Facultad de Ciencias

Implementación Multi-GPU de Algoritmos de Análisis de Metilación del ADN
(Multi-GPU Implementation of DNA Methylation Analysis Algorithm)

Trabajo de Fin de Máster
para acceder al

MÁSTER EN INGENIERÍA INFORMÁTICA

Autor: Adrián Celis Fernández

Director\es: José Luis Bosque Orero
Borja Pérez Pavón

Septiembre – 2023

Resumen

En la actualidad, las unidades de procesamiento gráfico (GPU) han experimentado un crecimiento en su aplicación, traspasando su función original dedicada a la aceleración de gráficos y juegos convirtiéndose en herramientas esenciales en la investigación científica.

En este aspecto, el grupo de Grupo de Redes y Entornos Virtuales (GREV) de la Universidad de Valencia conocido como desarrollo la aplicación HPG_DHunter dedicada a la detección de Regiones de Metilación Diferencial. Para poder realizar esto, el software cuenta con una implementación de la Transformada Wavelet Discreta (DWT) que se ejecuta sobre CUDA.

Este proyecto se centra en la implementación de DWT sobre múltiples GPUs a través de una herramienta llamada EngineCL. Esta implementación se lleva a cabo utilizando EngineCL, una herramienta desarrollada por el grupo de Arquitectura y Tecnología de Computadores (ATC) de la Universidad de Cantabria, que permite la utilización eficiente de múltiples GPUs para ejecutar un solo kernel mediante el uso del lenguaje OpenCL.

Una vez todos los desarrollos estén finalizados se realizarán una series de pruebas sobre la aplicación. Posteriormente se llevará un análisis sobre los resultados obtenidos para comprobar que configuración del sistema ofrece un mayor rendimiento usando desde una a varias GPUs y encontrar beneficios sobre la implementación inicial. El análisis que se va a realizar tendrá en cuenta el tiempo de comunicación que existe con las GPUs, el tiempo de ejecución de las GPUs, la memoria del sistema y el tiempo total de la aplicación.

Palabras clave: Multi-GPU, Transformada Wavelet Discreta, HPG_DHunter, EngineCL

Abstract

Nowadays, graphics processing units (GPUs) have seen a growth in their uses, moving beyond their original role in hardware acceleration and gaming to become essential tools in scientific research.

In this aspect, the group of Group of Networks and Virtual Environments (GREV) from the University of Valencia, has developed an application dedicated to the detection of Differential Methylation Regions known as HPG_DHunter. In order to do this, the software has an implementation of the Discrete Wavelet Transform (DWT) running on CUDA.

This proyect focuses on the implementation of DWT on multiple GPUs through a tool called EngineCL. This implementation is carried out using EngineCL, a tool developed by the Computer Architecture and Technology (ATC) group of the University of Cantabria, which allows the efficient use of multiple GPUs to run a single kernel under the OpenCL language.

Once all the implementations have been developed, a series of tests will be carried out on the application. Afterwards, an analysis of the results obtained will be performed to check which configuration of the system offers the best performance using one to several GPUs in order to find benefits over the initial implementation. The analysis will take into account the communication time with the GPUs, the execution time on GPUs side, the system memory and the total time of the application.

Keywords: Multi-GPU, Discrete Wavelet Transform, EngineCL, HPG_DHunter

Índice general

Índice de figuras

Capítulo 1

Introducción

1.1. Sistemas Heterogéneos

Hoy en día el uso de las GPUs está extendido por todos los ámbitos de la informática. Su alta capacidad de cálculo y de procesado de datos ha aumentado de manera drástica desde la aparición de este componente [4]. De manera inicial, las GPUs fueron concebidas para el procesamiento gráfico de aplicaciones como la visualización de imágenes o el simple hecho de mostrar una interfaz.

Tras la introducción de las GPUs en el ámbito académico se revolucionó la forma en que se llevan a cabo los problemas más complejos. Gracias a su gran capacidad de procesamiento se empezaron a introducir en análisis de datos, modelados, simulaciones y en investigación científica [2, 33].

Para poder llevar a cabo todos estos cambios empezaron a aparecer lo que se conoce hoy en día como Sistemas Heterogéneos [34]. Estos sistemas consisten en la integración de componentes con diferentes arquitecturas, funciones y capacidades que trabajan de forma conjunta para poder ejecutar las tareas indicadas. Estos sistemas están formados por la combinación de distintos tipos de procesadores, como puede ser una CPU y una GPU o FPGA [9]

Los sistemas heterogéneos han permitido conseguir un gran avance en las simulaciones y modelados. Gracias a la existencia de la CPU convencional y al paralelismo que se puede obtener mediante la GPU, se han podido reducir el tiempo para la ejecución de simulaciones [18]. Este avance se extiende al análisis de datos climáticos, a la modelización de fenómenos físicos y químicos y a la investigación genómica.[1].

Estos nuevos sistemas permiten ejecutar aplicaciones específicas que se pueden dividir en tareas independientes que pueden ser llevadas a cabo en paralelo [15]. El paralelismo de datos, que consisten en aprovechar al máximo las unidades de cómputo para ejecutar operaciones similares en diferentes conjuntos de datos, es un caso especialmente beneficioso para las GPUs. Esto se debe a su arquitectura, al estar centrada en el paralelismo pueden realizar estas operaciones en paralelo a una mayor velocidad que una CPU aumentando la capacidad de procesamiento de manera significativa [1].

[1]Disciplina científica que se centra en el estudio exhaustivo de los genomas, que son los conjuntos completos de material genético (ADN) presentes en los organismos.

Este cambio en la arquitectura también se aplicó a las supercomputadoras, que se refleja en el ranking de las supercomputadoras más potentes del mundo conocido como el TOP500. Este ranking, disponible en `www.top500.org`, proporciona un listado de las supercomputadoras más avanzadas y observando las máquinas que ocupan los primeros lugares se observa que están construidas cono nodos con múltiples GPUs. La presencia de GPUs en estos sistemas permite efectuar cálculos intensivos y de forma paralela a una velocidad inimaginable, y mejorando la eficiencia energética [26].

1.2. Nuevos modelos de programación

La aparición de los sistemas heterogéneos provocó la necesidad de disponer de un nuevo modelo de programación que permitiese desarrollar aplicaciones aprovechando los recursos y las capacidades del sistema. Estos modelos de programación buscan simplificar la coordinación y comunicación entre los distintos tipos de procesadores. Dos de estos modelos de programación que más se han extendido han sido CUDA, OpenCL, así como otros de mayor abstracción como SCYL [29] y OneAPI [20], que no se utilizarán en el trabajo.

CUDA[2], desarrollada por NVIDIA, permite a los desarrolladores escribir código en lenguajes de programación como C o C++ que pueden ser ejecutados en GPUs de NVIDIA[3]. Proporciona una jerarquía de hilos y bloques para aprovechar al máximo el paralelismo masivo de las GPUs para acelerar las operaciones matemáticas y las tareas que conllevan la utilización de grandes cantidades de datos. Cabe destacar que CUDA es cerrado y propietario, pertenece a NVIDIA y solo puede ser usado en sus GPUs.

OpenCL [3] nace en 2008 a manos del consorcio Khronos Group [14]. Es de código abierto y, a diferencia de CUDA, se enfoca en ofrecer versatilidad y puede utilizarse en muchísimas plataformas, incluyendo CPUs, GPUs, FPGAs y otros aceleradores y procesadores, siempre y cuando el fabricante proporcione el correspondiente driver de OpenCL para el hardware.

Un aspecto importante en estos modelos de programación y en la arquitectura de las GPUs es el aprovechamiento del paralelismo masivo de threads para ocultar las latencias en el acceso a la memoria que en GPU son muy costosas [12]. Uno de los desafíos más notables en el diseño de sistemas de alto rendimiento es la latencia de acceso a la memoria, que puede ralentizar significativamente la ejecución de tareas. En las GPUs esto se puede ocultar gracias a la gran cantidad de núcleos de procesamiento que tienen. Cada núcleo puede ejecutar varios threads de manera simultánea. Esto permite que, mientras un thread está en espera a que se realice el acceso a memoria, otro thread puede realizar sus cálculos. Gracias a este solapamiento se pueden ocultar las latencias del acceso a memoria, mejorando la eficiencia del sistema y no dejar a los núcleos en espera mientras se procede a acceder a memoria.

Otro aspecto importante es el cambio de contexto en las GPUs [16]. Gracias a los amplios bancos de registro de los que disponen, este cambio de contexto se logra realizar de manera eficiente y prácticamente "gratis". Los registros son unidades de almacenamiento muy rápidas y locales a cada núcleo, permitiendo realizar los cambios de contexto con un coste muy reducido. Esto habilita a que un thread deje de ejecutarse para que otro pase a ejecución sin que implique una penalización de tiempo significativa.

[2]Compute Unified Device Arquitecture
[3]Open Computing Language

También es destacable de la arquitectura de las GPUs que los espacios de memoria entre la CPU y la GPU son independientes [17]. A diferencia de las CPUs, que comparten memoria con el sistema operativo y otros componentes, las GPUs cuentan con su propia de memoria dedicada. Esto permite una separación entre las tareas llevadas a cabo por la CPU y las que ejecuta la GPU. Sin embargo, esta separación requiere tener que gestionar la comunicación entre la CPU y la GPU, sobre todo en aplicaciones que requieren un intercambio de datos.

1.3. Dificultades con las implementaciones multi-GPU

La aparición de las GPUs y de los sistemas heterogéneos permite construir sistemas que contengan más de una GPU. Los modelos de programación comentados anteriormente se centran en la gestión de una única GPU, por lo que poder utilizar todos los dispositivos disponibles queda a mano del desarrollador [25].

Programar implementaciones que puedan aprovechar varias GPUs es un proceso complicado, aunque es una forma efectiva de mejorar el rendimiento en tareas que necesitan una gran cantidad de procesado de datos. Existen varios problemas a la hora de realizar estas implementaciones:

1. **División y Coordinación de Tareas:** Una de las tareas básicas al programar para múltiples GPUs es dividir las tareas complejas en trabajos más pequeños que puedan ser procesadas por cada GPU. Esto conlleva descomponer el trabajo en fragmentos independientes y asignarlos a las GPUs de manera que cuanto más división y utilización de la GPU se realice, aprovecharemos al máximo la GPU.

