Universidad Zaragoza

Trabajo Fin de Máster

Optimización y Planificación en Sistemas Heterogéneos sobre SYCL/oneAPI

Autor

Raúl Herguido Sevil

Directores

Rubén Gran Tejero

Darío Suárez Gracia

ESCUELA DE INGENIERÍA Y ARQUITECTURA

2023

RESUMEN

Los requisitos actuales de cómputo y consumo energético requieren del uso de aceleradores de propósito específico en los que descargar trabajo de la CPU. Estos sistemas heterogéneos, compuestos por diversos dispositivos de cómputo, son difíciles de programar por las diferencias entre dispositivos. Existen modelos de programación como oneAPI para facilitar su programación.

Este trabajo explora y evalúa las diferentes opciones de optimización de *kernels* que ofrece oneAPI para FPGAs en múltiples *benchmarks*, centrándose en el patrón *parallel_for*. También evalúa la posibilidad de realizar coejecución CPU-FPGA con distintos planificadores. El soporte de oneAPI no está maduro, particularmente para la FPGA y su coejecución, por lo que se ha requerido un gran esfuerzo para entender y probar el funcionamiento de las características de oneAPI.

La implementación de diferentes optimizaciones, generalmente relacionadas con el acceso a memoria y el paralelismo en ejecución de bucles, ha reportado *speedups* desde 7.3 hasta 634.26 frente a versiones poco optimizadas. Se corrobora que pese a haber portabilidad funcional no la hay en rendimiento. Además, esta portabilidad se ve reducida con la implementación de optimizaciones, al igual que lo hace su usabilidad por requerir tamaños de problema concretos.

Las extensas pruebas de ejecución heterogénea con ambos dispositivos ejecutando oneAPI han funcionado para ejemplos sencillos pero no con *kernels* optimizados por lo que no se considera soportada durante la realizacion de este trabajo.

La potencia de la FPGA al emplear *kernels* optimizados dificulta la coejecución CPU-FPGA por la disparidad de rendimiento entre dispositivos, por lo que ninguno de los planificadores evaluados (estático, dinámico, hguided) ha resultado efectivo.

Se han podido probar las últimas versiones tanto *software* como *hardware* haciendo uso de la plataforma Devcloud proporcionada por Intel.

Índice

Capítulo 1

Introducción

1.1. Motivación

Existe desde el nacimiento de la computación un incesante incremento en la demanda de potencia de cómputo, y, para satisfacer dicha demanda, los ingenieros han de conseguir incrementos iguales o superiores en las prestaciones de los sistemas de computación. Dichos incrementos de rendimiento y las técnicas empleadas se muestran en la Figura 1.1.

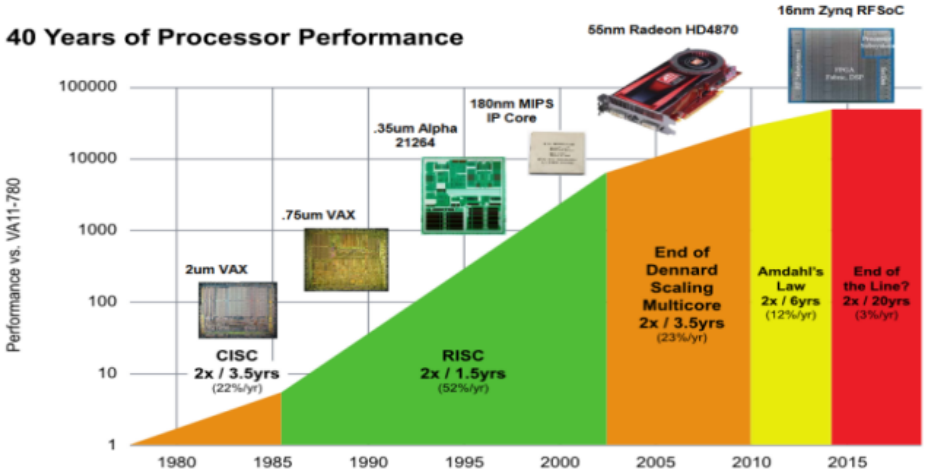


Figura 1.1: Evolución del rendimiento de los procesadores desde los años 80 a la actualidad

Fuente: *Computer Architecture: A Quantitative Aproach*

En el periodo de 1985 a 2005, se comenzó por aumentar el rendimiento de un procesador, doblando el número de transistores cada dos años [1] y reduciendo el tamaño de estos para conservar el mismo área de chip. También se aumentaba la frecuencia del procesador, ya que los transistores consumían menos energía al reducir su tamaño y emplear menor voltaje [2]. Adicionalmente, se desarrollaron multitud de mejoras en las micro-arquitecturas para aprovechar el paralelismo a nivel de instrucción.

Eventualmente se alcanzaron limitaciones físicas en la reducción del tamaño de los

transistores, el aumento de la corriente de fuga impidió seguir reduciendo el voltaje y aumentar la frecuencia, lo que conllevó un gran aumento del coste monetario y problemas de disipación de calor [3].

Cuando estos problemas frenan el incremento de rendimiento de un procesador, se cambia de filosofía y se comienza a explotar el paralelismo al emplear múltiples unidades de cómputo para trabajar cooperativamente. Esta solución aumenta drásticamente el rendimiento sobre la parte paralelizable del cómputo, pero no resulta efectiva cuando la sección secuencial es la limitante [4].

En el presente, la potencia de los procesadores es insuficiente en muchos casos [5], y es por ello que hacen uso de dispositivos de cómputo a los que descargar trabajo. Estos dispositivos de cómputo, a diferencia de los procesadores de propósito general, son de dominio específico por lo que el conjunto de tareas que puede realizar es reducido pero tienen grandes prestaciones energéticas y temporales ejecutando las tareas para las que han sido diseñados.

En la Figura 1.2 se comparan algunos de los dispositivos más comunes en base a su flexibilidad y rendimiento.

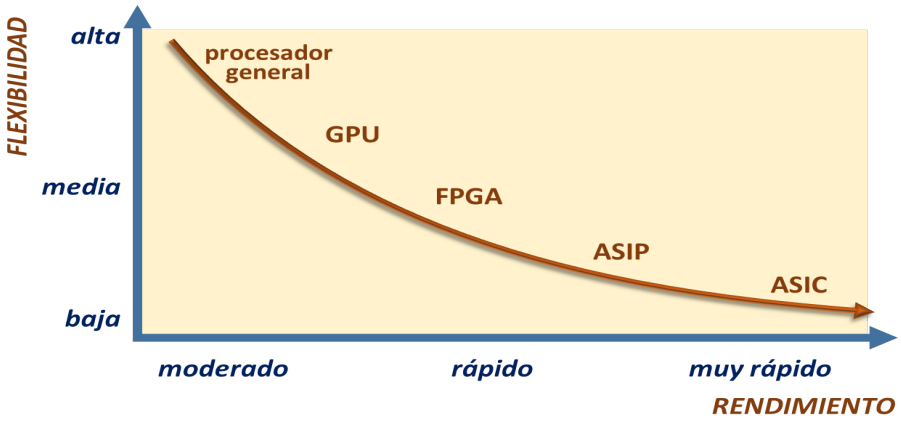


Figura 1.2: Flexibilidad frente a rendimiento de las principales tecnologías

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

El uso de estos dispositivos da lugar a sistemas heterogéneos, que se podrían definir como aquellos en los que conviven distintos dispositivos de cómputo, los cuales se emplean en la actualidad en todos los ámbitos de la computación, desde sistemas HPC hasta teléfonos móviles.

Si bien los sistemas heterogéneos están ampliamente extendidos por su rendimiento, su complejidad es mayor [6], siendo necesario tener en cuenta las diferencias entre las características de los distintos dispositivos que los componen: arquitectura, memoria, frecuencia de reloj, lenguaje de programación, bibliotecas, frameworks...

Idealmente, toda la complejidad añadida sería transparente al desarrollador y manejada por un software intermedio que interaccione con los diferentes dispositivos, lidiando con las características de cada uno y repartiendo eficientemente el trabajo [7].

Es este software intermedio el que pretende proporcionar Intel a través de su nuevo modelo de programación oneAPI [8], 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 intenta avanzar hacia la programabilidad de sistemas heterogéneos a nivel de portabilidad, no está tan claro si lo será en en portabilidad de rendimiento. Además 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 optimizar un conjunto de benchmarks con oneAPI para su ejecución en FPGA, evaluando las opciones de optimización de código existentes y permitiendo además valorar la portabilidad de rendimiento de oneAPI a FPGA al comparar el código base con el optimizado.

Con los benchmarks optimizados se realizará ejecución heterogénea entre CPU y FPGA, evaluando múltiples políticas de balanceo de carga.

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

- ¿Qué rendimiento tienen los *kernels*[1] oneAPI sin optizizar en FPGA?
- ¿Cómo se pueden optimizar los *kernels* oneAPI para FPGA? ¿Cómo de sencillo y efectivo resulta?
- ¿Podemos hacer cooperar a dos dispositivos para ejecutar los *benchmarks*? ¿Es mejor que ejecutar en uno único?

1.3. Alcance

En este TFM se ha partido de un trabajo previo [9] en el que se desarrolla una herramienta con múltiples *benchmarks* y planificadores de ejecución, funcional para la

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

ejecución heterogénea C++-FPGA con oneAPI. Partiendo de este código base y los objetivos del apartado anterior:

- Se ha extendido y migrado el proyecto a la plataforma devcloud de Intel.
- Se ha actualizado el código para operar con la última versión de oneAPI.
- Se han explorado y evaluado concienzudamente múltiples técnicas para la optimización de *kernels* en FPGAs. Así es posible resumir aquellas con mayor beneficio.
- Se han estudiado y probado exhaustivamente las herramientas ofrecidas por Intel para ejecución heterogénea, incluyendo el reporte de múltiples errores de las mismas.
- Se ha realizado ejecución heterogénea con múltiples políticas de balanceo de carga.

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 describe la plataforma empleada, se detalla la metodología seguida al realizar los experimentos y los *benchmarks* utilizados. Una vez definido el entorno del trabajo, en el Capítulo 4, se describen las diferentes optimizaciones consideradas y donde se aplican. En el Capítulo 5, se describen la herramienta de coejecución heterogénea CPU-FPGA que se empleará. Antes de finalizar, en el Capítulo 6 se exponen los diferentes experimentos realizados y sus resultados, tanto de optimización como de coejecución. Por último, en el Capítulo 7 se extraen las principales conclusiones del TFM y se proponen líneas de trabajo futuro.

Capítulo 2

Estado del arte

2.1. Sistemas Heterogéneos

Los sistemas heterogéneos están ampliamente extendidos en la actualidad. Resulta raro encontrarse con ordenadores personales o teléfonos móviles que no incluyan una tarjeta gráfica o diversidad de aceleradores para tareas concretas.

2.2. *Hardware* en sistemas heterogéneos

En este apartado se describe de manera simplificada la arquitectura de las CPUs y FPGAs, los dos dispositivos principales en este trabajo, permitiendo comprender la diferencia de rendimiento y flexibilidad entre ambos dispositivos.

2.2.1. CPU

Las CPUs (*Central Processing Unit*) son componentes *hardware* esenciales en casi todos los dispositivos electrónicos. Esto se debe a su alta flexibilidad ya que, empleando combinaciones de instrucciones sencillas, son capaces de implementar cualquier comportamiento.

Su organización suele estar basada en una ruta de datos segmentada (simplificada en la Figura 2.1) en la que se leen, decodifican y ejecutan instrucciones, guardando los resultados convenientemente. Es común el uso de mejoras microarquitectónicas como: ejecución fuera de orden, predicción de saltos, o ejecución superescalar. El objetivo de estas optimizaciones, y muchas otras, es explotar al máximo el paralelismo entre instrucciones, aprovechar todo el *hardware* posible para lograr el máximo rendimiento en las CPUs.

Cada CPU implementa una ISA (*Instruction Set Architecture*) en la que se documentan las caracterísicas subyacentes necesarias para generar binarios ejecutables, como son: los tipos de datos soportados, la gestión de memoria, los registros existentes...

