indexes.end);
        stopCpu = tick_count::now();
        engine.recordCPUTh(indexes.end - indexes.begin, (stopCpu - startCpu).seconds());
        return CPU;
    }
}
```

Figura 23: Implementación del cpuNode

- El gpuNode (Figura 24): Es implementado como un *opencl_node* y en lugar de medir tiempos en él, debemos hacerlo en su antecesor y su sucesor, ya que es un nodo interno de oneTBB y no podemos controlar su ejecución, solo la tarea que realiza, el *kernel* que ejecuta. Vemos que empleamos el *kernel* directamente desde el fuente proporcionado por los parámetros de entrada del planificador y el gpuSelector personalizado por nosotros.

```
gpuNode{
    graph,
    flow::opencl_program<>(flow::opencl_program_type::SOURCE, parameters.openclFile)
        .get_kernel(parameters.kernelName),
    gpuSelector
}
```

Figura 24: Implementación del gpuNode

- El gpuCallbackReceiver (Figura 25): Este nodo función recibe el evento de finalización

de la GPU y toma las medidas correspondientes de tiempo y encola otro *token* al nodo que activa la siguiente iteración de GPU.

```
gpuCallbackReceiver{
    graph,
    flow::unlimited,
    [&](t_index indexes) -> ProcessorUnit {
        stopGpu = tick_count::now();
        engine.recordGPUTh(ndexes.end - indexes.begin, (stopGpu - startGpu).seconds());
        return GPU;
    }
}
```

Figura 25: Implementación del gpuCallbackReceiver

- El `processorSelectorNode` (Figuras 26): Es aquí donde comienza y termina la ejecución del grafo, en el momento de inicializarse la ejecución, se introducen mediante un bucle implementado en el método de la Figura 28, tantos *tokens* de CPU y GPU como el usuario haya determinado y estos comienzan la ejecución del algoritmo. Cabe mencionar que aunque se realice el `try_put` secuencialmente, esto no garantiza la ejecución secuencial de su procesado. Para ello seleccionamos el tipo de concurrencia del nodo a serie, con lo que solo existe una única instancia del selector procesando `tokens` cada vez.

  Para encolar trabajo a la GPU, el nodo selector solicita al objeto `engine`, que implementa el algoritmo de particionado LogFit, el tamaño del *chunk* correspondiente a la GPU y construye mediante el método `getGPUArgs` la tupla que, junto con la referencia al `gpuNode`, conforman la entrada de la función auxiliar (Figura 27) detallada más adelante. Es esta función quien mediante técnicas de "desenrollado´´ de tuplas en tiempo de compilación, genera el código equivalente a un `try_put` puerto a puerto del *opencl_node*.

  Asimismo, cuando el *token* recibido es de tipo CPU, la ejecución se simplifica, pues una vez resuelto el tamaño del *chunk* y comprobado que no da desbordamiento sobre el total, se ejecuta un `try_put` corriente sobre `cpuNode`.

Concluimos la explicación detallada de los nodos explicando qué hace exactamente la función de nuestra librería `dataStructures::try_put`. Como observamos en el Código 27, la

```
processorSelectorNode{
...
if (token == GPU) {
    int gpuChunk = engine.getGPUChunk(begin, end);
    if (gpuChunk > 0) {
        int auxEnd =begin + gpuChunk;
        t_index indexes = {begin, auxEnd};
        begin = auxEnd;
        gpuNode.set_range(body.GetNDRange());
        auto args = body.GetGPUArgs(indexes);
        startGpu = tick_count::now();
        dataStructures::try_put<0, TArgs...>(&gpuNode, args);
    }
} else {
    int cpuChunk = engine.getCPUChunk(begin, end);
    if (cpuChunk > 0) {
        int auxEnd = begin + cpuChunk;
        auxEnd = (auxEnd > end) ? end : auxEnd;
        t_index indexes =  {begin, auxEnd};
        begin = auxEnd;
        cpuNode.try_put(indexes);
    }
...
}
```

Figura 26: Implementación del processorSelectorNode.

función se define por casos, el caso recursivo donde `I` es menor que el número de tipos definidos en la tupla argumento del *template* y el caso base, donde `I` es igual a la longitud de dicha lista.

Para comprender mejor ambos casos, supongamos que nuestro *kernel* solo utiliza el argumento obligatorio, `t_index`, donde por lo anteriormente explicado, la llamada al método comienza con *I valor cero*, pero la tupla conformada por los tipos `Tg...`, tiene al menos un elemento (`t_index`), esto provoca que la función ejecutada sea el caso recursivo, ya que la condición se cumple (*I de valor cero*, es menor que uno). Entonces, extraemos el argumento «Iésimo» de la tupla con la función de librería *get<I>* que nos devuelve `t_index` (recordemos I igual a cero), una vez tenemos el argumento que debemos pasar al `gpuNode`, utilizando también funciones propias de oneTBB, realizamos un `try_put` ordinario a su «Iésimo» puerto de entrada, con el argumento extraído y realizamos la invocación recursiva con `I más uno`, para

```
template<std::size_t I = 0, typename ...Tg>
inline typename std::enable_if<I == sizeof...(Tg), void>::type
try_put(
  tbb::flow::opencl_node<tbb::flow::tuple<Tg...>> *node,
  tbb::flow::tuple<Tg...> &args) { }

template<std::size_t I = 0, typename ...Tg>
inline typename std::enable_if<I < sizeof...(Tg), void>::type
try_put(
  tbb::flow::opencl_node<tbb::flow::tuple<Tg...>> *node,
  tbb::flow::tuple<Tg...> &args) {
    auto t = tbb::flow::get<I>(args);
    tbb::flow::interface11::input_port<I>(*node).try_put(t);
    try_put<I + 1, Tg...>(node, args);
}
```

Figura 27: Implementación del dataStructures::try_put

procesar cada argumento de la tupla.

El caso base es trivial, pues si *I es igual a la longitud de la tupla*, todos los argumentos han sido extraídos e introducidos en los puertos correspondientes del `gpuNode`. Pero, ¿por qué es necesario este *template* e implementación por casos para algo que un bucle de tipo *for* hace mucho mas fácilmente y sobre cualquier estructura de datos iterable? La respuesta es una restricción del lenguaje sobre el tipo tupla y como opera en C++11- Esto es, la tupla no puede ser accedida como si de un iterable corriente en tiempo de ejecución se tratase, debe ser definida estáticamente en tiempo de compilación tanto en su longitud (como la longitud de un array clásico) como en su tipado, por lo que sus posibles accesos a posiciones inexistentes (tupla de dos posiciones sobre la que se realiza un *get<3>*), nos daría un error en compilación. Pero si nuestro objetivo es hacer una librería que no requiera de que el programador interaccione con oneTBB, esto es un impedimento, ya que no solo tendría que proporcionarnos la tupla en la función `getGPUArgs`, debería proporcionarnos el método que implemente su propio `try_put`, tal como nuestro selector invoca la función aquí detallada.

Para solventar este problema, nuestra función "desenrolla´´ los argumentos de la tupla uno a uno en tiempo de compilación. Esto es, le estamos diciendo al compilador cuantas veces accederemos a la tupla y en qué posición de manera unívoca y predefinida, ya que I es conocida en todos sus posibles valores en el momento en el que se define el tipo de entrada del

gpuNode. Con este pequeño método alternativo al bucle *for*, podemos ocultar la existencia del *opencl_node* de la vista del usuario.