2. **Sincronización de Datos entre GPUs:** La sincronización de datos entre GPUs es fundamental para evitar resultados incorrectos o inconsistentes [28]. Las GPUs pueden trabajar en tareas diferentes que requieren acceso a los mismos datos compartidos. Coordinar el acceso y la modificación de estos datos compartidos de manera sincronizada es un desafío para garantizar la coherencia y la integridad de los resultados.

3. **Balanceo de Carga en Tiempo Real:** Dado que las tareas pueden variar en su complejidad y duración, el balanceo de carga es esencial para asegurar que cada GPU esté siendo utilizada de manera eficiente. Si una GPU está sobrecargada mientras que otra está inactiva, el rendimiento general de la aplicación se verá afectado [27].

4. **Comunicación y Transferencia de Datos:** La comunicación y la transferencia de datos entre dispositivos es un gran problema al trabajar con múltiples GPUs. Mover datos entre la memoria de la CPU y la memoria de cada GPU requiere tiempo y recursos del sistema, lo que puede impactar negativamente en el rendimiento [32].

5. **Depuración y Seguimiento en Entornos Multi-GPU:** La depuración y el seguimiento de código en un entorno multi-GPU es un gran problema. Identificar problemas de sincronización, acceso incorrecto a memoria y otros errores específicos de entornos paralelos puede ser más difícil. Además, la monitorización del rendimiento y la identificación de errores también son más complicados en sistemas con varias GPUs.

1.4. Motivación y explicación del problema

El objetivo de este trabajo es mejorar el rendimiento del software HPG_DHunter[7]. HPG_DHunter es una aplicación que permite la visualización de señales de metilación y la detección automá-

tica de Regiones de Metilación Diferencial (DMR), basándose en la información generada por la Transformada Wavelet Discreta (DWT). La aplicación fue creada por el Grupo de Redes y Entornos Virtuales (GREV) de la Universidad de Valencia. Esta ha sido validada y desarrollada basándose en un artículo previo de dicho grupo[8]. Además, se ha empleado en diversos artículos que abordan la implementación de una página web que hace uso de la aplicación[6], así como en investigaciones relacionadas con la diabetes[19].

La motivación detrás de esta mejora proviene de intentar reducir los tiempos de respuesta que se obtienen al procesar muestras complejas. A pesar de que el software ya está desarrollado en CUDA, permitiendo el uso de una única GPU para el procesamiento, la cantidad de las muestras a procesar por la aplicación puede llevar a tiempos de respuesta muy altos.

Una posible solución hubiera sido desarrollar desde cero toda la aplicación multi-GPU en CUDA, teniendo que tener en cuenta todos los problemas descritos en la sección anterior. Una alternativa más eficiente es trasladar el código a EngineCL[23], una herramienta desarrollada por el grupo de ATC (Arquitectura y Tecnología de Computadores) de la Universidad de Cantabria. Esta herramienta nos permitiría poder utilizar sistema heterogéneos con múltiples GPUs de manera sencilla, encargándose de realizar la gestión de los dispositivos, la distribución de tareas y de datos entre los dispositivos, entre otros beneficios, teniendo en cuenta el equilibrio de carga de trabajo, ya sea estático o dinámico, entre ellos. La herramienta ha sido utilizada en múltiples artículos[5, 21, 11, 10] con la premisa de mejorar el rendimiento debido a su implementación, pasando por un estudio con FPGAs[30] hasta la optimización y la eficiencia energética en sistemas paralelos[22]. Hay que comentar que EngineCL trabaja con OpenCL, por lo que será necesario, desarrollar primero una versión de la aplicación en OpenCL, en lugar de CUDA.

Este cambio conllevará una serie de beneficios. En primera instancia, la velocidad de ejecución debido a la utilización de múltiples GPUs que permitirá obtener los resultados de forma más rápida y ahorra tiempo. En segundo lugar, hay que tener en cuenta la escalabilidad que nos otorgará. La escalabilidad permite ejecutar instancia del problema más grandes, al poder dividir el conjunto de datos entre las diferentes memorias de las GPUs. Por último, al utilizar múltiples GPUs y adaptar la aplicación a este tipo de implementaciones, esta se ajustará dinámicamente a diversos sistemas en los cuales sea ejecutada, lo que otorgará una mayor portabilidad a la aplicación. La portabilidad nos permitirá ajustar el mismo código a diferentes sistemas (no necesariamente más grandes). Así, puedes cambiar no sólo el número de GPUs, sino también el modelo y la capacidad de proceso de cada GPU y la aplicación sigue siendo eficiente.

1.5. Objetivos

El objetivo principal del presente trabajo de fin de máster se centra en la implementación Multi-GPU de algoritmos de análisis de metilación del ADN partiendo de la aplicación HPG_DHunter desarrollada por la Universidad de Valencia. Ahora mismo esta aplicación hace uso de la plataforma de computación CUDA empleando GPUs de NVIDIA. Aunque usar CUDA permite la ejecución de los cálculos de manera rápida y potente por usar la GPU, sigue teniendo la restricción de un tiempo de espera alto cuando la carga de trabajo se lleva a un ejemplo real y hay que incluir la limitación de solo poder ejecutar la aplicación sobre tarjetas NVIDIA. Otra limitación con las GPUs reside en la memoria que disponen para poder almacenar datos ya que es limitado y no ampliable. Esto añade una limitación fuertes en las GPUs (en comparación con los servidores de alto rendimiento)

Teniendo todo esto en cuenta, los objetivos que se quieren obtener con este trabajo son:

- Estudio de la aplicación actual y cómo se hace uso de la GPU mediante CUDA.
- Proponer una nueva implementación sobre una plataforma Open Source, en concreto sobre OpenCL.
- Proponer una implementación multi-GPU con el uso de EngineCL.
- Mejorar el rendimiento y la escalabilidad de la aplicación, aprovechando varias GPUs.
- Evaluación sobre las nuevas implementaciones con el objetivo de comprobar si existe mejoría.

1.6. Plan de trabajo

De cara a cumplir los objetivos presentados en el anterior apartado, incrementando la portabilidad de la aplicación y la obtención de los resultados deseados, el trabajo se ha organizado de la siguiente manera:

- **Portado del código CUDA al modelo de programación de OpenCL:** El proceso de reimplementación implica analizar y adaptar el código de CUDA al marco de trabajo de OpenCL, asegurando que la funcionalidad y el rendimiento del código no se vean comprometidos. Esto conlleva realizar ajustes en el código, y en algunos casos, reescribirlo para que sea compatible con la estructura de OpenCL. Es importante destacar que el portado a OpenCL no solo hace que el código sea más accesible, sino que también puede mejorar su rendimiento y la eficiencia. OpenCL es una plataforma que se adapta a diferentes tecnologías y dispositivos, lo que permite que el código pueda ser ejecutado de forma más eficiente en diferentes entornos.
- **Realizar una implementación multi-GPU utilizando EngineCL:** Esta herramienta parte de la plataforma de OpenCL por lo que en primera instancia, y como se ha indicado antes, se necesita hacer una traducción de esta herramienta a OpenCL para poder implementarla. La herramienta dispone de algoritmos de equilibrio de carga tanto estáticos como dinámicos cambiando la distribución entre GPUs.
- **Validación de las nuevas implementaciones:** Las pruebas se realizarán con distintas cargas de trabajo y situaciones en las que se puede llegar a usar la aplicación para comprobar si el cambio realizado conlleva una mejoría con respecto a la implementación inicial. Para comprobar si existen mejorías se medirán los tiempos de comunicación entre el host y los devices y viceversa, los tiempos de ejecución para los cálculos sobre la GPU y la memoria de sistema tanto del host y de los devices.

1.7. Estructura del documento

Aparte de este capítulo de introducción, el documento se compone de otros 4 capítulos más con un total de 5. Los capítulos siguientes son los siguientes:

- **Capítulo 2: Background.** En este segundo capítulo se expondrán ciertos conocimientos para poder entender qué se intenta conseguir con la aplicación que estamos modificando, se comentaran las herramientas y plataformas que se han utilizado para poder realizar todos estos cambios y se mostrarán algunos trabajos relacionados con el software a transformar y las herramientas utilizadas para llevarlo a cabo.

- **Capítulo 3: Diseño e Implementación.** En este capítulo se explicarán en más profundidad el análisis inicial de la aplicación partiendo de la implementación de CUDA, se explicará el diseño y los cambios realizados para poder transcribir la aplicación a OpenCL y, por último, se explicará el diseño e implementación realizado para llevar a cabo la implementación en EngineCL.
- **Capítulo 4: Experimentos y resultados.** El capítulo se centra en las pruebas y resultados obtenidos sobre todas las implementaciones de las que disponemos, tanto en un solo dispositivo como en varios, lo que nos permitirá conocer si existen beneficios utilizando tanto la implementación realizada sobre OpenCL como EngineCL.
- **Capítulo 5: Conclusiones y trabajos futuros.** Este último capítulo mostrará las conclusiones obtenidas a partir de las implementaciones y pruebas realizadas sobre cargas de trabajo de diversos tamaños. También tendrá una explicación de los trabajos a futuro que se podrán llevar a cabo partiendo de este trabajo como base.

Capítulo 2

Background

Durante esta sección se explicarán ciertos conceptos básicos para entender el proyecto de HPG_DHunter que se intenta trasladar a EngineCL, y se darán ciertas nociones sobre la herramienta de EngineCL necesarios para poder entender posteriormente el beneficio que se intenta obtener.

2.1. Metilación del ADN

El análisis de metilación del ADN ha emergido como una valiosa herramienta en la investigación genómica, proporcionando información crucial sobre la regulación de genes y la epigenética[31]. El presente trabajo de fin de máster se centra en la Implementación Multi-GPU de Algoritmos de Análisis de Metilación del ADN, haciendo uso de la aplicación HPG_DHunter desarrollada por la Universidad de Valencia.

El ADN es una molécula que contiene toda la información genética de los seres vivos, como un libro de instrucciones que guía cómo funcionamos y nos desarrollamos. Pero, además del ADN, existen otros factores que afectan en cómo estas instrucciones tienen que ser interpretadas. Haciendo una similitud con la informática podríamos referirnos a ellos como las líneas de código que contiene un programa.