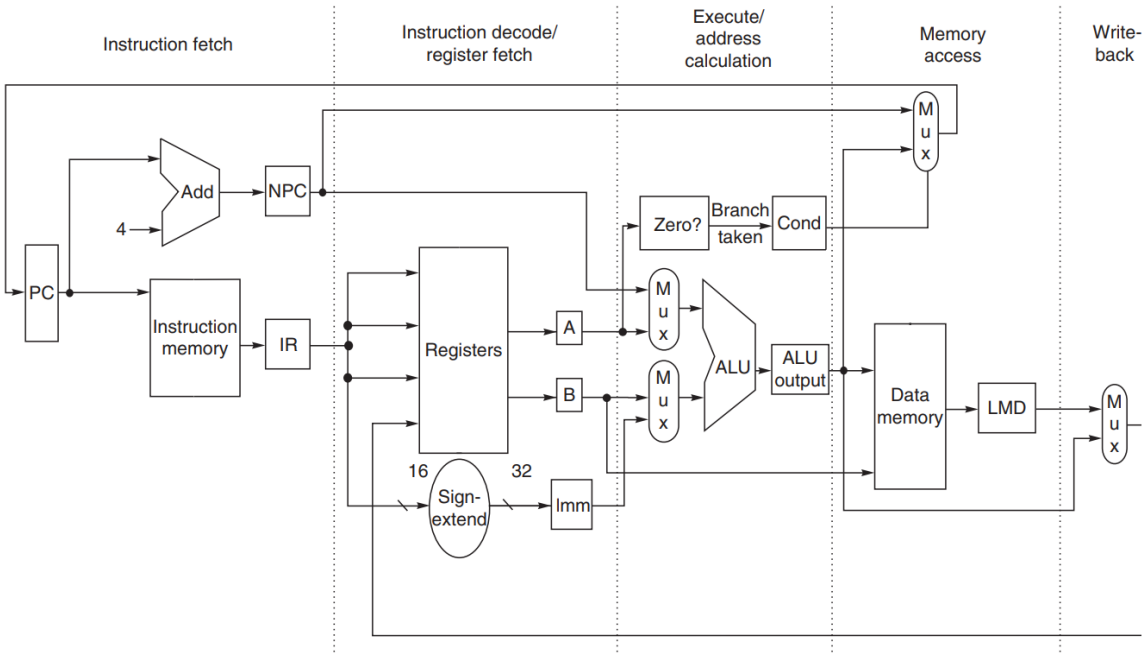


Figura 2.1: Ruta de datos segmentada

Fuente: *Computer Architecture: A Quantitative Aproach*

El uso de una ruta de datos común para la ejecución de diversos tipos de instrucciones siguiendo cualquier combinación conlleva a que, pese a ejecutar gran cantidad de instrucciones por segundo, la CPU pueda resultar muy ineficiente para secuencias de instrucciones con dependencias o que compiten por los mismos recursos. Por ejemplo, para ejecutar la multiplicación de la ecuación 2.1, sería necesario realizar, entre otras, dos instrucciones de multiplicación, que deben atravesar la ruta de datos entera y existiendo una dependencia que retrasa la ejecución de la segunda hasta que la primera proporciona su resultado. Si se diseñase una ruta de datos con esta operación en mente, se añadirían dos unidades de multiplicación encadenadas como en la Figura 2.2, para poder realizar la operación entera con un solo recorrido de la ruta de datos.

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

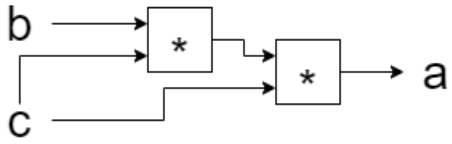


Figura 2.2: Circuito hardware para realizar la multiplicación de la Eq. 2.1 con multiplicadores de dos entradas

2.2.2. FPGA

El principal dispositivo acelarador usado en este trabajo es una FPGA (*Field-Programmable Gate Array*). Las FPGAs son dispositivos *hardware* reconfigurables cuya popularidad ha aumentado como consecuencia del incremento en el número de componentes configurables de los que disponen y debido también a una reducción en su precio.

Como muestra la Figura 2.3, las FPGAs están compuestas por multitud de bloques lógicos reprogramables para implementar desde puertas lógicas a funciones combinacionales complejas. Una jerarquía de interconexión permite configurar qué elementos se enlazan 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).

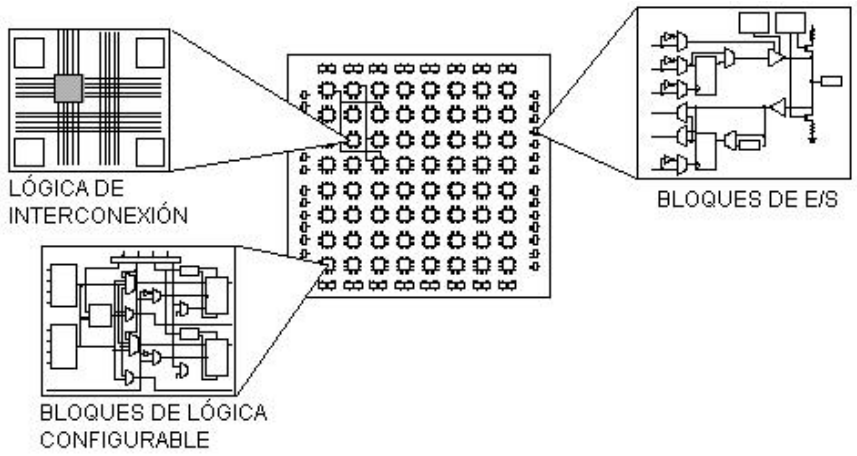


Figura 2.3: Bloques funcionales principales de la FPGA

Fuente: Implementación de una plataforma HW para la evaluación de predictores e saltos sobre arquitectura SPARC v8 [10].

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.

Originalmente la programación de las FPGAs se llevaba a cabo únicamente empleando lenguajes de descripción de *hardware* como VHDL. Hoy en día 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 evaluar 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 heterogéneo son las herramientas software que permiten su programación. En este apartado se presentan distintas alternativas.

2.3.1. OpenCL y CUDA

Previamente a oneAPI ya existían otros modelos de programación, como OpenCL y CUDA [11, 12], en los que se especifica la existencia de 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, ...

Aunque CUDA es específico para GPUs de Nvidia, OpenCL es un estándar abierto para ser utilizado por cualquier acelerador, incluido FPGAs. OpenCL se anunció por primera vez en la conferencia *SIGGRAPH* de 2008, y consigue definir un lenguaje común entre diferentes dispositivos, pero tiene múltiples desventajas:

- 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++) [13]. 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 reciente y no maduro, especialmente para FPGAs, 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 }
```

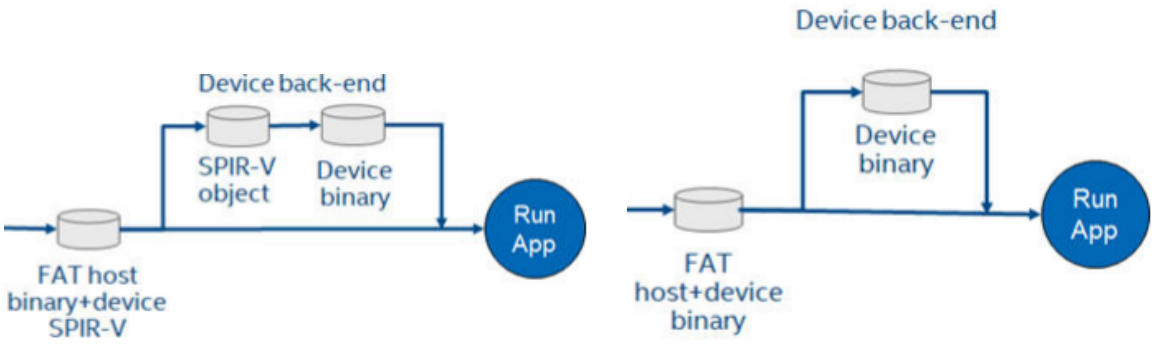
Figura 2.4: Programa oneAPI

Fuente: oneAPI Programming Model Manual
, `https://www.intel.com/content/www/us/en/develop/documentation/oneapi-programming-guide/top.html`

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

En la Figura 2.5a se observa que la compilación JIT genera un fichero con una parte



(a) Compilación *Just In Time* (b) Compilación *Ahead Of Time*

Figura 2.5: Opciones de compilación oneAPI

Fuente: oneAPI Programming Model

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, en binario ejecutable o bitstream de la FPGA, Intel se refiere a ellos como *FatBinary*. Las versiones testeadas de *FatBinary* han resultado imposibles de manejar por la cantidad de errores que generaban.

Paralelismo en oneAPI

En lugar de expresar la ejecución como bucles unidimensionales ejecutados secuencialmente en una única unidad de cómputo, oneAPI implementa el patrón *parallel_for*. Con el patrón *parallel_for* se expresa que las diferentes instancias (*work-items*) del *kernel* (que corresponderían a cada iteración en un bucle regular) son independientes entre sí y se pueden ejecutar en cualquier orden. De este modo se delega en oneAPI la planificación de la ejecución y el reparto de iteraciones entre las unidades de cómputo del dispositivo acelerador seleccionado. El patrón *parallel_for* se asocia a un único dispositivo por medio de una cola de tareas, por lo que no es posible repartir instancias entre múltiples dispositivos. En la Figura 2.6 se muestra una multiplicación de matrices dónde cada *work-item* realiza el cálculo de un valor de la matriz resultado.

Para tener más control sobre el orden de ejecución de los *work-items* existen los *kernels parallel_for ND-range* que permiten expresar cierta localidad entre instancias. Como muestra la Figura 2.7, en los *kernels ND-range* los *work-items* se agrupan en

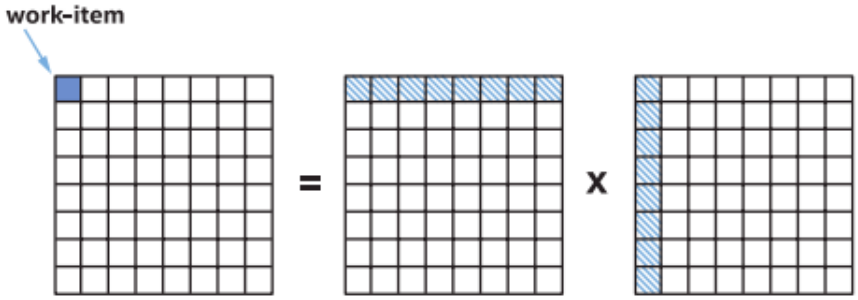


Figura 2.6: Multiplicación de matrices con *parallel_for*

Fuente: Data Parallel C++ - Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL

work-groups. Los *work-items* de un mismo *work-group* se ejecutan simultáneamente, tienen acceso a memoria local compartida por el grupo, a funciones de sincronización de grupo (como *barriers*) y a otras funciones de comunicación de grupo. Cuando se emplean directivas específicas de *work-groups*, el motor de ejecución de oneAPI garantiza que la ejecución de sus *work-items* es simultánea y no se entrelaza con la de otros *work-groups*. No obstante, el orden de ejecución de los *work-groups* o de cada *work-item* dentro de un mismo *work-group* sigue siendo indefinido.

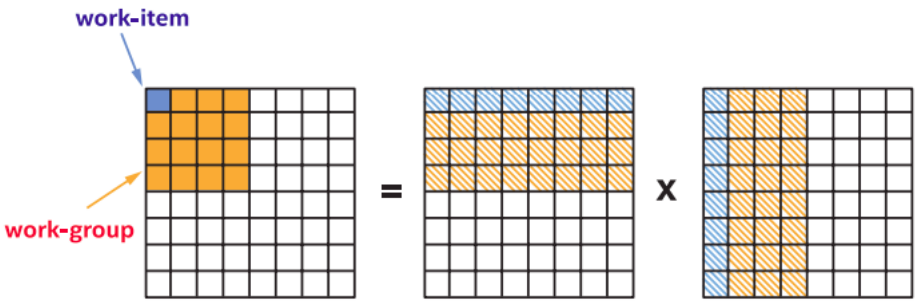


Figura 2.7: Multiplicación de matrices con *parallel_for ND-range*

Fuente: Data Parallel C++ - Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL

En caso de que el problema a implementar no sea altamente paralelizable o requiriera de situaciones especiales que empeoren el uso de *parallel_for*, se puede emplear *kernels* regulares de una única instancia, llamados *single-task*, que no expresan paralelismo.

Capítulo 3

Metodología

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

3.1. Devcloud

Todos los experimentos presentados en este trabajo se han llevado a cabo en Devcloud[1], aunque también se ha empleado una máquina local Macizo[2] para realizar pruebas. Devcloud es una plataforma proporcionada por Intel en la que se incluyen máquinas equipadas con diversidad de dispositivos aceleradores y las últimas versiones de todas las herramientas software necesarias para desarrollar aplicaciones para cualquier arquitectura evitando la costosa puesta en marcha y administración de este tipo de sistemas.

Si bien se requiere de la aprobación del proyecto o motivo de uso para poder acceder a Devcloud, su uso es gratuito siendo una excelente plataforma para probar a desarrollar en dispositivos aceleradores sin necesidad de invertir la gran cantidad de dinero que supondría comprarlos.

Como se muestra en la Figura 3.1, Devcloud cuenta con una serie de nodos *login* a los que se conectan los usuarios por medio de la terminal o entorno de desarrollo y en los que diseñar y mantener código. Para compilar y ejecutar experimentos se someten los *scripts* correspondientes a la cola de trabajos de Devcloud, la cual comprobará las etiquetas descriptivas especificadas al someter el trabajo y buscará una máquina de cómputo operativa que posea dichas etiquetas en la que ejecutar el *script*, devolviendo los resultados a la máquina *core*. Entre estas etiquetas suele incluirse el tipo de dispositivo acelerador que debe poseer el nodo computacional.

Si bien Devcloud ofrece muchas ventajas, también tiene inconvenientes ya que

[1] `https://www.intel.com/content/www/us/en/developer/tools/devcloud/overview.html`

[2] Macizo es una máquina del Grupo de Arquitectura de Computadores de la Universidad de Zaragoza equipada con una Stratix 10 GX a la que se tiene acceso

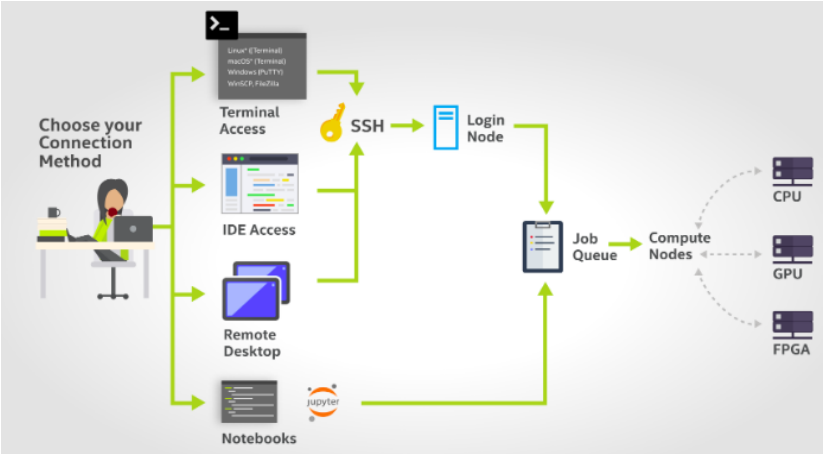


Figura 3.1: Infraestructura del Devcloud

Fuente: *Intel DevCloud for oneAPI*, `https://devcloud.intel.com/oneapi/`

no puedes elegir el momento en el que se aplican cambios de versiones, comandos o bibliotecas, lo cual ha resultado inconveniente en este trabajo. Además, las máquinas de cómputo se comparten entre todos los usuarios, por lo que pueden estar ocupadas y ralentizar el desarrollo del trabajo. Además, las máquinas se reinician o apagan temporalmente de manera frecuente, lo cual es especialmente problemático cuando se deben realizar compilaciones de varias horas. Es por estos inconvenientes y por la disponibilidad inicial de macizo que el desarrollo del trabajo se comenzó en local. No obstante, algunas de las directivas de optimización requerían de versiones más modernas de oneAPI que la instalada en macizo y se decidió cambiar al *cloud* ya que, al ser macizo una máquina compartida, no se podía modificar en ese momento la versión instalada.

En este trabajo se ha empleado la FGPA Intel Stratix 10 GX, para la cual existen 3 máquinas operativas en Devcloud. Dichas máquinas poseen una CPU Intel Xeon Platinum 8256 (8 núcleos, 2 hilos por núcleo, 3.80GHz, 16.5 MB cache y 1.45 TB RAM) y una FPGA Intel Stratix 10 GX[3] con 32GB de memoria principal, representada en la Figura 3.2 y con los recursos de la Tabla 3.1. Se emplea la versión 2023.0.0 de oneAPI, la 2022.3.1 de Intel FPGA SDK y la 19.2 de Quartus. Se aplica el máximo nivel de optimización, `-O3`, en todas las compilaciones.

3.2. *Benchmarks* utilizados

Se han evaluado cuatro *benchmarks* que se describen a continuación, correspondientes a algoritmos representativos de uso común. El código original proviene de Nozal y Bosque [14] y está diseñado para ejecución en CPU y GPU. Todos los *benchmarks*

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

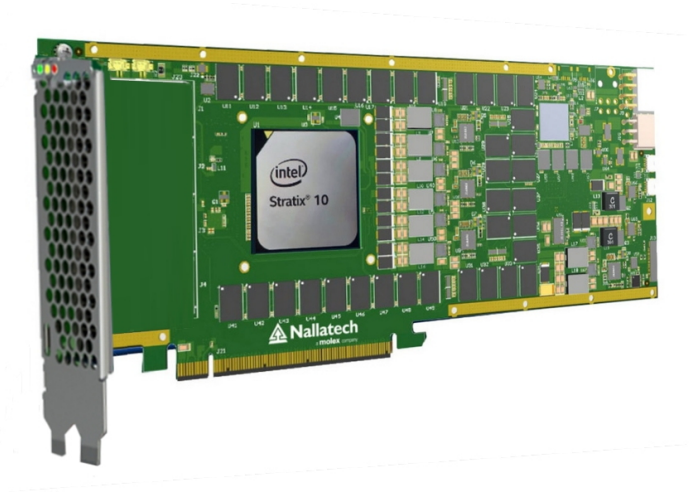


Figura 3.2: BittWare 520C Intel Stratix 10 FPGA accelerator
`https://www.zerif.co.uk/fpga-main-board-accelerators/intel-stratix-10/bittware-520c-intel-stratix-10-gx-2800-10-tflops`

	Stratix 10 Gx
Elementos lógicos (LE)	2753000
Módulos lógicos adaptables (ALM)	933120
Registros del módulo lógico adaptativo (FF)	3732480
Bloques de procesamiento de señal digital (DSP)	5760

Tabla 3.1: Recusros de la FPGA Stratix 10 GX 2800

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 la mayor potencia de 2 cuyo tamaño quepa en la memoria de la FPGA y de modo que tenga un tiempo de ejecución razonablemente alto pero no excesivo.

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.

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.

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.

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.

3.3. Medición del tiempo de ejecución

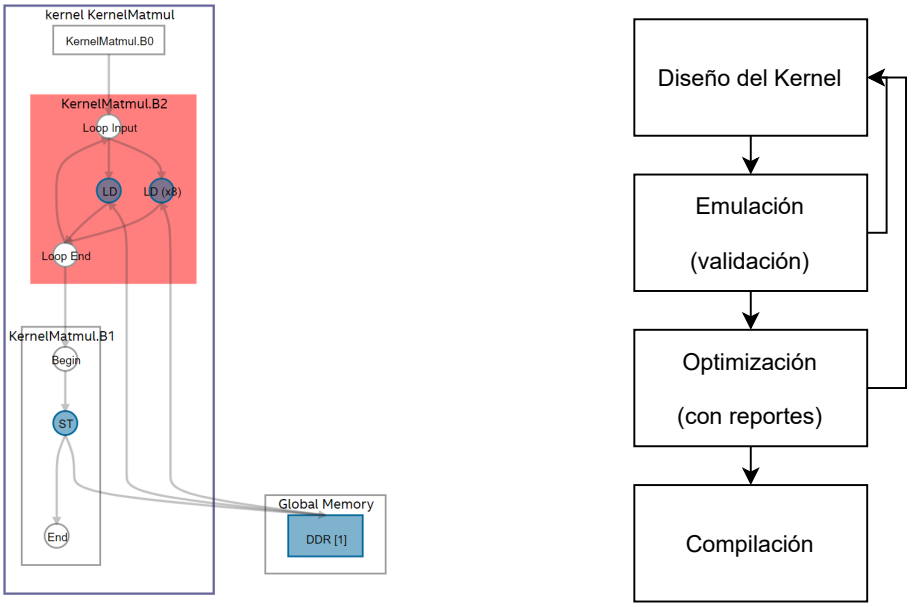
Para cada experimento se mide la diferencia de tiempo entre el momento en que se llama al planificador y el momento en el que este termina. Las mediciones emplean la biblioteca "*chrono*" de C++. Con fin de evitar una alta variabilidad en los resultados, cada experimento se repite múltiples veces[4], 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. Por tanto es el último paso a realizar, cuando se está convencido de que el diseño del *kernel* es el adecuado. 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

[4]Se calculó sobre los experimentos base (Sección 6) 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.

iniciación, desenrollado,...) o la frecuencia de reloj máxima estimada. La Figura 3.3a 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 3.3b. Este flujo resulta de gran utilidad ya que evita realizar el lento proceso de compilación hasta que no se considera que el *hardware* a generar pueda ser suficientemente rápido.



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

(b) Flujo de compilación de FPGA

Figura 3.3: Compilación en FPGA

Capítulo 4

Optimización

En este capítulo se describen las diferentes cuestiones a tener en cuenta para la optimización de los diferentes *kernels* para su ejecución en la FPGA. Así mismo, se analiza el impacto de estas técnicas sobre los *benchmarks*. A lo largo de este TFM se han explorado multitud de técnicas para mejorar el rendimiento de código oneAPI en FPGA generalmente descritas en la guía de Intel[1], pero por limitaciones de espacio, solo se van a mostrar aquellas que han mostrado más potencial de mejora.

4.1. Rendimiento en FPGA

El rendimiento de un *kernel* en FPGA se va a ver condicionado por cuatro factores principales, que serán objetivo principal de las optimizaciones:

- Número de operaciones que se realizan en paralelo.
- Ancho de banda a memoria de la implementación.
- Número de operaciones por ciclo de reloj que realiza el *hardware*.
- Frecuencia del reloj.

La síntesis de un *kernel* en una FPGA adopta la forma de una FIFO de procesamiento. El trabajo a realizar se divice en elementos, *items*, que son procesados en esta FIFO por etapas o fases. El tiempo de ejecución de esta organización puede ser modelado analíticamente con la siguiente ecuación[15]:

$$T_{ejecucion} = \frac{(items - 1) \times II \times \frac{1}{F}}{vector\_factor \times unroll\_factor} \tag{4.1}$$

Donde *items* representa el número de elementos a procesar por el kernel, II la tasa de iniciación y F la frecuencia del kernel. La tasa de iniciación es el número de ciclos que transcurren entre que 2 *items* consecutivos pueden empezar su ejecución.

[1] https://www.intel.com/content/www/us/en/develop/documentation/oneapi-fpga-optimization-guide/top.html

4.2. Guías de diseño

Se presentan a continuación una serie de guías de diseño de *kernels* que se han tenido en cuenta en todo momento para facilitar que el compilador sea capaz de analizar eficientemente el código y generar el mejor diseño de la FIFO de procesamiento posible. Además, se comentará cuando sea pertinente a que término de la ecuación 4.1 afectan.

Evitar solapamiento de punteros Si los *kernels* reciben varios punteros a memoria como argumentos, el compilador no puede descartar a priori que distintos punteros accedan a la misma posición de memoria. Aunque este comportamiento es raramente deseado por el programador, acaba limitando el análisis de dependencias necesario para generar código optimizado.

Suponiendo un código de ejemplo como el de la Figura 4.1, es posible que los punteros no compartan direcciones de memoria como se ejemplifica en la Figura 4.2a y por tanto sea completamente paralelizable. Si no se ayuda al compilador este no puede asumir que no se esté dando el caso representado en la Figura 4.2b, donde se comparten la mayoría de direcciones de memoria y se requiere una ejecución secuencial para obtener un resultado correcto.

Para facilitar el análisis del compilador, como se hace en la Figura 4.1, podemos especificar al compilador que ninguno de los argumentos se solapan mediante la directiva *kernel_args_restrict*. Esto afectará reduciendo el II y permitiendo aumentar el *vector_factor* y/o el *unroll_factor*

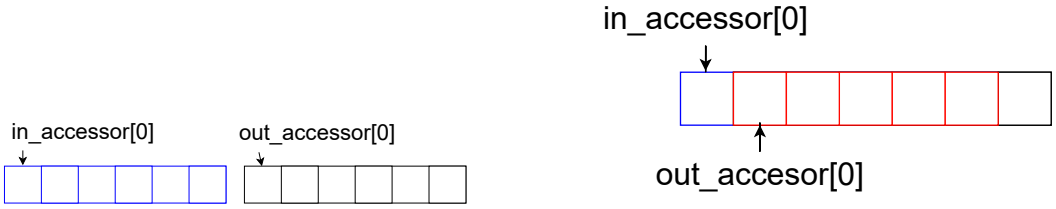
```
device_queue.submit([&](handler& cgh) {
  // create accessors from global memory
  accessor in_accessor(in_buf, cgh, read_only);
  accessor out_accessor(out_buf, cgh, write_only);