```
template<typename TExectionBody, typename TSchedulerEngine, typename ...TArgs>
class GraphScheduler : public IScheduler {
private:
    .........
public:
    .........
    void StartParallelExecution() {
        begin = 0;
        end = body.GetVsize();
        engine.reStart();
        for (int i = 0; i < parameters.numcpus; ++i) {
            processorSelectorNode.try_put(CPU);
        }
        for (int j = 0; j < parameters.numgpus; ++j) {
            processorSelectorNode.try_put(GPU);
        }
        graph.wait_for_all();
    }
};
```

Figura 28: Implementación del método abstracto del planificador basado en flow_graph

```
GraphScheduler(Params p, TExectionBody &body, TSchedulerEngine &engine) :
    IScheduler(p), body{body}, engine{engine}, begin{0}, end{body.GetVsize()},
    cpuNode{ ... }, gpuNode{ ... }, gpuCallbackReceiver{ ... }, processorSelectorNode{ ... }
{
  flow::make_edge(cpuNode, processorSelectorNode);
  flow::make_edge(gpuCallbackReceiver, processorSelectorNode);
  flow::make_edge(flow::output_port<0>(gpuNode), gpuCallbackReceiver);
}
```

Figura 29: Implementación del constructor del planificador basado en flow_graph

3.4. Implementacion en oneAPI

Para implementar el planificador en oneAPI, hemos adaptado tanto el *pipeline* original como nuestro *template* basado en Flow Graph, donde en el primero, se han eliminado las fun-

ciones de soporte a compilación y ejecución de OpenCL asi como los manejadores de memoria, pues nos basamos en el modelo USM. En la adaptación del Flow Graph, el nodo OpenCL ha sido reemplazado por un nodo función que es el encargado de ejecutar la función `OperatorGPU`. Ambos planificadores basados en oneAPI, heredan del `IScheduler` (Figura 21) anteriormente presentado, pero difieren en la implementación de la función `StartParallelExecution`, siendo el *pipeline* semejante al actual Heterogeneous Parallel_For, donde se ha modificado la función `executeOnGPU` del filtro paralelo por una sin funciones de soporte para compilar código OpenCL (Figura 40).

3.4.1. Uso y funcionamiento

Al igual que el Flow Graph hemos implementado la solución basada en DPC++ de forma modular tal que el usuario de la librería solo requiere de implementar el *body* y de seleccionar el algoritmo y el planificador correspondientes. Al introducir este nuevo planificador que requiere de un *body* diferente, nuestra implementación inicial debe extenderse para asegurar la seguridad y facilidad de su uso, por eso, hemos introducido una capa de abstracción que permite emplear un planificador u otro de manera mas sencilla. Esta abstracción esta basada en el patrón de diseño *factory* o factoría en español, que consiste en proveer de una clase estática sobre la que delegamos la creación y destrucción de objetos de un mismo tipo o, como en este caso, planificadores con comportamientos comunes. En la figuras: 31, 32 y 33 se detalla su implementación, volveremos a ellas más adelante. Mediante el uso de esta factoría, el usuario no necesita manejar punteros para el algoritmo, el planificador ni el *body*, pues la fábrica se encarga de proveer instancias y de eliminarlas así como sus componentes (ver ejemplo de la Figura 30).

Con la factoría, seleccionar nuestro planificador es más sencillo, aunque deberemos emplear la interfaz correcta de `IBody`, pues un *body* para Flow Graph requiere de la función `GetGPUArgs` mientras que un *body* de oneAPI, no necesita esta pero si de `OperatorGPU`, hemos declarado dos interfaces herederas de `IBody` que harán las veces de base para nuestras implementaciones adaptadas al problema. Estas restricciones de tipo que nos obliga la factoría se han implementado para otorgar una mayor seguridad en la codificación de aplicaciones por parte del usuario, pues impide que errores de tipos, frecuentemente enmascarados hasta el *runtime*, debido a las diversas capas de abstracción, sean capturados lo antes posible, en

```
namespace HelperFactories {
  struct SchedulerFactory {
    template<typename TScheduler, typename TEngine, typename TBody>
    static inline void deleteInstance (TScheduler *instance) {
        delete ((TBody*)instance->getBody());
        HelperFactories::EngineFactory::deleteInstance<TEngine>(
            (TEngine*)instance->getEngine()
        );
        delete (instance);
    }
  };
}
```

Figura 30: Implementación del borrado en el patrón factoría.

tiempo de compilación y, así solventados de forma más eficiente. También nos proporciona un azucarillo sintáctico que orienta sobre qué tareas y métodos deben ser implementados y quién los ejecutará posteriormente.

La funcionalidad clave que nos permite llevar a cabo la implementación de la factoría fue introducida en C++ en su versión 11, «std::enable_if»[1], esta llamada a librería permite que en tiempo de compilación, el preprocesamiento del código nos habilite o no un método o cualquier otro tipo de estructura de datos. De esta forma, en el fuente aparecen todos los métodos, pero, al *runtime* objetivo solo serán accesibles aquellos que cumplen las condiciones, estos son los asociados a cada planificador y *body* que el usuario elige implementar.

Ambas soluciones en oneAPI son fácilmente implementables al tener un *template* sencillo, es decir, requieren de un `TBody` y un `TEngine`, mientras que `GraphScheduler`, puede tener indefinidos argumentos adicionales (la entrada al *kernel* de OpenCL), por lo que necesitamos de la función auxiliar `is_graph_body<TBody>()` que nos realiza una comprobación de que el `TBody` es efectivamente un `IGraphBody<Args ...>`, coincidiendo en tipo con los `Args` del `GraphScheduler`

Una vez heredado `IOneApiBody` y definido nuestro *kernel* en C++ tal como se especifica en DPC++. De esta forma, el código mínimo a implementar por el usuario es el *main* principal (Figura 34) y el *body* (Figura 35) que explicaremos a continuación.

```
namespace HelperFactories {
  struct SchedulerFactory {
    template<typename TScheduler, typename TEngine, typename TBody, typename ...Args>
    static inline typename std::enable_if<std::is_same<TScheduler,
        GraphScheduler<TEngine, TBody, Args ...>>::value
        && is_graph_body<TBody>(), TScheduler *>
    ::type getInstance(Params p, TBody *body) {
        auto engine{ HelperFactories::EngineFactory::getInstance<TEngine>(
            p.numcpus, p.numgpus, 1, 1
        )};
        return new GraphScheduler<TEngine, TBody, Args...>(p, *body, *engine);
    }
  };
}
```

Figura 31: Implementación del patrón factoría. GraphScheduler

```
namespace HelperFactories {
  struct SchedulerFactory {
    template<typename TScheduler, typename TEngine, typename TBody, typename ...Args>
    static inline typename std::enable_if<std::is_same<TScheduler,
        OneApiScheduler<TEngine, TBody>>::value
        && is_base_of<IOneApiBody, TBody>::value, TScheduler *>
    ::type getInstance(Params p, TBody *body) {
        auto engine{ HelperFactories::EngineFactory::getInstance<TEngine>(
            p.numcpus, p.numgpus, 1, 1
        )};
        return new OneApiScheduler<TEngine, TBody>(p, *body, *engine);
    }
  };
}
```

Figura 32: Implementación del patrón factoría. OneApiScheduler.

3.4.2. Ejemplo de uso.

Comenzaremos a detallar como utilizar el *template* con la factoría para, como en el ejemplo anterior, realizar una suma de dos vectores.

- El usuario programador debe crear un fichero que implemente la clase `Body` especifica para su problema, esta clase ha de ser heredera del `IOneApiBody` para verse obligada

```
namespace HelperFactories {
  struct SchedulerFactory {
    template<typename TScheduler, typename TEngine, typename TBody, typename ...Args>
    static inline typename std::is_same<TScheduler,
        OnePipelineScheduler<TEngine, TBody>>::value
        && is_base_of<IOneApiBody, TBody>::value, TScheduler *>
    ::type getInstance(Params p, TBody *body) {
        auto engine{ HelperFactories::EngineFactory::getInstance<TEngine>(
            p.numcpus, p.numgpus, 1, 1
        )};
        return new OnePipelineScheduler<TEngine, TBody>(p, *body, *engine);
    }
  };
}
```

Figura 33: Implementación del patrón factoría. OnePipelineScheduler.

a implementar sus métodos abstractos que serán requeridos por el planificador basado en DPC++. Si bien `OperatorCPU` será invocado directamente por la CPU, el método `OperatorGPU` requiere del usuario para su especificación, ha de crear y manejar una cola (`cl::sycl::queue`) e inicializarla al dispositivo GPU o cualquiera de los que requiera su implementación.

Una vez definida la función en CPU y el *kernel* en C++, así como el resto de funciones obligatorias, como el rango de iteraciones o el `ShowCallback`. El resto de funciones comunes al Flow Graph tienen el mismo significado y no entraremos en detalles redundantes.

- Para concluir con nuestro ejemplo, en la Figura 34 hemos implementado una función *main* mínima que se sirve del patrón factoría y semejante al caso anterior, recibe como argumentos de entrada el número de tareas simultáneas de CPUs y GPU que se desean activar. Aunque el ejemplo aquí mostrado utiliza el `OnePipelineScheduler` como planificador, debido a que ambas implementaciones basadas en oneAPI están modularizadas, el *body* es válido tanto para la implementación basada en pipeline, como en *flow_graph*, OneApiScheduler.