La metilación[13] es una modificación química del ADN y otras moléculas que puede conservarse a medida que las células se dividen para generar más células. Cuando se produce en el ADN, la metilación puede alterar la expresión génica. En este proceso, etiquetas químicas denominadas grupos metilo se unen a un sitio en particular en el ADN, donde activan o desactivan un gen, y de esa forma regulan la producción de las proteínas que el gen codifica. La metilación del ADN, en un intento de acercarlo a la similitud anterior, sería como las comprobaciones que se realizan en la líneas de código sobre variables de control las cuales permiten modificar el funcionamiento del programa dependiendo de lo que se le indique.

En resumen, la metilación del ADN es un proceso que modifica el ADN sin cambiar su secuencia, y funciona como un interruptor que controla qué genes se activan y cuáles se apagan en nuestras células. Este mecanismo es esencial para el funcionamiento adecuado de nuestro cuerpo y tiene un papel importante en cómo crecemos, nos desarrollamos y respondemos al ambiente que nos rodea.

2.2. Aplicación HPG_DHunter

HPG_DHunter es una aplicación gráfica fácil de usar que permite la visualización interactiva de señales de metilación y la detección automática de Regiones de Metilación Diferencial (DMR), basándose en la información generada por la Transformada Wavelet Discreta (DWT). Esta aplicación de software es capaz de detectar y mostrar DMRs en diferentes escalas de muestras diversas.

Figura 2.1: Interfaz de la aplicación. Fuente artículo HPG_DHunter[7]

El funcionamiento de la aplicación se basa en transformar la información del nivel de metilación (y/o hidroximetilación) de cada base en el genoma de referencia en una señal de metilación. Luego, se aplica la Transformada Wavelet Discreta a esta señal para detectar DMRs en el espacio de transformación. La DWT realiza un promedio local de la señal de metilación en distintos niveles de resolución, similar a los enfoques de suavizado utilizados en otros métodos. Esta media local tiene en cuenta la variabilidad biológica presente en los mapas de metilación, al tiempo que proporciona una medición más precisa del nivel de metilación para segmentos más extensos de ADN. Al utilizar una descomposición dicádica de la DWT, esta estrategia logra una reducción logarítmica en la cantidad de valores utilizados para comparar señales y detectar DMRs en comparación con otros métodos basados en suavizado. Esto resulta en una significativa reducción de la carga computacional requerida para detectar DMRs.

2.2.1. Implementación de la Transformada Wavelet Discreta

La Transformada Wavelet Discreta de Haar (DWT) es una técnica de procesamiento de señales utilizada para descomponer una señal en diferentes componentes de frecuencia, lo que

permite analizar y representar la señal en diferentes niveles de detalle. Fue propuesta por primera vez por Alfréd Haar en 1909 y es una de las transformadas wavelet más simples y básicas.

La DWT se aplica a señales discretas y utiliza una función wavelet especial llamada "wavelet de Haar", que es una función rectangular escalada y desplazada. Esta wavelet es única en el sentido de que solo tiene dos valores distintos, +1 y -1, en ciertos intervalos y cero en todos los demás lugares.

En esta aplicación la DWT se aplica a la señal de metilación del ADN. La señal de metilación del ADN se suele entender como una secuencia de valores numéricos que representan los niveles de metilación en diferentes regiones del ADN. En la implementación utilizada se denota el uso de un filtro de paso bajo. La idea detrás de aplicar un filtro de paso bajo después de la DWT en la señal de metilación del ADN es reducir el ruido o las fluctuaciones de alta frecuencia que puedan haber sido capturadas en los niveles de la transformada. Esto puede ser útil para obtener una representación más suave y clara de la señal de metilación, lo que puede facilitar la detección de patrones biológicos y características relevantes.

2.3. Herramienta EngineCL

EngineCL es una herramienta de runtime basada en OpenCL que simplifica la programación de sistemas heterogéneos y que simplifica notablemente la co-ejecución de un único kernel masivamente paralelo de datos en todos los dispositivos de un sistema heterogéneo. La implementación llevada a cabo tiene como objetivo aumentar el nivel de abstracción del programador de aplicaciones multi-GPU. Se encarga de detectar los dispositivos existentes en el sistema, hacer la partición de los datos de entrada, distribuirlos entre los dispositivos y ensamblar los datos de salida proporcionados por cada dispositivo. También permite la integración con distintos planificadores que distribuyen la carga de trabajo entre distintos dispositivos de forma proporcional a su capacidad de cómputo. En la actualidad se han implementados dos algoritmos de planificación. EL primero es estático y divide la carga de trabajo, antes de iniciar el cómputo, en función de parámetros de entrada proporcionados por el programador. El segundo método es dinámico, ya que divide la caga de trabajo en una gran cantidad de paquetes que asigna a los dispositivos bajo demanda. Para poder suportar múltiples arquitecturas, no solo NVIDIA, EngineCL se basa en OpenCL permitiendo hacer uso de cualquier dispositivo que disponga de un driver para ello.

EngineCL ha sido desarrollado en C++, utilizando principalmente características modernas de C++11 para reducir la sobrecarga y el tamaño del código al proporcionar un mayor nivel de abstracción. EngineCL cuenta con una arquitectura multi-hilos que combina las mejores técnicas medidas en cuanto a la gestión de colas, dispositivos y buffers de OpenCL. Estos mecanismos son utilizados internamente en tiempo de ejecución y están ocultos al programador para lograr ejecuciones eficientes y una gestión transparente de dispositivos y datos.

El runtime desarrollado se divide en tres capas. La primera y segunda capa son aquellas accesibles por el programador donde se sitúa todo el diseño de la aplicación teniendo en cuenta todo lo que esto conlleva. En la primera capa nos encontramos con el *Engine* que es una clase dedicada a la ejecución del programa donde se aplica un patrón de diseño conocido como **Facade**, que permite la abstracción de toda la compleja estructura y arquitectura del programa permitiendo un punto de acceso sencillo para el programador que lo usa. Este *Engine* se configura mediante un *Program* que contiene el kernel de ejecución y los buffers de entrada y salida de los datos. Pasando a la segunda capa nos encontramos con que el *Engine* hace uso del *Runtime* para poder

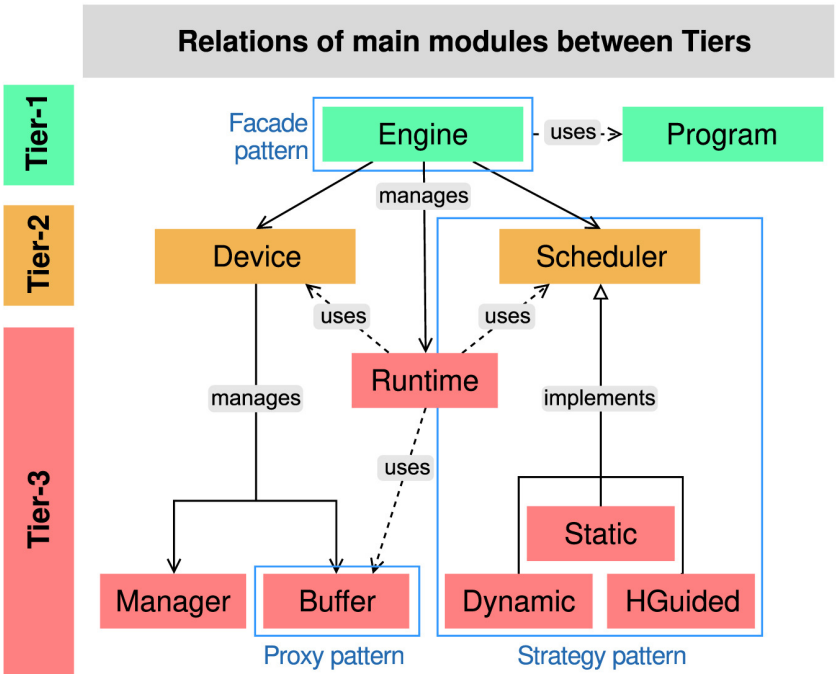

Figura 2.2: Relaciones entre capas de EngineCL. Fuente artículo EngineCL[23]

llevar a cabo todas las tareas necesarias. Este *Runtime* hace uso de las clases *Device* y *Scheduler*. El *Device* hace referencia a los dispositivos utilizados para poder llevar a cabo las tareas mandadas por el *Runtime* (*Manager* y *Buffer*). El *Scheduler* hacer referencia a los balanceadores de carga sobre los que la aplicación puede hacer uso si están implementados (*Static* y *Dynamic*).

Recordando los problemas comentados anteriormente sobre las implementaciones multi-GPU, EngineCL soluciona muchos de estos problemas encargándose de distribuir y coordinar las tareas entre los dispositivos, sincronizar correctamente los datos entre los dispositivos, balancear la carga de trabajo entre GPUs, etc.

2.4. Trabajos relacionados

2.4.1. EngineCL

EngineCL es un potente sistema en tiempo de ejecución basado en OpenCL que se ha utilizado en diversos trabajos de investigación para simplificar la programación de sistemas heterogéneos y permitir la co-ejecución eficiente de un único kernel masivamente paralelo de datos en todos los dispositivos de un sistema heterogéneo. Los siguientes trabajos resaltan la integración y aplicación de EngineCL en diversos escenarios de computación:

Un grupo de trabajos de investigación ha destacado la integración y aplicación de EngineCL en distintos escenarios de cómputo. Uno de ellos explora cómo EngineCL facilita la asignación de recursos de cómputo en dispositivos heterogéneos en el borde de la red[5]. Esto beneficia la implementación de algoritmos de aprendizaje federado, mejorando la red en sí.

En otro estudio, se describe cómo EngineCL se ha unido a un enfoque eficiente de programación heterogénea para FPGAs[30]. Esto ha demostrado ser particularmente efectivo en

situaciones donde la naturaleza reconfigurable de las FPGAs necesita ser gestionada de manera óptima para obtener un rendimiento óptimo.

Además, hay una investigación que explora cómo EngineCL puede mejorar los kernels híbridos en simulaciones de dinámica molecular[21]. Aquí, se introduce un algoritmo de equilibrio de carga que distribuye de manera eficaz la carga computacional en recursos de cómputo heterogéneos, utilizando las capacidades de EngineCL.

En otro contexto, EngineCL desempeña un papel clave en un marco de trabajo llamado Lightning[11]. Este marco amplía las capacidades del modelo de programación de GPU más allá de una única GPU, permitiendo que varias GPU trabajen juntas de manera cooperativa y fluida en entornos de cómputo heterogéneos.

