Universidad Zaragoza

Trabajo Fin de Grado

Evaluación del modelo de programación oneAPI para ejecución heterogénea con CPU y FPGA

Autor

Raúl Herguido Sevil

Directores

Darío Suárez Gracia

Rubén Gran Tejero

ESCUELA DE INGENIERÍA Y ARQUITECTURA
2021

TRABAJOS DE FIN DE GRADO / FIN DE MÁSTER

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD

(Este documento debe entregarse en la Secretaría de la EINA, dentro del plazo de depósito del TFG/TFM para su evaluación).

D./Dª. Raúl Herguido Sevil ,en aplicación de lo dispuesto en el art. 14 (Derechos de autor) del Acuerdo de 11 de septiembre de 2014, del Consejo de Gobierno, por el que se aprueba el Reglamento de los TFG y TFM de la Universidad de Zaragoza, Declaro que el presente Trabajo de Fin de (Grado/Máster) Grado (Título del Trabajo)

Evaluación del modelo de programación oneAPI para ejecución heterogénea con CPU y FPGA

es de mi autoría y es original, no habiéndose utilizado fuente sin ser citada debidamente.

Zaragoza, 22 de Junio de 2021

Fdo: Raúl Herguido Sevil

AGRADECIMIENTOS

Quiero agradecerle la oportunidad de realizar este TFG y el trabajo semana a semana a mis dos directores, Darío y Rubén, y a Angélica Dávila, que no siendo directora ha invertido tanto tiempo como cualquiera. También agradecer a Raúl Nozal y José Luis Bosque de la Universidad de Cantabria por proporcionarnos su código fuente para realizar el TFG.

Título del resumen

RESUMEN

La demanda de potencia de cómputo continúa aumentando a la vez que se exige también un mejor consumo energético. Desgraciadamente, en la actualidad, los procesadores de propósito general no satisfacen ambos requerimientos. Una solución empleada es el uso de dispositivos de dominio específico, que ofrecen grandes prestaciones para un conjunto reducido de tareas. Al juntar múltiples dispositivos aceleradores se crean sistemas heterogéneos, que mejoran la eficiencia a costa de dificultar la programación. Para abstraer al programador de las diferencias entre dispositivos han surgido varios modelos de programación, siendo oneAPI uno de los más recientes.

Este trabajo evalúa oneAPI adaptando un *runtime* heterogéneo para ejecutar el patrón *parallel_for* sobre CPU-FPGA. El soporte de oneAPI no está maduro, y ha requerido un gran esfuerzo entender su funcionamiento y las diferentes opciones que ofrece. Para evaluar el rendimiento y la programabilidad, se ha comparado una nueva implementación en C++ de múltiples *benchmarks* frente a una implementación oneAPI ejecutada en CPU y en FPGA.

Se ha mejorado el código oneAPI de los *benchmarks* (proveniente de la colaboración con la Universidad de Cantabria y optimizado para GPU) para su ejecución en FPGA, obteniendo *speedups* entre 1.43 y 116.7 frente a la versión no optimizada. También se ha optimizado la ejecución en CPU, obteniendo entre un 3.55 y 6.28 de mejora frente a la versión sin optimizar.

Con fin de aprovechar el sistema al completo, se ha evaluado el rendimiento de la ejecución cooperativa entre CPU (que ejecuta C++ nativo paralelizado) y FPGA (que ejecuta oneAPI optimizado para FPGA), empleando tres planificadores de ejecución diferentes (estático, dinámico y hguided). Los resultados demuestran hasta 1.26 veces mejor rendimiento (sobre un máximo de 1.29) que la mejor versión no cooperativa, lo que supone una eficiencia del 98 %.

Se han alcanzado todos los objetivos iniciales del Trabajo Fin de Grado, a pesar de las dificultades técnicas que Intel oneAPI presenta debido a su reciente lanzamiento.

Índice

1. Introducción **1**

1.1. Motivación 1

1.2. Objetivos 3

1.3. Alcance 3

1.4. Estructura del documento 4

2. Estado del arte **6**

2.1. Sistemas Heterogéneos 6

2.2. *Hardware* en sistemas heterogéneos 6

2.2.1. CPU 6

2.2.2. FPGA 8

2.3. *Software* en sistemas heterogéneos 9

2.3.1. OpenCL 9

2.3.2. Intel OneAPI 10

3. Metodología **12**

3.1. Plataforma experimental 12

3.2. *Benchmarks* utilizados 12

3.3. Medición del tiempo de ejecución 14

3.4. Flujo de trabajo con la FPGA 14

4. Coejecución con Intel oneAPI **16**

4.1. Trabajo previo 16

4.1.1. Planificadores 16

4.1.2. Paralelismo en la coejecución 17

4.2. Limitación de oneAPI con la FPGA 17

4.3. Portabilidad entre Intel oneAPI y C++ 18

4.3.1. *Benchmarks* 18

4.3.2. Planificadores 19

4.4. Portabilidad para la FPGA 20

4.4.1. Matadd 20
4.4.2. Matmul, Rap y Nbody 20
4.4.3. Gaussian 20

5. Resultados experimentales **21**
5.1. Rendimiento por dispositivo 21
5.2. Optimización 23
5.2.1. Paralelización de CPU++ 23
5.2.2. Optimización FPGA 23
5.2.3. Comparación entre optimizaciones 27
5.3. Ejecución cooperativa 28
5.3.1. Planificador estático 28
5.3.2. Planificador dinámico 30
5.3.3. Planificador hguided 31
5.3.4. Comparativa de planificadores 32
5.4. Resumen de la experimentación 33

6. Conclusiones y Trabajo Futuro **34**

7. Bibliografía **36**

Lista de Figuras **38**

Lista de Tablas **40**

Anexos **41**

A. Cronología **42**

B. Experimentos y Figuras adicionales **45**
B.1. Estudio de la sobrecarga oneAPI 45
B.2. Efectividad de la paralelización CPU++ 46
B.3. Paralelización oneAPI y nativa 47
B.4. Gráficas de Matmul y Rap para los planificadores dinámico y hguided . 48
B.5. Exploración exhaustiva del parámetro K de hguided 49

Capítulo 1

Introducción

1.1. Motivación

Es extremadamente complicado hacer más rápidos a los procesadores de propósito general. Previamente, la tecnología mejoraba a tal velocidad que se conseguía que el aumento en potencia de cómputo fuese superior al aumento de la demanda, pero ya no es así, tal y como muestra la Figura 1.1.

Como estimó Moore en 1965, el número de transistores por chip se doblaba cada dos años, reduciéndose el tamaño de los mismos para no aumentar el área total del chip [1]. Al reducir el tamaño, la distancia entre transistores y el voltaje, se reducía la energía que consumían, lo que permitía aumentar la frecuencia del procesador [2]. Estos dos factores, junto con los avances micro-arquitectónicos protagonizaron las increíbles mejoras en procesadores desde su creación.

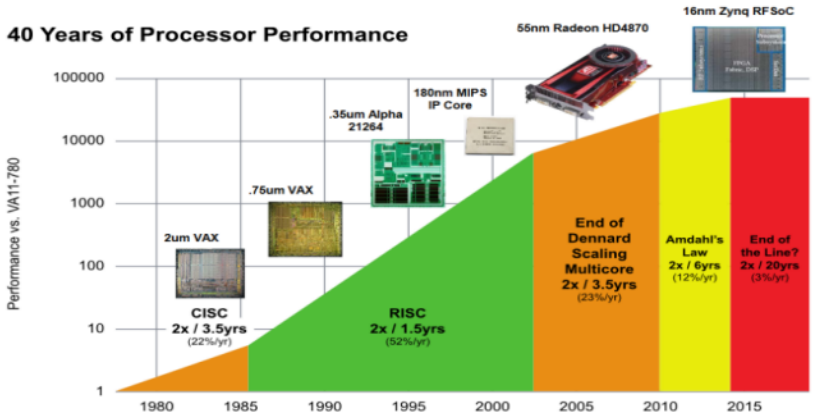


Figura 1.1: Evolución del rendimiento de los procesadores desde los años 80 a la actualidad

Fuente: *Computer Architecture: A Quantitative Aproach*

Eventualmente, pese a que el número de transistores empleados por chip seguía aumentando, la tasa de aumento era mucho menor ya que el tamaño de los transistores

llegaba a límites físicos y la fabricación era más difícil y cara [3]. Además, a tamaños de transistor tan pequeños es muy difícil reducir el voltaje, y la corriente de fuga acaba aumentando la energía total requerida, por lo que ya no se aplica el escalado de Dennard y es demasiado caro aumentar la frecuencia o directamente imposible por no poder disipar el calor. En lugar de diseñar procesadores más potentes, se recurrió a explotar el paralelismo, aumentando el número de unidades de cómputo y haciéndolas trabajar de manera cooperativa, pero llega cierto punto en el que acelerar la parte paralelizable ya no proporciona mejoras significativas [4]. Si, por ejemplo, un 90 % de una aplicación es paralelizable, el *speedup* máximo es 10 con independencia del número de procesadores que se empleen en la paralelización.

En la actualidad la mejora en capacidad de cómputo es lenta e insuficiente y el consumo de energía y la disipación de potencia se ha convertido en un problema todavía mayor. Por tanto, se han de buscar nuevas soluciones [5]. Una de las estrategias que se está empleando para atacar ambos problemas es el uso de dispositivos de dominio específico, en lugar de propósito general. Al estar especializados en un conjunto más reducido de tareas, son capaces de ofrecer altas prestaciones tanto en tiempo de ejecución como en consumo energético. En la Figura 1.2 se comparan algunos de los dispositivos más comunes en base a su flexibilidad y rendimiento.

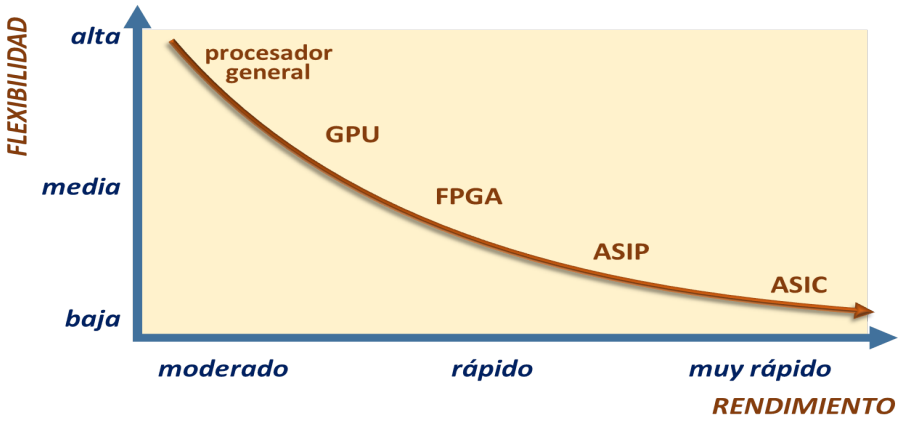


Figura 1.2: Flexibilidad frente a rendimiento de las principales tecnologías

Fuente: Adaptada de *Hardware Design of Embedded Systems for Security Applications*

Estos dispositivos están dando muy buenos resultados y se están construyendo sistemas heterogéneos que los incluyen (desde sistemas HPC hasta teléfonos móviles de uso común), pero traen consigo un nivel de complejidad extra [6]. Cada dispositivo es (o puede ser) diferente: arquitectura, memoria, frecuencia de reloj, ..., y por lo tanto se trabaja con él de manera diferente: lenguaje, bibliotecas, herramientas,.... Justamente es esta diversidad la que da el nombre de *sistemas heterogéneos* a estos entornos.

Lo que el programador querría es poder diseñar un único código fuente y que fuese el sistema heterogéneo el encargado de compilar un binario específico para cada dispositivo y repartir eficientemente el trabajo entre ellos. Y esto es precisamente lo que pretende conseguir Intel con su nuevo modelo de programación oneAPI [7], un modelo abierto, gratuito y estandarizado que simplifica el uso de sistemas heterogéneos. Conseguir abstraer al programador de los dispositivos subyacentes no es sencillo, pero todavía menos lo es maximizar las prestaciones. El rendimiento puede desplomarse por diversas razones: mala comunicación de datos entre dispositivos, mala localidad de los accesos a memoria, mal uso de los recursos del dispositivo, ...

Si bien oneAPI es un claro avance en la programabilidad de sistemas heterogéneos, carece de una capa de nivel superior que le permita valorar en qué dispositivo/s ejecutar cada carga de trabajo y cómo hacerlos trabajar conjuntamente.

1.2. Objetivos

El objetivo principal es evaluar el rendimiento y programabilidad que ofrece oneAPI para un sistema heterogéneo compuesto por CPU y FPGA al emplear el patrón `parallel_for`, y también valorar la efectividad de la ejecución cooperativa empleando diferentes políticas de balanceo de carga entre estos dos dispositivos.

Persiguiendo este objetivo principal se busca responder a las siguientes cuestiones:

- ¿Cómo de fácil es programar con oneAPI? ¿Cómo de portables son los *kernels*[1]?
- ¿Cuánto rendimiento es capaz de extraer oneAPI manteniendo la portabilidad? ¿Es suficiente como para suponer una mejora frente a la ejecución tradicional en CPU? ¿Se acerca al rendimiento máximo que se puede obtener de cada dispositivo?
- ¿Existen facilidades para la optimización de *kernels* para dispositivos concretos? ¿Cómo de sencillas de usar y efectivas son?
- ¿Podemos hacer cooperar a dos dispositivos para ejecutar los *benchmarks*? ¿Es mejor que ejecutar en uno único?

1.3. Alcance

En este TFG se ha partido de un trabajo previo [8] en el que se desarrolla una herramienta con múltiples *benchmarks* y planificadores de ejecución, funcional para la ejecución heterogénea CPU-GPU con oneAPI. Partiendo de este código base y los objetivos del apartado anterior:

[1] El *Kernel* es la sección de código que se ejecuta en los dispositivos aceleradores

– Se ha extendido la herramienta para poder trabajar con sistemas heterogéneos basados en CPU-FPGA.

– Se ha evaluado la programabilidad y portabilidad de oneAPI al cambiar de GPU a FPGA.

– Se ha comparado el rendimiento del código oneAPI sin optimizar y completamente agnóstico del dispositivo (ejecutado tanto en CPU cómo en FPGA) frente al de una implementación C++ tradicional.

– Se han probado algunas de las opciones de optimización implementables con oneAPI y comparado el rendimiento de la implementación oneAPI original frente a una versión optimizada para la FPGA.

– Se ha mejorado la arquitectura de la herramienta, tanto *benchmarks* como planificadores, añadiendo la posibilidad de ejecutar C++ nativo en lugar de oneAPI[2].

– Se ha paralelizado la implementación nativa añadida a la herramienta, permitiendo comparar esta optimización con las aplicadas al código oneAPI al refinarlo para FPGA. También permite evaluar la efectividad de la paralelización implícita en el código oneAPI inicial.

– Se ha evaluado el rendimiento de la ejecución cooperativa CPU-FPGA usando las versiones optimizadas para cada dispositivo y tres planificadores de ejecución distintos.

El código fuente del coejecutor CPU-FPGA desarrollado se hará público cuando Raúl Nozal publique el de la herramienta original.

1.4. Estructura del documento

En el Capítulo 2 se describe el estado del arte de los sistemas heterogéneos al comienzo de este trabajo. A continuación, en el Capítulo 3, se detalla la metodología empleada al realizar los experimentos y los *benchmarks* utilizados. Una vez definido el entorno del trabajo, en el Capítulo 4, se describen los diferentes pasos realizados desarrollar una nueva herramienta de coejecución heterogénea CPU-FPGA a partir de un trabajo previo que lo hacía para CPU-GPU. Antes de finalizar, en el Capítulo 5

[2]Pese a que el objetivo inicial era realizar ejecución heterogénea empleando dos dispositivos oneAPI, CPU y FPGA, ésta ha tenido que realizarse usando C++ nativo en la CPU y oneAPI en la FPGA ya que en el momento de realización del trabajo no estaba soportada, en la actualidad tampoco, la generación de binarios para múltiples dispositivos oneAPI si uno de ellos es una FPGA.

se exponen los diferentes experimentos realizados y sus resultados. Por último, en el Capítulo 6 se extraen las principales conclusiones del TFG y se proponen líneas de trabajo futuro.

Capítulo 2

Estado del arte

2.1. Sistemas Heterogéneos

Los sistemas heterogéneos son la norma hoy en día. Todo ordenador personal incluye una GPU integrada si no una discreta, los *smartphones* llevan varios procesadores distintos y de propósito específico para ahorrar batería (mi propio móvil, Xiaomi Mi A1, tiene un procesador gráfico Adreno 506, un DSP Hexagon, una CPU ARM para el GPS,...), y, con la reciente popularidad de la inteligencia artificial, existen aceleradores específicos para tareas comunes como el entrenamiento de redes neuronales (Huawei Ascend 910).