```
#include <scheduler/SchedulerFactory.h>
#include <utils/Utils.h>
#include "Implementations/Bodies/TestOneApiBody.h"

using namespace std;
using namespace tbb;
using MySchedulerType = OnePipelineScheduler <LogFitEngine, TestOneApiBody>;

int main(int argc, char **argv) {
    Params p = ConsoleUtils::parseArgs(argc, argv);

    auto logFitOneApiScheduler{ HelperFactories::SchedulerFactory
        ::getInstance<MySchedulerType, LogFitEngine, TestOneApiBody, vector<double>>
        (p, new TestOneApiBody())
      };
    logFitOneApiScheduler->startTimeAndEnergy();
    logFitOneApiScheduler->StartParallelExecution();
    logFitOneApiScheduler->endTimeAndEnergy();
    logFitOneApiScheduler->saveResultsForBench();

    HelperFactories::SchedulerFactory
        ::deleteInstance<MySchedulerType,LogFitEngine,TestOneApiBody> (logFitOneApiScheduler);
    return EXIT_SUCCESS;
}
```

Figura 34: Implementación del *main* mínimo para oneAPI utilizando el patrón factoría.

3.4.3. Implementación

Ahora que hemos presentado cómo utilizar y cuáles son las analogías que presenta el uso de oneAPI frente a *flow_graph* o el *pipeline* anterior, pormenorizaremos cuales son las partes que componen estos planificadores, sus semejanzas a los anteriores y sus diferencias clave. También desarrollaremos el `IOneApiBody` y como hemos realizado el ejemplo presentado.

3.4.4. IOneApiBody

La clase abstracta `IOneApiBody` nos sirve de interfaz para poder desarrollar una clase Body que sea compatible con los planificadores basados en oneAPI. Esto implica la implementación del método abstracto y fundamental, el `operatorGPU`, pues es quien define el *kernel* en C++ que será ejecutado por el dispositivo acelerador.

```
class TestOneApiBody : public IOneApiBody {
public:
    void OperatorCPU(int begin, int end) {
        for (int i = begin; i < end; i++) {
            C[i] = A[i] + B[i];
        }
    }
    void OperatorGPU(int begin, int end) {
        using namespace cl::sycl;
        gpu_queue.submit([&](handler& handler){
            handler.parallel_for(range<1>{vsize}, [=](id<1> id){
                auto idx = id[0];
                C[idx] = A[idx] + B[idx];
              }
            );
        });
        gpu_queue.wait();
    }
    size_t GetVsize() {
        return vsize;
    }
};
```

Figura 35: Ejemplo del Body para la suma de dos vectores en DPC++

Para implementar este *kernel* en C++, se requiere del compilador de DPC++, así como de crear una `cl::sycl::queue` asociada al dispositivo. A diferencia de las estructuras de datos necesarias para los planificadores anteriores, el `IOneApiBody` no requiere de redefiniciones de tipos ni de declaración de variables en OpenCL para trabajar, ya que nos basamos en el modelo USM donde el dispositivo anfitrión, o *host* en inglés, y aceleradores comparten acceso a memoria. Como ejemplo, el anteriormente referido `TestOneApiBody` (Figura 35)

3.4.5. OneApiScheduler

Para implementar el planificador hemos partido de nuestro `GraphScheduler`, simplificando las operaciones relativas al nodo GPU. Esto se aprecia en el diagrama de flujo de datos de oneAPI basado en *flow_graph* 36. La diferencia frente al `GraphScheduler` es que en la reimplementación de grafo, este ahora consta de tres nodos, el `processorSelectorNode`, el nodo CPU y el GPU, que ahora en lugar de tratarse de un nodo OpenCL, es un nodo función. Asimismo

el método `StartParallelExecution` 37 es prácticamente el mismo que en `GraphScheduler`.

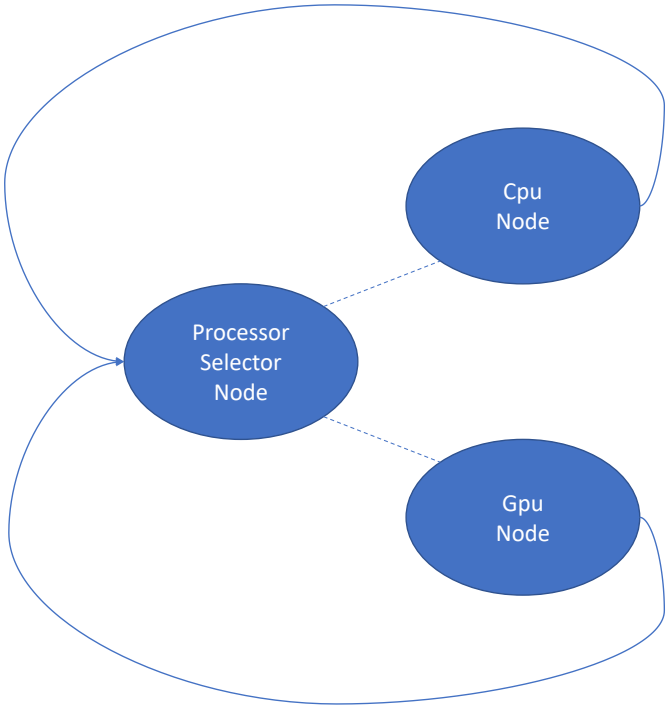

Figura 36: Esquema de la arquitectura de la solución basada en oneAPI sobre flow_graph.

Esta simplificación en el modelo, permite utilizar el `IOneApiBody` de forma mas sencilla, si bien, utilizar un nodo función bloquea una hebra de computación al invocar el kernel en el dispositivo, hemos optado por esta pequeña concesión a cambio de simplificar enormemente la tarea del usuario programador. Desarrollaremos esta posible mejora en el capítulo 4.

Debido a la completa analogía en el resto de los componentes, no entraremos en redundancia sobre como se han implementado, más allá del reconstruido nodo GPU, como se aprecia en la Figura 38. Este nodo es ahora prácticamente igual que el nodo CPU, así como enormemente más sencillo de comprender y extender si fuese necesario.

3.4.6. OnePipelineScheduler

En el caso del planificador oneAPI basado en el *pipeline*, hemos eliminado todas las funciones competentes a OpenCL, sus colas de comandos y sus variables asociadas tanto a dispositivos como a los tamaños del WG y los WI. Reduciendo enormemente la complejidad del planificador, como veremos en el Capítulo 4. Respecto de las funciones, clases y flujo de datos,

```
template<typename TExectionBody,
         typename TSchedulerEngine>
class OneApiScheduler : public IScheduler {
private:
    .........
public:
    .........
    void StartParallelExecution() {
        begin = 0;
        end = body.GetVsize();
        engine.reStart();
        for (int i = 0; i < parameters.numcpus; ++i) {
            processorSelectorNode.try_put(CPU);
        }
        for (int j = 0; j < parameters.numgpus; ++j) {
            processorSelectorNode.try_put(GPU);
        }
        graph.wait_for_all();
    }
};
```

Figura 37: Implementación del método abstracto del planificador basado en DPC++ sobre flow_graph

```
gpuNode {
    graph, flow::unlimited,
    [&](t_index indexes) -> ProcessorUnit {
    startGpu = tick_count::now();
    body.OperatorGPU(indexes.begin, indexes.end);
    stopGpu = tick_count::now();
    engine.recordGPUTh(indexes.end - indexes.begin, (stopGpu - startGpu).seconds());
    return GPU;
    }
}
```

Figura 38: Implementación del gpuNode

como apreciamos en el diagrama de flujo de datos de la implementación en la Figura 39 la diferencia frente al `GraphScheduler` es que necesitamos de dos clases auxiliares que implementan las funciones a procesar en cada etapa del *pipeline*. Estos filtros son quienes redistribuyen el flujo de datos hacia CPU o GPU según corresponda el dispositivo disponible.

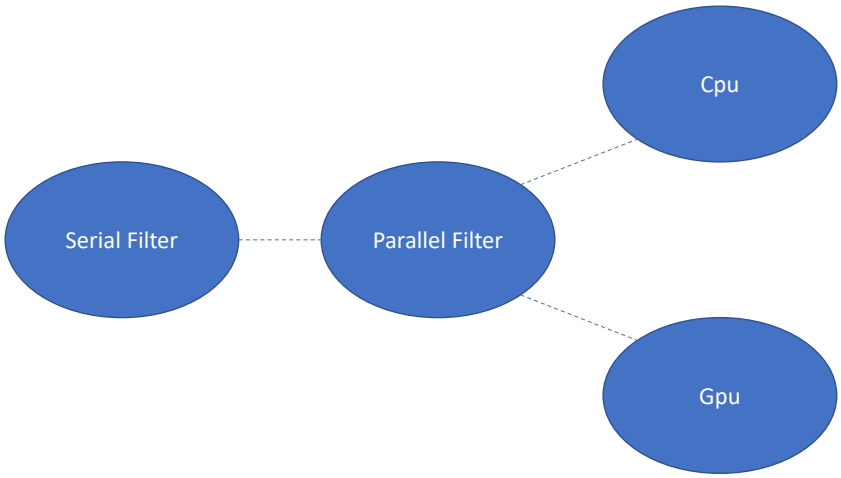

Figura 39: Esquema de la arquitectura de la solución basada en oneAPI sobre pipeline.

La implementación de los filtros parte de los actuales en Heterogeneous Parallel_For, pero eliminando todo el código de soporte de OpenCL (ver Figura 40). El `SerialFilter` ha sido reutilizado, al ser el encargado de generar *bundles*, sin conocer realmente como se ejecutarán esos *bundles* por el `ParallelFilter`.

También debemos de implementar el método `StartParallelExecution`, tomando como referencia el *pipeline*(ver Figura 41). Si bien, los cambios consisten en referenciar los nuevos filtros y añadir los métodos que éstos necesitan del *pipeline*.

De esta forma, quedaría nuestra librería finalizada. La Figura 42 resume de forma gráfica el diagrama de clases simplificado que hemos generado mediante YUML[2].