En el ámbito del aprendizaje automático, un grupo de autores ha propuesto un programador de tareas basado en aprendizaje automático que hace uso de EngineCL[10]. Este programador asigna tareas de manera dinámica a los recursos de cómputo adecuados en sistemas heterogéneos, optimizando así la utilización de recursos y maximizando el rendimiento del sistema mientras se minimiza el consumo de energía.

Por último, otra investigación se ha centrado en la optimización del rendimiento y la eficiencia energética en sistemas masivamente paralelos[22]. EngineCL ha sido un elemento esencial en este proceso, contribuyendo a alcanzar la informática de alto rendimiento con un consumo mínimo de energía en plataformas heterogéneas. En conjunto, estos estudios muestran cómo EngineCL se ha convertido en una herramienta crucial en el mundo de la programación heterogénea y la optimización del rendimiento en sistemas complejos de cómputo.

2.4.2. HPG_DHunter

HPG_DHunter ha sido aplicado exitosamente en diversas investigaciones, proporcionando una valiosa aplicación para la detección de Regiones de Metilación Diferencial y la visualización interactiva de señales de metilación en estudios epigenéticos y análisis de ADN. Los siguientes trabajos resaltan la integración y aplicación de HPG_DHunter:

Uno de esos proyectos se centró en crear una plataforma web que permitiera el acceso a HPG_DHunter de manera online[6], lo que facilitó su uso para un amplio público. Esta aplicación, diseñada para analizar muestras de diferentes escalas, se destacó por su capacidad automatizada para identificar DMRs.

En otro rincón de la investigación, se exploró cómo las arquitecturas paralelas podrían impulsar el análisis de metilación del ADN[24]. Aquí, HPG_DHunter entró en juego en entornos de computación paralela. La incorporación de esta aplicación aceleró y optimizó el proceso de detección de DMRs en conjuntos masivos de datos, resaltando su eficiencia en análisis de metilación complejos.

En un estudio relacionado con la diabetes tipo II[19], los investigadores recurrieron a HPG_DHunter para analizar las regiones de metilación diferencial en el exoma de pacientes. Este enfoque les permitió identificar y visualizar DMRs asociados con la enfermedad. El uso de HPG_DHunter proporcionó una nueva perspectiva sobre la epigenética y la patogénesis de la diabetes tipo II.

Así, a lo largo de estas investigaciones, HPG_DHunter demostró su versatilidad y potencial para contribuir en el análisis de metilación y en la comprensión de procesos biológicos complejos. Su presencia en estos estudios no solo simplificó el proceso de detección de DMRs, sino que también amplió nuestra visión de la genética y la epigenética en diferentes contextos científicos.

Capítulo 3

Diseño e Implementación

En este capítulo se van a presentar el diseño e implementación inicial de CUDA y las nuevas implementaciones usando OpenCL y EngineCL que cambian cómo la aplicación se comunica con una (CUDA y OpenCL) o varias GPUs (EngineCL) para realizar el procesado de datos en GPU. El documento se estructura en varios apartados clave. En el primero, se presenta la arquitectura cliente-servidor necesaria para la utilización de la aplicación HPG_DHunter. El segundo apartado ofrece un panorama general del flujo de trabajo de la aplicación HPG_DHunter. El tercer segmento analiza la fase inicial de implementación de la aplicación utilizando la tecnología CUDA. El cuarto apartado se enfoca en exponer tanto el diseño como la implementación de la aplicación, enfocado en la migración de CUDA a OpenCL. El quinto apartado muestra el diseño y desarrollo específico para la integración de EngineCL en lugar de OpenCL. Por último, el sexto apartado enumera las limitaciones identificadas durante la experiencia de emplear EngineCL.

3.1. Arquitectura servidor-cliente para el uso de la aplicación

En esta primera sección se va a analizar cuál es la estructura utilizada para poder dar uso a la aplicación de HPG_DHunter y poder llevar los datos necesarios a la GPU, realizar los cálculos y traer de vuelta el resultado. Teniendo en cuenta lo anterior, se va a explicar cuál es el flujo a seguir para poder hacer uso de HPG_DHunter y cuál es su flujo de trabajo.

Como se observa en la figura 3.1 se dispone una arquitectura dedicada para poder utilizar la aplicación:

1. El primer servicio presentado levanta un servidor local en la máquina donde se está ejecutando, y permite recibir llamadas para poder llevar a cabo las ejecuciones. Tras recibir la llamada el servidor decide el tipo de tarea que tiene que realizar dependiendo del tipo de operación que se le ha indicado mediante la llamada.

2. El segundo servicio esta dedicado a mandar las peticiones al servidor y en el cuerpo de la llamada mandar toda la información y parámetros necesarios para que la aplicación ejecute correctamente.

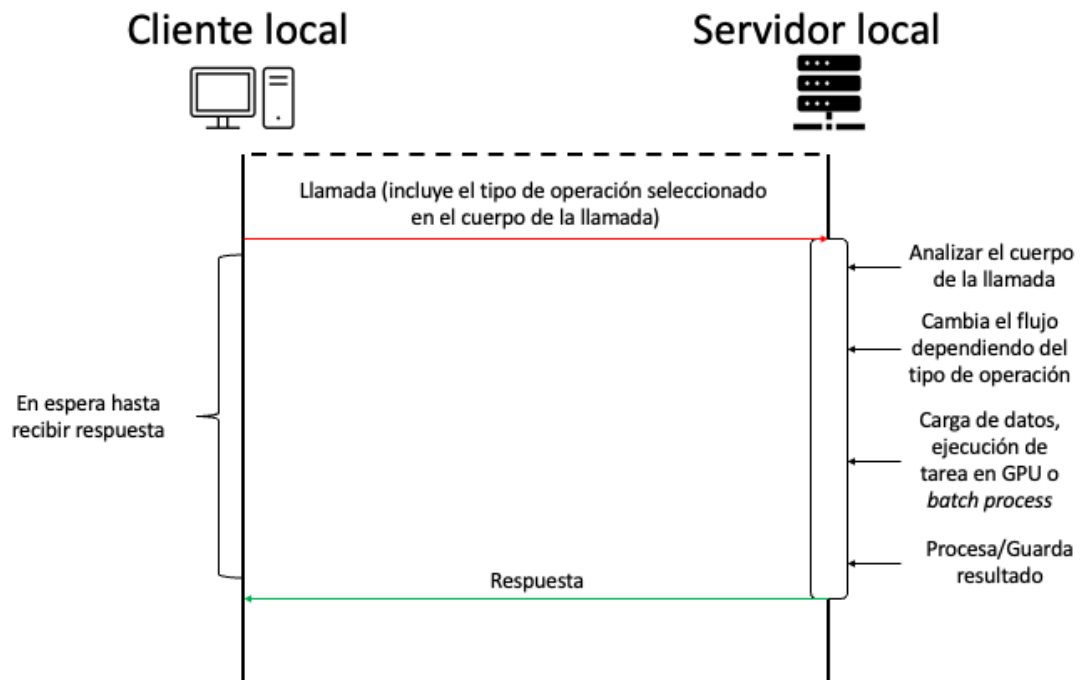

Figura 3.1: Arquitectura servidor-cliente implementada.

En la figura 3.2 se muestra cómo esta arquitectura permite la creación de colas de trabajo para poder acumular varias tareas del mismo tipo y realizar varias tareas distintas al mismo tiempo, permitiendo a varios clientes interactuar con el servidor. Dependiendo de la tarea que se indique, el servidor mandará el trabajo a la cola correspondiente a ese tipo de trabajo. Si no hay ninguna ejecución en el momento o la cola de trabajos esta vacía se llevará a cabo la tarea de manera inmediata.

Si la operación consiste en la *carga de datos* (parte inferior de la figura 3.2), se encarga de generar una cola de tareas de carga de ficheros sin que afecte a las operaciones sobre la GPU, lo que permite solapar el cómputo y las comunicaciones entre CPU y GPU. Si la tarea acaba correctamente se devuelve un mensaje satisfactorio al cliente; sin embargo si no se dispone de suficiente memoria para llevar a cabo la carga de datos se notifica al cliente mediante un mensaje de error indicando cuál es el problema.

Si la operación consiste en una *tarea de GPU* (parte media de la figura 3.2) se tiene que verificar que la GPU se encuentra disponible antes de cargar los datos y realizar los cálculos para la Transformada Discreta Wavelet (DWT) de la señal de metilación.

Existe una ultima operación definida como *batchprocess* (parte superior de la figura 3.2). Esta es la operación que nos interesa para seguir el flujo que ejecuta la aplicación que estamos modificando. Esta operación es larga y no hay ningún tipo de interacción, aparte del inicial donde se indican las variables de control. Básicamente el servidor se encarga de leer y preparar todos los datos y variables de control necesarias para poder ejecutar la aplicación.

La operación *batchprocess* es el que tenemos que llevar a cabo para ejecuta la herramienta de HPG_Dhunter. Esta aplicación tiene la limitación de que no tiene ningún tipo de interacción, a parte de los logs que va escribiendo. Las ejecuciones pueden llevar bastante minutos o

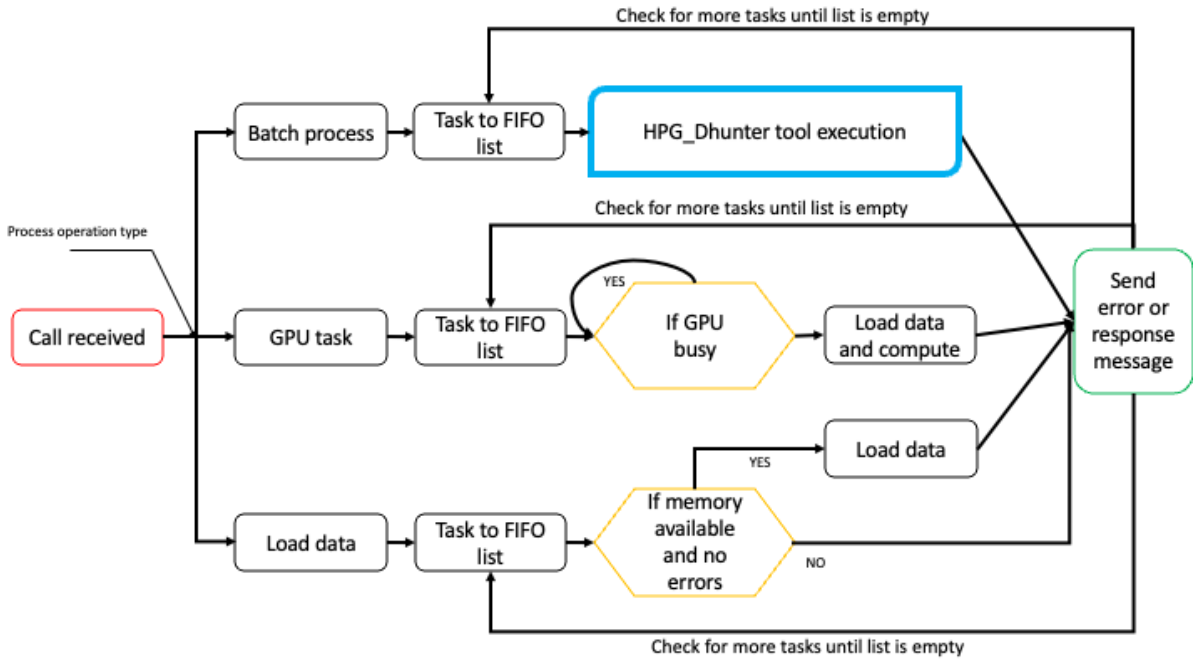

Figura 3.2: Flujo de trabajo del servidor. Fuente artículo HPG_DHunter[7]

incluso horas y aunque los tiempos de espera son altos no se obtiene ningún resultado hasta la finalización de ejecución. Si la aplicación no puede seguir por el flujo apropiado el error se refleja en los logs y mediante un envío de un mensaje al cliente, quien se tiene que quedar en espera hasta que el servidor acabe el proceso.