2.2. *Hardware* en sistemas heterogéneos

En este apartado se hace una descripción sintética de la arquitectura de los dos dispositivos de cómputo que se utilizan en este trabajo: CPU y FPGA. Comprender su arquitectura es esencial para entender la diferencia de rendimiento y flexibilidad entre dispositivos.

2.2.1. CPU

Las CPUs (*Central Processing Unit*) son componentes *hardware* vitales en prácticamente cualquier dispositivo electrónico. Su tan extendido uso se debe a la alta flexibilidad que aportan, ya que son capaces de implementar cualquier comportamiento a partir de combinaciones de instrucciones simples.

Su organización suele estar basada en una ruta de datos segmentada (simplificada en la Figura 2.1) a través de la que se leen, decodifican y ejecutan las instrucciones, guardando los resultados convenientemente. Otras optimizaciones frecuentes son la ejecución fuera de orden, la predicción de saltos o la ejecución superescalar. Con estas técnicas y muchas otras se consigue explotar al máximo el paralelismo entre instrucciones,

aprovechándose todo el *hardware* posible y otorgando buen rendimiento a las CPUs.

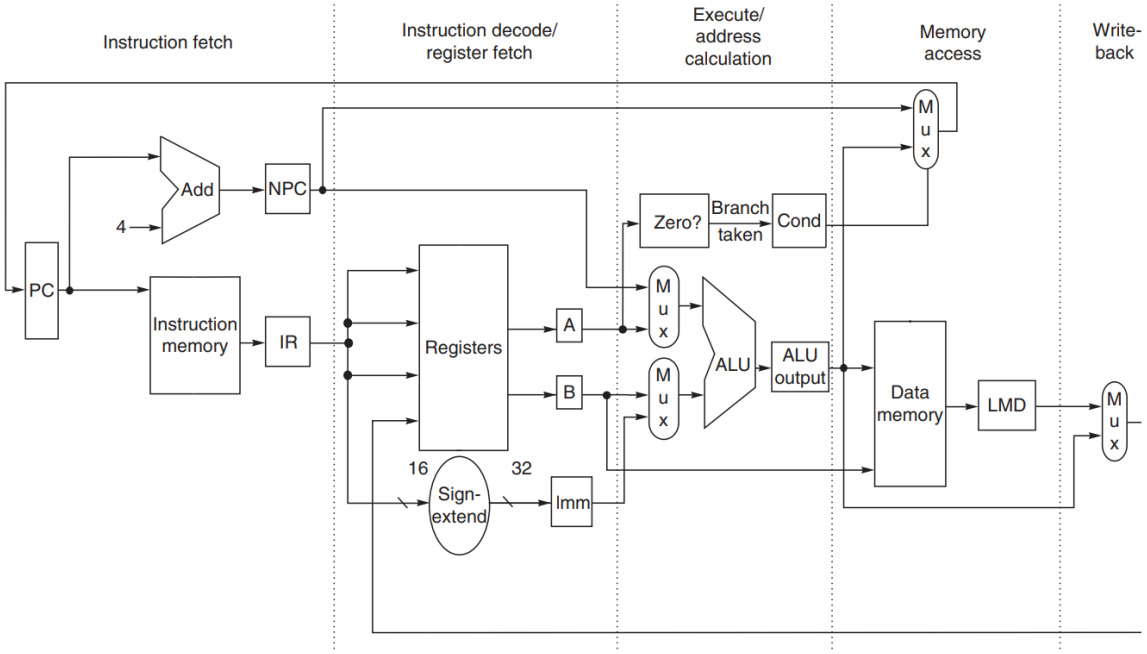


Figura 2.1: Ruta de datos segmentada

Fuente: *Computer Architecture: A Quantitative Aproach*

Cada CPU implementa una ISA (*Instruction Set Architecture*) en la que se detalla todo lo que necesita conocer el programador para generar binarios ejecutables, como por ejemplo podrían ser las instrucciones que la CPU es capaz de entender y ejecutar o los registros disponibles.

Pese a ejecutar gran cantidad de instrucciones por segundo, todas ellas deben atravesar las diferentes etapas de la ruta de datos, e instrucciones dependientes entre sí se bloquean hasta poder ejecutar. Además, al ser un conjunto de instrucciones y un hardware concreto y limitado, no es posible implementar cualquier algoritmo de manera óptima y puede ser necesario ejecutar muchas instrucciones. Por ejemplo, para ejecutar la operación de la ecuación 2.1, además de los accesos a memoria sería necesario realizar dos instrucciones de multiplicación, recorriendo ambas la ruta de datos entera y no pudiendo ejecutarse la segunda hasta que la primera le suministra su resultado, generalmente en el banco de registros. Si esta fuese una operación común en el código que se debe ejecutar y se tuviera que diseñar un ASIC (*Application-Specific Integrated Circuit*), se añadirían dos multiplicadores, el segundo tomando como entrada la salida del primero, para poder realizar la operación entera con un solo recorrido de la ruta de datos (como en la Figura 2.2).

$$\mathbf{a = b * c * c} \tag{2.1}$$

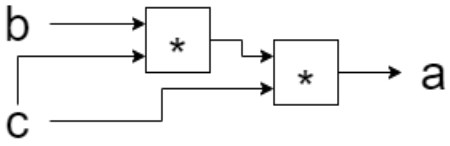


Figura 2.2: Circuito hardware para realizar la multiplicación de la Eq. 2.1 con multiplicadores de dos entradas

2.2.2. FPGA

El principal dispositivo usado en este trabajo es una FPGA. Las FPGAs (*Field-Programmable Gate Arrays*) son dispositivos *hardware* reconfigurables que están en auge ya que su precio ha disminuido y el número de componentes configurables ha aumentado.

Como se representa en la Figura 2.3, las FPGAs están compuestas por una gran cantidad de bloques lógicos que pueden ser programados para implementar desde puertas lógicas a funciones combinacionales complejas. Una jerarquía de interconexión permite configurar qué elementos se conectan entre sí. Incluyen también bloques de entrada/salida y suelen tener bloques fijos dedicados a memoria RAM o DSPs (*Digital Signal Processor*).

Estos dispositivos permiten implementar cualquier diseño *hardware* (si se dispone de los recursos suficientes), siendo idóneos para hacer prototipado. De manera similar a los ASIC, permiten diseñar unidades lógicas y rutas específicas para acelerar operaciones concretas (como la ecuación 2.1).

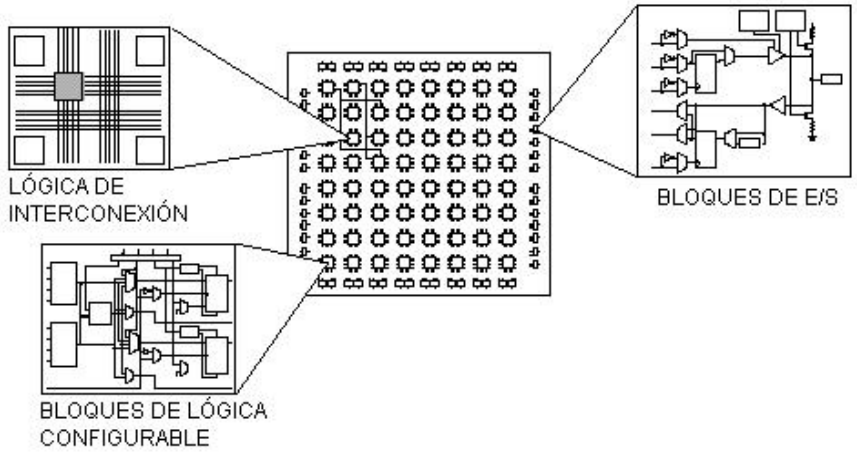


Figura 2.3: Bloques funcionales principales de la FPGA

Fuente: Implementación de una plataforma HW para la evaluación de predictores e saltos sobre arquitectura SPARC v8 [9].

Se encuentran en un punto intermedio entre una CPU y un ASIC (ver Figura 1.2), ofreciendo mejor rendimiento y menor consumo que la CPU y más flexibilidad y

facilidad de diseño que un ASIC. Por contrapartida, las CPU son más flexibles que las FPGAs, y los ASIC ofrecen mejor rendimiento y consumo ya que no tienen el excesivo interconexionado de las FPGAs.

En primera instancia la programación de las FPGAs se realizaba únicamente empleando lenguajes de descripción de *hardware* como VHDL. Actualmente existen herramientas de *High-Level Synthesis* (HLS) que generan el diseño de la FPGA a partir de código de alto nivel (como C++). Pese a que estas herramientas facilitan enormemente la programación para FPGAs, el proceso de generación de una configuración de FPGA válida sigue siendo costoso en tiempo, generalmente en el rango de varias horas, ya que se han de procesar todos los recursos de la FPGA para encontrar una configuración que cumpla las especificaciones y guardarla en un binario (*bitstream*). oneAPI aprovecha las capacidades de las herramientas de HLS para generar los binarios.

2.3. *Software* en sistemas heterogéneos

Una parte fundamental de cualquier sistema computacional son las herramientas software que permiten su programación. Los sistemas heterogéneos no son una excepción. En este apartado se presentan distintas alternativas.

2.3.1. OpenCL

Previamente a oneAPI ya existían otros modelos de programación, como OpenCL [10], en los que se definió que ha de existir un anfitrión (*host*) que gestione la ejecución y descargue las tareas de cómputo (*kernels*) a uno o más dispositivos aceleradores (*devices*), similar al patrón *master-worker*. El *host* debe gestionar los problemas derivados de la heterogeneidad: diferentes arquitecturas, memoria, frecuencia, ...

OpenCL se anunció por primera vez en la conferencia *SIGGRAPH* de 2008, y consigue definir un lenguaje común entre diferentes dispositivos, pero tiene múltiples carencias:

- Bajo nivel de abstracción. Requiere que el programador haga explícito todo el paralelismo de datos existente.
- Los *kernels* se escriben en C99 y deben separarse del código del *host* (a otro fichero).
- El rendimiento de los *kernels* depende mucho de optimizarlos para cada dispositivo.
- La coordinación entre los distintos dispositivos del sistema heterogéneo debe ser explícitamente programada.

2.3.2. Intel OneAPI

Intel oneAPI se define como un modelo de programación que simplifica el uso de CPUs y aceleradores en sistemas heterogéneos. Emplea directivas de C++ moderno para expresar paralelismo, con un lenguaje de programación llamado *Data Parallel C++* (DPC++) [11]. Este lenguaje permite reutilizar código tanto del *host* como de los aceleradores, que pueden estar mezclados en un único fichero. Incluye directivas para expresar dependencias de ejecución y memoria. Permite seleccionar fácilmente el dispositivo dónde ejecutar cada tarea, pudiéndose también usar el *host* como dispositivo.

Es importante mencionar que oneAPI es un modelo de programación muy reciente y no maduro, que fue anunciado por primera vez en la *Intel HPC Developer Conference* en noviembre de 2019.

En la Figura 2.4 se muestra un ejemplo funcional de código oneAPI. En tan solo 21 líneas se incluye la incorporación de la biblioteca con las clases oneAPI (línea 1), la declaración de un *buffer* a partir de datos ya existentes (línea 9), la declaración de una cola (asociada al dispositivo por defecto) y adición de una tarea (línea 11) a la misma, la declaración de un *accessor* al *buffer* para el dispositivo (línea 12), el código a ejecutar en el dispositivo (línea 14) que se ha de ejecutar para todos los "idx" en el rango especificado al llamar a *parallel_for* (línea 13) y por último la declaración de un *accessor* para leer los resultados en el *host*. Se aprecia que la sintaxis es muy similar a C++ moderno.

```
1  #include <CL/sycl.hpp>
2  #include <iostream>
3
4  constexpr int num=16;
5  using namespace sycl;
6
7  int main() {
8    auto r = range{num};
9    buffer<int> a{r};
10
11   queue{}.submit([&](handler& h) {
12     accessor out{a, h};
13     h.parallel_for(r, [=](item<1> idx) {
14       out[idx] = idx;
15     });
16   });
17
18   host_accessor result{a};
19   for (int i=0; i<num; ++i)
20     std::cout << result[i] << "\n";
21 }
```

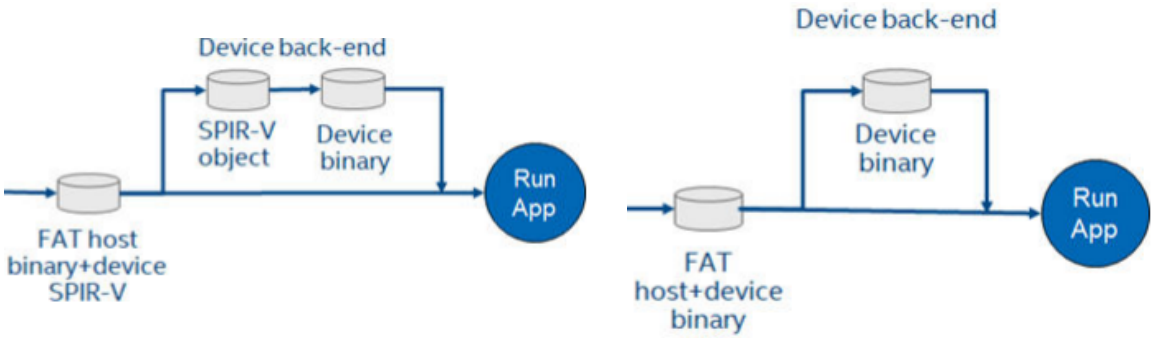
Figura 2.4: Programa oneAPI

Fuente: oneAPI Programming Model

Compilación

La compilación para diferentes dispositivos es uno de los principales desafíos a resolver. En una compilación tradicional se genera código para una única arquitectura objetivo, en la cual puede ejecutarse directamente el binario resultante. Para poder compilar cuando existen múltiples arquitecturas objetivo, oneAPI define dos esquemas

de compilación nuevos: *Just In Time* (JIT) y *Ahead Of Time* (AOT).



(a) Compilación *Just In Time*

(b) Compilación *Ahead Of Time*

Figura 2.5: Opciones de compilación oneAPI

Fuente: oneAPI Programming Model

En la Figura 2.5a se observa que la compilación JIT genera un fichero con una parte directamente ejecutable en el *host* y otra en un lenguaje intermedio (SPIR-V) que es compilado para cada dispositivo en concreto en tiempo de ejecución.

Para las ocasiones donde no es admisible compilar en tiempo de ejecución (compilaciones largas de FPGA o requisitos temporales muy estrictos) existe el modelo AOT (Figura 2.5b), en el que se genera un binario que contiene una parte compilada para el *host* y otra para el/los dispositivos.

Tanto si el fichero resultante de la compilación tiene la parte de los dispositivos en SPIR-V o en binario ejecutable, Intel se refiere a ellos como *FatBinary*.

Capítulo 3

Metodología

En este capítulo se presentan los aspectos relativos al entorno de trabajo, *benchmarks* utilizados y metodología experimental.

3.1. Plataforma experimental

Los experimentos se han llevado a cabo en un sistema heterogéneo compuesto por una CPU Intel Xeon Bronze 3204 (6 núcleos/subprocesos, 1.90GHz, 8.25 MB cache y 98GB RAM) y una FPGA Intel Stratix 10 GX con 2GB de memoria externa DDR4[1]. Se emplea la versión 2021.1.1 de oneAPI, la 20.3.0 de Intel FPGA SDK y la 20.2 de Quartus. Se aplica el máximo nivel de optimización, "$-O3$", en todas las compilaciones.

3.2. *Benchmarks* utilizados

Se han evaluado cinco *benchmarks* que se describen a continuación. Todos ellos incluyen una función de verificación con la que se validan los resultados obtenidos en todos los experimentos. Los tamaños de *benchmark* elegidos en los experimentos corresponden a cinco potencias de 2 contiguas, de modo que la mayor tenga un tiempo de ejecución razonablemente alto pero no excesivo. En algunos casos el tamaño mayor ha sido reducido de 16384 a 13000 por falta de memoria en la FPGA (la memoria requerida se detalla en la Tabla B.1 del Anexo B.1).

Suma de Matrices (Matadd) Suma de dos matrices de *floats*, cuadradas y de igual tamaño. Sin optimizaciones. Se permite elegir el tamaño del problema seleccionando el tamaño de los lados de las matrices. Usa un tamaño de fragmento mínimo de 32 filas para el planificador hguided (explicado en la sección 4.1.1).