  // run the task (note the use of the attribute here)
  cgh.single_task<KernelArgsRestrict>([=]() [[intel::kernel_args_restrict]] {
    for (int i = 0; i < N; i++) {
      out_accessor[i] = in_accessor[i];
    }
  });
});
```

Figura 4.1: Uso del atributo kernel_args_restrict

Fuente: *oneAPI DPC++ FPGA optimization guide*

Se puede especificar, con grano más fino que *kernel_args_restrict*, que la memoria accedida por un *accessor* en concreto no va a ser modificada por otros, sino que siempre se hará indexando por medio de ese mismo *accessor*. Para ello se emplea la directiva *no_alias*, como se muestra en la Figura 4.3.

El uso de *kernel_args_restrict* equivale al uso de *no_alias* para todos los argumentos.



(a) Punteros no solapados (b) Punteros solapados

Figura 4.2: Solapamiento de punteros

```
ext::oneapi::accessor_property_list PL{ext::oneapi::no_alias};
accessor acc(buffer, cgh, PL);
```

Figura 4.3: Uso del atributo no_alias

Fuente: *oneAPI DPC++ FPGA optimization guide*

Construir bucles "bien formados" Emplear bucles con incrementos y condiciones de salida sencillos ayuda al compilador a analizarlos eficientemente, generando mejor diseño que permitirá que sucesivas iteraciones del bucle se lancen con menor latencia de iniciación (reducción de II). Si nos fijamos en la implementación original del bucle de gaussian de la Figura 4.4a podemos observar que se trata de un bucle "mal formado" ya que el valor de la variable *middle* no se conoce en tiempo de compilación (corresponde con la mitad truncada del tamaño del filtro) y existen condicionales para terminar la iteración prematuramente con `continue`. La Figura 4.4a se asemeja al *kernel* final empleado, con un bucle "bien formado".

```
for (int i = -middle; i <= middle; ++i){ // rows
    for (int j = -middle; j <= middle; ++j){ // columns
        int h = r + i;  int w = c + j;
        if (h > height || h < 0 || w > width || w < 0)
            continue;
        float pixelX = input[w + cols * h].x() // current pixel
```

(a) Bucles "mal formados"de gaussian

```
#define FILTERWIDTH 5
// *fusionados y desenrollados*
for (int i = 0; i < FILTERWIDTH; ++i){ // rows
    for (int j = 0; j < FILTERWIDTH; ++j){ // columns
        float pixelX = input_local[local_x + i][local_y + j].x());
```

(b) Bucles "bien formados"de gaussian

Figura 4.4: Cambio en el código de gaussian

Minimizar las dependencias entre iteraciones La existencia de instrucciones que dependen de otras instrucciones ejecutadas en iteraciones previas reduce las prestaciones del kernel puesto que la instrucción dependiente no podrá ejecutarse hasta que la dependencia se resuelva, y por tanto se limita el paralelismo alcanzable. Resolver estas dependencias suele aumentar mucho la tasa de iniciación del modelo.

Fusionar bucles anidados Convertir bucles perfectamente anidados en un único bucle equivalente reduce el uso de hardware y la sobrecarga para iniciar iteraciones.

Declarar las variables en el ámbito más cercano posible El compilador tiene que asegurarse de mantener el valor de las variables a lo largo de todo el ámbito en el que están declaradas, por lo que gastará mayor cantidad de recursos *hardware* para aquellas que estén declaradas en un ámbito más amplio del necesario.

4.3. Técnicas de optimización evaluadas

Si bien se han realizado optimizaciones menores siguiendo las guías de diseño e implementado alguna otra técnica puntualmente, se describen a continuación las técnicas implementadas con mayor impacto en el código y rendimiento. Dichas técnicas se aplican incrementalmente en los experimentos finales.

4.3.1. Usar memoria local para accesos repetidos

Existen múltiples tipos de memoria en oneAPI:

- **Memoria global**: la memoria correspondiente a *buffers* asignados en el *host*. Tiene un gran tamaño, pero una mala latencia y ancho de banda ya que la FPGA ha de acceder a ella a través del *hardware* de interconexión.
- **Memoria local privada**: se encuentra implementada dentro del *kernel* posiblemente como registros o en bloques RAM. Es la empleada al declarar variables dentro del *kernel*.
- **Memoria local de grupo**: similar a la memoria local privada, pero en este caso es visible y compartida por todos los *work-items* del *work-group*. Se declara con la directiva *group_local_memory_for_overwrite*.

El mal uso de la memoria es uno de los mayores problemas de rendimiento al trabajar con una FPGA ya que existe una gran penalización por acceder a memoria global, que es la memoria accedida por defecto en un *kernel* genérico. Cualquier dato que se deba emplear múltiples (localidad) veces es generalmente preferible que sea cargado en primera instancia en memoria local, con bajo consumo y latencia, y posteriormente utilizado desde esta memoria en lugar de utilizar la lejana memoria global.

Ejemplificando sobre el algoritmo gaussian en la Figura 4.5, empleando un filtro de 3x3 para la convolución de un pixel, se requiere la lectura de 9 píxeles adyacentes para su cálculo, pero muchos de ellos se comparten con la convolución del pixel vecino (intersección entre el marco azul y amarillo en la imagen). Con el uso de memoria local, en lugar de cargar desde memoria principal cada píxel inicial 9 veces, es mejor cargarlo una única vez y reutilizarlo en memoria local.

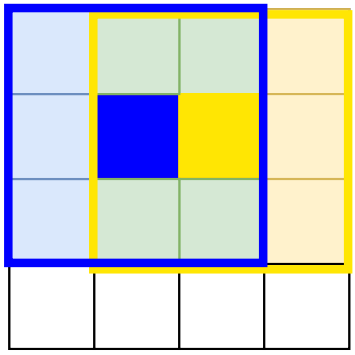


Figura 4.5: Reutilización de memoria en gaussian. Con el uso de memoria local cada píxel se lee una vez desde memoria global y hasta 9 desde memoria local. Sin el usar memoria local, el píxel se leería hasta 9 veces desde memoria global.

Siguiendo esta idea, se han modificado los *kernels* para que empleen memoria local. En primer lugar, al ser *kernels parallel_for*, se han incluido los atributos *reqd_work_group_size* y *max_work_group_size* para especificar el tamaño de los *work-groups* en los que agrupar el espacio de ejecución. Seguidamente se ha declarado la memoria local de grupo correspondiente al tamaño del *work-group*. Finalmente, se ha modificado el código para que cada *work-item* cargue el valor de memoria correspondiente a su rango en el espacio de ejecución y se sincronice con su *work-group* mediante *barriers* para delimitar los periodos de carga y de cómputo. Este cambio permite reducir la tasa de iniciación y el ancho de banda a memoria requerido. La directiva de memoria local de grupo no se encontraba disponible la versión 2021.1.1 de oneAPI que es la que se encontraba instalada en la máquina "macizo"de la Universidad de Zaragoza que se había empleado previamente y fue uno de los motivos que forzó al cambio a Devcloud.

4.3.2. Vectorización

Al existir alto grado de independencia y secuencialidad en las operaciones realizadas, tanto matemáticas como de cargado y guardado de datos, resulta ineficiente que cada *work-item* trabaje de manera individual sobre un solo dato. Esto es especialmente importante en las operaciones de acceso a memoria ya que, pese a estar fomentando el reuso de datos en memoria local, sigue existiendo una gran cantidad de accesos de poco tamaño a memoria global. Mediante el atributo *num_simd_work_items(N)* especificamos que se puede realizar la vectorización de N *work-items* contiguos en uno único. Por ejemplo, *num_simd_work_items(16)* sobre un *work-group* de 32x32 causaría un *work-group* resultante de 32x2. Al emplear la vectorización, aumentamos el valor de $vector\_factor$ en la ecuación 4.1 y se pueden conseguir grandes mejoras.

4.3.3. Desenrollado de bucles

El desenrollado de bucles permite especificar al compilador que agrupe la ejecución de múltiple iteraciones en una única. De este modo, el número de iteraciones totales a ejecutar disminuye por el factor de agrupamiento, las cuales emplearán más *hardware* para realizar el mismo trabajo en menos tiempo. Si se compila un código de ejemplo como el de la Figura 4.6 se observa que existen dos unidades de suma y una de multiplicación para ejecutar cada iteración, que esperar al resultado de la anterior acumulación para empezar la siguiente. Al añadir la directiva *unroll* el compilador es capaz de agregar unidades funcionales para realizar todas las operaciones con el mayor grado de paralelismo y atravesando una única vez la ruta de datos.

```
queue.submit([&](handler &cgh) {
  accessor x(x_buf, cgh, read_only);
  accessor sum(sum_buf, cgh, write_only);
  cgh.single_task<class unoptimzed>([=]() {
    int accum = 0;