3.2. Flujo de trabajo de HPG_DHunter

Como se ha comentado en anteriores capítulos la aplicación HPG_DHunter permite mostrar de manera gráfica la visualización interactiva de señales de metilación y la detección automática de Regiones de Metilación Diferencial (DMR), basándose en la información generada por la Transformada Wavelet Discreta (DWT).

Para poder analizar el código que nos interesa nos centraremos en una aplicación dedicada conocida cómo HPG_DHunter_batch que comprende todo lo relacionado con la búsqueda de los DMRs. La aplicación se centra en la lectura de datos, el análisis de la señal de metilación y la búsqueda de los DMRs. Para poder ejecutar esta aplicación hay que indicar al servidor que la operación que queremos realizar es "batchprocess".

Como se muestra en la figura 3.3, la aplicación divide el trabajo en dos: **Trabajo del host y trabajo del dispositivo.** El trabajo del host es el que se ejecuta en la CPU mientras que el trabajo del dispositivo es el que se ejecuta en GPU.

El primer trabajo a realizar por la CPU es preparar los datos necesarios que son cargados en memoria y verifica que los parámetros de control han sido definidos. Estos parámetros de control definen desde los directorios donde tiene que acceder la aplicación para leer los datos y escribir los resultados y todos aquellos elementos que tienen que ver con la metilación, los limites que se imponen para aplicar DWT y ciertos elementos definidos para poder detectar las

DMRs.

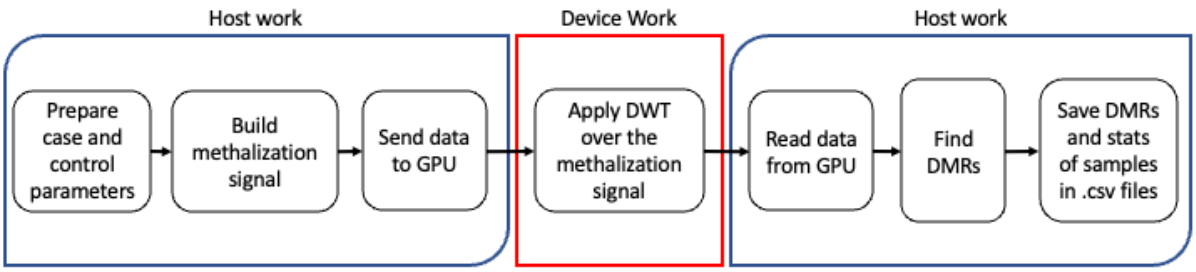

Figura 3.3: Flujo de trabajo de la aplicación. Fuente artículo HPG_DHunter[7]

Tras la recopilación de los datos necesarios y la preparación de los parámetros de control pertinentes, la aplicación avanza y construye la señal de metilación. Esto implica la manipulación y combinación de los datos leidos para generar la representación de la señal. Una vez construida la señal de metilación, el siguiente paso consiste en transferirla a la GPU para llevar a cabo los cálculos. Esta transferencia a la GPU aprovecha su capacidad para realizar cálculos en paralelo, lo que acelera significativamente el procesamiento de datos y análisis.

Después de que los datos se hayan enviado a la GPU, es necesario comunicarle las tareas que debe realizar. Una vez que la GPU recibe estas instrucciones, el sistema se queda en espera mientras la GPU completa las tareas indicadas. Esta espera tiene como objetivo asegurar la sincronización entre el sistema y la GPU para poder utilizar los resultados generados por la GPU. Una vez que la GPU ha realizado las tareas que se le han indicado se procede a obtener los resultados. Estos resultados se transfieren desde la GPU a la memoria del sistema, permitiendo que los datos estén disponibles nuevamente para su posterior manipulación y análisis.

Después de la transferencia de datos desde la GPU, la aplicación de análisis se concentra en la identificación de las DMRs. Estas DMRs representan áreas del genoma donde se observan cambios significativos en los patrones de metilación entre las muestras analizadas.

Finalmente, una vez que se han identificado y caracterizado las DMRs, la aplicación almacena los resultados obtenidos junto con las estadísticas relacionadas de las muestras en archivos CSV para su posterior revisión y análisis.

En el siguiente apartado se desarrollarán los pasos relacionados con la copia de datos a la

GPU, la definición y envío de tareas a la GPU y la posterior copia de datos al sistema. Todos estos pasos están relacionados con el uso de la plataforma CUDA que es la parte modificada para realizar el resto de implementaciones.

3.3. Análisis de la implementación inicial en CUDA

Esta sección se centra tanto el proceso de diseño como la implementación concreta realizada en CUDA por los autores originales, desde la migración eficiente de datos hasta la ejecución paralela optimizada de los cálculos de la DWT en la señal de metilación. El resultado del uso de CUDA es una aceleración significativa en la transformación de la señal de metilación, lo que permite encontrar DMRs de manera más eficiente.

3.3.1. Encapsulación del código CUDA

Dentro del diseño original, **existe un archivo de código dedicado a todo lo relacionado con la GPU**. Este archivo actúa como un contenedor principal que **encapsula todas las funcionalidades relacionadas con la plataforma CUDA**. Esto ofrece varios beneficios:

1. **Organización:** Tener todo el código CUDA encapsulado facilita la navegación teniendo definido donde se sitúa todo el código relacionado con el uso de la GPU. Cada función y método está diseñado para llevar a cabo tareas específicas relacionadas con la GPU, lo que hace que el código sea más estructurado y legible.

2. **Mantenibilidad:** La separación de la lógica del código relacionado con CUDA aparte permite realizar modificaciones y mejoras en el código sin afectar otras partes de la aplicación. Esto será muy útil a posterior cuando tengamos que realizar las nuevas implementaciones ya que solo implicará manejar este archivo.

3. **Pruebas y depuración:** La encapsulación de todo el código de CUDA facilita la depuración de problemas relacionados con la parte del host y de los datos que se envían a la GPU. Controlar y hacer depuración en GPU es otro mundo completamente diferente y más complicado de realizar.

Gracias a esta encapsulación solo se requieren cuatro líneas de código para utilizar la GPU. Estas cuatro líneas de código encapsulan todo el proceso, desde la preparación, que conlleva la copia de datos a la GPU, hasta la finalización, que conlleva la ejecución de los cálculos en la GPU y la copia de datos de vuelta a la memoria del sistema.

```
1 // libera la memoria de la GPU
2 cuda_end(cuda_data);
3
4 // envia el total de los datos a la GPU
5 cuda_send_data(cuda_data);
6
7 // procesado de los datos
8 cuda_calculo_haar_L(cuda_data);
9 cuda_main(cuda_data);
```

Figura 3.4: Uso de las funciones dedicados a GPU en el flujo principal de la aplicación.

Como se puede observar en el listado 3.4, la **línea 2** se encarga de liberar la memoria reservada en la GPU, garantizando que no se mantiene ningún dato utilizado previamente. Esta

limpieza es esencial para evitar posibles conflictos o errores de memoria que podrían surgir durante la ejecución.

En la **línea 5** se utiliza un método dedicado a la copia de los datos necesarios que son enviados desde la memoria principal a la memoria de la GPU. Esta operación es fundamental para que los cálculos se realicen en la GPU y aprovechen su capacidad de procesamiento paralelo.

Posteriormente en la **línea 8** nos encontramos con un método dedicado a obtener el número de datos que se necesitan obtener a posterior de la matriz de resultados con el nivel indicado para aplicar la DWT. Este número sirve como delimitador a la hora de copiar los resultados de vuelta en la CPU.

Y por ultimo, en la linea 9 tenemos la ejecución del método principal de la clase el cual se dedica a lanzar el kernel dedicado a la GPU y realiza la sincronización con la misma para posteriormente realizar la copia de datos de la memoria de la GPU a la memoria del sistema.

3.3.2. Exploración del código CUDA

Como se ha podido observar en el anterior apartado la inclusión del uso de la GPU se vuelve muy sencilla teniendo solo que añadir cuatro líneas de código. Aunque en la ejecución del código solo se añaden esas líneas **existen varias funciones ocultas que son necesarias y obligatorias para el correcto uso de la aplicación.** Dentro de este archivo con la encapsulación del código tenemos varias funciones que podemos dividir en dos conjuntos:

- **Funciones ejecutadas por el host:** cuda_init(), cuda_end(), cuda_send_data(), cuda_calculo_haar_L(), cuda_main(),
- **Funciones ejecutadas por el device:** wavedec(), transform(), copyValues()

Cabe destacar que en el código se define una estructura de datos fija la cual contiene todas las variables y elementos de control utilizados en las implementaciones, este parámetro se recibe en todos los métodos referentes al host y es donde se obtienen y se guardan datos.