[1] `https://ark.intel.com/content/www/es/es/ark/products/210291/intel-stratix-10-gx-2800-fpga.html`

Problema de Asignación de Recursos (Rap) Algoritmo de reserva de recursos simplificado e implementado por Raúl Nozal [8]. Para cada índice (i) en un espacio de iteraciones determinado por el tamaño de problema elegido, se comprueba que i esté por debajo de un límite especificado. En caso de ser así, se recorren los elementos de dos vectores entre 0 e i, cada uno en un sentido, sumándolos y calculando el máximo de esas sumas. Esto supone que el acceso a memoria no es secuencial para ambos vectores y que cada índice tiene una cantidad de cómputo diferente (algoritmo con intensidad computacional dinámica). El problema se prueba con números enteros (*int*) y el tamaño empleado en los experimentos corresponde al límite a partir del cual no se calcula y a la raíz cuadrada del número de elementos de cada vector. Usa un tamaño de fragmento mínimo de 128 elementos para el planificador hguided.

Multiplicación de Matrices (Matmul) Multiplicación de dos matrices de *floats*, cuadradas y de igual tamaño. Sin optimizaciones. Se permite elegir el tamaño del problema seleccionando el tamaño de los lados de las matrices. Usa un tamaño de paquete mínimo de 8 filas para el planificador hguided.

Problema de los N Cuerpos (Nbody) Simulación N-body que aproxima numéricamente la evolución de un sistema de cuerpos donde cada cuerpo interacciona constantemente con todos los demás. Para cada cuerpo se calcula el efecto que tienen todos los demás sobre él, y a partir de ahí su siguiente posición y velocidad. Optimizado originalmente con desenrollado de bucles que se retira para medir el rendimiento base. Tanto la posición como la velocidad se representan en tres dimensiones con *float4*. Se permite elegir el tamaño del problema seleccionando el número de cuerpos de la simulación. Usa un tamaño de fragmento mínimo de 64 cuerpos para el planificador hguided.

Filtro Gaussiano (Gaussian) Filtro que calcula el valor de cada píxel de salida en base a una media ponderada del píxel de entrada y sus adyacentes en un radio proporcional al tamaño del filtro. Sin optimizar. Se permite elegir el tamaño de la imagen (del lado, dando siempre lugar a una imagen cuadrada) y del filtro (del lado de la matriz, que será una matriz cuadrada), que debe ser impar. Los píxeles de la imagen de entrada y de salida son *uchar4*, y el filtro son *floats*. En los experimentos siempre se usa un filtro de 5x5. Usa un tamaño de paquete mínimo de 128 filas para el planificador hguided.

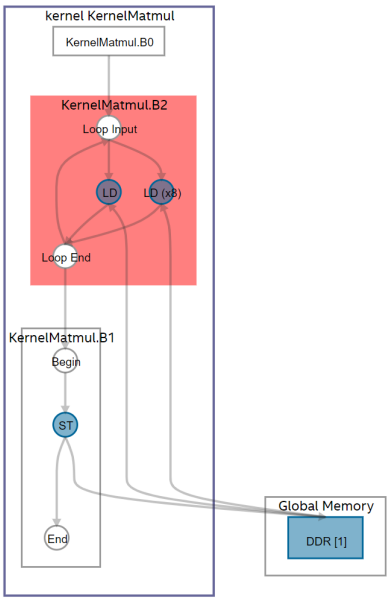
3.3. Medición del tiempo de ejecución

Para cada experimento se mide la diferencia de tiempo entre el momento en que se llama al planificador y el momento en el que este termina. Las mediciones emplean la biblioteca "*chrono*" de C++. Con fin de evitar una alta variabilidad en los resultados, cada experimento se repite múltiples veces[2], siendo el resultado válido la media de todas, y se presta particular atención a la desviación estándar. En los experimentos con FPGA se realiza una ejecución previa (no contabilizada en la media) para que la FPGA se reconfigure, y así evitar penalizaciones temporales asociadas al primer uso (*warm-up*). Para los experimentos heterogéneos, además del tiempo de ejecución del planificador se mide el tiempo de terminación de cada dispositivo y el trabajo que ha realizado. Esto nos permitirá tener una medida sobre el desbalanceo existente en la ejecución.

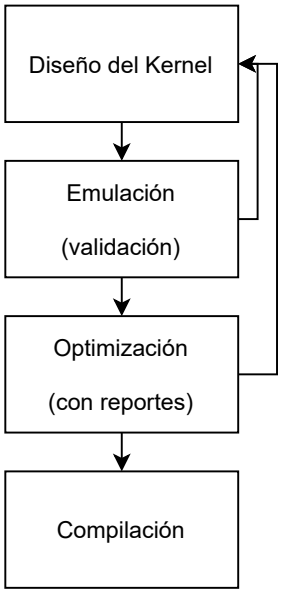
3.4. Flujo de trabajo con la FPGA

Como se ha mencionado previamente, la compilación para FPGA es un proceso que lleva múltiples horas (en la experiencia de este trabajo se tarda aproximadamente cinco horas por *kernel*). Por tanto es el último paso a realizar, cuando se está convencido de que el diseño del *kernel* es el deseado. Para ello, oneAPI ofrece un dispositivo emulador de FPGA que permite validar los resultados del *kernel*. Una vez es funcionalmente correcto, se realiza una compilación intermedia que genera un reporte sobre cómo se va a implementar funcionalmente el *kernel*. En este reporte se pueden detectar cuellos de botella analizando los accesos a memoria, la implementación de los bucles (intervalo de iniciación, desenrollado,...) o la frecuencia de reloj máxima estimada. La Figura 3.1a representa el grafo del sistema generado en el reporte para Matmul con desenrollado de grado 8. La sección roja indica que no ha sido capaz de segmentar ese bucle, pero si se hiciera clic sobre el *load* se vería que realiza un acceso a memoria de 256 bits correspondiente a 8 *floats*. Tras valorar con el reporte que las optimizaciones se están aplicando como se desea, se realiza la compilación completa para generar el *bitstream*. El flujo de compilación entero se representa en la Figura 3.1b.

[2]Se calculó sobre los experimentos base (Sección 5.1) que con cinco repeticiones se obtiene un coeficiente de variación bajo. No obstante, se calcula siempre la desviación estándar de las repeticiones y se ejecutan experimentos adicionales si esta no se considera aceptable.



(a) Grafo del *kernel* completo en un reporte de optimización



(b) Flujo de compilación de FPGA

Figura 3.1: Compilación en FPGA

Capítulo 4

Coejecución con Intel oneAPI

En este capítulo se describen los diferentes pasos realizados para portar una herramienta de coejecución heterogénea CPU-GPU a CPU-FPGA. Entendemos por herramienta de coejecución heterogénea aquella que es capaz de dividir el problema a resolver en múltiples fragmentos y repartir dichos fragmentos entre los distintos dispositivos, de modo que resuelvan el problema inicial de manera cooperativa.

4.1. Trabajo previo

En este trabajo fin de grado se parte de una herramienta de coejecución desarrollada por Raúl Nozal de la Universidad de Cantabria [8]. Dicha herramienta era capaz de realizar ejecución heterogénea entre CPU y GPU usando oneAPI. La herramienta incluye un conjunto de *benchmarks* del ámbito científico (multiplicación de matrices, filtro gaussiano, ...) descritos en el capítulo 3.2, e implementa tres planificadores de ejecución (estático, dinámico y hguided) descritos a continuación.

4.1.1. Planificadores

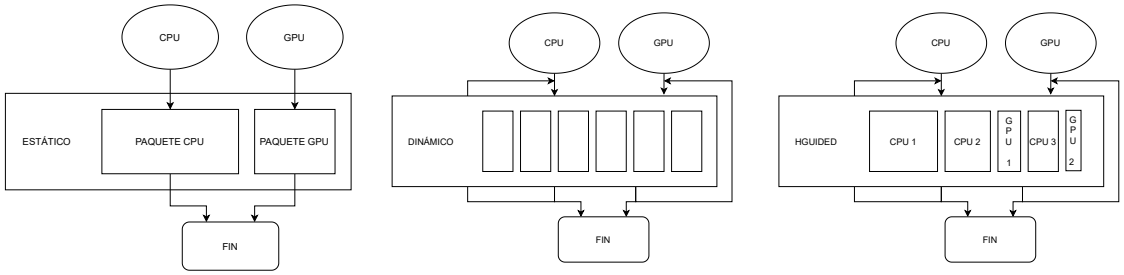
Planificador estático (Figura 4.1a) Divide el problema en dos fragmentos, uno para CPU y otro para GPU. El tamaño de los fragmentos es ajustable por el usuario según la potencia de cálculo que le estime a cada dispositivo.

Planificador dinámico (Figura 4.1b) Divide el problema en N fragmentos de igual tamaño que se reparten de uno en uno a los distintos dispositivos conforme van quedando ociosos.

Planificador hguided (Figura 4.1c) Combinando ideas de los planificadores anteriores se reparten fragmentos de manera dinámica, pero en este caso no se elige partir el problema en N fragmentos iguales, sino que se elige la potencia de cálculo

estimada de cada dispositivo y éstos van ejecutando fragmentos de tamaño proporcional a esa potencia. Además, los fragmentos son más pequeños cuanto menos problema queda por resolver. Para permitir ajustar el número de paquetes se incluye un parámetro, K, por el que se divide el tamaño de los paquetes cuando se planifican. Existe también un tamaño mínimo de fragmento ajustable para cada *benchmark* (especificado en la sección 3.2). Más concretamente, el tamaño de cada paquete viene determinado por la ecuación (4.1).

$$Fragmento = max(MinFrag, \frac{Restante \cdot PotenciaComputo}{K}) \tag{4.1}$$



(a) Estático (b) Dinámico (c) Hguided

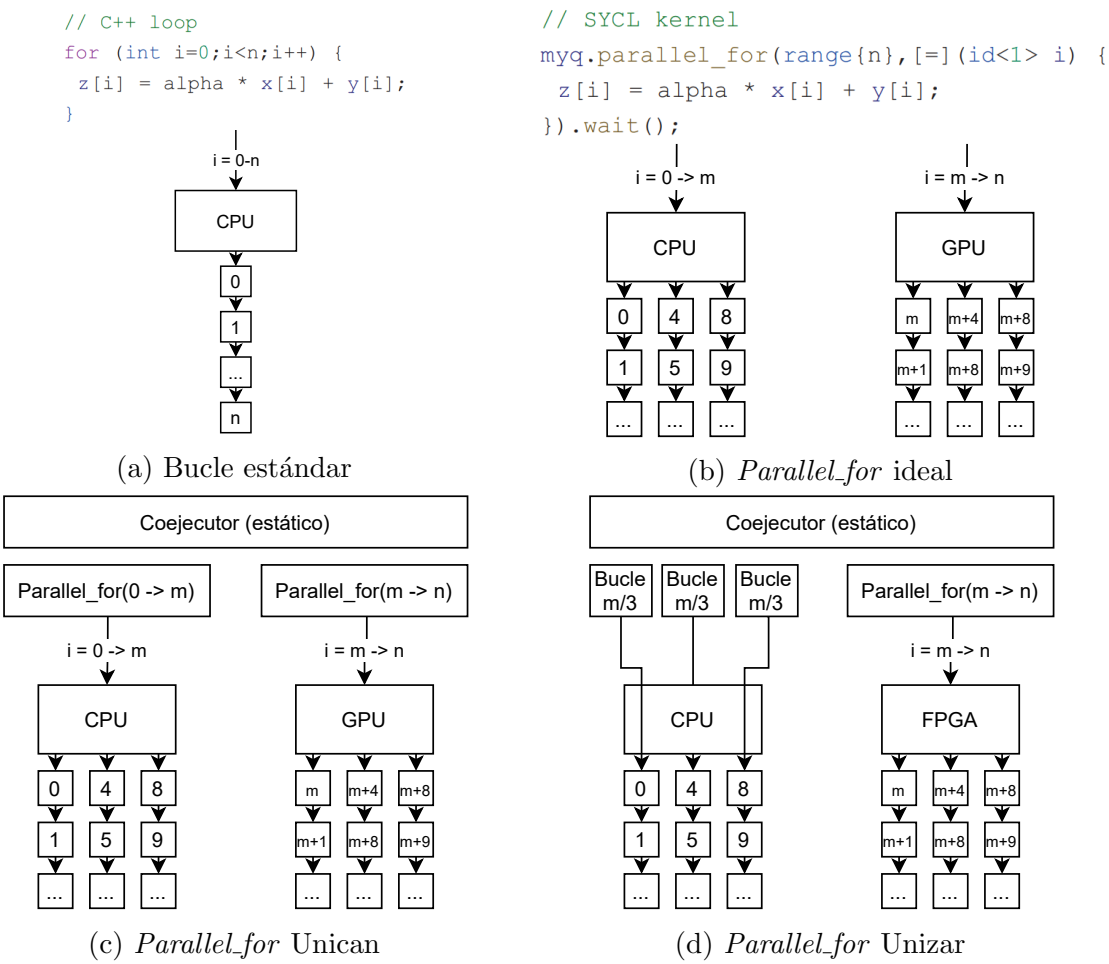
Figura 4.1: Planificadores originales

4.1.2. Paralelismo en la coejecución

Para realizar coejecución se emplea el patrón *parallel_for*. En la Figura 4.2 se representa una iteración secuencial de un bucle regular C++ (4.2a) junto con su traducción a oneAPI. La estructura *parallel_for* permite expresar que las iteraciones son independientes entre sí. De este modo, de manera ideal se podrían repartir las diferentes iteraciones independientes entre múltiples dispositivos (4.2b). El *runtime* de oneAPI no permite realizar este reparto ideal (el propio *parallel_for* se invoca sobre una cola de dispositivo único) automáticamente, por lo que la herramienta de la Universidad de Cantabria divide el espacio de iteraciones manualmente en dos (estático) o más (dinámico y hguided) porciones que se asignan a las colas individuales de los dispositivos (4.2c). La paralelización de la herramienta final, tras extenderla en este trabajo, se muestra en la Figura 4.2d para el planificador estático. En este caso, se emplean bucles regulares paralelizados manualmente (motivo explicado en la sección 4.2) y una FPGA.

4.2. Limitación de oneAPI con la FPGA

Al cambiar la GPU por una FPGA resulta obligatorio emplear el modelo de compilación AOT en lugar de JIT (2.3.2). Este modelo consigue combinar el código del



(a) Bucle estándar

(b) *Parallel_for* ideal

(c) *Parallel_for* Unican

(d) *Parallel_for* Unizar

Figura 4.2: Paralelización con *Parallel_for*

host con el *bitstream* de la FPGA en un mismo fichero y permite ejecutar *kernels* en la FPGA. Por desgracia, al añadir la CPU como segundo dispositivo la compilación no genera correctamente el fichero ejecutable. Eventualmente se confirmó desde Intel que, pese a existir tutoriales al respecto y estar descrito en los manuales, no estaba soportada la generación de un *FatBinary* con código de FPGA y otro dispositivo.

Para superar este contratiempo se hizo necesaria la sustitución del dispositivo oneAPI CPU por una implementación C++ nativa que se ejecuta como código del *host* en la CPU.

4.3. Portabilidad entre Intel oneAPI y C++

4.3.1. *Benchmarks*

Debido a los problemas descritos en la sección anterior fue necesario portar el código de los *benchmarks*, que originalmente estaban escritos en Intel oneAPI (usando DPC++), a C++ para su ejecución en la CPU. Este fue un paso relativamente sencillo puesto que

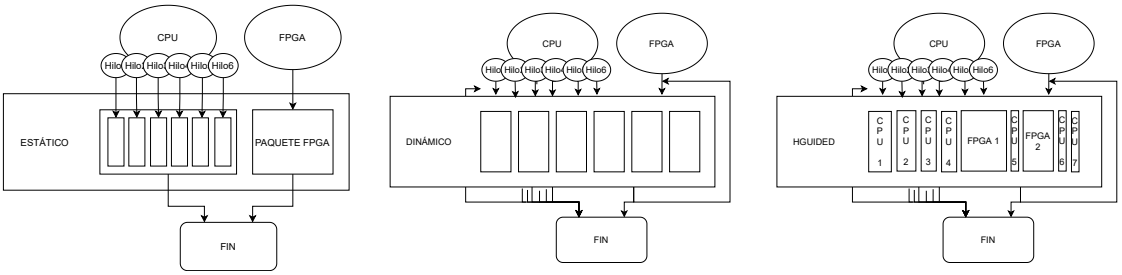
ambos lenguajes comparten una sintaxis parecida. Así mismo, para su posterior análisis experimental, se añadió una opción mas a los modos de ejecución de la herramienta (CPU++), quedando en total los siguiente modos:

- CPU++ (dispositivo único CPU con código C++)
- oneCPU (dispositivo único CPU con código oneAPI)
- oneFPGA (dispositivo único FPGA con código oneAPI)
- Heterogéneo (CPU + FPGA con código C++ y oneAPI)

4.3.2. Planificadores

El cambio de lenguaje oneAPI a C++ requirió cambios más profundos en la implementación de los planificadores de trabajo (4.1.1). La razón es que el código oneAPI expresa implícitamente el paralelismo en los *kernels*, siendo el propio entorno oneAPI el encargado de aprovechar ese paralelismo haciendo uso del *hardware* subyacente, en el caso de la CPU empleando todos sus *cores*. C++ por el contrario no expresa ni aprovecha el paralelismo automáticamente, sino que ha tenido que explicitarse manualmente implementando el patrón de diseño *thread-pool* utilizando los hilos y mecanismos de sincronización de la biblioteca estándar de C++11. En la Figura 4.3 se representan los planificadores modificados. Los, en este caso, seis hilos de CPU atacan a los planificadores. En el caso del estático se debe subdividir el paquete de CPU en seis paquetes de igual tamaño. Para el hguided la potencia de la CPU debe ser ajustada en base al número de hilos (en este caso un sexto de la potencia inicial). La ecuación (4.2) muestra la nueva fórmula para la CPU.

$$Fragmento = max(MinFrag, \frac{Restante \cdot \frac{PotenciaComputo}{NumeroHilos}}{K}) \tag{4.2}$$



(a) Estático (b) Dinámico (c) Hguided

Figura 4.3: Planificadores nuevos

4.4. Portabilidad para la FPGA

Como parte del trabajo se realizó un esfuerzo de adaptación para que oneAPI hiciese un uso mas óptimo de la FPGA, por ese motivo se buscaron transformaciones en el código que mejoraran el rendimiento.