```
ParallelFilter(int begin, int end, TScheduler *scheduler) :
    filter(false), begin{begin}, end{end}, scheduler{scheduler} {}

void *operator()(void *item) {
    if (item == NULL) return NULL;
    Bundle *bundle = (Bundle *) item;
    if (bundle->type == GPU) {
        executeOnGPU(bundle);
        scheduler->gpuStatus++;
    } else {
        executeOnCPU(bundle);
    }
    delete bundle;
    return NULL;
}
void executeOnGPU(Bundle *bundle) {
    scheduler->setStartGPU(tbb::tick_count::now());
    scheduler->getTypedBody()->OperatorGPU(bundle->begin, bundle->end);
    scheduler->setStopGPU(tbb::tick_count::now());
    float time = (scheduler->getStopGPU() - scheduler->getStartGPU()).seconds();
    scheduler->getTypedEngine()->recordGPUTh((bundle->end - bundle->begin), time);
}
void executeOnCPU(Bundle *bundle) {
    scheduler->setStartCPU(tbb::tick_count::now());
    scheduler->getTypedBody()->OperatorCPU(bundle->begin, bundle->end);
    scheduler->setStopCPU(tbb::tick_count::now());
    float time = (scheduler->getStopCPU() - scheduler->getStartCPU()).seconds();
    scheduler->getTypedEngine()->recordCPUTh((bundle->end - bundle->begin), time);
}
```

Figura 40: Implementación del filtro selector de dispositvo en DPC++ pipeline.