    #pragma unroll
    for (size_t i = 0; i < 4; i++) {
      accum += x[i + get_global_id(0) * 4];
    }
    sum[get_global_id(0)] = accum;
  });
});
```

Figura 4.6: Uso del atributo *unroll*

Fuente: *oneAPI DPC++ FPGA optimization guide*

4.4. Otras técnicas de optimización implementadas

4.4.1. Patrón de acceso a memoria

Aunque los circuitos de memoria modernos, por ejemplo RAM DDR, permite acceder consecutivamente a posiciones aleatorias, este patrón tiene severas penalizaciones en el rendimiento frente acceder consecutivamente a valores almacenados en direcciones adyacentes[2]. Para mejorar el patrón de acceso a memoria y que se realicen peticiones del máximo tamaño a datos contiguos, en algoritmos como la multiplicación de matrices en los que se accede a datos por columnas (en este caso a la matriz B), resulta más eficiente acceder a bloques por filas, en la memoria global. Si además se invierte cada

[2]El motivo es que las memorias se organizan en bloques, que contienen varios datos consecutivos. Un acceso secuencial aprovecha todos los datos de un bloque, mientras que un acceso aleatorio debe acceder a un nuevo bloque para cada dato accedido.

fila y se guarda como una columna en la memoria local, al querer acceder a lo que sería originalmente una columna en la matriz B se realiza un acceso secuencial ya que está invertida. Esta secuencia de accesos a memoria se representa en la Figura 4.7

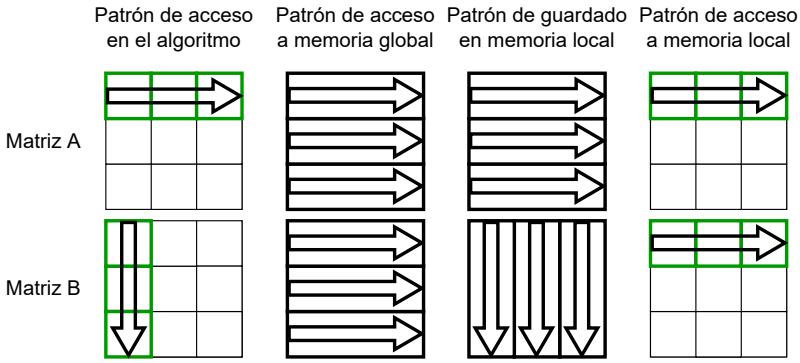


Figura 4.7: Cambio en el patrón de acceso a memoria

4.4.2. *Buffers* de uso específico

Los *buffers* empleados para mandar y recibir datos a los *kernels* pueden ser marcados con atributos que especifican el uso que se va a hacer de los mismos. Entre los atributos posibles se encuentra los 3 empleados en este trabajo: *access::mode::read* junto con *access::target::constant_buffer* y *access::mode::discard_write*. Los dos primeros permiten al compilador ubicar dicho *buffer* en una sección de solo lectura, ya que no será modificado. El segundo atributo le permite emplear cualquier zona de memoria ya que los contenidos serán descartados.

4.4.3. Alineamiento de memoria

La declaración y transferencia de *buffers* resulta más eficiente si estos se declaran a partir de datos alineados en memoria, por lo que se declaran las estructuras de datos de entrada o salida con la función *aligned_alloc* a direcciones múltiplo de 1024 *bytes*.

4.4.4. Operaciones con números decimales

Para facilitar la generación de *hardware* para ejecutar operaciones con tipos de datos decimales se permite el uso de FMAs, la reasociación de operaciones y saltarse conversiones y redondeos intermedios. Actualmente estas opciones se emplean por defecto, por lo que ya no es necesario incluir las correspondientes opciones de compilación.

4.5. Resumen

En la experiencia de este trabajo, las optimizaciones más efectivas han resultado ser el desenrollado de bucles y la vectorización, seguidos del uso de memoria local. Para el aprovechamiento de estas optimizaciones, las iteraciones de los bucles no deben presentar interdependencias, se debe acceder a posiciones contiguas de memoria y los datos accedidos deben presentar localidad para que la memoria local pueda explotar el reúso. El alineamiento de memoria también ha resultado efectivo para permitir al compilador inferir un mejor acceso a memoria principal, pero poco efectivo si ya se estaba empleando vectorización.

El resto de optimizaciones o bien se han mantenido a lo largo de todo el proceso de optimización siguiendo las recomendaciones de la guía o bien han resultado de poca o ninguna mejora pese a proporcionar información extra al compilador.

En la Tabla 4.1 se muestran las diferentes optimizaciones y su aplicación a cada *kernel*. Los recuadros verdes implican que la optimización se ha aplicado activamente al *kernel*, los azules que el código ya cumplía con esa optimización y se ha hecho un esfuerzo por mantenerla y gris que la optimización no era aplicable al *kernel*.

	matadd	**matmul**	**nbody**	**gaussian**
Evitar solapamiento de punteros				
Construir bucles "bien formados"				
Minimizar las iteraciones dependientes				
Fusionar bucles anidados				
Variables de ámbito cercano				
Memoria local para accesos repetidos				
Vectorización				
Desenrollado de bucles				
Patrón de acceso a memoria				
Buffers de uso específico				
Alineamiento de memoria				
Operaciones con números decimales				

Tabla 4.1: Resumen de optimizaciones aplicadas por *kernel*

Capítulo 5

Coejecución con Intel oneAPI

En este capítulo se describe la herramienta de coejecución heterogénea CPU-FPGA empleada y el trabajo realizado para intentar migrar la sección de CPU de C++ nativo a oneAPI. 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.

5.1. Coejecutor

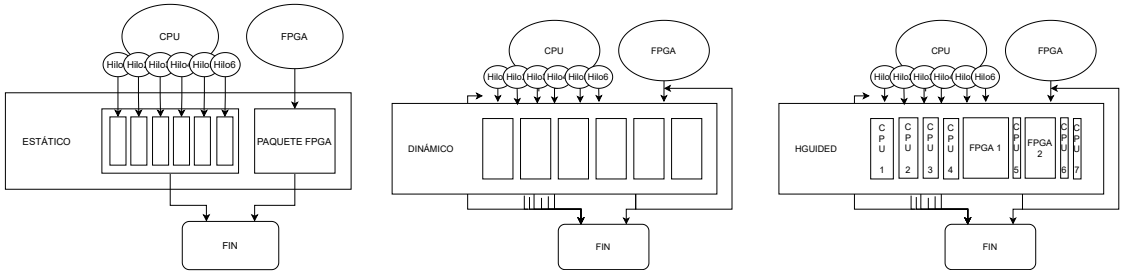
En este trabajo fin de máster se parte de una herramienta de coejecución desarrollada originalmente por Raúl Nozal [16] y adaptada por mí[9]. Dicha herramienta era capaz de realizar ejecución heterogénea entre CPU y FPGA usando C++ nativo para CPU y oneAPI para FPGA. La herramienta incluye un conjunto 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.

5.1.1. Planificadores

Planificador estático (Figura 5.1a) Divide el problema en dos fragmentos, uno para CPU y otro para FPGA. El tamaño de los fragmentos es ajustable por el usuario según la potencia de cálculo que le estime a cada dispositivo. El fragmento de CPU se subdivide entre el número de hilos elegido.

Planificador dinámico (Figura 5.1b) 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 5.1c) 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 (5.1).



(a) Estático (b) Dinámico (c) Hguided

Figura 5.1: Planificadores nuevos

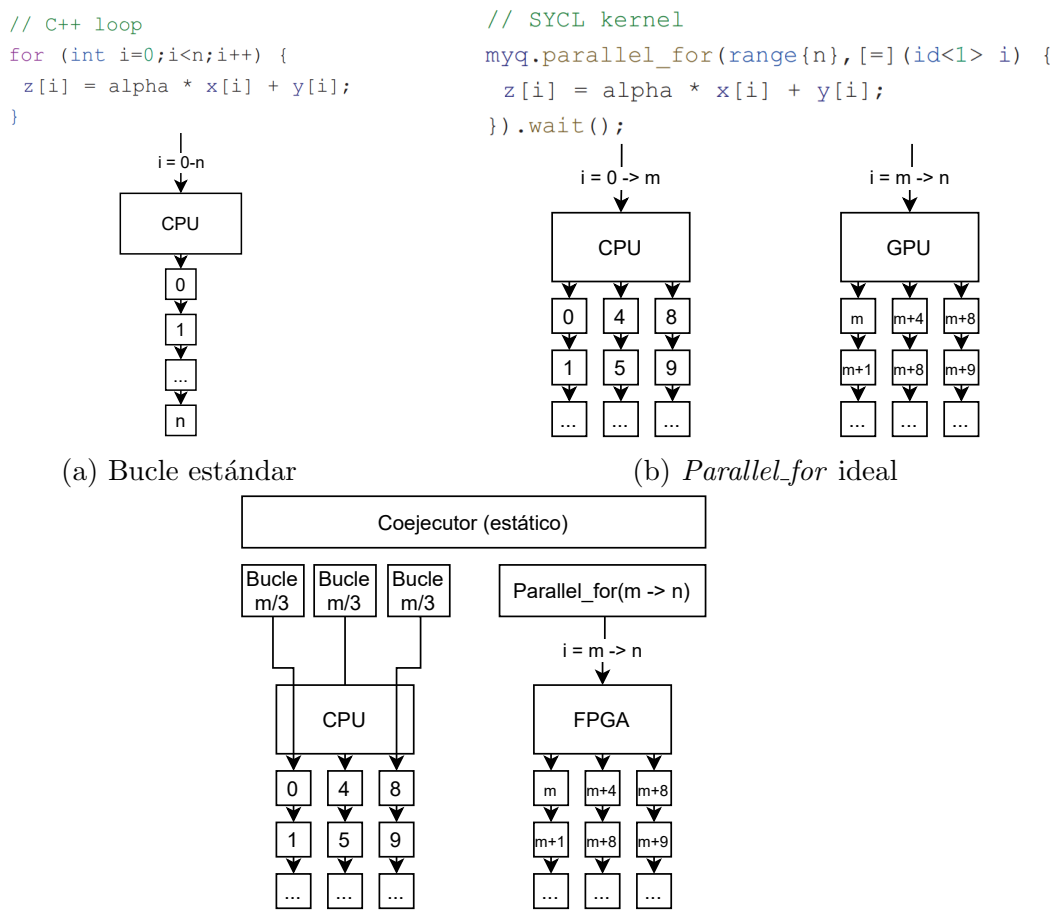
$$Fragmento = max(MinFrag, \frac{Restante \cdot \frac{PotenciaComputo}{NumeroHilos}}{K}) \tag{5.1}$$

5.1.2. Paralelismo en la coejecución

En la Figura 5.2 se representa una iteración secuencial de un bucle regular C++ (5.2a) junto con su traducción a un *kernel parallel_for* con oneAPI. Idealmente se podrían repartir los diferentes *work-items* entre múltiples dispositivos (5.2b), pero, como se ha mencionado previamente (Sección 2.3.2), esto no está soportado por oneAPI. Para lograr este reparto entre dispositivos el coejecutor divide el espacio de ejecución manualmente para los dos dispositivos, asignando bucles a cada *core* de la CPU y *parallel_for* a la FPGA 5.2c.

5.2. Coejecución oneAPI con la FPGA

Al emplear la FPGA como dispositivo resulta obligatorio emplear el modelo de compilación AOT, descrito en la sección 2.3.2. Este modelo consigue combinar el código del *host* con el *bitstream* de la FPGA en un mismo fichero y permite ejecutar *kernels* en la FPGA. Se ha conseguido añadir la CPU como segundo dispositivo incluido en el



(a) Bucle estándar (b) *Parallel_for* ideal

(c) *Parallel_for* Unizar

Figura 5.2: Paralelización con *Parallel_for*

fatbinary para códigos con dos *kernels* muy simples. Por desgracia, al emplear *kernels* optimizados más complejos se encuentra un error en ejecución ya que, pese a que el *kernel* de CPU termina correctamente, el *kernel* de FPGA ejecuta indefinidamente y se obtienen mensajes informativos de que el *kernel* puede estar colgado. Se intentó extensamente solucionar este problema modificando los parámetros de compilación y empleando diferentes niveles de complejidad en el *kernel*, pero al ser necesaria una compilación de varias horas para cada prueba (ya que la versión de emulador no se queda atascada) no se llegó a encontrar una solución al problema. Por este motivo y dado que la intención es trabajar con los *kernels* optimizados para FPGA, se ha mantenido la ejecución de la CPU en C++ nativo.

A finales de diciembre hubo una actualización en Devcloud que cambió del comando de compilación de `dpcpp` a `icpx -fsycl`. Este cambio causó, ya en enero, que los *kernels* ya compilados no pudieran ejecutarse por no encontrar bibliotecas de sistema, siendo necesario recompilarlos usando el nuevo comando. Al ejecutar nuevamente el proyecto de prueba de coejecución con los *kernels* más complejos si que se consiguió ejecutar,

quedando pendiente como trabajo futuro explorar de nuevo si la coejecución 100 % oneAPI es ahora posible y adaptar el coejecutor al uso de CPU en oneAPI.

Capítulo 6

Resultados experimentales

En este capítulo se muestra primero los resultados experimentales de la optimización de los *benchmarks* y posteriormente se analizan los distintos planificadores.

6.1. Optimización

En esta sección se listan las optimizaciones (explicadas en la Sección 4) aplicadas a cada uno de los *benchmarks* y los resultados obtenidos.

6.1.1. Matadd

Matadd_base Suma de matrices estándar sin ninguna optimización. Matadd es el *kernel* más sencillo y por tanto el que menos margen de mejora tiene, ya que ni siquiera existe reuso de datos para implementar memoria local.

Matadd_simd Como optimización principal se ha implementado un tamaño de *work-group* de 16x16 para permitir emplear un tamaño SIMD de 16. De este modo los accesos a memoria cargan 16 valores por petición, con una anchura de 512 bytes, que es el tamaño del *burst* máximo desde memoria [6].También se garantiza que los parámetros de entrada no se superponen con `kernel_args_restrict`.

La diferencia en el tiempo de ejecución del *benchmark* al aplicar dicha optimización se muestran en la Tabla 6.1. También se muestra la diferencia en los recursos empleados por la FPGA en la Tabla 6.2. Se observa que sobrecarga en el uso de hardware para implementar vectorización es mínima y la mejora en el rendimiento muy significativa, alcanzando un *speedup*[1] de 7.3.

[1]Se entiende por *speedup* la diferencia entre el tiempo de ejecución original y el nuevo

Benchmark	**Tiempo (ms)**	***Speedup***
matadd_base	19559	1
matadd_simd	2680	7.3

Tabla 6.1: Optimización de Matadd

	matadd_base	**matadd_simd**
ALUTs	0 %	0 %
FFs	25 %	26 %
RAMs	26 %	27 %
MLABs	1 %	1 %
DSPs	22 %	23 %
Frecuencia (MHz)	377	408

Tabla 6.2: Recursos *hardware* y frecuencia de Matadd. Porcentajes con respecto a la Tabla 3.1

6.1.2. Matmul

Matmul_base El algoritmo inicial de *Matmul* corresponde con una multiplicación de matrices ingenua, siguiendo el algoritmo descrito en la Figura 6.1.