4.4.1. Matadd

Se han añadido atributos de optimización a nivel de *kernel*, que indican que en lugar de paralelizar la ejecución para todo el rango (sumar cada elemento de las matrices por separado y en paralelo), se agrupe la ejecución de 16 en 16 elementos y emplee instrucciones SIMD. Se consigue un *speedup* pequeño (1.43 veces más rápido) ya que gran parte del tiempo de ejecución se dedica a la transferencia de datos (y preparación del entorno oneAPI), el cual no se ve afectado por las mejoras aplicadas al *kernel* (Anexo B.1).

4.4.2. Matmul, Rap y Nbody

Se ha dividido el bucle principal en dos bucles anidados, de manera que el interno realiza 8 iteraciones desenrolladas y el externo comprueba que queden suficientes iteraciones restantes. La principal ventaja de este desenrollado es que permite a la FPGA lanzar múltiples accesos a memoria en paralelo y, si estos están indexados con el iterador del bucle interno y son secuenciales en memoria, le permite agrupar múltiples accesos pequeños en uno de mayor tamaño. Se consiguen *speedups* de 4.6 para Matmul, 1.65 para Rap (sobrecarga similar a Matadd) y 8.04 para Nbody, y podrían conseguirse *speedups* mayores con un grado de desenrollado mayor.

4.4.3. Gaussian

Para Gaussian se ha intentado aplicar un desenrollado similar, pero la FPGA no conseguía sacar rendimiento, por lo que se ha fijado el tamaño del filtro a 5x5 en tiempo de compilación y aplicado el desenrollado al bucle original (sin dividirlo en dos). No se ha conseguido que la FPGA agrupe múltiples accesos pequeños en uno de mayor tamaño, pero con la optimización se lanzan en paralelo. Se alcanza un *speedup* de 2.53.

Capítulo 5

Resultados experimentales

En este capítulo se explican los diferentes experimentos realizados y los resultados obtenidos. Se da comienzo con un estudio del rendimiento de cada dispositivo por separado y cómo afecta al rendimiento el lenguaje de programación y algunas optimizaciones. Finalmente, se utilizarán los tres planificadores presentados anteriormente (Sección 4.1.1) para obtener resultados de ejecución cooperativa.

5.1. Rendimiento por dispositivo

En la Figura 5.1 se muestra el tiempo de ejecución base de cada *benchmark* para cinco tamaños de problema y tres entornos diferentes: CPU++, oneCPU y oneFPGA. CPU++ se refiere a ejecución de código nativo C++ sobre CPU, mientras que oneCPU y oneFPGA representan la ejecución de código oneAPI sobre CPU y FPGA, respectivamente.

A partir de estos resultados se pueden dividir los *benchmarks* en tres categorías:

- Ligeros (Matadd y Rap): los tiempos de ejecución son muy pequeños en la versión nativa (máximo 0.51 segundos), por lo que la sobrecarga añadida por la comunicación de datos entre el *host* y el dispositivo en las versiones oneAPI es demasiado significativa y las hace considerablemente más lentas (la sobrecarga se analiza en más detalle en el Anexo B.1).

- Intermedio (Gaussian): los tiempos de ejecución son de rango medio, comportándose como los *benchmarks* ligeros para tamaños de problema pequeños y como los pesados para los grandes. No obstante, incluso en los mayores tamaños, el rendimiento en CPU++ no queda tan por detrás de oneFPGA como en los pesados. Entre oneCPU y oneFPGA, oneAPI es capaz de extraer mejor rendimiento en el dispositivo más genérico, la CPU.

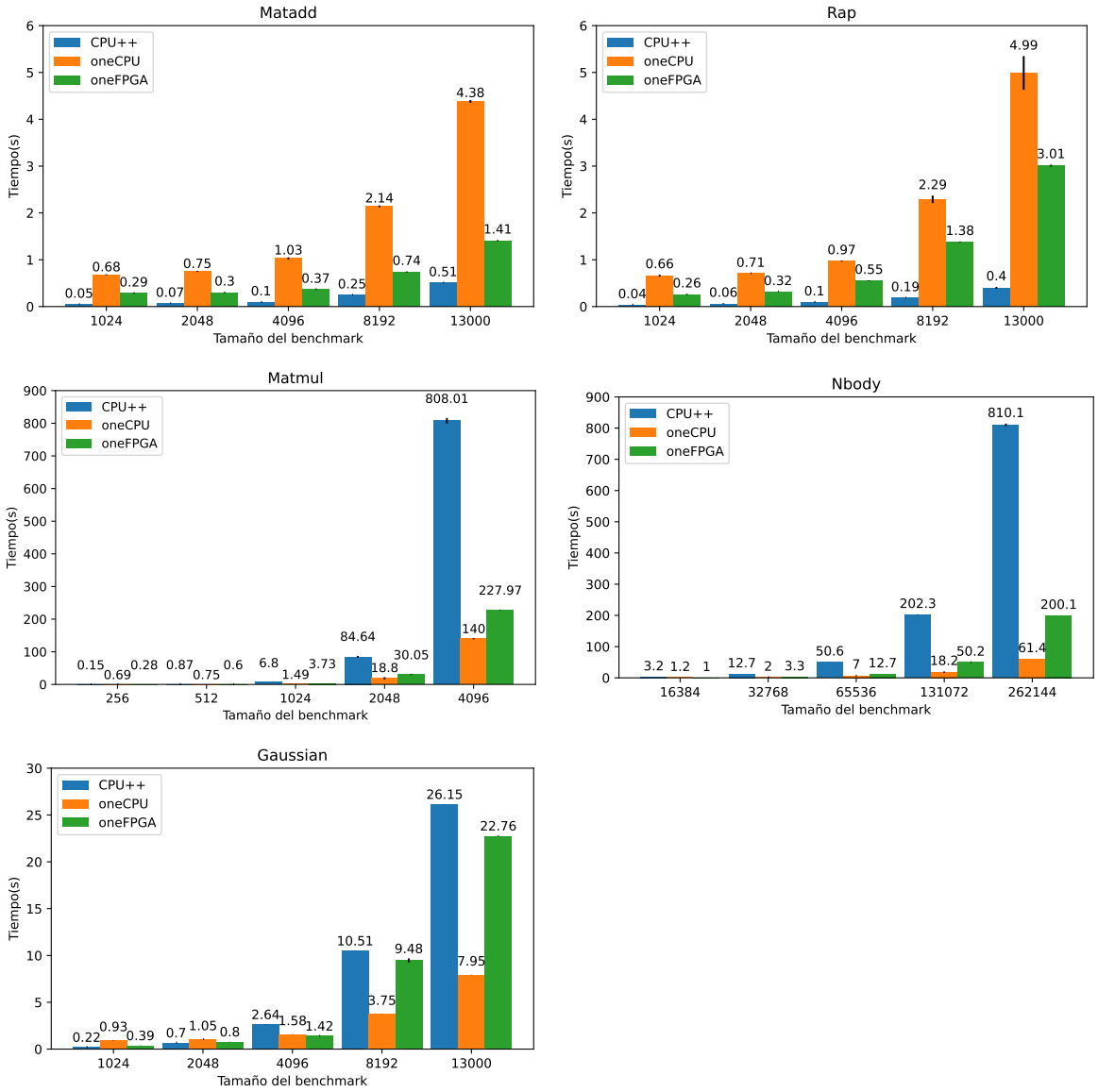


Figura 5.1: Comparativa del rendimiento base en cada dispositivo. CPU++ emplea un *core* mientras que oneCPU emplea los 6 disponibles.

– Pesados (Matmul y Nbody): los tiempos de ejecución son muy altos para la versión nativa, permitiendo a la FPGA ser aproximadamente 4 veces más rápida. En el caso de oneCPU se observan *speedups* máximos de 13.2 respecto a CPU++.

En general, las ejecuciones lo suficientemente pesadas como para conllevar 1 segundo o más de cómputo en un hilo de CPU++ ejecutan mejor en cualquiera de los dos dispositivos oneAPI.

Con este experimento se verifica que oneAPI es capaz de adaptar un mismo código a su ejecución en múltiples dispositivos (por separado), manteniendo buen rendimiento siempre que la carga de trabajo sea lo suficientemente grande como para despreciar la sobrecarga del *runtime*.

5.2. Optimización

Una vez tomados los rendimientos base, se ha optimizado la implementación oneAPI de los *benchmarks* para la ejecución en un dispositivo concreto, la FPGA, valorando así las opciones que ofrece el modelo de programación y permitiendo estimar el margen de mejora del código base. Asimismo, se ha optimizado la versión nativa, permitiendo comparar la efectividad de las optimizaciones aplicadas al código oneAPI. También permite evaluar el uso del paralelismo que hace oneAPI (Anexo B.3).

5.2.1. Paralelización de CPU++

Al migrar oneCPU a CPU++ se pierde la paralelización implícita presente en el código oneAPI. En este apartado se evalúa la efectividad de reimplementar esa paralelización en CPU++ manualmente mediante la adición de un *thread-pool*.

En la Figura 5.2 se muestra la comparativa, para los diferentes problemas y tamaños, entre el tiempo de ejecución base de la versión nativa secuencial y el de la versión nativa paralelizada con 6 hilos (ya que la CPU empleada dispone de 6 *cores*). Cada *benchmark* se divide en seis fragmentos de igual tamaño y cada fragmento es asignado a un hilo.

Se observa que la optimización resulta muy efectiva (analizado en más detalle en el Anexo B.2). Los *benchmarks* de mayor cómputo (Matmul, Nbody y Gaussian) se acercan al *speedup* máximo para seis hilos a lo largo de los diferentes tamaños de problema. En Matadd también se observan mejoras cercanas al máximo de 6, mientras que en Rap sólo se alcanza un 3.5. Esto probablemente se deba a que al tener los tiempos de ejecución más cortos y ser un problema de intensidad computacional dinámica (fragmentos de igual tamaño tienen diferente cantidad de cómputo), el desbalanceo entre los diferentes hilos es más aparente.

En el Anexo B.3 se compara esta versión paralelizada en C++ con la versión base oneAPI ejecutada en CPU y FPGA.

5.2.2. Optimización FPGA

Como parte del trabajo de adaptación del código de Intel oneAPI para su ejecución en la FPGA, se realizaron las siguientes mejoras: usar grupos de trabajo y SIMD, desenrollar bucles, y mejorar los accesos a memoria, tal y como se describió en la sección 4.4.

Optimización de *kernels* En la Figura 5.3 se muestra el rendimiento que oneAPI es capaz de extraer de la FPGA para los *kernels* sin optimizar (caso base) y se compara

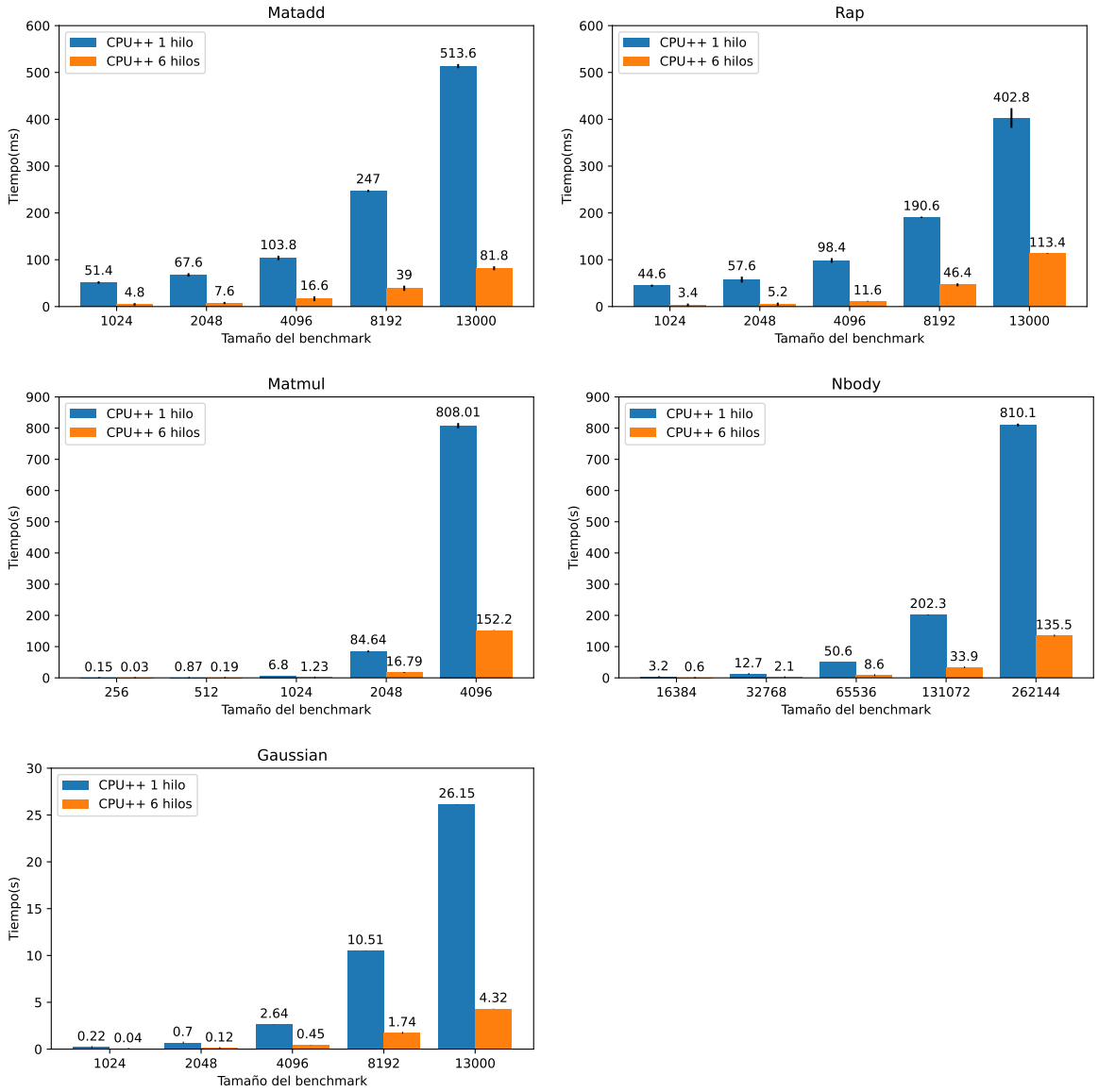


Figura 5.2: Rendimiento en C++ de un hilo frente a seis

con el de los *kernels* optimizados para FPGA, empleando los diferentes tamaños y *benchmarks*.