```
void StartParallelExecution() {
    int begin = 0;
    int end = body.GetVsize();
    engine.reStart();
    if (parameters.numgpus < 1) {
        tbb::parallel_for(tbb::blocked_range<int>(begin, end),
            [&](const tbb::blocked_range<int> &range) {
                startCpu = tbb::tick_count::now();
                body.OperatorCPU(range.begin(), range.end());
                stopCpu = tbb::tick_count::now();
                },
                tbb::auto_partitioner());
    } else {
        tbb::pipeline pipe;
        SerialFilter<OnePipelineScheduler> serial_filter(begin, end, this);
        ParallelFilter<OnePipelineScheduler> parallel_filter(begin, end, this);
        pipe.add_filter(serial_filter);
        pipe.add_filter(parallel_filter);
        pipe.run(parameters.numcpus + parameters.numgpus);
        pipe.clear();
    }
}
```

Figura 41: Implementación del método abstracto del planificador basado en DPC++ sobre pipelines.

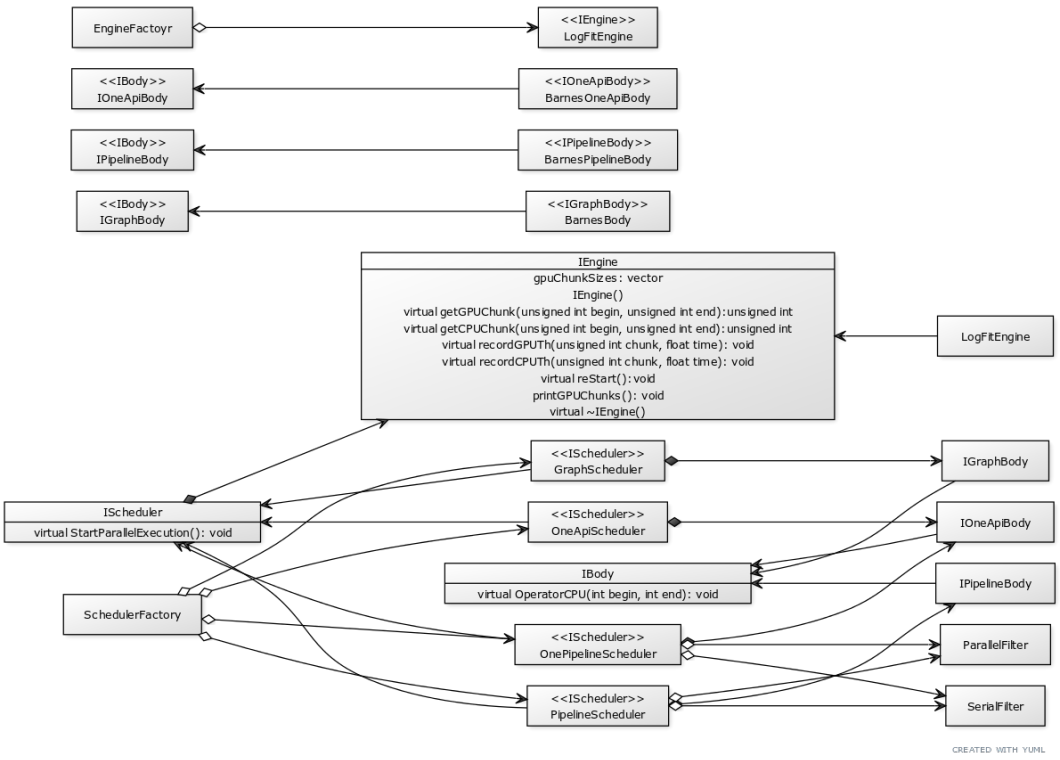

Figura 42: Diagrama de clases de la librería. Incluye los bodies para el benchmark Barnes Hut

4

Resultados experimentales

4.1. Introducción al *benchmarking*

El *bechmarking* o pruebas de rendimiento, han sido planteadas de forma que se enfrente al mismo conjunto de datos y empleando como único algoritmo de particionado, el algoritmo LogFit. Para ello, hemos utilizado el algoritmo de *benchamarking* Barnes Hut [8] para solucionar el problema de los n-cuerpos. Consiste en una simulación gravitacional de cuerpos durante un número de intervalos de tiempo. Cada uno de esos intervalos puede implementarse como una ejecución paralela donde se calcula la próxima posición y velocidad de cada cuerpo en el espacio tridimensional. La irregularidad del *benchmark* reside en que el número de computaciones para cada partícula depende de la distancia de las partículas entre si. El benchmark se ejecuta con una entrada de 100.000 cuerpos lo que resulta en un bucle paralelo de 100.000 iteraciones que se deben repartir entre CPU y GPU en caso de que esa ejecución heterogénea sea ventajosa.

Asimismo, la cantidad de trabajo a realizar varía entre iteraciones por el propio movimiento de las partículas. Para comprender como afecta esta variabilidad y carga en la GPU y su rendimiento, hemos realizado los experimentos un total de cinco veces con la misma configuración cada modelo de *scheduler* para obtener medidas estándar que no reflejen picos de trabajo derivados de otros procesos del sistema operativo.

Se ha empleado un procesador intel core i7-7700HQ de cuatro núcleos con *hyper-threading*[25], así como una unidad gráfica integrada modelo HD Graphics 630 ejecutando el *driver* OpenCL

[25]Tecnología que permite ejecutar hasta dos hilos de ejecución por cada núcleo físico del procesador, dando la sensación de tener el doble de capacidad de cómputo. En realidad los cores lógicos adicionales no ofrecen el mismo rendimiento que los cores físicos ya que comparten unidades funcionales con estos últimos

compatible con oneAPI y TBB.

Las gráficas resultado de nuestros experimentos mostradas a lo largo de este capítulo muestran la comparativa en velocidad entre ejecuciones utilizando desde uno hasta ocho hilos de CPU y su comparación cuando la GPU colabora. El eje x esta numerado del uno al nueve, para indicar el número de unidades de computación. En las gráficas en las que se muestran rendimiento de CPU+GPU, el "1" indica GPU, y a partir de ahí cada incremento en el eje x añade el apoyo de 1 hilo de CPU. En las gráficas con ejecución homogénea en CPU, mantenemos la coherencia añadiendo hilos de CPU a partir de la etiqueta "2".

4.2. Ejecución en solo CPU

Para comenzar nuestra comparativa, partiremos del pipeline original ejecutándose en un único núcleo de CPU. El tiempo obtenido es nuestro *baseline* o el tiempo que debemos mejorar en el resto de configuraciones del *scheduler*. Este tiempo es 12.962,7 milisegundos, un tiempo que tomaremos como referencia para el resto de las implementaciones utilizando un único núcleo de procesamiento.

Pipeline	Flow Graph	oneAPI Graph	oneAPI Pipeline
12.962,7	12.806	12.886	12.929,4

Tabla 2: Tabla comparativa sobre el mínimo tiempo (milisegundos) de ejecución empleando una unidad de proceso.

Como se aprecia en la Tabla 2 empleando una única unidad de proceso, a priori, los cuatro planificadores se comportan de manera análoga. Al añadir más núcleos debemos medir menores tiempos de ejecución. Para ello, configuraremos las distintas implementaciones con un número creciente de hilos de CPU, hasta cubrir los ocho hilos de ejecución que como máximo soporta nuestra plataforma.

La Tabla 3 muestra la escalabilidad de la ejecución cuando aumentamos el número de hilos de ejecución. Hay que destacar que a partir de 5 hilos las reducciones en tiempo de ejecución dejan de escalar ya que a partir de ese momento estamos usando los cores lógicos que proporciona la tecnología Intel "Hyperthreading".

Aunque levemente, también se aprecia una mejora de rendimiento del *pipeline* clásico basa-

Número de cores	Pipeline	Flow Graph	oneAPI Graph	oneAPI Pipeline
1	12.962,7	12.806	12.886	12.929,4
2	6.828,51	7.037,04	7.069,68	6.944,16
3	4.742,25	4.974,42	4.965,95	4.826,35
4	3.730,15	3.866,06	3.877,52	3.762,99
5	3.390,87	3.517,46	3.521,86	3.370,96
6	3.107,36	3.283,39	3.296,42	3.088,1
7	2.898,41	3.053,14	3.068,17	2.855,26
8	2.703,51	2.940,53	2.938,19	2.726,62

Tabla 3: Tabla comparativa sobre el mínimo tiempo (milisegundos) de ejecución empleado.

do en *tbb::parallel_for* respecto de las otras tres implementaciones. Esta característica también la hereda la reimplementación en oneAPI Pipeline, que también recorta décimas a los planificadores basados en el modelo *flow_graph*.

Hacemos notar también que en realidad las principales diferencias entre los cuatro planificadores aquí expuestos se basan en como reparten la carga de trabajo entre la CPU la GPU, ya que es donde la tecnología subyacente a cada uno cobra importancia. Procederemos por tanto a mostrar y comentar los resultados de las ejecuciones solo-GPU y heterogéneas a continuación.

4.3. Ejecución solo en GPU