```
for (int row = 0; row < N; row++)
   for (int col = 0; col < N; col++)
       for (int k = 0; inner < N; inner++)
           C[row][col] += A[row][k] * B[k][col];
```

Figura 6.1: Multiplicación de matrices ingenua en C++

Matmul_lm El primer paso de optimización fue modificar el algoritmo por una multiplicación de matrices por bloques, empleando *work-groups* de 32x32 *items* para cargar cada bloque (de equivalente tamaño) en memoria local y operar con los datos locales. Al acceder a ambas matrices de entrada se emplea una lectura por filas, pero al guardarlas en memoria local se transpone la matriz B, de manera que al leer la memoria local posteriormente el acceso vuelve a ser por filas, maximizando la localidad.

Matmul_lm_simd Seguidamente se emplea SIMD 16 para mejorar la anchura de los accesos a memoria global, tanto de lectura como escritura, así como vectorizar los cálculos del bucle principal.

Matmul_lm_simd_lu Por último se desenrollan las 32 iteraciones del bucle principal, paralelizando los cálculos.

Los resultados en rendimiento de la aplicación aditiva de estas optimizaciones se muestran en la Tabla 6.3. También se muestra la variación en el uso de recursos *hardware*

y otros factores en la Tabla 6.4. Se observa que existe suficiente reuso de datos y densidad de operaciones como para que el uso de memoria local (de 32x32 que es el máximo para el que se consigue generar una imagen FPGA) ya suponga un 3.48 de *speedup* incluso si el acceso a memoria global no es eficiente. Si bien las optimizaciones son progresivamente más efectivas, cada una de ellas se respalda en la anterior ya que SIMD optimiza las operaciones de carga de datos a memoria local y envío de resultados a memoria global, y el desenrollado de bucles se aprovecha del rápido acceso a memoria local para poder suministrar los operandos rápidamente a todas las iteraciones desenrolladas. Se observa que, en este caso, el uso de FFs y RAMs aumenta significativamente al emplear SIMD, y que los DSPs aumentan principalmente al hacer desenrollado de bucles, como cabría esperar. Pese a que no se supere el 40 % de utilización de ninguno de los recursos, es imposible aumentar el grado de desenrollado sin aumentar el tamaño de los *work-groups*, y aumentando dicho tamaño a la siguiente potencia de 2 no se consigue generar la imagen FPGA por violaciones de *timing*[2]. Al trabajar con *kernels parallel_for*, el reporte indica para cada bucle la latencia y la cantidad de hilos de ejecución en lugar del intervalo de iniciación. Además, al añadir memoria local se incluye un bucle externo de carga de datos, quedando entonces en el bucle interno un acceso a memoria local. Se observa que la latencia siempre es menor que la capacidad de hilos, y que disminuye drásticamente al cambiar el acceso a memoria a una local. Se observa el efecto de SIMD sobre el bucle interno, reduciendo la cantidad de hilos por el valor de SIMD y aumentando la latencia como consecuencia del uso de operaciones vectoriales. También se observa la diferencia en latencia del acceso a memoria del bucle externo al aplicar SIMD.

Benchmark	**Tiempo (ms)**	***Speedup parcial***	***Speedup total***
matmul_base	800015.33	1.00	1.00
matmul_lm	229910.00	3.48	3.48
matmul_lm_simd	28454.67	8.08	28.12
matmul_lm_simd_lu	1261.33	22.56	634.26

Tabla 6.3: Optimización de Matmul

6.1.3. Nbody

Nbody_base Simulación Nbody sin ninguna optimización

Nbody_lm Se emplean *work-groups* de 1x128 *work-items* que cargan los datos en memoria local, alternando etapas de carga y de cómputo mediante *barriers*

[2]No se supera el límite de recursos pero el uso es suficientemente grande para que no se consiga encontrar una configuración que cumpla las especificaciones y las restricciones de tiempo

	matmul base	matmul lm	matmul lm_simd	matmul lm_simd_lu
ALUTs	0 %	0 %	0 %	0 %
FFs	25 %	25 %	38 %	39 %
RAMs	27 %	29 %	36 %	34 %
MLABs	1 %	1 %	1 %	2 %
DSPs	22 %	23 %	23 %	32 %
Frecuencia (MHz)	351	329	216	259
Bucle interno: Latencia	846	15	28	-
Bucle interno: Hilos	847	1024	64	-
Bucle externo: Latencia	-	1871	923	1010
Bucle externo: hilos	-	4096	1088	1024

Tabla 6.4: Recursos *hardware*, frecuencia y bucles de Matmul. Porcentajes con respecto a la Tabla 3.1

Nbody_lm_simd Se emplea SIMD 16 para mejorar la anchura de los accesos a memoria global, tanto de lectura como escritura, así como los cálculos matemáticos.

Nbody_lm_simd_lu Se desenrollan 4 iteraciones del bucle principal, que es el máximo para el cual los recursos requeridos permitían generar una imagen de FPGA.

Los resultados de la aplicación aditiva de estas optimizaciones de muestran en la Tabla 6.5, junto con los recursos empleados, frecuencia y latencia y cantidad de hilos por bucle en la Tabla 6.6. Se observa que en este caso el uso de memoria local por si sola supone una perdida en el rendimiento, esto podría intentar solucionarse aumentando el tamaño de los *work-groups* para explotar mayor reúso, pero teniendo en cuenta que habilita SIMD y que con SIMD se consigue un *speedup* total de 10.04, se prefiere no gastar más recursos y reservarlos para el desenrollado de bucle, que ya de por si solo puede desenrollar 4 iteraciones antes de encontrar violaciones de *timing* o superar el máximo de recursos.

Benchmark	**Time (ms)**	***Speedup parcial***	***Speedup total***
nbody_base	198930.67	1.00	1.00
nbody_lm	205759.50	0.97	0.98
nbody_lm_simd	19810.67	10.39	10.04
nbody_lm_simd_lu	5264.25	3.76	37.79

Tabla 6.5: Optimización de Nbody

6.1.4. Gaussian

Gaussian_base Cálculo del difuminado gaussiano. En este último *benchmark* se alteró el orden de aplicación de las optimizaciones, ajustando en primera instancia el

	nbody base	nbody lm	nbody lm_simd	nbody lm_simd_lu
ALUTs	0 %	0 %	0 %	0 %
FFs	26 %	26 %	54 %	56 %
RAMs	30 %	29 %	26 %	50 %
MLABs	1 %	1 %	2 %	3 %
DSPs	22 %	22 %	29 %	42 %
Frecuencia (MHz)	348	338	221	223
B. interno: Latencia	879	86	86	91
Bucle interno: Hilos	880	128	88	96
Bucle externo: Latencia	-	937	941	941
Bucle externo: Hilos	-	1280	1040	1048

Tabla 6.6: Recursos *hardware*, frecuencia y bucles de Nbody. Porcentajes con respecto a la Tabla 3.1

alineamiento de memoria a 1024 bytes de los *buffers* de entrada. Se comprueba que, si bien esta optimización no afectaba a los otros *benchmarks* al añadirse en los últimos pasos de optimización, al aplicarla desde el principio tiene un efecto similar a SIMD permitiendo al compilador inferir accesos a memoria de mayor anchura.

Gaussian_lm Se modifica el algoritmo para que emplee memoria local. En este caso, por las características del *kernel* es necesario traer a memoria local más valores que el tamaño del *workgroup* por lo que, pese a no estar recomendado, es necesario que un *work-item* en concreto se encargue de cargar los bordes. Además, para no requerir comprobar que la indexación de la memoria se salga de rango, se incluye un anillo exterior con valor 0 que no altera el resultado y evita condicionales extra en el bucle.

Gaussian_lm_lu Se desenrolla por completo el bucle interno, 5x5 iteraciones.