Se observan mejoras en todos los *benchmarks* y casi todos los tamaños, pero estas son más irregulares que las obtenidas para CPU++ (5.2.1), variando el *speedup* en los tamaños más grandes desde 1.43 en Matadd hasta 8.28 en Nbody. La irregularidad vuelve a ser fruto de la densidad de cómputo de cada *benchmark*, pero no exclusivamente. Entre los dos *benchmarks* pesados (Matmul y Nbody) se observa una diferencia de *speedup* de casi el doble. Esto se debe a que la optimización más efectiva es la agrupación de múltiples accesos a memoria en uno sólo de mayor tamaño. Esta optimización requiere que los accesos a memoria iniciales sean secuenciales, el cual es el caso para Nbody pero no del todo para Matmul, ya que realiza un acceso secuencial (recorriendo las filas) y

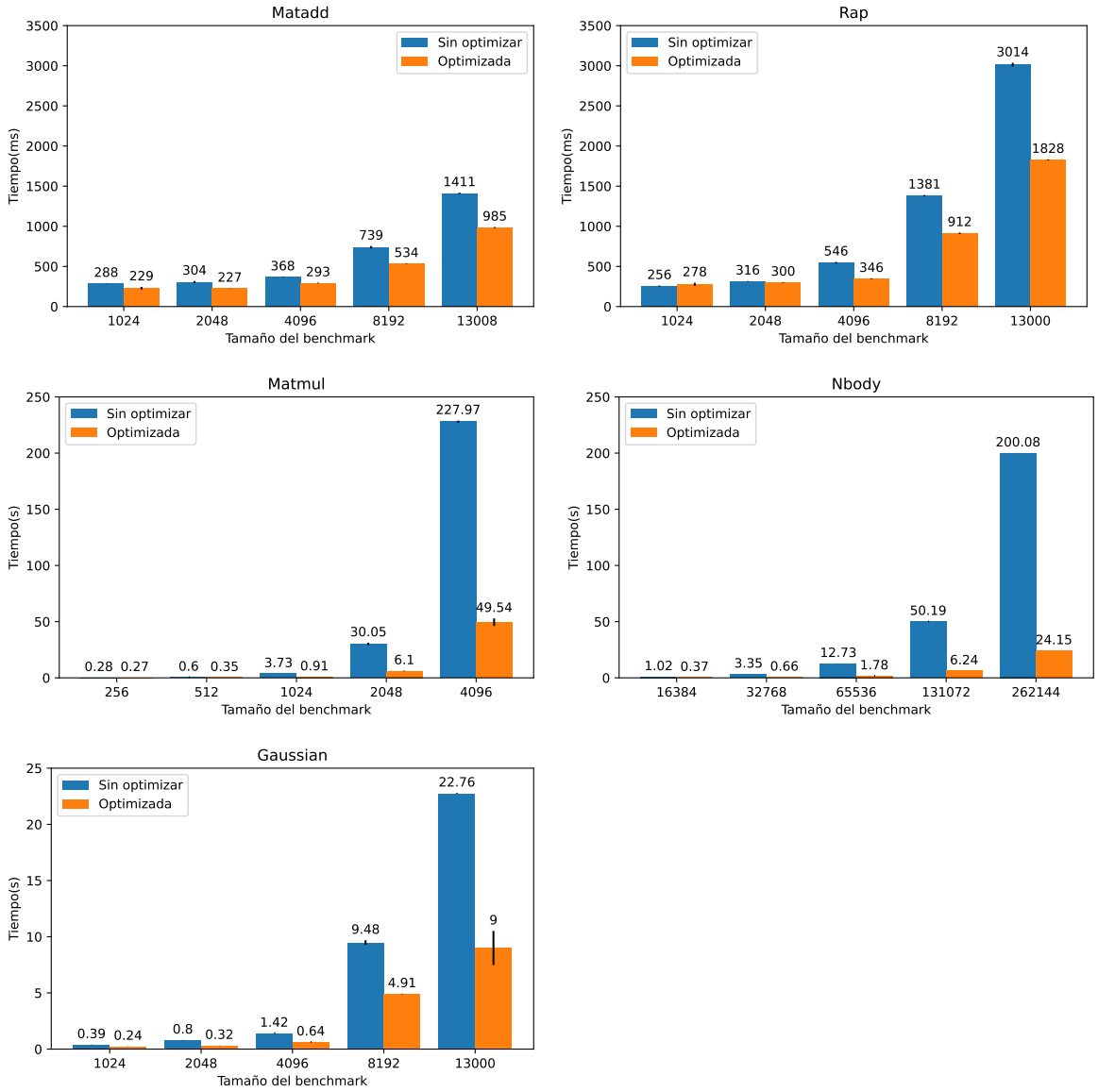


Figura 5.3: Optimización de los *kernels* para FPGA

otro que no lo es (recorriendo las columnas).

A lo largo de los diferentes *benchmarks*, oneAPI desaprovecha rendimiento que se puede conseguir con optimizaciones relativamente sencillas. Analizando los reportes de compilación se observa que oneAPI no consigue mejorar los accesos a memoria (tamaño de los mismos y realizarlos en paralelo) sin intervención específica del programador.

División en fragmentos En la Figura 5.4 se aplica una segunda optimización a la FPGA, dividiendo cada *benchmark* en diferente cantidad de fragmentos (todos los fragmentos de un mismo *benchmark* son del mismo tamaño) para que la FPGA tenga más oportunidades para solapar cómputo y transferencias de resultados. [1]

[1]La división en fragmentos del tamaño 262144 de Nbody da lugar a errores de cálculo en FPGA, por lo que se emplea 131072 como tamaño máximo siempre que se ha de dividir el problema asignado

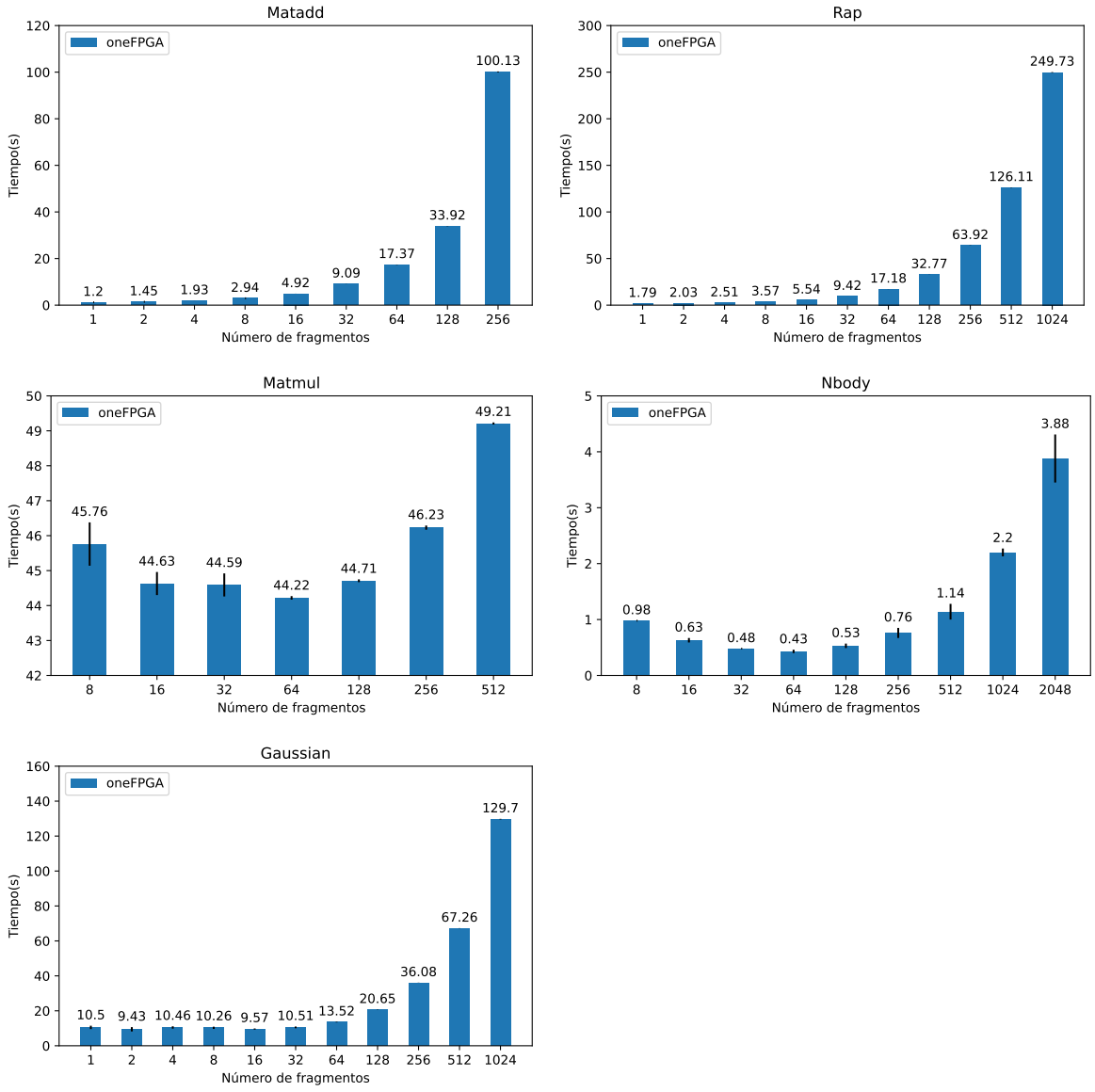


Figura 5.4: Optimización del tamaño de paquete para FPGA

Se observa que el solape de cómputo y comunicaciones es efectivo en los *benchmarks* pesados (*speedups* de 1.12 en Matmul y 14.19 en Nbody), pero no en el resto.

Tras aplicar esta segunda optimización, los recursos usados de la FPGA siguen siendo bajos y por tanto el número de operaciones de punto flotante realizadas está lejos del máximo. Para Matadd se usa menos de un 5 % de la FPGA y se realizan 3.1GFLOPS de un máximo de 5.1TFLOPS que puede realizar la FPGA según Intel. El objetivo de las optimizaciones no era conseguir la máxima eficiencia sino probar las opciones de optimización y comprobar que en los *kernels* hay mucho margen de mejora si los optimizas manualmente. En un futuro se podrían valorar optimizaciones más agresivas como desenrollados del máximo grado posible (hasta que se usan todos

a la FPGA.

los recursos de la FPGA).

5.2.3. Comparación entre optimizaciones

Para facilitar la comparación entre optimizaciones se resumen los *speedups* máximos obtenidos en la Tabla 5.1. También se muestra, en la Figura 5.5, el tiempo de ejecución de cada *benchmark* en su versión oneFPGA optimizada, normalizado respecto a la versión CPU++ paralelizada.

Problema	*Speedup* CPU++	*Speedup* oneFPGA
Matadd	6.28	1.43
Matmul	5.3	5.15
Rap	3.55	1.68
Nbody	5.97	116.7
Gaussian	6.05	2.53

Tabla 5.1: *Speedups* de las optimizaciones

Tras aplicar las optimizaciones, la superioridad de la versión nativa para los *benchmarks* ligeros (Matadd y Rap) se acentúa ya que la optimización de FPGA no afecta a la sobrecarga del *runtime*, por lo que las versiones nativas tienen mejores *speedups*. En el caso del *benchmark* intermedio (Gaussian), se obtiene muy poco *speedup* para FPGA ya que el *kernel* es complejo de optimizar. Por el contrario, la mejora obtenida en CPU++ ha sido la máxima posible, lo que causa que sea dos veces más rápida que la de FPGA (sin optimizaciones Gaussian era mas rápido en la FPGA). En los *benchmarks* pesados (Matmul y Nbody) la FPGA sigue siendo la mejor opción, habiendo sido las mejoras en CPU++ y FPGA aproximadamente equivalentes en Matmul y muy favorables a la FPGA en Nbody.

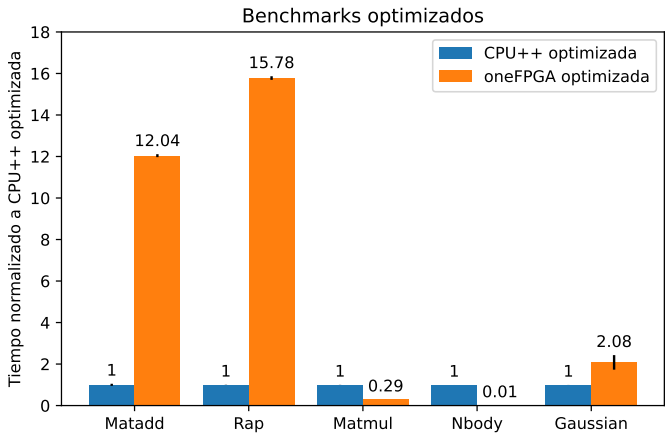


Figura 5.5: Comparación CPU++ y oneFPGA tras optimizar

En vista de la magnitud de las diferencias temporales entre versiones para cada *benchmark* (Figura 5.5), Matadd y Rap están claramente descompensados en favor de la versión nativa e igualmente Nbody en favor de la FPGA. Por el contrario, Matmul y Gaussian están más balanceados (aunque con preferencia por oneFPGA y CPU++ respectivamente), y serán los que más se beneficien de la ejecución cooperativa. No obstante, pese al esfuerzo realizado en optimizar ambas versiones, sigue habiendo margen de mejora pendiente como trabajo futuro. Optimizando más la versión CPU++ de Nbody o mejorando el uso de memoria de la FPGA para Matadd y Rap (solo reservando para un fragmento a la vez) y aumentando el tamaño del problema, se podría igualar más el rendimiento de ambos dispositivos, facilitando la ejecución heterogénea cooperativa.

5.3. Ejecución cooperativa

En este apartado se evalúa la posibilidad de utilizar la CPU y la FPGA para la resolución cooperativa del mismo problema. Se emplean las versiones optimizadas para cada dispositivo, con 6 hilos de ejecución para la CPU y *kernels* optimizados para la FPGA.

Aunque el trabajo se distribuya entre los dos dispositivos, los datos de entrada serán comunicados a las memorias privadas de cada dispositivo. Esto causa que en los problemas ligeros el tiempo de terminación del dispositivo sea demasiado elevado pese a computar poco porcentaje.

En los planificadores por paquetes (dinámico y hguided), descritos en la subsección 4.1.1, se crea una gran variabilidad al usar un número de paquetes pequeño dependiendo del orden en el que cada dispositivo coge un paquete, por lo que se ha decidido omitir las gráficas de Matadd y Rap (se incluyen en el Anexo B.4). Además, la existencia de seis hilos de C++ para computar la parte nativa da resultado a que generalmente coja paquetes de seis en seis, por lo que con menor número total de paquetes la CPU tiende a desbalancear la finalización del problema al ser el dispositivo más lento para los *benchmarks* pesados y dejar ociosa a la FPGA.

5.3.1. Planificador estático

En la Figura 5.6 se representa en el eje Y el tiempo de terminación de cada dispositivo (los 6 hilos CPU++ en azul y oneFPGA en naranja), siendo el tiempo de terminación del planificador equivalente o ligeramente superior al más alto de los dos (en gris). En el eje X se representa la porción del problema que ha de ser computada por la FPGA (0.2 corresponde a un 20 % ejecutado en FPGA), siendo el resto ejecutado por la versión nativa.