Como en el caso anterior, la marca de referencia es la que arroja el pipeline original, cuya mejor marca empleando solo el coprocesador gráfico es de 2.094,98 milisegundos, una marca que supera el rendimiento de incluso ocho hebras de procesamiento de CPU trabajando de forma concurrente. A continuación comparamos todas las alternativas en la Tabla 4.

Pipeline	Flow Graph	oneAPI Graph	oneAPI Pipeline
2.094,98	1.911,6	1.969,32	1.899,46

Tabla 4: Tabla comparativa sobre el mínimo tiempo (milisegundos) de ejecución empleando la GPU.

De forma análoga al caso anterior, a priori, estos rendimientos solo-GPU no son conclu-

yentes para indicarnos como un planificador optimiza la ejecución del kernel en GPU frente a los demás. Es aquí donde el aparente cuádruple empate se podría inclinar por un planificador u otro, en el caso heterogéneo.

4.4. Ejecución heterogénea en CPU y GPU

En el caso heterogéneo, nuestros planificadores deberán ejecutar código del usuario tanto en CPU como en GPU. Cada planificador emplea una metodología por la que la GPU recibe el *kernel* a ejecutar, así como el consiguiente esfuerzo por parte del usuario del *template*. Desarrollaremos este último aspecto relacionado con la programabilidad más adelante.

Número de cores	Pipeline	Flow Graph	oneAPI Graph	oneAPI Pipeline
1	1.792,38	1.746,04	1.705,43	1.725,45
2	2.106,03	2.048,18	1.739,69	2.058,58
3	1.855,18	2.051,83	1.946,54	1.972,96
4	1.983,72	1.603,59	1.969,35	1.897,2
5	1.858,11	1.883,63	1.882,55	1.993,28
6	1.843,66	1.620,09	1.753,63	1.870,53
7	1.708,29	1.609,92	1.764,32	1.815,46
8	1.888,14	2.941,32	1.616,21	1.815,99

Tabla 5: Tabla comparativa sobre el mínimo tiempo (milisegundos) de ejecución empleado.

Tal y como se aprecia en la Tabla 5, las soluciones basadas en oneAPI son ligeramente mejores en la mayor parte de los experimentos realizados en el caso heterogéneo, destacando el modelo basado en flow_graph, probablemente debido a la mejora realizada en la optimización del grafo frente al *parallel_pipeline*. También cabe destacar como en Flow Graph, el *opencl_node* de oneTBB maneja las hebras internamente ya que el kernel de GPU se ejecuta desde una hebra adicional. En este caso y a juzgar por el incremento observado en el planificador Flow Graph, el hilo que alimenta la GPU se mantiene en una "espera activa" hasta la finalización de la ejecución del acelerador, "entorpeciendo" por "oversubscription" la labor de las otras ocho (recordemos que nuestro máximo número de hebras concurrentes es ocho). Este problema no aparece en las implementaciones oneAPI, ya que se evita al utilizar el paradigma de fuente

única y USM, pues el grafo de oneAPI sustituye el mencionado nodo openCL por un nodo función que realiza el *submit* del kernel SYCL directamente a la cola del dispositivo de GPU.

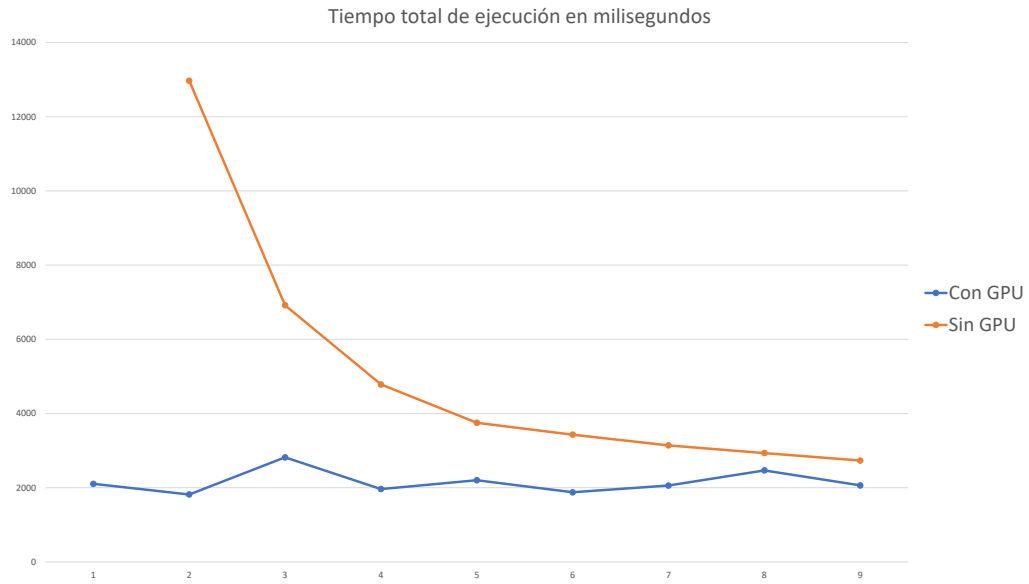

Figura 43: Gráfico de la mediana del tiempo de ejecución del pipeline.

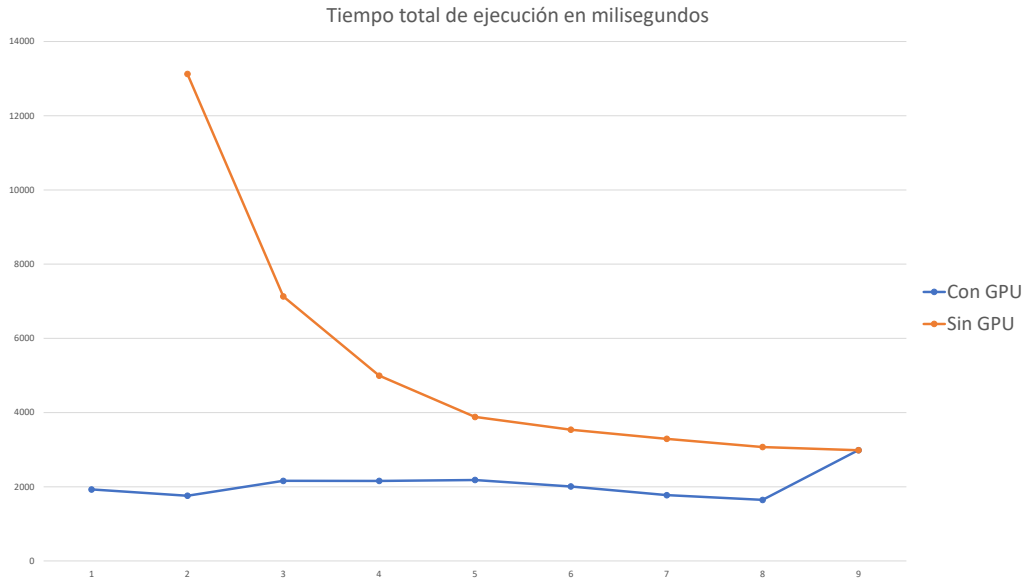

Figura 44: Gráfico de la mediana del tiempo de ejecución del Flow Graph.

En los siguientes gráficos mostrados en las Figuras 43, 44, 45 y 46, podemos ver los tiempos de ejecución en milisegundos del *benchmark* para cada planificador variando el número de unidades de proceso y si participa o no la GPU.

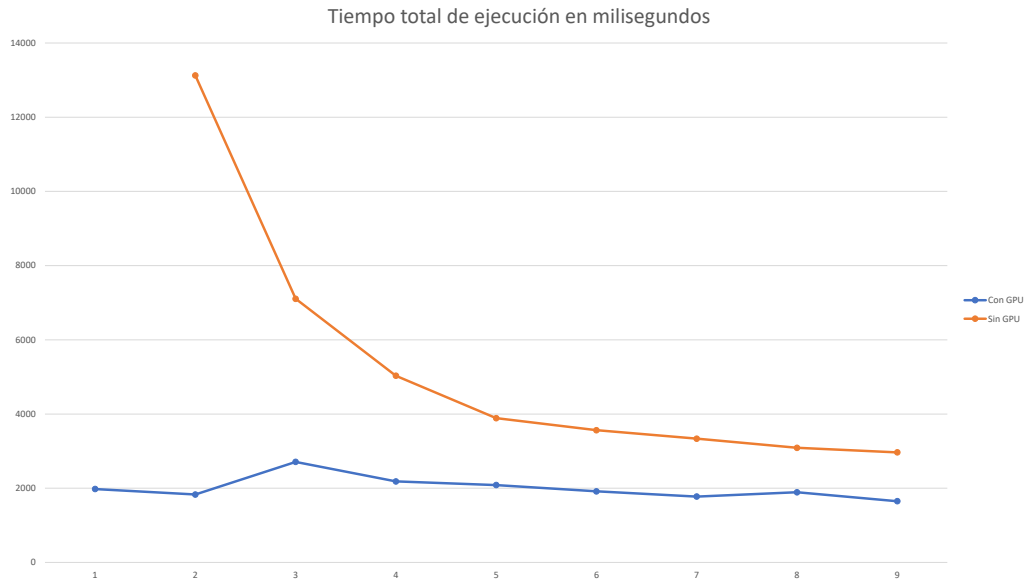

Figura 45: Gráfico de la mediana del tiempo de ejecución del oneAPI Graph.

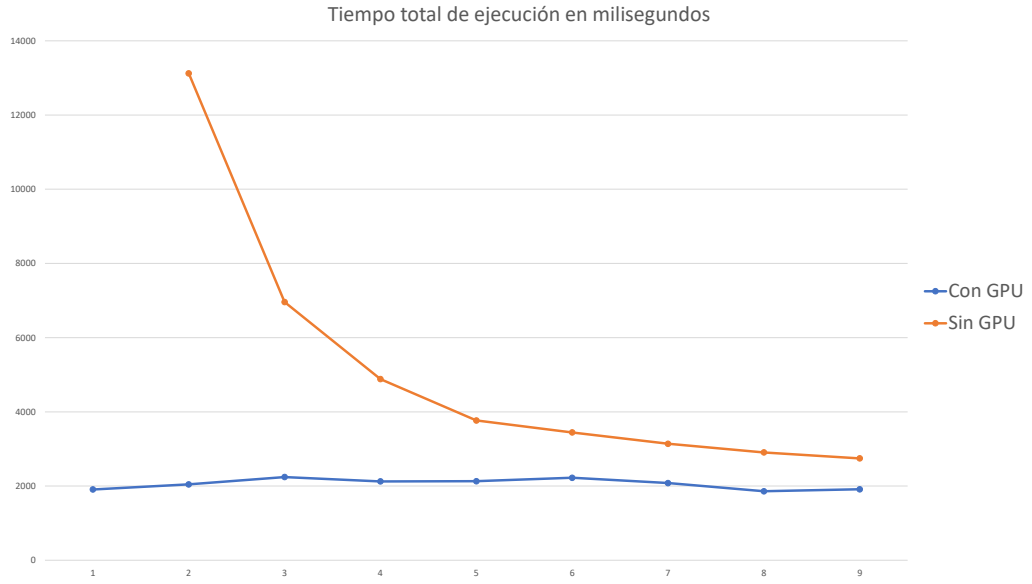

Figura 46: Gráfico de la mediana del tiempo de ejecución del oneAPI pipeline.

4.5. Diferencias clave entre Flow Graph y oneAPI

Cabe destacar, a pesar de la enorme similitud de los planificadores basados en *flow_graph*, el modelo basado en opencl_node tiende a empeorar su rendimiento cuando tratamos de ejecutar tantos *tokens* de CPU como unidades lógicas de procesamiento tenemos, esto es así porque en la implementación subyacente al opencl_node debe existir un mecanismo de espera activa

o *polling* que obliga al hilo asignado a la GPU a utilizar recursos de procesamiento, entorpeciendo a los otros ocho hilos que no deberían sufrir cambios de contexto que sí están dándose. oneAPI parece haber solventado este inconveniente mediante una espera pasiva que bloquea la hebra asociada a la ejecución en GPU hasta la finalización del kernel.

Para demostrar esta hipótesis, nos basamos en los resultados experimentales, donde hemos medido cuántas veces se solicita al algoritmo Logfit un *chunk* de GPU. En los casos comunes, entre cero y siete *tokens*/hilos de CPU no apreciamos diferencia (ver Figura47), pero cuando llegamos al punto de saturación, la hebra de la GPU, deja de solicitar *tokens* al planificador en Flow Graph mientras que sigue funcionando con normalidad para oneAPI. Pero ¿cómo es esto posible si ambos utilizan el mismo algoritmo y gran parte del grafo? Esta anomalía en el comportamiento del nodo de GPU se debe a como se realiza la espera activa en *opencl_node* y como es bloqueante/pasiva para DPC++. La espera activa de un hilo que constantemente está haciendo "spinning" para preguntar si ha terminado de ejecutarse el kernel, no solo reduce la eficiencia de la ejecución, sino que además, por su frecuencia, prácticamente ocupa un core, que aunque sea lógico se debe compartir con el octavo hilo de CPU. Esto hace que en términos efectivos, esté ejecutándose con un hilo menos de CPU y con una GPU lastrada por la ineficiencia con la que recibe carga de trabajo. Debemos recordar que Logfit explora el espacio de iteraciones asignando tamaños más grandes cuanto más rápido es el cómputo. Si este cómputo se ralentiza por los constantes cambios de contexto, aumenta la probabilidad de que las hebras de CPU acaben con el trabajo antes incluso de que la GPU concluya su iteración primera.

4.6. Rendimientos entre los distintos *schedulers*