Gaussian_st Dado que para cargar datos a memoria local se requiere considerar el caso especial del anillo exterior e incluir código específico para algunos *work-items* se ha probado a cambiar el código de *parallel_for* a *single-task* (con memoria local y desenrollado de bucles).

Los resultados de la aplicación aditiva de las primeras optimizaciones y del cambio a *single-task* se muestran en la Tabla 6.7 y se complementan con los recursos *hardware* empleados de la Tabla 6.8. Se observa que la adición de memoria local ya existiendo accesos a memoria eficientes supone un *speedup* relativamente bajo de 1.54, si bien se podría aumentar incrementando el tamaño de *work-group* y por consiguiente de bloque de memoria local. El *speedup* mayor se obtiene en este caso de la aplicación de desenrollado en el bucle principal. En el caso de *single-task* si que existe II en lugar de

latencia e hilos. Se omite este valor en la tabla ya que existen más de 5 bucles, pero todos tienen II de 1 o 2 ciclos.

Benchmark	**Tiempo (ms)**	***Speedup parcial***	***Speedup total***
gaussian_base	77628.33	1.00	1.00
gaussian_lm	50380.67	1.54	1.54
gaussian_lm_lu	8741.67	5.76	8.88
gaussian_st	10121.67	0.86	7.67

Tabla 6.7: Optimización de Gaussian

	gaussian base	**gaussian lm**	**gaussian lm_lu**	**gaussian st**
ALUTs	0 %	0 %	0 %	0 %
FFs	25 %	26 %	42 %	29 %
RAMs	27 %	29 %	39 %	33 %
MLABs	0 %	0 %	0 %	0 %
DSPs	22 %	22 %	24 %	24 %
Frecuencia (MHz)	371	320	212	258
B. interno: Latencia	856	885	-	-
Bucle interno: Hilos	857	886	-	-

Tabla 6.8: Recursos *hardware*, frecuencia y bucles de Gaussian. Porcentajes con respecto a la Tabla 3.1

6.1.5. Resumen

Como resume la Tabla 6.9, aplicando las técnicas de optimización descritas se han conseguido *speedups* significativos en todos los *benchmarks*, que varían entre 7.3 en *kernels* en los que no se incrementa casi el uso de recursos y 654.26 en aquellos que si permiten explotar mejor la potencia de la FPGA.

	Tiempo base (ms)	**Tiempo optimizado (ms)**	***Speedup***
matadd	19559	2680	7.3
matmul	800015	1261	634.26
nbody	198931	5264	37.79
gaussian	77628	8742	8.88

Tabla 6.9: Resumen de optimización

6.2. Resultados de la Coejecución CPU y FPGA

En esta sección se describen los resultados de la coejecución de los diversos *kernels* para su ejecución en CPU y FPGA. Se recuerda que la ejecución en CPU se realiza en

C++ nativo mientras que la de FPGA se hace con oneAPI y utilizando la versión más optimizada del kernel vista en la sección anterior.

Se incluye en la Tabla 6.10 un resumen de los mejores tiempos de ejecución en FPGA obtenidos en la fase de optimización, así como el tiempo de ejecución base en CPU empleando los 16 núcleos.

Benchmark	**Tiempo CPU (ms)**	**Tiempo FPGA (ms)**
matadd	379	2680
matmul	191803	1261
nbody	64090	5264
gaussian	5001	8742

Tabla 6.10: Mejores tiempos en CPU y FPGA

Se emplean los *kernels* matadd, matmul y nbody, descartando *gaussian* ya que las optimizaciones implementadas dificultan su fragmentación y reparto y requerirían un cambio drástico en el planificador. Se observa que los 3 *kernels* restantes tienen tiempos de ejecución muy dispares entre dispares entre dispositivos, lo que dificulta enormemente su coejecución.

6.2.1. Planificador estático

En la Figura 6.2 se representa en el eje Y el tiempo de terminación de cada dispositivo (los 16 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 otras palabras, la altura de la barra gris representa el desbalanceo del sistema e idealmente no debería aparecer.

El eje X 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 CPU++. La exploración del porcentaje del problema a asignar a cada dispositivo comienza en un reparto inicial de 50 %/50 % y en ejecuciones subsiguientes se aumenta el porcentaje asignado al dispositivo que antes termina. Los porcentajes elegidos garantizan que se cumpla el requisito de que el fragmento asignado a la FPGA sea múltiplo del tamaño de *work-group*.

Se observa que el tiempo de ejecución de la FPGA siempre es el limitante en matadd ya que por muy pequeño que sea el fragmento que mandemos al acelerador, hay que esperar la latencia de transferencia. En matmul y nbody la FPGA es dominante, siendo la CPU siempre el dispositivo más lento. Este resultado era de esperar ya que el tiempo de ejecución en FPGA es mucho menor tras las optimizaciones.

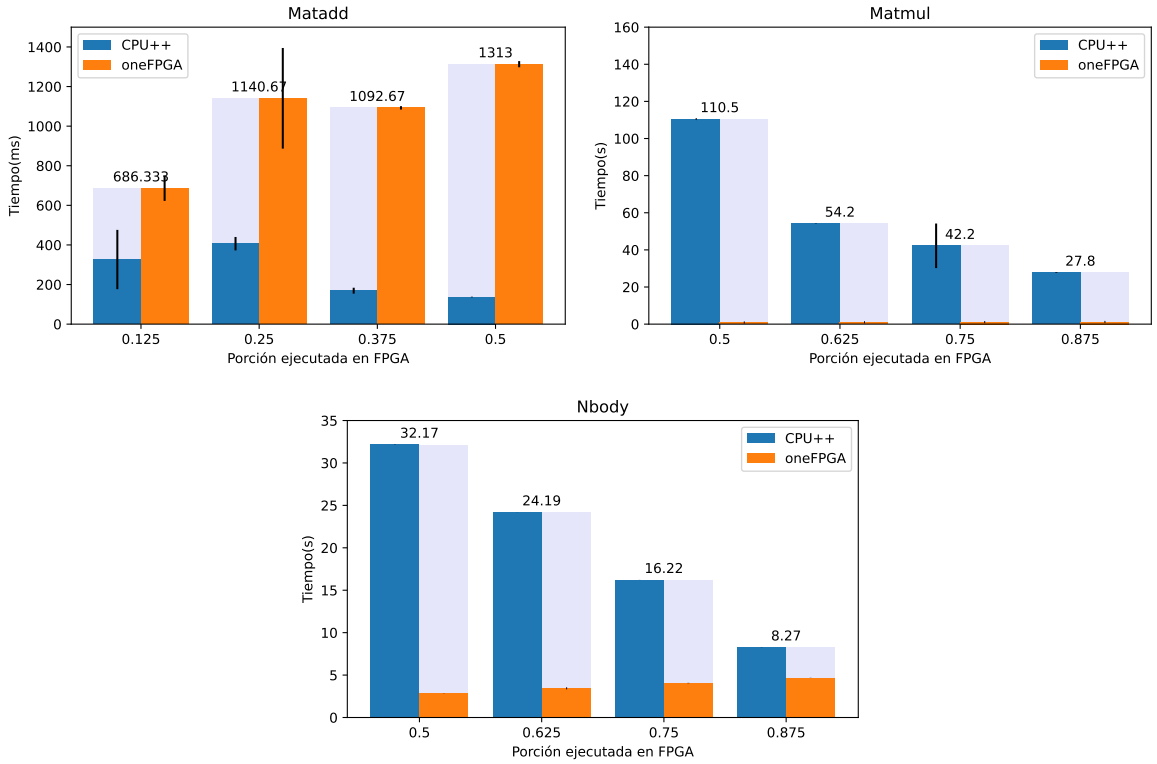


Figura 6.2: Planificador estático

6.2.2. Planificador dinámico

De manera similar al planificador estático, en la Figura 6.3 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 fragmentos (de igual tamaño) en los que se ha dividido el problema. Dado que el reparto de trabajo se realiza de forma dinámica, en el eje X de la parte superior de la gráfica se representa el porcentaje que se ha entregado de manera efectiva a la versión CPU++.

Se observa que, de manera general, conforme se aumenta el número de paquetes el desbalanceo disminuye. También se observa que el tiempo de ejecución de los dispositivos (CPU en matadd y FPGA en nbody) es mayor que el tiempo de ejecución en ejecución individual. Esto se corresponde con el retardo añadido entre la finalización de cada fragmento y el comienzo de la ejecución del siguiente fragmento asignado.

6.2.3. Planificador *hguided*

Para emplear el planificador hguided, que reparte la ejecución en fragmentos de tamaño decreciente, se debe asegurar que los fragmentos repartidos a los *kernels* optimizados para FPGA son de tamaño múltiplo del tamaño de *work-group* empleado. Por tanto se establece como tamaño múltiplo, y también mínimo, 16, 32 y 128 para

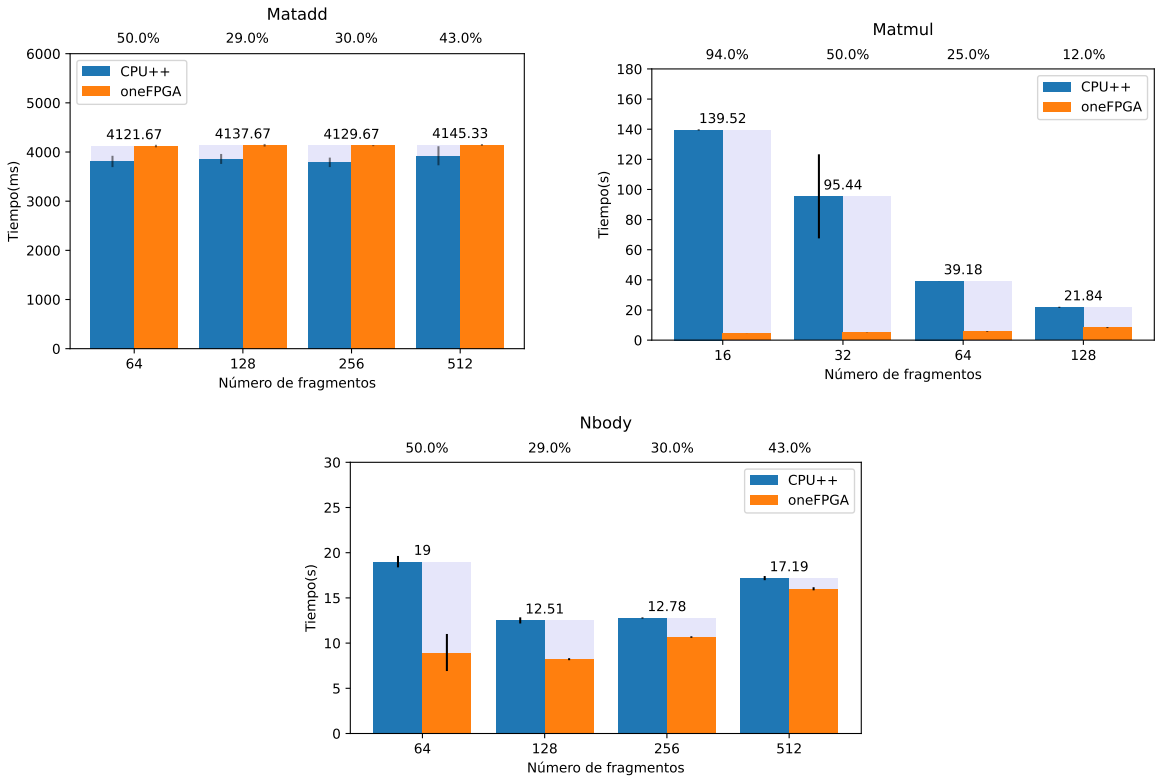


Figura 6.3: Planificador dinámico

matadd, matmul y nbody respectivamente.

En la Figura 6.4 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 (explicado en la sección 5.1.1), el cual se aumenta en intervalos de 0.5 desde 1 hasta 4.5. Se emplea el mejor porcentaje de reparto de problema obtenido con el planificador estático para asignar las potencias esperadas de los dispositivos.

Se observa que matadd presenta un comportamiento similar al planificador dinámico para todos los valores de K. En el caso de matmul, por las limitaciones que presenta la relación entre el tamaño del problema y el tamaño mínimo de fragmento, el planificador siempre reparte el problema en un número demasiado pequeño de fragmentos por lo que se asignan demasiados a la CPU. nbody ejemplifica muy bien el potencial de este planificador, alcanzando tan buen balanceo (en este caso incluso mejor) como el planificador dinámico pero empleando menor número de paquetes por lo que la sobrecarga es significativamente menor, obteniendo mejor tiempo que el planificador dinámico pero no mejor que exclusivamente en FPGA.

6.2.4. Resumen

La Tabla 6.11 incluye un resumen del mejor tiempo de ejecución de cada una de las opciones, con la mejor versión para cada *benchmark* recuadrada de verde.

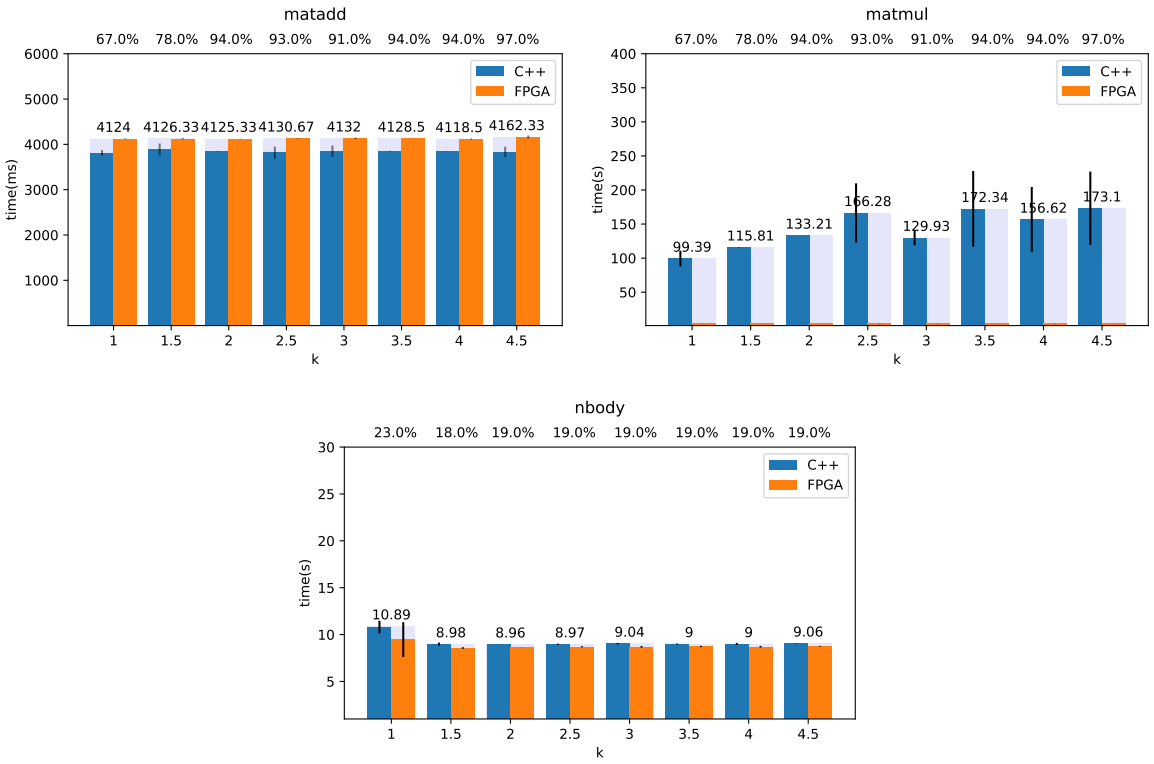


Figura 6.4: Planificador hguided

	CPU	FPGA	Estático	Dinámico	Hguided
matadd	379	2680	686	4121	4118
matmul	191803	1261	278000	21840	99390
nbody	64090	5264	827000	12510	8960
gaussian	5001	8742	-	-	-

Tabla 6.11: Mejor tiempo de terminación en milisegundos para cada versión de ejecución de cada *benchmark*