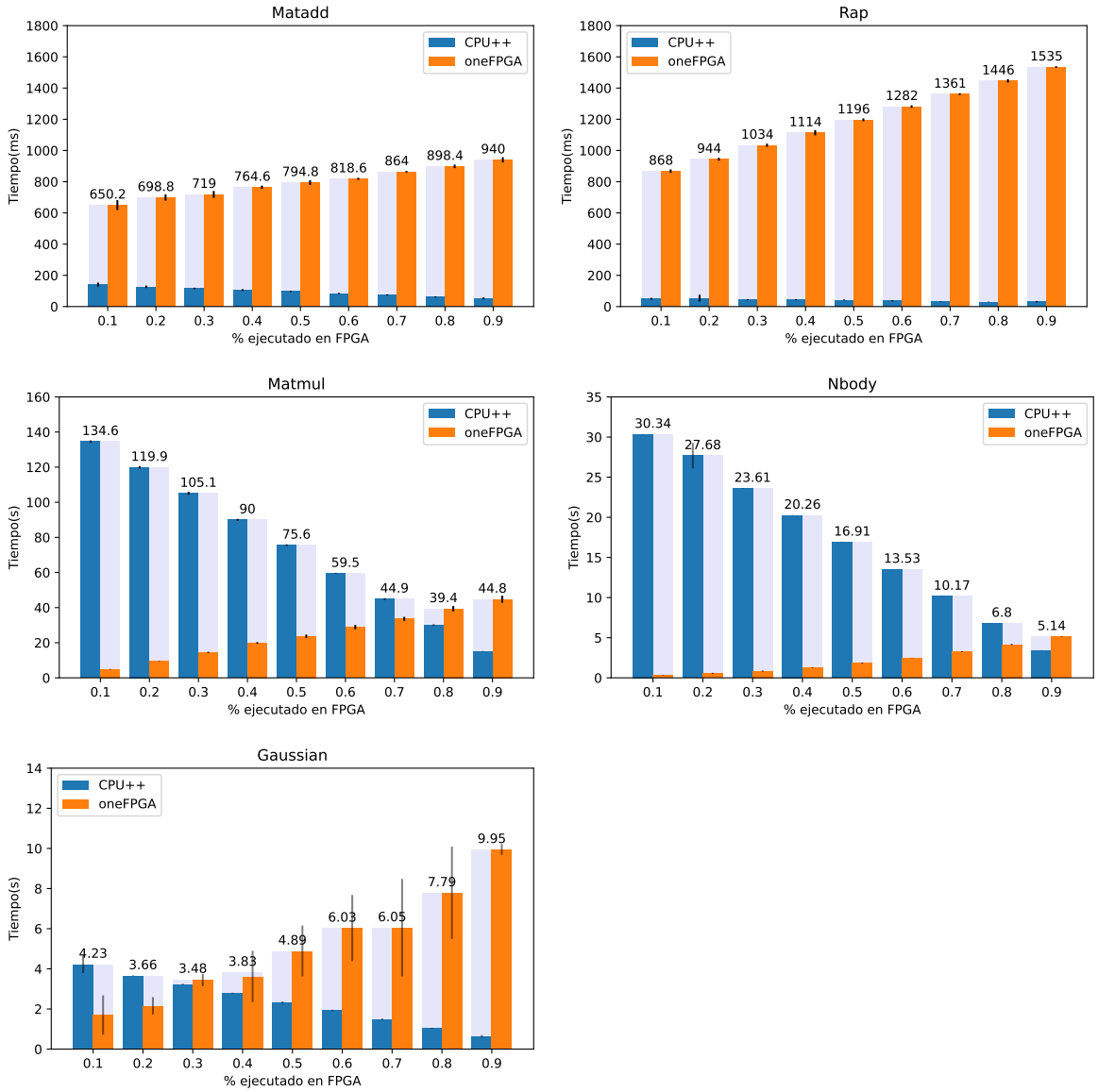


Figura 5.6: Rendimiento de los *benchmarks* con el planificador estático

Para los problemas ligeros (Matadd y Rap), el tiempo de terminación siempre viene condicionado por oneFPGA, siendo mejor ejecutar la totalidad del problema en CPU++. Era lo esperado ya que la sobrecarga del *runtime* para oneFPGA dominaba el rendimiento para tiempos tan pequeños. En los otros tres *benchmarks* la coejecución supone una mejora en el rendimiento [2], resolviendo la FPGA la mayoría del problema en Matmul y Nbody (80 % y 90 % respectivamente) y tan solo un 30 % en Gaussian.

[2]Con respecto a la versiones estáticas, que no hacen solapamiento entre cómputo y comunicación para FPGA

5.3.2. Planificador dinámico

De manera similar al planificador estático, en la Figura 5.7 se muestra el tiempo de terminación de cada dispositivo en ejecución heterogénea y del planificador dinámico. En este caso el eje X representa el número de paquetes (de igual tamaño) en los que se ha dividido el problema. Dado que el reparto de trabajo se realiza de forma dinámica, en la parte superior de la gráfica, en éste eje X, se representa el porcentaje que se ha entregado de manera efectiva a la versión C++ para resolver.

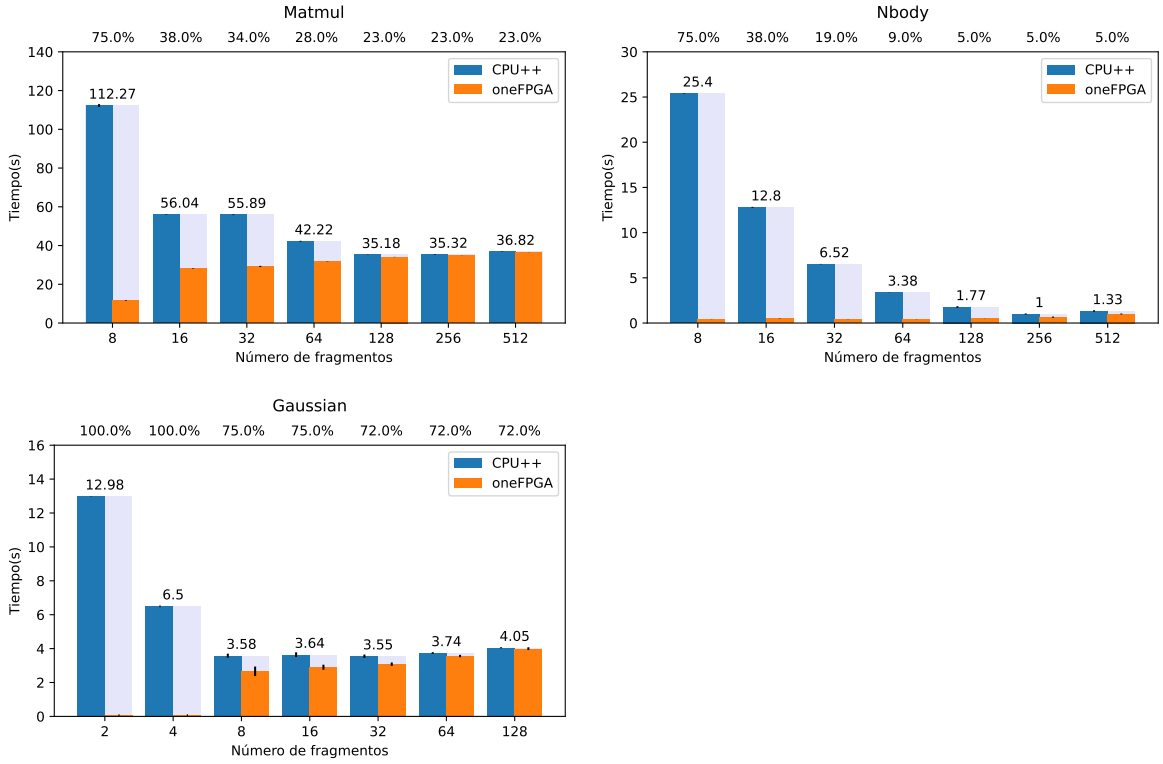


Figura 5.7: Rendimiento de los *benchmarks* con el planificador dinámico

Se aprecia que con menor número de paquetes la CPU ejecuta demasiada porción del problema ya que son 6 hilos pidiendo trabajo frente a uno de FPGA, además los hilos de CPU son menos eficientes que la FPGA, provocando desbalanceo. Esto supone que la CPU acabe siendo el condicionante de los altos tiempos de ejecución. Conforme se usa mayor número de paquetes los tiempos de terminación se equilibran, obteniéndose mejores resultados que en la versión estática sin necesitar hacer un barrido por los diferentes porcentajes de reparto. En el caso de Nbody se observa que el dispositivo más lento sigue siendo la versión nativa por bastante margen para todos los repartos, pero sigue suponiendo una mejora con respecto al estático ya que la FPGA se beneficia enormemente de la paquetización del cómputo.

5.3.3. Planificador hguided

El planificador hguided es similar al dinámico, pero entrega paquetes de diferente tamaño según la potencia de cálculo que se asigne a cada dispositivo, la cantidad de problema restante y el parámetro de ajuste K (previamente explicado en la sección 4.1.1). Al poderse usar ahora múltiples hilos de CPU, se ha adaptado el planificador para que pondere la potencia de cómputo de la CPU en base al número de hilos empleados.

En la Figura 5.8 se representan los diferentes tiempos de terminación del mismo modo que para el planificador dinámico, aunque en este caso el eje X corresponde al parámetro de ajuste K, el cual se ha ido aumentando en intervalos de 0.5 (entre los valores límite del código original de 1.5 y 4).

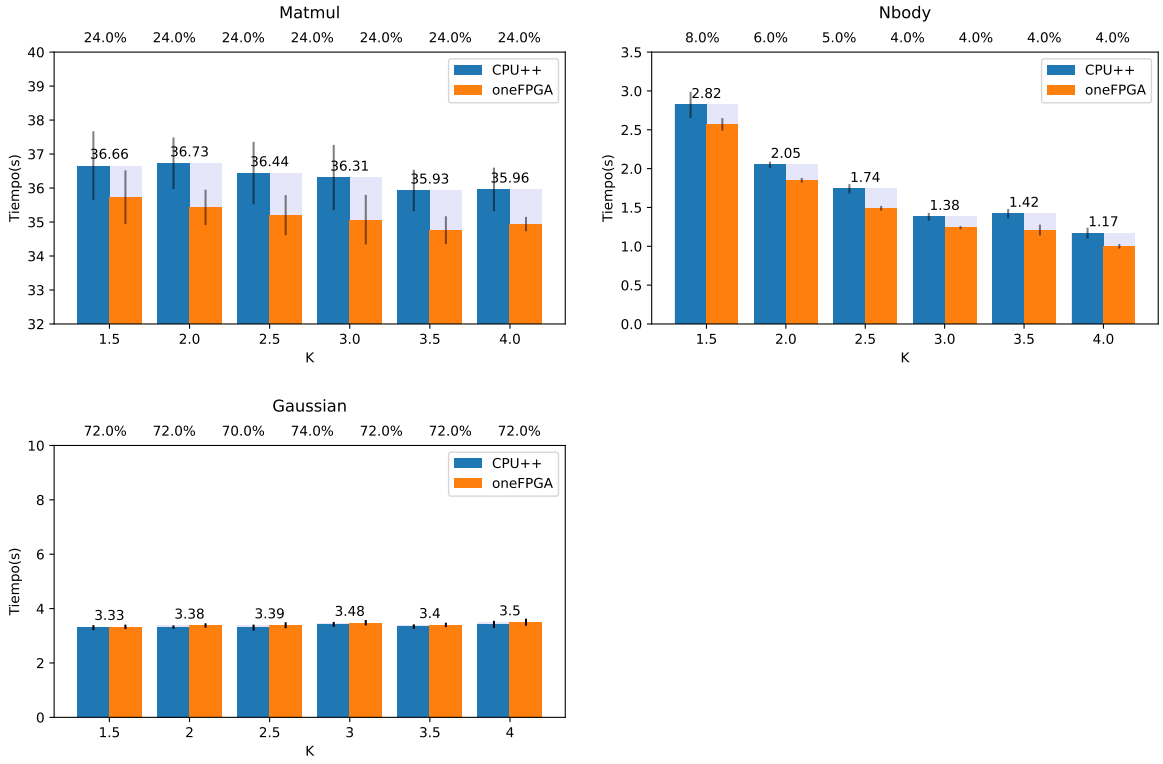


Figura 5.8: Rendimiento de los *benchmarks* con el planificador hguided

Se observa que pese a emplear como potencia de cada dispositivo los mejores porcentajes obtenidos con el planificador estático (5.6) y ajustar la potencia de la CPU++ al número de hilos, el tiempo de terminación de los *benchmarks* pesados se ve condicionado por la CPU++ en todos los casos. No obstante sigue suponiendo una mejora sobre cualquier opción no heterogénea para Matmul. Como ambos *benchmarks* pesados obtienen mejores resultados con valores de K mayores, se está realizado un estudio más exhaustivo cuyos resultados preliminares se pueden ver en el Anexo B.5. Para Gaussian hguided consigue mejor tiempo que cualquier otra versión o planificador

y buen rendimiento y balanceo en general para todos los valores de K.

5.3.4. Comparativa de planificadores

En la Tabla 5.2 se muestra el *speedup* ideal de la ejecución heterogénea junto al obtenido por cada *benchmark* y planificador[3]. Se entiende por *speedup* ideal el resultante de dividir la suma del rendimiento de ambos dispositivos (de la mejor versión de cada uno) entre el rendimiento del mejor de ellos (ecuación (5.1)). El *speedup* de cada planificador es su rendimiento respecto a la mejor versión no heterogénea (ecuación (5.2)). La eficiencia heterogénea corresponde a la relación entre el *speedup* conseguido y el ideal (ecuación (5.3)).

$$speedup\_ideal = \frac{mejorCPU + mejorFPGA}{max(mejorCPU, mejorFPGA)} \tag{5.1}$$

$$speedup\_planificador = \frac{max(mejorCPU, mejorFPGA)}{mejorPlanificador} \tag{5.2}$$

$$eficiencia\_heterogenea = \frac{speedup\_planificador}{speedup\_ideal} \cdot 100 \tag{5.3}$$

	Speedups				Eficiencia heterogénea		
Problema	Ideal	Estático	Dinámico	Hguided	Estático	Dinámico	Hguided
Matadd	1.08	0.13	-	-	12 %	-	-
Rap	1.06	0.13	-	-	12 %	-	-
Matmul	1.29	1.12	1.26	1.23	87 %	98 %	95 %
Nbody	1.01	0.08	0.43	0.37	8 %	43 %	37 %
Gaussian	1.48	1.24	1.22	1.3	84 %	82 %	90 %

Tabla 5.2: Speedup y eficiencia heterogénea de cada *benchmark* y planificador

En los *benchmarks* Matadd, Rap y Nbody hay poco margen de mejora ya que el rendimiento de ambos dispositivos está muy descompensado, siendo muy malos los resultados obtenidos en coejecución.

En Matmul y Gaussian sigue habiendo un dispositivo favorito (FPGA y CPU respectivamente) pero no por demasiada diferencia, siendo más aptos para la ejecución heterogénea y consiguiendo *speedups* con todos los planificadores.

El planificador estático consigue buena eficiencia para ambos *benchmarks*, pero requiere explorar los diferentes porcentajes de reparto y no es el mejor para ninguno de los dos.

El planificador dinámico prácticamente obtiene la eficiencia máxima para Matmul, siendo el mejor planificador para este *benchmark*, y buena para Gaussian. En este caso

[3]Matadd y Rap se omiten por tener una variabilidad muy alta (sus figuras se incluyen en el Anexo B.4).

se requiere un estudio del número de fragmentos óptimo, pero es más fácil estimar un valor que obtenga un rendimiento bueno (con 128 ó 256 fragmentos se obtienen buenos resultados en todos los *benchmarks*).

El planificador hguided obtiene una eficiencia cercana a la del dinámico en Matmul y mejor eficiencia que los otros dos planificadores para Gaussian. Pese a que teóricamente hguided no requiere de un ajuste preciso de la potencia de cada dispositivo, en la práctica ha supuesto diferencias significativas, consiguiendo *speedups* de 1.19 y 4.41 (para Matmul y Nbody respectivamente, con K=4) al ajustar la potencia de la CPU en base al número de hilos empleados. Adicionalmente hay que ajustar el parámetro K, que en Nbody puede llegar a suponer un *speedup* de 2.41[4].

5.4. Resumen de la experimentación

Entre los *benchmarks* seleccionados (cuyos tiempos de ejecución para las diferentes versiones se resumen en la Tabla 5.3[5]): Matadd y Rap han demostrado que para problemas poco costosos computacionalemente es muy complicado sacar provecho de los aceleradores, siendo mejor ejecutar solo en CPU. Nbody ha demostrado la gran superioridad que suponen los aceleradores para ciertos problemas, siendo mejor ejecutar el problema en su totalidad en la FPGA. Matmul y Gaussian son los más beneficiados por la ejecución heterogénea ya que rinden bien en ambos dispositivos. Matmul ha dado los mejores resultados con el planificador dinámico y paquetes pequeños, entregando un 77 % del problema a la FPGA. Gaussian por el contrario se beneficia más del planificador hguided y la CPU, ejecutando sólo un 28 % en la FPGA.

	Tiempo de ejecución (ms)							
	Base			Optimizado		Heterogéneo		
Problema	CPU++	oneCPU	oneFPGA	CPU++	oneFPGA	Estático	Dinámico	Hguided
Matadd	513	4380	1410	81.8	985	650	-	-
Rap	400	4990	3010	113.4	1790	868	-	-
Matmul	808.01	139.76	227.97	152.25	44.22	39.4	35.18	35.93
Nbody	202.3	18.2	50.2	33.9	0.43	5.14	1	1.17
Gaussian	26.15	7.95	22.76	4.32	9	3.48	3.55	3.33

Tabla 5.3: Mejor tiempo obtenido para cada *benchmark* en las diferentes etapas del trabajo

[4]Para valores de K entre 1.5 y 4, con valores más altos cómo en el Anexo B.5 puede suponer más aún.

[5]Las versiones heterogéneas con planificador dinámico y hguided se omiten para Matadd y Rap por tener una variabilidad muy alta (sus figuras se incluyen en el Anexo B.4).

Capítulo 6

Conclusiones y Trabajo Futuro

En este trabajo se ha conseguido desarrollar una de las primeras implementaciones de balanceo de carga CPU-FPGA sobre oneAPI, estando este problema todavía abierto en la comunidad investigadora.