La Figura 47 muestra el número de peticiones de *chunk* que realiza la GPU y que es equivalente al número de veces que se invoca el kernel de GPU. En principio un menor número de invocaciones implica un menor overhead de lanzamiento del kernel, pero también repercute en un peor balanceo con los 8 hilos de CPU. La Figura 48 precisamente muestra el tamaño medio de los *chunks* que está inversamente relacionado con el número de *chunks* que la GPU solicita. Por tanto un mayor tamaño medio implica una mayor probabilidad de que el trabajo no se reparta bien entre los distintos dispositivos, pero a cambio también indica un menor número de invocaciones al planificador. De estas figuras podemos extraer varias conclusiones:

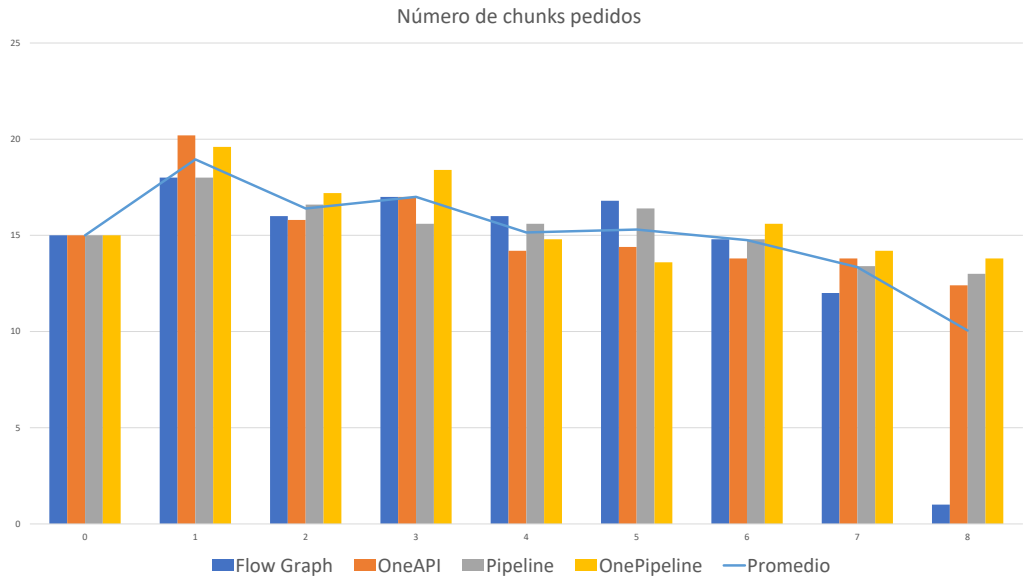

Figura 47: Histograma de número de solicitudes de *chunks* de GPU.

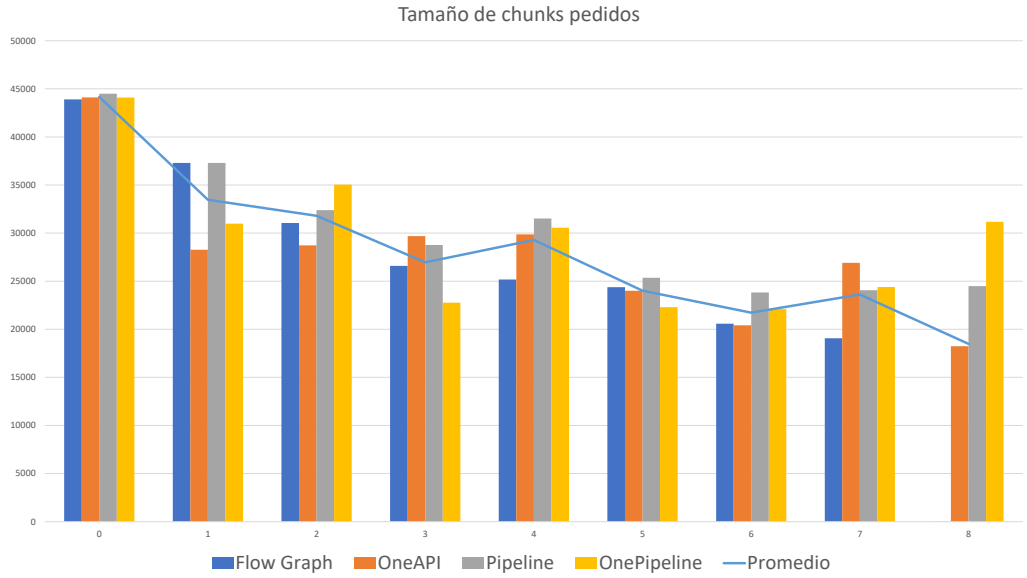

Figura 48: Histograma de tamaño medio de *chunks* de GPU entregados.

- En el uso de la GPU exclusivamente, los cuatro planificadores rinden de forma similar, pues están basados en el mismo algoritmo y su única tarea de "planificación.es evitar que la GPU se quede ociosa, con lo que cabe esperar un rendimiento similar y un tamaño de *chunk* promedio aproximado.

- En nuestro computador y con las limitaciones actuales del propio benchmark, no hemos

conseguido saturar la GPU. Es decir, el planificador de LogFit no encuentra un número suficiente de iteraciones para ocupar todas las unidades funcionales de la GPU y por tanto no encuentra el codo de la función logarítmica en la que el throughput deja de crecer rápidamente al aumentar el tamaño del *chunk*. Al tratarse de una GPU con más capacidad de computo con con veinticuatro unidades de ejecución, las 100.000 iteraciones del bucle paralelo se quedan escasas para poder repartirlas entre la CPU y GPU. Cuando hemos intentado aumentar el tamaño del problema, nos hemos encontrado con un problema de limitación en los buffer de memorias que la CPU y la GPU pueden compartir. Es por ello que el chunk máximo en todos los planificadores para todas las opciones posibles de número de CPU y GPU está cercano a 100.000, que es el tamaño total del número de iteraciones.

- No nos sorprende como sin ser el planificador más veloz, el basado en pipeline y sobre oneAPI es el que más *chunks* consume, y de mayor tamaño, por lo que se esperaría que fuese el más rápido en concluir. Esto no es así, por la sobrecarga asociada a la gestión del pipeline y los filtros que lo componen, frente a la simplicidad del modelo de grafos de oneTBB.

4.7. Complejidad ciclomática y esfuerzo de programación

Aunque el rendimiento de los planificadores pueda estar algo reñido y no dictaminar un claro resultado a favor o en contra, si podría ser decisivo para nuestro *template* que sea o no usado por la comunidad a la que va dirigido. Esto es, debe ser fácil de usar y en menor importancia, pero no carente de ella, fácil de extender y probar. Para ello, nuestra solución es modular como bien hemos comentado, tal que una librería única nos proporciona cuatro planificadores y, al menos un algoritmo de particionado del espacio de iteraciones, en nuestro caso, Logfit. Añadir o modificar el algoritmo de particionado es además hasta cierto punto inmediato.

Con esto en mente desde el primer momento, surge Flow Graph, para simplificar los aspectos a los que el pipeline no llega, la ejecución de openCL. Por su parte, oneAPI ha encapsulado todo bajo DPC++, con el paradigma de fuente única para código anfitrión y *kernel* de aceleradores.

Utilizamos como métricas de programabilidad la Complejidad Ciclomática y el Programming Effort [4]. La complejidad ciclomática representa el número de caminos posibles en el grafo de ejecución de un código y coincide con el número de predicados más 1. El Programming Effort es una función del número de operandos únicos y totales así como de operadores únicos y totales. Es deseable que estas magnitudes sea lo más pequeñas posibles ya que eso indica una mayor facilidad de programación. Estas métricas se han medido para los códigos fuente de las distintas implementaciones del *benchmark* que aquí exponemos. Las medidas se han realizado con el software C3MS [4] y procederemos a explicar los resultados.

C3MS	Esfuerzo de programación	Complejidad ciclomática
Pipeline	3.647.966	72
Flow Graph	3.498.900	64
oneAPI Graph	3.201.406	57
oneAPI Pipeline	3.592.930	56
Pipeline sin modificar	5.704.702	117

Tabla 6: Tabla comparativa sobre la complejidad ciclomática y el esfuerzo de programación necesarios para usar los *templates*.

Como apreciamos en la Tabla 6, el esfuerzo de programación necesario para utilizar el *pipeline* frente a *flow_graph*, es unas 150.000 unidades mayor en cuanto a esfuerzo de programación. Por otro lado un usuario de oneAPI, tendría un esfuerzo significativamente menor y produciría un código de menor complejidad y mas simple. Estas reducciones tan drásticas en la complejidad ciclomática se deben a la no necesidad de escribir código en OpenCL, así como a no gestionar sus estructuras de datos. Si comparamos además con el Heterogeneous Parallel_For original (implementación de partida) antes de realizar ninguna modificación, vemos como hemos reducido sustancialmente tanto el esfuerzo de programación como la complejidad ciclomática a nivel usuario.

También hemos medido la propia librería desarrollada 7 y comparamos también con la implementación original de partida. Además hemos dividido la comparativa entre la complejidad de la parte que afecta al usuario y la complejidad total que afectaría a la extensión y actualización de la librería. De esa forma contemplamos también el coste de mantenimiento y actualizaciones de la misma.

C3MS	Esfuerzo de programación	Complejidad ciclomática
librería (include)	2.985.369	120
librería (include y lib)	4.225.463	162
librería sin modificar	10.948.225	262

Tabla 7: Complejidad ciclomática y esfuerzo de programación de la librería.

5

Conclusiones y Líneas Futuras

5.1. Conclusiones sobre el proyecto

Con la incorporación de las GPUs a los dispositivos de uso cotidiano comenzó una nueva época de computación heterogénea. Además, el grado de heterogeneidad ha ido aumentado ya que muchos de estos dispositivos actuales incluyen también aceleradores de redes neuronales o FPGAs. Nuestro proyecto ha partido de la base implementada por el Departamento de Arquitectura de Computadores de la Universidad de Málaga, el *Heterogeneous Parallel_For* que aporta la primera capa de abstracción sobre el cómputo en GPU, y ha sido adaptado para poder ser utilizado con la versión actual de la librería oneAPI, que a su vez contiene oneTBB, base del *parallel_pipeline* y *opencl_node*.