Funciones ejecutadas por el host

Son aquellas funciones que se ejecutan en la CPU del sistema pertenecientes al programa que se esta ejecutando. Estas funciones son quienes realizan el envío de datos a la GPU, indican qué tarea tienen que realizar y posteriormente recuperan el resultado de la GPU.

cuda_init()

Esta es una función sencilla que no recibe ningún parámetro y cuya función es obtener información básica de los dispositivos conectados al sistema y mostrarla por pantalla. Cierta de la información que recopila tiene que ver con el número de dispositivos conectados, la GPU a ser utilizado por la API de CUDA, el nombre de la GPU y ciertas propiedades básicas sobre la GPU que va a utilizar.

cuda_end()

Es una función dedicada a la gestión de memoria de la GPU. Libera la variable del objeto que contiene el vector de datos en la GPU, y hace lo respectivo con el vector de ayuda que se

emplea durante la ejecución de los kernels de la GPU. Este método es el que se encuentra en la línea 2 de la figura 3.4 para mantener la memoria de la GPU limpia entre ejecuciones evitando conflictos o errores de memoria en la ejecución.

cuda_send_data()

Esta función realiza dos acciones principales: Reserva el espacio en la memoria de la GPU y copia la matriz de datos a la memoria recién asignada en la GPU. Esta primera acción es llevada a cabo utilizando la función **cudaMallocPitch** que permite realizar una reserva de memoria en GPU de manera un poco especial ya que aplica un desplazamiento final en los datos introducidos para que se alineen con la memoria de la GPU y reducir los fallos de memoria al obtener datos en GPU. Este desplazamiento calculado se guarda en la estructura de datos del programa para ser utilizada posteriormente.

La segunda acción de esta función se encarga de enviar los datos desde la memoria principal a la GPU y es llevada a cabo por la función **cudaMemcpy2D**.

cuda_calculo_haar_L()

Esta función tiene como objetivo encontrar el número de datos resultantes tras aplicar la DWT multinivel. Por cada nivel que se quiera ejecutar se calcula el número de datos que devuelve la DWT. Todo esto lo realiza aplicando un bucle en el que se van calculando aplicando las mismas limitaciones que se aplican a la hora de realizar DWT.

Al solo tener que calcular el número de datos resultantes no es necesario aplicar DWT para obtener el resultado que se busca. Con solo realizar una división por dos del número inicial, y aumentar el resultado en uno si es impar, se obtiene el número de datos de resultados del siguiente nivel. Haciendo esto de manera reiterativa hasta que lleguemos al nivel que buscamos nos permite obtener el número de datos que obtenemos como resultado a ese nivel al aplicar DWT. Esta operación permite una distribución eficiente de los datos a lo largo de los niveles de la DWT, garantizando una transformación coherente y precisa.

Por último, si ha encontrado el último nivel o alguna limitación que le obliga a salir del cálculo de los números, guarda este número final en el primer elemento de una cola en la estructura de datos del programa para que, posteriormente, sea fácil de obtener para cuando haga falta.

cuda_main()

Esta función se encarga de procesar los datos en la GPU. Abarca una serie de pasos cruciales que preparan la memoria, ejecutan el kernel y recuperan los resultados. Durante esta exploración, se explicarán en detalle cada uno de los pasos para lograr una comprensión completa del proceso.

1. **Reserva de memoria en la CPU:**
   La función empieza reservando memoria de forma contigua para la matriz de datos que contendrá los resultados tras aplicar DWT. Esta reserva de memoria contiene una matriz con el número de filas de la matriz original y usara el número de datos por nivel calculado antes, como se muestra en la figura 3.4 llamando previamente al método cuda_calculo_haar_L(), para indicar el número de elementos por fila.

2. **Reserva de memoria en la GPU:**
   Utilizando las dimensiones de la matriz original se realiza una reserva de memoria en la GPU utilizando la función cudaMallocPitch. Esta matriz se usa de apoyo para poder guardar los resultados de los cálculos tras aplicar DWT.

3. **Ejecución del Kernel wavedec:**
   El método wavedec se llama para manejar la GPU y ejecutar los cálculos para realizar la transformación wavelet multinivel. Según se indica el kernel wavedec se ejecutará con 1 bloque y el segundo parámetro indica el número de threads por bloque. Esto significa que por cada fila de la matriz se lanzará un thread dedicado que hará el trabajo indicado por el kernel llamado wavedec.

4. **Esperar a la Sincronización de la GPU:**
   Se espera a que la GPU termine sus tareas antes de continuar, lo que garantiza que los resultados estén disponibles y listos para copiarlos de vuelta a la memoria del sistema.

5. **Recuperación de los datos:**
   Los resultados de la transformación se recuperan desde la memoria de la GPU hacia la CPU utilizando cudaMemcpy2D. Estos datos se copian de vuelta en la matriz que hemos hecho primeramente y se utiliza el número de elementos calculado previamente para acortar los datos que hay que copiar.

6. **Liberación de la memoria:**
   Por último se libera la memoria temporal que se ha reservado previamente para poder realizar los cálculos en la GPU.

En conjunto, esta función representa un flujo completo, desde la preparación de la memoria hasta la recuperación de la memoria pasando por la ejecución del kernel, todo ello con el objetivo de lograr una transformación wavelet multinivel en la plataforma CUDA.

Funciones ejecutadas por el device

Son aquellas funciones que se ejecutan en la GPU. Estas funciones son lo denominado ”kernels” que son lanzados desde la CPU e indican a la GPU el trabajo a realizar. En este caso a la GPU se le pide hacer una división del trabajo tratando de explotar al máximo la paralelización que se puede obtener de la GPU y para ello se divide al mínimo las acciones que tiene que realizar cada thread.

En la figura 3.5 se puede observar un diagrama de secuencia que ejemplifica el diseño y la implementación realizada para incluir el uso de GPUs mediante CUDA en la aplicación. Hay que tener en cuenta que la secuencia de ejecución es la misma para todos los threads relacionados con la ejecución de los kernels.

A continuación se dará la explicación de como funciona el código de los kernel y como se obtienen de vuelta los resultados para posteriormente poder compararlo con el resto de implementaciones que se han realizado.

wavedec()

Este es el kernel principal que se encarga de calcular y ordenar las partes del vector para su transformación wavelet multinivel. En esta función se realiza el procesamiento de los datos en la GPU, dividiendo el vector de datos en segmentos y aplicando la transformación wavelet en

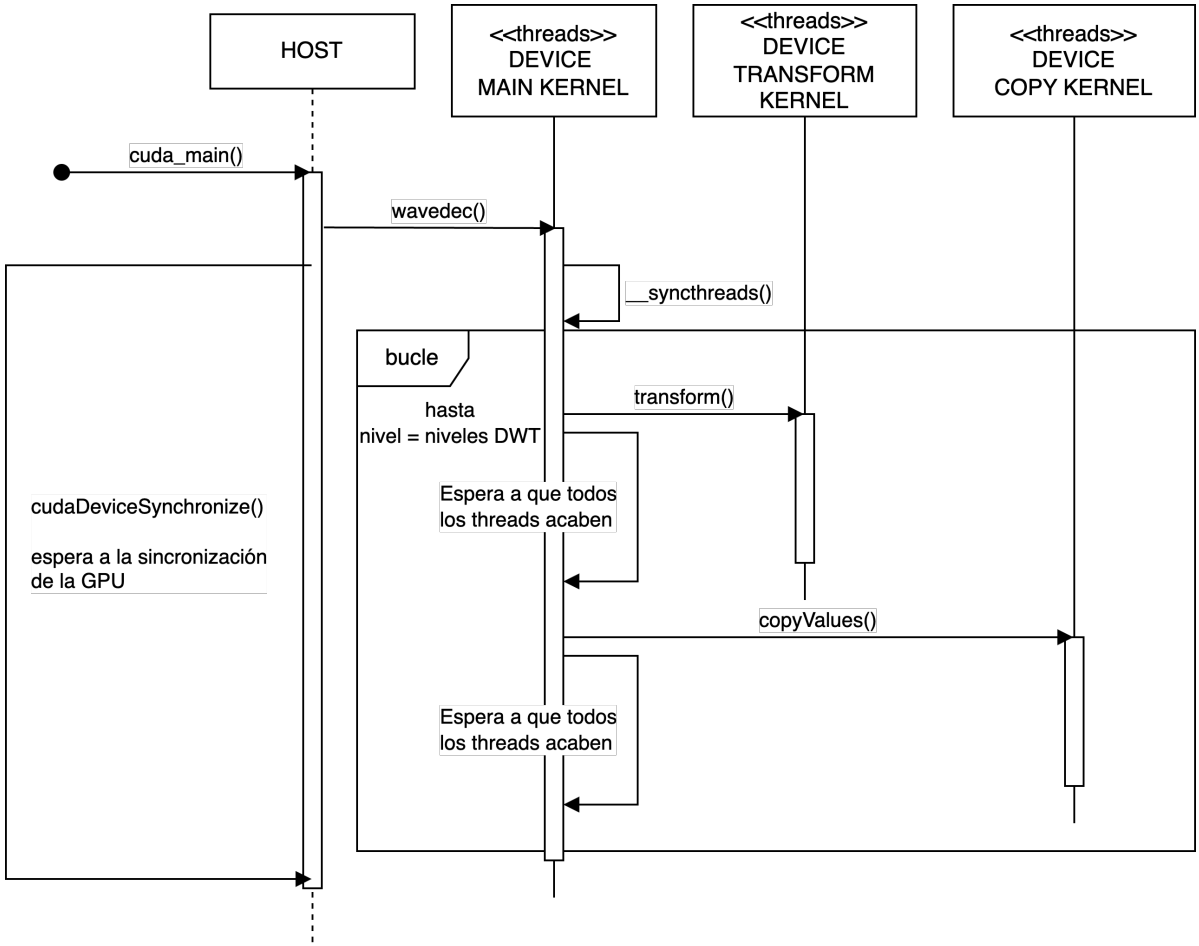

Figura 3.5: Diagrama de secuencia aplicado a la paralelización en GPU en CUDA.

cada uno de ellos. La función es paralelizada en la GPU utilizando hilos y bloques para procesar cada uno de los segmentos de datos. Los parámetros de entrada son los siguientes:

- Puntero a matriz de datos a transformar y a matriz de datos auxiliar.
- Desplazamiento óptimo en memoria GPU para alojar cada muestra y el cálculo auxiliar.
- Número total de posiciones del vector.
- Número de niveles a computar y de muestras a analizar.
- Posición inicial del segmento de datos a analizar.