Trabajar con oneAPI ha sido bastante cómodo teniendo ya experiencia con C++. El manejo de los diferentes dispositivos y datos usando colas, tareas y *buffers* es sencillo. Por contrapartida, el proceso de instalación de la FPGA y el soporte en general ha dejado mucho que desear. Fue necesario invertir un mes en solucionar un *bug* con las variables de entorno y otro para que Intel anunciara que no estaba soportada la generación de un *FatBinary* para oneCPU+oneFPGA, pese a que había tutoriales para ello. No obstante, estos obstáculos han servido para dar realimentación a Intel y que mejoren oneAPI. Los *kernels* son muy portables. Todos los *benchmarks* eran funcionalmente equivalentes ejecutando oneAPI en CPU, emulador de FPGA y FPGA.

oneAPI consigue buen rendimiento para *kernels* agnósticos al dispositivo si la carga computacional es grande. Estos *kernels*, ejecutados tanto en CPU como en FPGA, consiguen mejor rendimiento que su correspondiente implementación en C++ no paralelizada. Al paralelizar manualmente la versión C++, oneCPU sigue siendo más rápida para *kernels* pesados, pero no así oneFPGA, que para competir necesita un *kernel* optimizado para FPGA. A partir de estos resultados se concluye que oneAPI es un modelo eficiente para la programación, ya que se expresa paralelismo implícitamente y al ejecutar en CPU tiene mejor rendimiento que el paralelismo manual en C++. En el caso de la FPGA, pese a que los resultados no quedan muy por detrás del C++ tradicional paralelizado, al ser un dispositivo más específico, se desaprovechan mucho los recursos del mismo.

oneAPI facilita la optimización de *kernels* no sólo con los diferentes parámetros y atributos sino también con un buen flujo de trabajo para FPGA. Tras probar algunas optimizaciones básicas se han obtenido mejoras muy variadas en los diferentes *benchmarks*, que refuerzan la idea de que hacer un uso óptimo de la FPGA es complejo

y debe hacerse manualmente. No obstante, se ha sido bastante conservador aplicando las optimizaciones y queda como trabajo futuro probar a sacrificar usabilidad de los *benchmarks* en favor de rendimiento (por ejemplo con *unrolls* de mayor grado, forzando al tamaño del problema a ser múltiplo de este y usando todos los recursos de la FPGA) y probar otras optimizaciones.

Al realizar ejecución heterogénea, Matadd y Rap han demostrado que para problemas poco costosos computacionalemente es muy complicado sacar provecho de los aceleradores, siendo mejor ejecutar solo en CPU. Nbody ha demostrado la gran superioridad que suponen los aceleradores para ciertos problemas, siendo mejor ejecutar el problema en su totalidad en la FPGA. Matmul y Gaussian son los más beneficiados por la ejecución heterogénea ya que rinden bien en ambos dispositivos. Matmul ha dado los mejores resultados con el planificador dinámico y paquetes pequeños, entregando un 77 % del problema a la FPGA. Gaussian por el contrario se beneficia más del planificador hguided y la CPU, ejecutando sólo un 28 % en la FPGA.

Se propone como trabajo futuro la extensión del repositorio para la realización de ejecución 3-heterogénea con CPU, GPU y FPGA, empleando oneAPI para los tres dispositivos cuando Intel dé soporte a la generación de *FatBinaries*, y pudiendo mantener una versión en C++ nativo para problemas ligeros. Sigue existiendo margen de mejora en la optimización de los *kernels* para cada dispositivo, pudiendo hacerse uso de las versiones de biblioteca incluidas con oneAPI e incluso el *profiler* VTune. En el trabajo se han evaluado cinco de los ocho *benchmarks* existentes inicialmente, por lo que se propone experimentar con los restantes.

Capítulo 7

Bibliografía

[1] GORDON E. MOORE. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 1965.

[2] Mark Bohr. A 30 year retrospective on dennard's mosfet scaling paper. *IEEE Solid-State Circuits Society Newsletter*, 12(1):11–13, 2007.

[3] M. Horowitz, E. Alon, D. Patil, S. Naffziger, Rajesh Kumar, and K. Bernstein. Scaling, power, and the future of cmos. In *IEEE InternationalElectron Devices Meeting, 2005. IEDM Technical Digest.*, pages 7 pp.–15, 2005.

[4] Mark D. Hill and Michael R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, 2008.

[5] John Hennessy and David Patterson. A new golden age for computer architecture: domain-specific hardware/software co-design, enhanced security, open instruction sets, and agile chip development,, 2017.

[6] Maria Dávila, Raúl Nozal, Rubén Gran Tejero, María Villarroya, Darío Suárez Gracia, and Jose Bosque. Cooperative cpu, gpu, and fpga heterogeneous execution with enginecl. *The Journal of Supercomputing*, 75, 03 2019.

[7] Intel oneAPI. `https://www.oneapi.com/`.

[8] Raúl Nozal and Jose Luis Bosque. Exploiting co-execution with oneapi: heterogeneity from a modern perspective, 2021.

[9] Francisco Irigoyen, Alfredo Villalba, and Enrique Sedano Algarabel. Implementación de una plataforma hw para la evaluación de predictores e saltos sobre arquitectura sparc v8. 01 2008.

[10] Benedict R. Gaster, Lee Howes, David R. Kaeli, Perhaad Mistry, and Dana Schaa. Chapter 2 - introduction to opencl. In Benedict R. Gaster, Lee Howes, David R.

Kaeli, Perhaad Mistry, and Dana Schaa, editors, *Heterogeneous Computing with OpenCL (Second Edition)*, pages 15–38. Morgan Kaufmann, Boston, second edition edition, 2013.

[11] James Reinders, Ben Ashbaugh, James Brodman, Michael Kinsner, John Pennycook, and Xinmin Tian. In *Data Parallel C++*. Apress, first edition, 2021.

Lista de Figuras

1.1. Evolución del rendimiento de los procesadores desde los años 80 a la actualidad 1
1.2. Flexibilidad frente a rendimiento de las principales tecnologías 2

2.1. Ruta de datos segmentada 7
2.2. Circuito hardware para realizar la multiplicación de la Eq. 2.1 con multiplicadores de dos entradas 8
2.3. Bloques funcionales principales de la FPGA 8
2.4. Programa oneAPI 10
2.5. Opciones de compilación oneAPI 11

3.1. Compilación en FPGA 15

4.1. Planificadores originales 17
4.2. Paralelización con *Parallel_for* 18
4.3. Planificadores nuevos 19

5.1. Comparativa del rendimiento base en cada dispositivo. CPU++ emplea un *core* mientras que oneCPU emplea los 6 disponibles. 22
5.2. Rendimiento en C++ de un hilo frente a seis 24
5.3. Optimización de los *kernels* para FPGA 25
5.4. Optimización del tamaño de paquete para FPGA 26
5.5. Comparación CPU++ y oneFPGA tras optimizar 27
5.6. Rendimiento de los *benchmarks* con el planificador estático 29
5.7. Rendimiento de los *benchmarks* con el planificador dinámico 30
5.8. Rendimiento de los *benchmarks* con el planificador hguided 31

A.1. Tiempo dedicado a cada tarea 44

B.1. Sobrecarga oneAPI 45
B.2. Rendimiento en CPU++ variando el número de hilos 46
B.3. Comparación de 6 hilos CPU++ frente a oneAPI base 47

B.4. Matadd y Rap con planificador dinámico 48
B.5. Matadd y Rap con planificador hguided 48
B.6. Exploración más exhaustiva del parámetro K de hguided para Matmul y Nbody 49

Lista de Tablas

5.1. *Speedups* de las optimizaciones 27
5.2. Speedup y eficiencia heterogénea de cada *benchmark* y planificador . . 32
5.3. Mejor tiempo obtenido para cada *benchmark* en las diferentes etapas del trabajo 33

B.1. Memoria requerida para el tamaño máximo de cada *benchmark* 46

Anexos

Anexos A

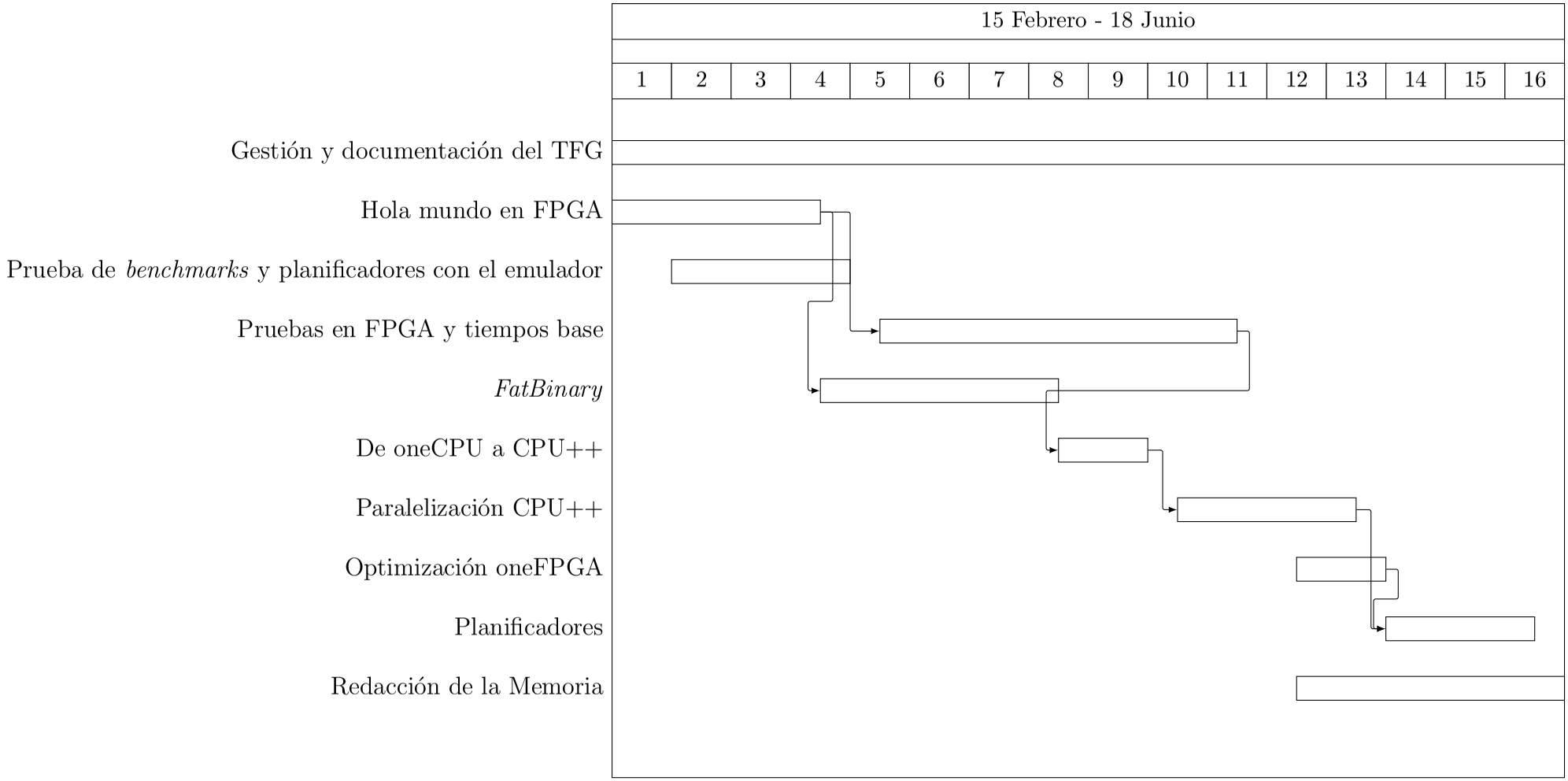
Cronología

Los primeros dos meses se encontraron dos grandes dificultades.

- Primero con la compilación de un simple Hola Mundo en FPGA, que obligó a contactar a Intel tras haber comprobado que todas las variables de entorno que oneAPI especifica estaban inicializadas correctamente. Eventualmente Intel respondió que se trataba de un *bug* con el idioma de los números decimales (representación con una coma o con un punto).
- La segunda consistió en que no se conseguía generar un *FatBinary* para CPU+FPGA siguiendo el ejemplo de la web de Intel y el manual. Tras enviarles un proyecto sencillo para el que fallaba el tutorial cambiaron la web acordemente, pero seguía sin funcionar. Más tarde declararon que no estaban soportados los *FatBinary* aún, por lo que comenzó la traducción a C++ nativo.

Mientras se intentaba compilar el Hola Mundo se probó el coejecutor con el emulador de FPGA, y mientras se probaban los *FatBinary* se compilaron versiones para un solo dispositivo con fin de ir probando los *benchmarks* en FPGA y extrayendo los resultados base.

Tras portar el código a C++ se realizaron todos los experimentos, primero optimizando y luego probando los planificadores de ejecución heterogénea. Finalmente se redactó la memoria.


15 Febrero - 18 Junio
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
Gestión y documentación del TFG
Hola mundo en FPGA
Prueba de *benchmarks* y planificadores con el emulador
Pruebas en FPGA y tiempos base
FatBinary
De oneCPU a CPU++
Paralelización CPU++
Optimización oneFPGA
Planificadores
Redacción de la Memoria

La Figura A.1 representa una estimación del tiempo dedicado a cada tarea, las cuales se pueden agrupar en 5 grupos:

- (25h, 7.5 %) Gestión del TFG: tiempo dedicado a discutir y documentar el progreso y los resultados.
- (80h, 23.8 %) Compilación para FPGA: exploración de los parámetros de configuración para conseguir compilaciones funcionales.
  - Hola mundo en FPGA
  - FatBinary
- (75h, 22.4 %) Probar el coejecutor: probar al completo el código fuente de unican.
  - Pruebas con emulador
  - FPGA y tiempos base
- (105h, 31.4 %) Experimentación: modificación del coejecutor y realización de experimentos.
  - De oneCPU a CPU++
  - Paralelización CPU++
  - Optimización FPGA
  - Planificadores
- (50h, 14.9 %) Memoria: redacción de la memoria.

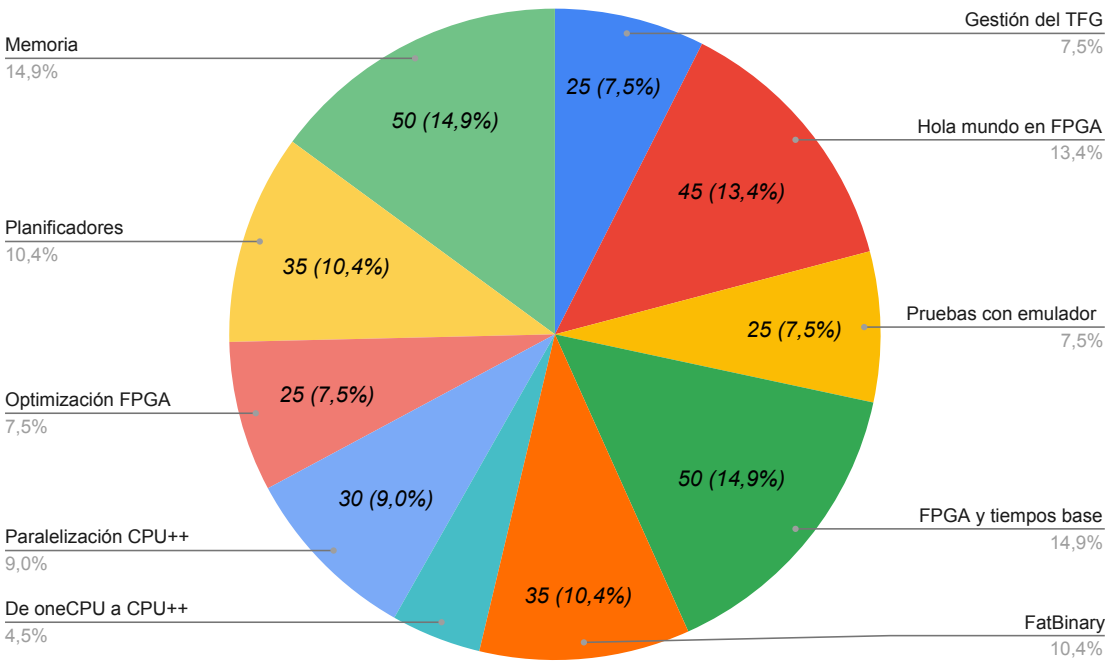


Figura A.1: Tiempo dedicado a cada tarea

Anexos B

Experimentos y Figuras adicionales

B.1. Estudio de la sobrecarga oneAPI

La Figura B.1 representa el tiempo de transmisión de datos a la FPGA, el tiempo de cómputo del *kernel* y el tiempo total hasta que termina el planificador. Se emplea el tamaño máximo de cada *benchmark* y se ejecuta asignando el 100 % del *benchmark* a la FPGA con el planificador estático. Los *kernels* son los optimizados previamente para FPGA (en el apartado 5.2.2).