Capítulo 7

Conclusiones y Trabajo Futuro

oneAPI ofrece un gran repertorio de optimizaciones para mejorar el desempeño de *kernels* en FPGA. Se han probado multitud de las opciones y recomendaciones mencionadas en la guia de Intel, dando lugar a incrementos sustanciales de rendimiento en todos los *kernels* optimizados que varían entre 7.3 veces y 634.26 veces más rendimiento que la versión poco o nada optimizada. Estos resultados corroboran que, pese a existir portabilidad funcional, se sigue estando lejos de obtener portabilidad de rendimiento.

Entre las optimizaciones probadas se distinguen diferentes niveles de impacto obtenido, siendo más efectivas aquellas que mejoran el acceso a memoria (mayor ancho de banda o menor necesidad de acceso) o permiten paralelizar los cálculos matemáticos de los bucles internos. Otras directivas que otorgan información extra al compilador tienen un impacto menor, que puede depender del resto de optimizaciones ya incluidas, o son ignoradas. Se han considerado un total de 12 optimizaciones diferentes, aplicándose la mayoría de ellas a todos los *kernels*, siempre y cuando el algoritmo lo permitiese. Entre estas optimizaciones se encuentran 5 guías generales de diseño de bucles en *kernels* y 7 optimizaciones de diverso tipo.

La aplicación de optimizaciones tiene un impacto negativo en la portabilidad y usabilidad de los *kernels*, resultando en un fallo de ejecución al ser lanzado en otros dispositivos y requiriendo de restricciones adicionales para su correcto funcionamiento en FPGA como tamaños de entrada múltiplos del *work-group* o uso de valores conocidos en compilación.

La reducción de usabilidad junto con la disparidad en el rendimiento de ambos dispositivos como consecuencia de la optimización dificulta obtener un balanceo adecuado con independencia de la política de planificación empleada. En aquellos casos en los que sí que se logra balancear la terminación se observa que la fragmentación del problema en paquetes disminuye el rendimiento de los dispositivos, particularmente de la FPGA que ha de pagar la latencia de transferencia de datos por el bus PCIe. Como resultado,

ningún *kernel* obtiene en este caso beneficio de la coejecución.

Devcloud resulta una muy buena plataforma para acceder y probar las ultimas versiones de las herramientas y *hardware* de Intel, no obstante puede ser bastante tediosa para trabajar con FPGAs debido a la contención por las máquinas resultado de los largos tiempos de compilación y de los constantes reinicios o apagados de las máquinas. Las múltiples actualizaciones tanto de la herramienta en Devcloud como de la documentación dificultan el trabajo, habiendo coincidido una gran actualización con las semanas anteriores a la finalización de este trabajo. La documentación para FPGA es algo pobre, particularmente en el uso de *kernels parallel_for* y aún más en cómo realizar co-ejecución con *fatbinary*, no existiendo ejemplos de ello.

Queda como trabajo futuro probar si la última actualización verdaderamente facilita el uso estricto de oneAPI para coejecución. Podría también evaluarse la efectividad de algunas de las optimizaciones por separado, ya que la aplicación aditiva podría estar mitigando sus efectos como es el caso del alineamiento de memoria. Existen también otros *kernels* y optimizaciones que se podrían probar, particularmente para el modelo *single-task*.

Capítulo 8

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] Swapnil Ghike, Rubén Gran, María J. Garzarán, and David Padua. Directive-based compilers for gpus. In James Brodman and Peng Tu, editors, *Languages and Compilers for Parallel Computing*, pages 19–35, Cham, 2015. Springer International Publishing.

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

[9] Raúl Herguido. Evaluación del modelo de programación oneapi para ejecución heterogénea con cpu y fpga. Trabajo Fin de Grado. Universidad de Zaragoza, 2021.

[10] 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.

[11] 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.

[12] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda: Is cuda the parallel programming model that application developers have been waiting for? *Queue*, 6(2):40–53, mar 2008.

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

[14] Raúl Nozal and Jose Luis Bosque. Straightforward heterogeneous computing with the oneapi coexecutor runtime. *Electronics*, 10(19):2386, Sep 2021.

[15] Andrés Rodríguez, Angeles Navarro, Kris Nikov, Jose Nunez-Yanez, Rubén Gran, Darío Suárez Gracia, and Rafael Asenjo. Lightweight asynchronous scheduling in heterogeneous reconfigurable systems. *Journal of Systems Architecture*, 124:102398, 2022.

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

Lista de Figuras

Lista de Tablas

Anexos

Anexos A

Cronología

Se comenzó probando las opciones de optimización y coejecución en macizo, así como algunos *kernels single-task* con los que no se había trabajado.

Tras comprobar que algunas optimizaciones básicas (como memoria local) así como los comandos de coejecución no funcionaban en macizo, se comenzó a probar todo en el Devcloud.

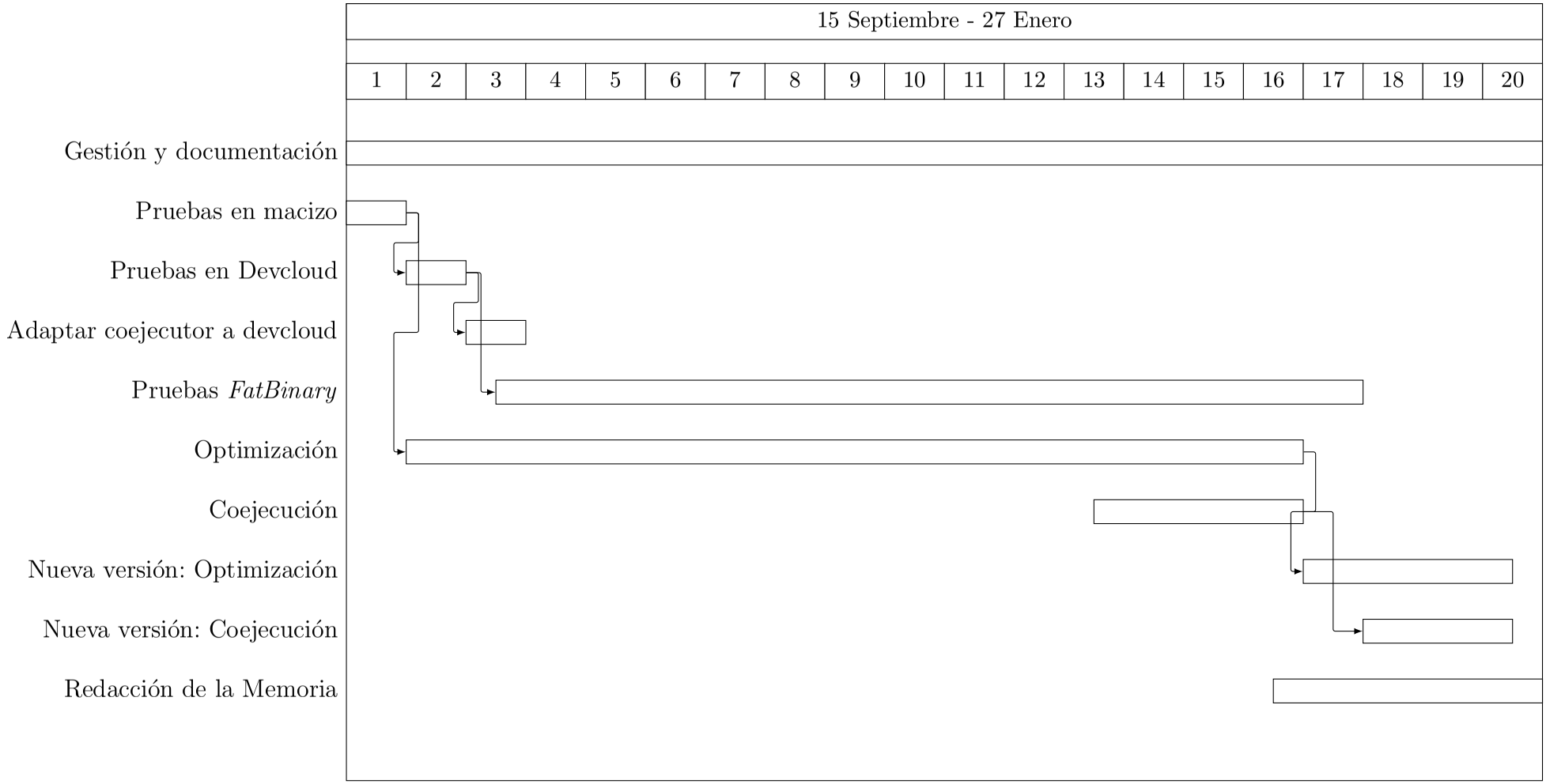
Al comprobar que era necesario emplear la versión de Devcloud y no poder actualizar la de macizo, se procedió a migrar el proyecto.

Paralelamente se realizan pruebas de funcionamiento de coejecución en Devcloud junto a optimizaciones (las primeras semanas también en macizo)

Tras muchas optimizaciones y muchas pruebas de coejecución fallidas, se comienza a adaptar el coejecutor y los *kernels* para poder realizar planificación C++-oneAPI.

Tras la actualización de versión a finales de Diciembre en Devcloud se deben recompilar todos los binarios y se realizan los experimentos con las optimizaciones más relevantes y la coejecución

Finalmente se redacta la memoria.


15 Septiembre - 27 Enero
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
Gestión y documentación
Pruebas en macizo
Pruebas en Devcloud
Adaptar coejecutor a devcloud
Pruebas *FatBinary*
Optimización
Coejecución
Nueva versión: Optimización
Nueva versión: Coejecución
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:

- (30h, 8.6 %) Gestión del TFM: tiempo dedicado a discutir y documentar el progreso y los resultados.
- (34h, 9,7 %) Exploración de plataformas: Pruebas iniciales de las diferencias entre versiones y plataformas y migración
  - Pruebas macizo
  - Pruebas Devcloud
  - Migración
- (80h, 23 %) *Fatbinary*: pruebas de coejecución 100 % oneAPI con diferentes proyectos, comandos, ...
- (130h, 37.2 %) Optimización: proceso de prueba de las diferentes optimizaciones de manera aditiva y toma de resultados finales tras el cambio de versión
  - Optimización
  - Optimización con la nueva versión
- (50h, 14.3 %) Coejecución: proceso de prueba de la funcionalidad de la coejecución al ir optimizando y toma de resultados finales tras el cambio de versión
  - Coejecución
  - Coejecución con la nueva versión
- (25h, 7.2 %) Memoria: redacción de la memoria.

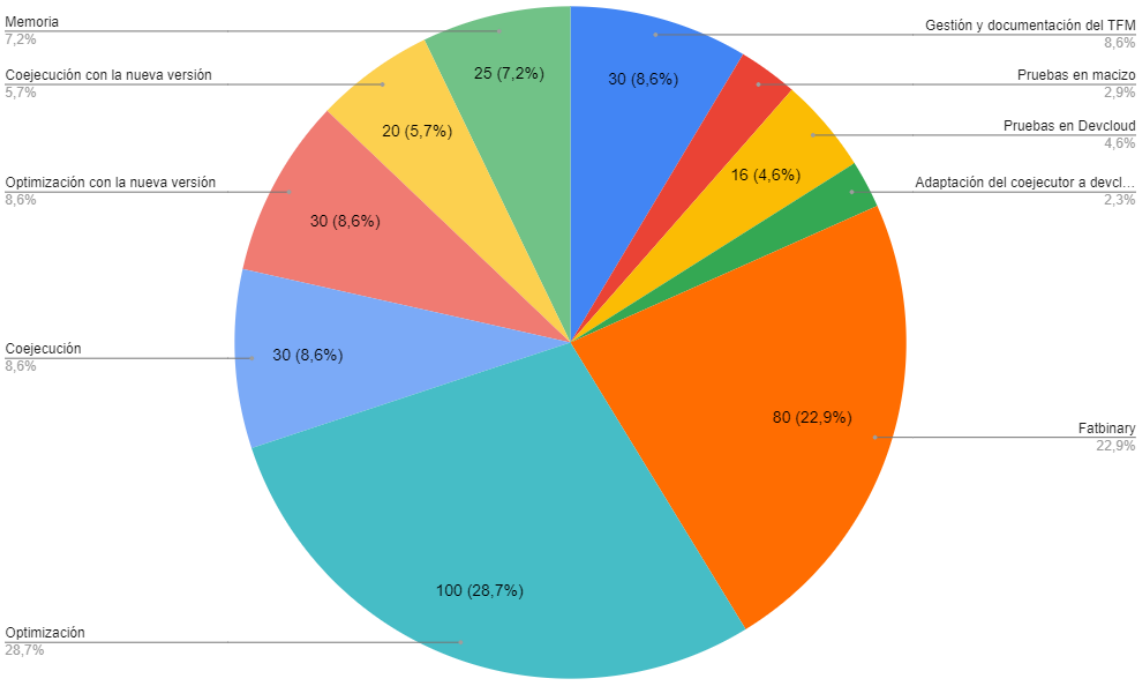


Figura A.1: Tiempo dedicado a cada tarea