El código utiliza una estrategia de paralelización de hilos sobre todo el vector, limitando el número de hilos al de muestras. En el caso de que el hilo se encargue de una muestra concreta, se procede a separar los datos por muestras y se realiza la transformación wavelet multinivel para cada muestra.

La función comienza calculando el índice del hilo en el vector, que es la suma del índice del hilo en el bloque y el índice del bloque multiplicado por el tamaño del bloque. Luego, limita el número de hilos al número de muestras a analizar y asigna un hilo a cada muestra.

A continuación, la función separa los datos por muestra y llama a una función hija para copiar el segmento de vector a transformar en una matriz de coeficientes auxiliares. Acto seguido, se realiza la transformación wavelet multinivel del vector de datos utilizando un bucle while, que repite la transformación tantas veces como niveles se han solicitado.

Dentro del bucle, se llama a una función hija para realizar la transformación wavelet en el nivel correspondiente. Esta función utiliza una arquitectura padre-hijo, lo que significa que se dividen los datos en bloques más pequeños para que cada bloque se transforme en paralelo. Después de la transformación, se actualizan las variables de nivel y se llama a otra función hija para copiar los resultados en un vector auxiliar.

Finalmente, se actualiza el número de datos que se tienen que procesar para el siguiente nivel y se ajusta el número de datos si es impar. La función se sincroniza con todos los hilos antes de finalizar.

transform()

La función recibe como entrada un puntero que apunta a un vector de datos que se van a transformar, un puntero que apunta a un vector para guardar los resultados intermedios de la transformada, y el número de posiciones del vector.

El kernel utiliza una variable para identificar la posición de cada hilo en el vector global. Se calcula sumando el índice de hilo threadIdx.x al índice de bloque blockIdx.x multiplicado por el tamaño de bloque blockDim.x. De esta manera, cada hilo se encarga de transformar una posición diferente del vector ya que obtienen su id global y único en la ejecución.

El código utiliza la transformada wavelet de Haar para calcular los coeficientes de cada nivel de descomposición. Cada hilo se encarga de calcular el coeficiente de una posición par en el vector, utilizando los valores de las posiciones pares e impares adyacentes, aplicando el filtro paso-bajo para escalar y filtrar la señal. El resultado se guarda en el vector auxiliar.

Es importante destacar que este kernel no realiza la transformada multinivel completa, sino solo un nivel. Para obtener el resultado final, es necesario ejecutar una secuencia de kernels que se encargan de realizar la transformada en diferentes niveles de descomposición y combinar los resultados en un solo vector.

copyValues()

Esta función es responsable de copiar los valores escalados de un vector de datos temporales a un vector de datos proporcionando sincronización de datos para seguir realizando los cálculos para los siguiente niveles de DWT y posteriormente ser copiados a la memoria del sistema.

La función toma tres argumentos: un puntero a un vector de datos a transformar, un puntero a un vector de datos temporales a copiar, y el número de datos a copiar. Dentro de la función, se define una variable igual que en la función de transform() indica el identificador único del thread. Esta variable se utiliza como un índice en los dos vectores para poder acceder a los datos.

Después, se comprueba si el índice es menor que el número de datos a copiar, lo que significa que el hilo actual está dentro del rango de elementos que se deben copiar. Si es así, el valor de

la matriz temporal se copia en la matriz principal.

3.4. Diseño de la aplicación en OpenCL

Esta sección se centra en el diseño e implementación llevada a cabo para la traducción del código a OpenCL desde la migración eficiente de datos hasta la ejecución de los cálculos de la DWT en la señal de metilación. Existen bastantes cambios en comparación con la implementación inicial incluyendo cambios en los tres kernels que existían en la implementación inicial, reduciendo su número solo a uno, y nuevas estructuras de datos para guardar información sobre OpenCL. Estas nuevas estructuras de datos permiten obtener y generar ciertos elementos necesarios de forma previa para OpenCL de manera que no se tenga que obtener todo cada vez que se quiera hacer uso de la GPU.

Esta implementación es necesaria como paso intermedio a causa de que EngineCL utiliza OpenCL para poder comunicarse con los dispositivos por lo que es necesario realizar una implementación con OpenCL que nos sirva como base.

3.4.1. Encapsulación del código OpenCL

Al igual que en la implementación anterior todo el código relacionado con el uso de la GPU esta encapsulado. En este caso, en vez de agrupar todo el código en funciones se procedió a realizar una clase que mantenía las mismas funciones pero convertidas para hacer uso de las nuevas estructuras de datos creadas específicamente para la implementación en OpenCL.

Mediante la encapsulación anterior mantenemos las cuatro líneas de código para llevar a cabo la utilización de la GPU. En este caso el nombre de los métodos a utilizar han sido modificados para diferenciarlos.

```
1 // libera la memoria de la GPU
2 haar_opencl_service.opencl_end();
3
4 // envia el total de los datos a la GPU
5 haar_opencl_service.opencl_send_data(opencl_data);
6
7 // procesado de los datos
8 haar_opencl_service.opencl_calculo_haar_L(opencl_data);
9 haar_opencl_service.opencl_main(opencl_data);
```

Figura 3.6: Métodos dedicados a GPU para OpenCL en el flujo principal de la aplicación.

Como se puede observar en la listado 3.6, el código es, en esencia, el mismo pasando de funciones a métodos de una clase concreta que ejecutan las mismas acciones explicadas en el apartado 3.3.1. Aunque observemos que el diseño usado para utilizar la GPU es prácticamente el mismo (salvando las distancias con el uso de la nueva clase), la implementación y la arquitectura que existe por debajo de esta interfaz que proporcionamos a través de la clase ha sido cambiada por completo para poder acomodar la implementación con OpenCL.

3.4.2. Exploración del código OpenCL

El código de OpenCL esta recogido en una clase con nombre HaarOpenCL. Esta clase contiene las mismas funciones que nos encontramos en CUDA, convertidas en métodos y reescritas para acomodar la implementación al framework de OpenCL. Aparte de estos métodos también existen otras adiciones necesarias a la clase: un método que nos permita leer el kernel de un archivo(read_kernel() y read_file()) y un método que nos permita descubrir la plataforma y los dispositivos que pueden ser utilizados para tomar uno de ellos. También se han tenido que añadir varias estructuras de datos necesarias para OpenCL.

En OpenCL al seleccionar un dispositivo sobre el que trabajar hay que mantener su identificador. Este identificador es usado para poder obtener información sobre el dispositivo al que identifica, para crear el contexto a utilizar, obtener la cola de comandos del dispositivo y hacer el build del programa que contiene el kernel que se quiere ejecutar. Todos estos elementos mencionados son necesarios y esenciales para poder hacer uso de OpenCL. Dentro de esta nueva implementación y clase dedicada podemos dividir los métodos, de nuevo, en dos conjuntos:

- **Funciones ejecutadas por el host:** discover_platforms_and_devices(), read_kernel(), read_file(), opencl_init(), opencl_end(), opencl_send_data(), opencl_calculo_haar_L(), opencl_main()
- **Funciones ejecutadas por el device:** transform_wavedec()

Funciones ejecutadas por el host

Son aquellas funciones que se ejecutan en la CPU del sistema pertenecientes al programa que se esta ejecutando. Estas funciones son quienes realizan el envío de datos a la GPU, indican que tarea tienen que realizar y posteriormente recuperan el resultado de la GPU. En adición a lo previo, en esta implementación la GPU también se tiene que encargar de seleccionar el dispositivo que va a ser utilizado para OpenCL, realizar la lectura del archivo del kernel y generar un programa dedicado al kernel y al dispositivo.

Para mantener la encapsulación anterior con solo las cuatro líneas de código todo el descubrimiento de las plataformas y dispositivos, junto a la lectura del kernel son ejecutados en el constructor de la clase que se encarga de esto y guardar el contexto, la cola de comando y el programa en la estructura de datos de la clase. Con esto conseguimos que se pueda acceder a estos datos a posteriori, cuando haya que realizar alguna acción, y no haya que volver a preparar todo, lo que ahorra tiempo en ejecución y simplifica la implementación.

discover_platforms_and_devices()

Este método **se encarga de obtener el número de plataformas de las que dispone el sistema, posteriormente de estas plataformas selecciona aquellos dispositivos que están disponibles y se queda con el de la GPU y con su identificador.** A partir de aquí obtiene el contexto del dispositivo, del cual se consigue la cola de comandos del dispositivo y guarda toda la información en la estructura de datos de la clase.

read_kernel() y read_file()

Estos dos métodos se encargan de crear un kernel OpenCL y establecer sus argumentos. Para esto es necesario que el kernel sea leído del archivo en donde esta definido que

es de lo que se encarga el método de read_file() que es utilizado desde read_kernel().

Una vez el kernel ha sido leído, se crea un programa partiendo del contexto y del kernel obtenidos previamente. Con este programa pueden hacer una build del mismo que se asocia con el dispositivo mediante su identificador y permite crear el kernel indicando el nombre de la función definida en el archivo del kernel.

opencl_init()

Este método no recibe ningún parámetro y **sirve para mostrar información sobre todas las plataformas y dispositivos que se encuentran en el sistema**. Mediante el framework de OpenCL puede acceder a toda la información que proporcionan tanto las plataformas como los dispositivos disponibles.

opencl_end()

Es una función dedicada a la gestión de memoria de la GPU. Libera la variable del objeto que contiene el vector de datos en la GPU. Este método es el que se encuentra en 3.6 en la línea 2 para mantener la memoria de la GPU limpia entre ejecuciones evitando conflictos o errores de memoria en la ejecución.

opencl_send_data()

Esta función realiza dos acciones principales: Crear un buffer para el dispositivo y escribir la matriz de datos al buffer recién creado. Esta primera acción es llevada a cabo utilizando la función **clCreateBuffer** que permite crear un buffer indicando con el número de bytes a escribir las dimensiones del buffer donde habría que incluir también el desplazamiento.

Como se explicó en la sección 3.3.2 el uso de cudaMallocPitch hace la reserva de memoria calculando de forma automática un desplazamiento para hacer alineación de datos con la memoria de la GPU. Esto en OpenCL no es calculado automáticamente por lo que es necesario calcularlo de forma previa. Este desplazamiento se puede calcular obteniendo del dispositivo su parámetro **CL_DEVICE_MEM_BASE_ADDR_ALIGN** que contiene el tamaño del alineamiento de memoria preferido del dispositivo en bits. Con esto se obtiene el número bytes de datos por fila que tenemos en la matriz y transformar de bit a bytes la alineación de la memoria. Al acabar estas transformaciones solo resta obtener el número de datos en bytes más cercano al tamaño de datos de la fila que tengo el desplazamiento justo para que la memoria este alineada.