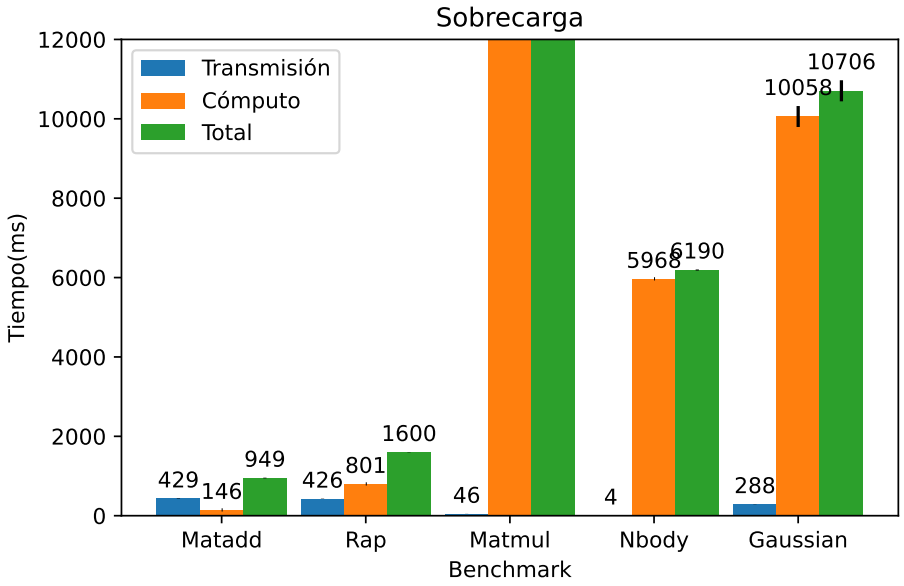


Figura B.1: Sobrecarga oneAPI

Se observa que para Matadd y Rap el tiempo de cómputo solo supone un 15 % y 50 % del tiempo total respectivamente, mientras que en los otros tres supone un mínimo del 94 %. Se puede ver que en el caso de Matadd y Rap la mayor sobrecarga es el tiempo de transmisión, pero no es la única. En todos los *benchmarks* el tiempo total es superior a la suma de transmisión y cómputo ya que el coejecutor tiene que crear la cola y los

buffers del dispositivo. Los tiempos de transmisión son consistentes con la cantidad de memoria transferida, que se representa en la Tabla B.1.

Benchmark	Tamaño	Memoria (GB)
Matadd	13000	2.028
Rap	13000	2.028
Matmul	4096	0.201
Nbody	131072	0.008
Gaussian	13000	1.352

Tabla B.1: Memoria requerida para el tamaño máximo de cada *benchmark*

B.2. Efectividad de la paralelización CPU++

La Figura B.2 muestra el tiempo de ejecución en CPU++ de los *benchmarks* pesados e intermedio, empleando como tamaño de problema el tamaño máximo de cada *benchmark* en la Figura 5.1 y variando el número de hilos. Los problemas se dividen en tantos fragmentos de igual tamaño como hilos. El número de hilos máximo se ha limitado a 6 ya que es el máximo paralelismo que puede gestionar la CPU empleada.

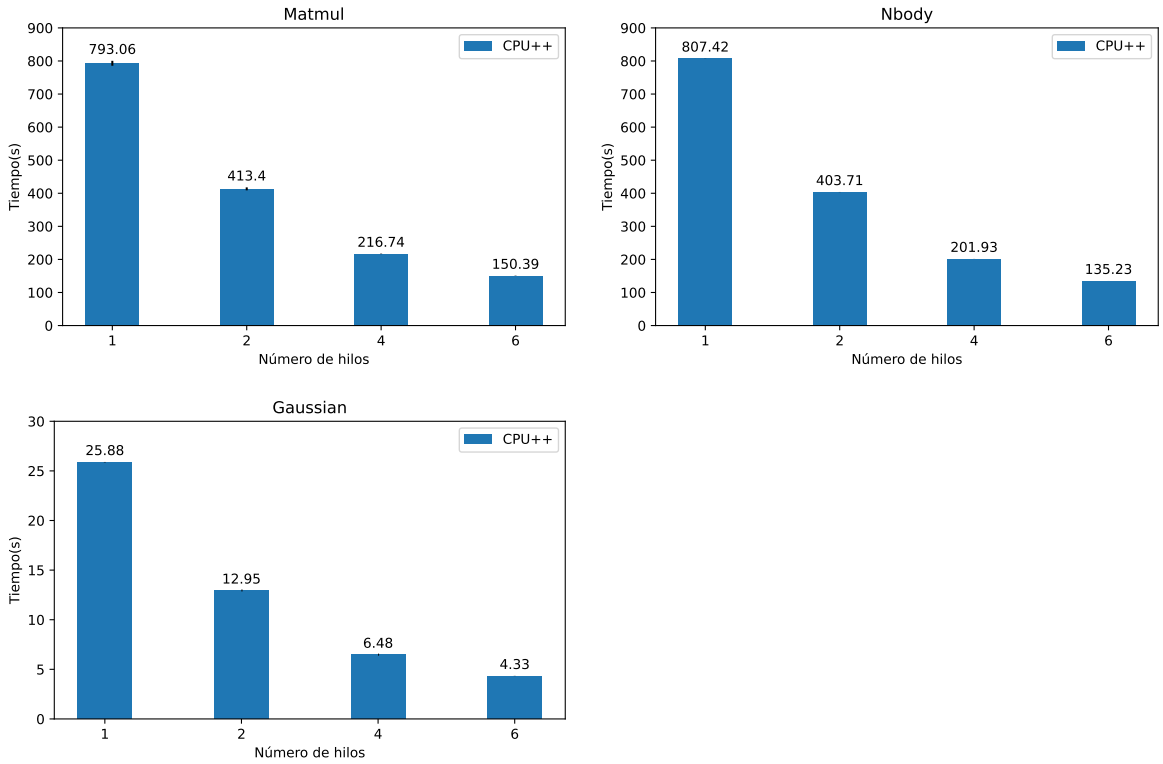


Figura B.2: Rendimiento en CPU++ variando el número de hilos

Se observa que la optimización resulta muy eficiente ya que el tiempo de ejecución se divide por el número de hilos empleados, alcanzando el *speedup* máximo ideal de 6

para Nbody y Gaussian y un 5.3 para Matmul.

B.3. Paralelización oneAPI y nativa

En la Figura B.3 se compara la versión nativa paralelizada (CPU++ 6 hilos) con la versión oneAPI base (que implícitamente están paralelizada), tanto oneCPU como oneFPGA.

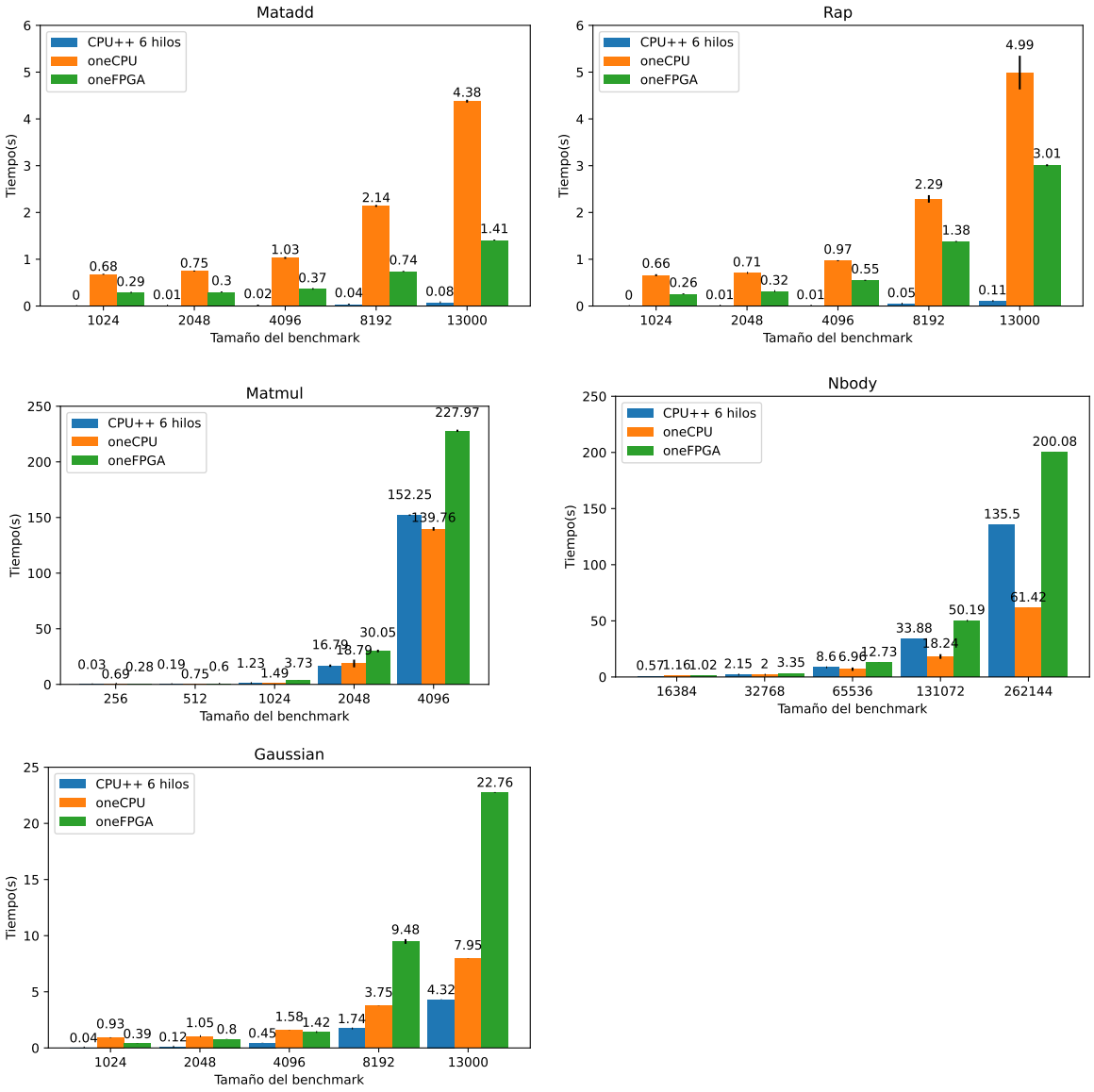


Figura B.3: Comparación de 6 hilos CPU++ frente a oneAPI base

Se observa que, pese a que los resultados son más cercanos, oneCPU sigue siendo mejor para los *benchmarks* pesados (oneFPGA pasa a ser peor que CPU++). En el caso de los *benchmarks* ligeros, CPU++ sigue siendo mejor al no tener tanta sobrecarga. Es en el *benchmark* intermedio (Gaussian) donde se nota el mayor cambio ya que CPU++

pasa de ser 3 veces peor a ser casi 2 veces mejor que oneCPU.

B.4. Gráficas de Matmul y Rap para los planificadores dinámico y hguided

Las Figuras B.4 y B.5 muestran las gráficas de Matadd y Rap para los planificadores dinámico y hguided respectivamente. Como se ha mencionado previamente, existe una gran variabilidad dependiendo de si se asigna algún paquete a la FPGA o no, ya que es considerablemente mas lenta que la versión CPU++ por la sobrecarga de transmisión de datos.

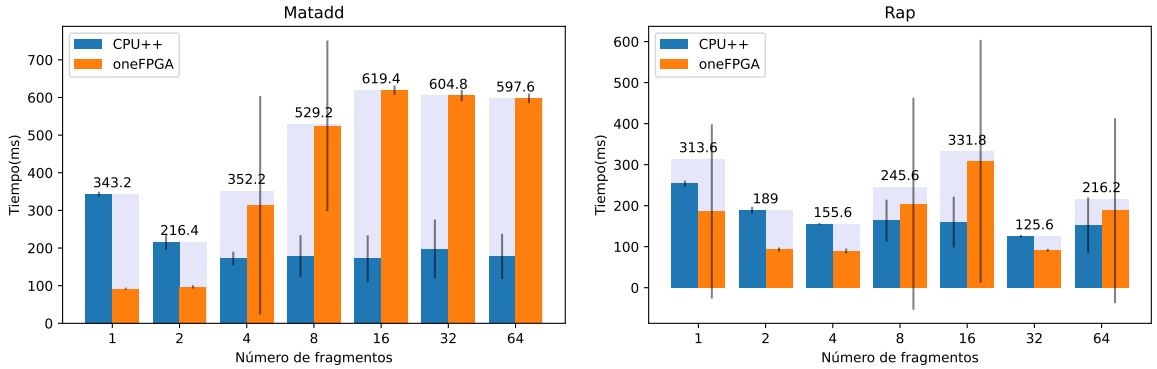


Figura B.4: Matadd y Rap con planificador dinámico

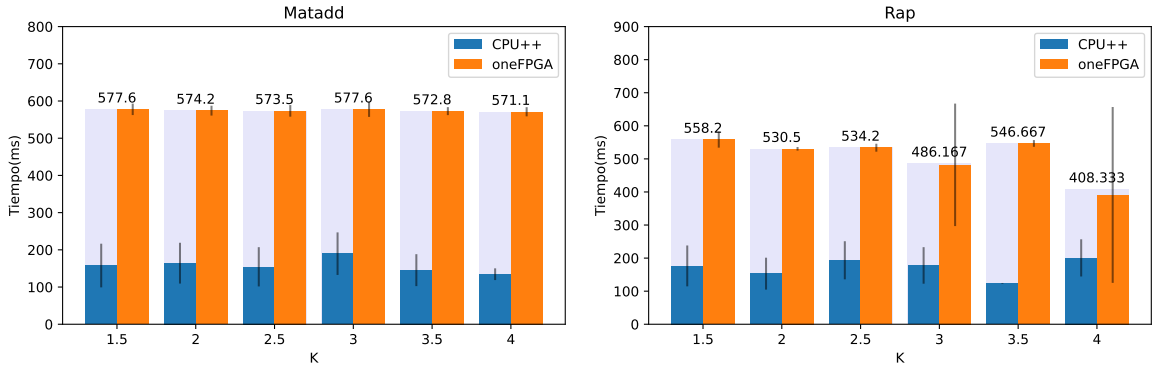


Figura B.5: Matadd y Rap con planificador hguided

Se observan barras de tiempo de oneFPGA bajas cuando no coge ningún fragmento la FPGA, altas cuando siempre coge alguno (es el caso más común para hguided), e intermedias con alta variabilidad cuando en algunas repeticiones del experimento coge y en otras no.

B.5. Exploración exhaustiva del parámetro K de hguided

En la Figura B.6 se muestra una exploración del tiempo de terminación (eje Y) de Matadd y Nbody para valores del parámetro K (eje X) superiores a 4. Pese a que los tiempos de ejecución mostrados se creen verídicos, los resultados de los experimentos de esta exploración no han podido ser verificados todavía, por lo que no se puede garantizar.

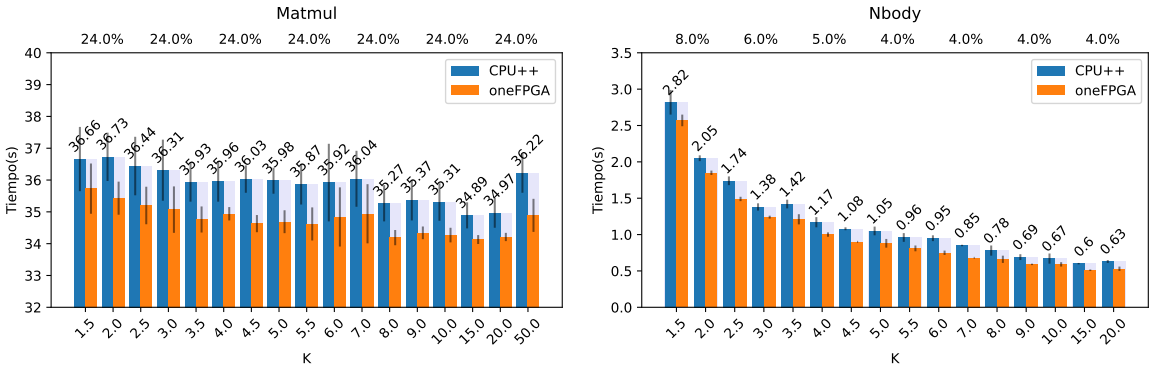


Figura B.6: Exploración más exhaustiva del parámetro K de hguided para Matmul y Nbody

Los resultados demuestran que valores de K superiores consiguen mejores resultados. En Nbody esta mejora es de mayor calibre ya que es el *benchmark* que más se beneficia de dividir el espacio de iteraciones en muchos fragmentos pequeños (5.2.2).