Dicha base se sigue utilizando actualmente, con modificaciones por supuesto, para proyectos actuales sobre investigación en el ámbito de la computación heterogénea. Por la propia naturaleza de nuestro problema, el planificador basado en *flow_graph* requería de una implementación desde cero pero que utilizara el algoritmo LogFit, fuertemente acoplado al pipeline original.

Esta fue la principal motivación para desarrollar la librería de forma modular. La segunda ha sido que se pueda extender por parte de futuros usuarios, profesores, alumnos e investigadores y que sea fácilmente incluida en aplicaciones reales. Ante esta situación, presentamos nuestro proyecto con las siguientes conclusiones:

- En la elaboración del Flow Graph, hemos reestructurado como interacciona el planificador con el algoritmo, LogFit en nuestro caso. Esta reestructuración y adaptación a un C++ moderno hace que sea más mantenible y extensible.

- oneAPI y DPC++ permiten una intercomunicación prácticamente inmediata con el uso del compilador propio y las librerías de SYCL de forma que su inclusión en la librería resultó sencilla, y por tanto, su uso en programas de usuario que utilicen el *template*.

- Al comparar las implementaciones, hemos cuantificado y demostrado como se comporta cada uno, así como sus ventajas e inconvenientes. Si bien el pipeline y Flow Graph no requieren de un compilador especial, deben de ser utilizados por usuarios acostumbrados a codificar en OpenCL, así como a gestionar sus estructuras de datos. Por otra parte, el modelo de oneAPI, es tan agnóstico del código a ejecutar en el kernel que permite reutilizar el mismo `TBody` tanto para un pipeline como para un grafo. Es aquí donde mejor se aprecian las ventajas de las implementaciones basadas en oneTBB, en su interoperabilidad aún requiriendo de un compilador que genere binarios tanto para CPU como para aceleradores.

Por último destacamos que todo el código está disponible en el repositorio público LogFit[7], donde se aceptan *pull-requests* y se promueve su uso.

5.2. Lineas futuras

La computación heterogénea lleva entre nosotros mucho tiempo y en lugar de irse acotando a un modelo unificado, se observa una tendencia al uso de hardware especifico para cada problema específico. Es aquí donde soluciones como la propuesta a lo largo de este TFG nos simplificarán el trabajo a los programadores de hoy y de mañana.

5.2.1. Modularización e integración

La estructura del *template* aquí desarrollada aporta la ventaja de poder añadir planificadores, algoritmos de planificación y problemas a resolver a voluntad y necesidad del usuario. Si en el futuro se utiliza un algoritmo de planificación distinto o un problema con otra distribución del espacio de iteraciones, se podrá realizar un análisis cualitativo y cuantitativo sobre sus ventajas e inconvenientes frente al estado del arte.

5.2.2. Nuevos dispositivos

Las FPGAs ya son una realidad que en cuestión de tiempo llegarán a la electrónica de consumo, como hizo anteriormente la GPU y como recientemente ha ocurrido con las NPUs[26] en dispositivos móviles. Todos estos aceleradores se irán soportando paulatinamente por las librerías como oneAPI y nuestro *template* ofrecerá acceso a ellos de forma transparente al desarrollador de aplicaciones heterogéneas.

5.2.3. Futuros estándares de C++

Como mencionamos anteriormente en el capítulo segundo, oneAPI se sirve de DPC++ y del estándar SYCL y su paradigma de fuente única y memoria unificada compartida. Estas dos tecnologías sobre C++, forman un súper-conjunto del lenguaje que pretende extenderlo y aspira a formar parte del estándar ISO de C++.

[26]Unidad de procesamiento neuronal, o dispositivos aceleradores optimizados para el entrenamiento e inferencia de redes neuronales.

Referencias

[1] *C++ enable_if*. https://en.cppreference.com/w/cpp/types/enable_if. 2020.

[2] *Diagramas de clases online*. https://yuml.me/. 2021.

[3] *DPC++ SDK*. https://spec.oneapi.com/versions/latest/elements/dpcpp/source/index.html. Sep. de 2020.

[4] Carlos H. González y Basilio B. Fraguela. «A Generic Algorithm Template for Divide-and-Conquer in Multicore Systems». En: *12th IEEE International Conference on High Performance Computing and Communications*. HPCC 2010 (Melbourne, Australia). Sep. de 2010, págs. 79-88.

[5] David Kaeli y col. *Heterogeneous Computing with OpenCL 2.0*. 1.ª ed. Estados Unidos: Morgan Kaufmann, 2015. ISBN: 9780128014141.

[6] Ray Lischner. *Exploring C++ 11*. 2.ª ed. Estados Unidos: Apress, 2013. ISBN: 9781430261940.

[7] *LogFit Github*. https://github.com/TheUnixRoot/Logfit. 2021.

[8] Angeles Navarro y col. «Heterogeneous parallel_for Template for CPU–GPU Chips». En: *International Journal of Parallel Programming* (ene. de 2018).

[9] *OneAPI*. https://software.intel.com/content/www/us/en/develop/tools/oneapi.html. 2021.

[10] *OneTBB GitHub*. https://github.com/01org/tbb/tree/tbb_2020. 2020.

[11] *oneTBB SDK*. https://spec.oneapi.com/versions/latest/elements/oneTBB/source/nested-index.html. Sep. de 2020.

[12] *OpenCL buffers*. https://downloads.ti.com/mctools/esd/docs/opencl/memory/buffers.html. Dic. de 2019.

[13] *OpenCL node*. https://software.intel.com/en-us/blogs/2015/12/09/opencl-node-overview. 2015.

[14] James Reinders y col. *Data Parallel C++. Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL*. 1.ª ed. Berkeley, CA: Apress, 2021. ISBN: 9781484255735.

[15] *SYCL API*. https://intel.github.io/llvm-docs/doxygen/group__sycl__api.html. 2020.

[16] *SYCL especificación*. https://www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf. Abr. de 2020.

[17] Michael Voss, Rafael Asenjo y James Reinders. *Pro TBB. C++ Parallel Programming with Threading Building Blocks*. 1.ª ed. Berkeley, CA: Apress, 2019. ISBN: 9781484243978.

UNIVERSIDAD DE MÁLAGA | uma.es

E.T.S de Ingeniería Informática
Bulevar Louis Pasteur, 35
Campus de Teatinos
29071 Málaga

E.T.S. DE INGENIERÍA INFORMÁTICA