Una vez obtenido este desplazamiento hay que crear el buffer con **clCreateBuffer** con los siguientes parámetros:

- **Contexto:** Contexto válido obtenido del dispositivo para crear el buffer.
- **Flags:** Flags de control para indicar que acciones pueden ser realizadas sobre el buffer y donde tiene que ser asignado.
- **Tamaño del buffer:** El número de bytes del tamaño de la memoria que va a ser asignada al buffer.

- **Puntero al buffer:** Puntero al buffer de datos en memoria asignado en la aplicación. En nuestro caso esto se define a NULO ya que tenemos que hacer la copia de datos teniendo en cuenta el desplazamiento.
- **Dirección de variables para retornar error:** Dirección de la variable a la que indicar su ha ocurrido un error a la hora de crear el buffer.

La segunda acción de esta función se encarga de escribir los datos en el buffer creado previamente. Esta segunda función es llevada a cabo por la función **clEnqueueWriteBufferRect** que se encarga de copiar los datos. Este método necesita de catorce parámetros para poder llevarse a cabo. Se van a comentar los que son más importantes dejando de lado la lista de comando, el buffer, los relacionados con los eventos y aquellos relacionados con el offset inicial de datos en el buffer y de la memoria del sistema. Con esto nos quedan los siguientes parámetros:

- **Región de datos a copiar:** Es una estructura que indica el número de bytes que hay que copiar de cada fila, el número de filas y la profundidad de la matriz que al ser un matriz de dos dimensiones tiene profundidad de 1.
- **Desplazamiento para alineamiento:** Parámetro en el que se indica el número de datos en bytes de una fila cuando tiene aplicado el desplazamiento para el alineamiento de la memoria.

opencl_calculo_haar_L()

Este método recibe la misma explicación que la recibida en 3.3.2. En resumen, se encarga de encontrar el número de datos por fila tras aplicar la DWT multinivel.

opencl_main()

Este método se encarga de procesar los datos en la GPU. Abarca un serie de pasos que, excepto el primero, son completamente diferentes con respecto a lo explicado en la sección 3.3.2:

1. **Reserva de memoria en la CPU:**
   Al igual que en el apartado de CUDA, el método empieza por la reserva de memoria de forma contigua para la matriz de datos que contendrá el resultado de la matriz tras aplicar DWT. De igual manera que antes, es una reserva de memoria para una matriz con un número de filas igual al de la matriz original y que el tamaño de la fila depende del número de datos por fila en un cálculo previo realizado en el método opencl_calculo_haar_L() que se ejecuta antes que este método como podemos observar en 3.6
2. **Creación de buffer de apoyo:**
   Al igual que se explica en la sección 3.4.2 se necesita crear un buffer nuevo que sirve de apoyo para la ejecución del kernel y, de nuevo, tenemos que hacer un cálculo del desplazamiento para tener la estructura de datos alineada en memoria de la GPU. Tenemos que recalcular este desplazamiento ya que el número de datos por fila cambia al utilizar el calculado previamente en vez de usar el original.
3. **Ejecución del Kernel transform_wavedec**
   Para la ejecución de este kernel hay que tener en cuenta cómo es la secuencia que sigue el código para poder llevar a cabo los cálculos usando la GPU.

Antes en CUDA el control de cuando tenia que dejar de realizar cálculos se encontraba en el mismo kernel donde se comprobaba si se había llegado al nivel elegido tras aplicar de manera iterativa DWT sobre los datos. Tener el control en el kernel provocaba que tras realizar los cálculo se necesitase copiar los datos de la matriz de apoyo a la principal por cada nivel para seguir ejecutando.

Ahora en la ejecución de OpenCL tenemos ese control del bucle en CPU. Eso implica que por cada iteración que se hace del bucle se necesita lanzar el kernel de nuevo. Con esto conseguimos no tener que copiar el resultado de la matriz de apoyo a la matriz original debido a que nos basamos en si la iteración que se esta realizando es par o impar se intercambian los bufferes de posición.

Con este cambio evitamos tener que copiar de vuelta el resultado de la matriz de apoyo a la principal por cada nivel como ocurría en la implementación de CUDA teniendo solo que hacer una única copia al host del buffer correspondiente.

Estos cambios también permite enviar todo el trabajo directamente desde el host. La implementación de CUDA lleva la ejecución de los kernels que hacen la ejecución de los datos a un kernel adicional el cuál es lanzado con un thread por cada fila. Esto implica que el trabajo de los cálculos se lanza desde distintos threads y haya que forzar una sincronización entre los threads de las filas. Con la implementación de OpenCL se ha conseguido llevar esa sincronización y lanzamiento de trabajo al host de manera que no haya que dividir el trabajo ni calcular a que fila pertenece cada thread, ya que se encarga el kernel de hacerlo de manera individual reduciendo la complejidad y evitando la copia de datos al acabar de ejecución de cada nivel.

En la primera iteración (pares) se manda como matriz principal de datos el buffer que contiene esa matriz y para la de apoyo se usa el buffer de apoyo, mientras que en la segunda iteración (impares) se manda como matriz principal de datos el buffer que contiene la matriz de apoyo ya que es donde están los resultados de la anterior iteración y, por lo tanto, en la matriz de apoyo se indica el buffer con la matriz original. Teniendo en cuenta la iteración en la que estamos también indicamos en otra variable qué buffer de datos hay que leer si está es la última iteración que se hace.

Gracias a esto conseguimos evitarnos copiar la matriz entera de datos cada vez que se computa un nivel de DWT y nos permite paralelizar al máximo el cómputo de datos ya que por cada nivel lanzamos tantos threads como elementos tiene la matriz en bloques de 1024 (Es un número variable siempre que se encuentre entre múltiplos de 2)

También tener en cuenta que la división de los elementos que son necesarios para la siguiente iteración se realizan aquí en vez de en la GPU, por lo que nos permite que tras acabar la ejecución se proceda a la copia de datos de la GPU a la memoria del sistema

4. **Recuperación de datos:**
   Los resultados de la transformación se recuperan desde los buffers de GPU hacia la CPU utilizando clEnqueueReadBufferRect. Gracias a la nueva implementación, con el cambio en la ejecución del kernel, se conoce de que buffer hay que leer los datos. Con los cálculos previos de los números de datos por nivel conocemos cuantos datos hay que traer, por lo que de los parámetros más importantes a indicar serian los siguientes:

   - **Región de datos a copiar:**
     Es una estructura que indica el número de bytes que hay que copiar de cada fila, el número de filas y la profundidad de la matriz que al ser un matriz de dos dimensiones tiene profundidad de 1. El número de bytes por fila solo tiene en cuenta la cantidad

de bytes de los datos reales a copiar, no tiene que tener en cuenta el desplazamiento. Gracias al cálculo previo de estos valores en 3.4.2 sabemos la cantidad de datos que se necesitan obtener.

- **Desplazamiento para alineamiento:**
  Parámetro en el que se indica el número de datos en bytes de una fila que existe en la memoria de GPU para traer los datos al sistema sin desplazamiento.

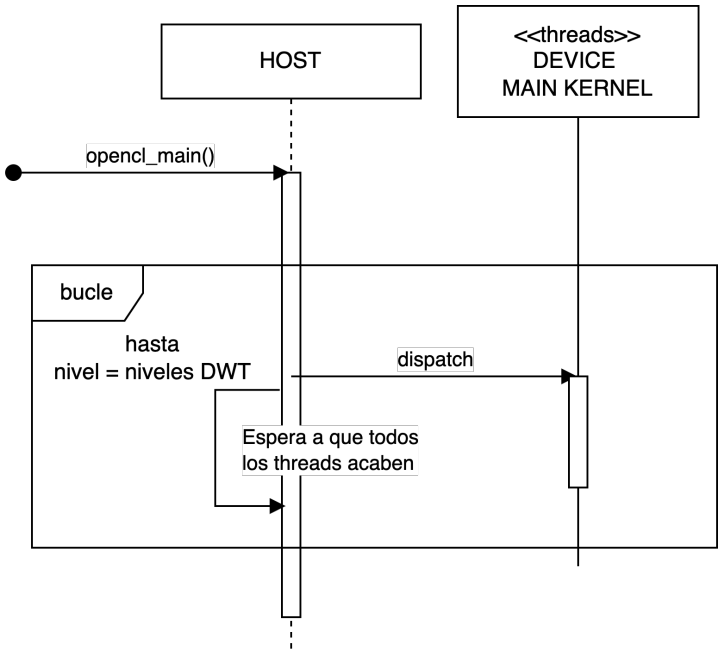

Figura 3.7: Diagrama de secuencia aplicado a la paralelización en GPU en OpenCL.

Debido a todos estos cambios el diagrama de secuencia en la figura 3.7 se simplifica bastante, con respecto al de la implementación original en CUDA mostrado en la figura 3.5, teniendo en cuenta que solo un kernel es lanzado y ejecutado por los threads de la GPU y no hace un jerarquía padre-hijo entre los propios hilos del kernel principal. Para poder conseguir esto, además del cambio realizado aquí, también se ha tenido que realizar un gran cambio en los kernels, que nos ha permitido acortarlo todo a un solo kernel que es el que se encarga de realizar el calculo y guardar el resultado.

Funciones ejecutadas por el device

Son aquellas funciones que se ejecutan en la GPU. En este caso a la GPU se le pide hacer una división del trabajo tratando de explotar al máximo la paralelización que se puede obtener de la GPU y para ello se divide al mínimo las acciones que tiene que realizar cada thread.

transform_wavedec()

Al haber realizado todos estos cambios sobre la implementación del código en CPU, el código que queda para la GPU es menor, ya que toda la distribución del trabajo se realiza previamente y en GPU solo hay que realizar los cálculos correspondientes.

El kernel recibe como parámetros los siguientes elementos:

- **Puntero a la matriz de datos:** De aquí es donde obtenemos los datos necesarios para poder realizar las operaciones con ellos.
- **Puntero a la matriz auxiliar:** Este puntero hace referencia a la matriz auxiliar que es donde se guardan los resultados de las operaciones.
- **Tamaño de una fila de la matriz de datos:** Hace referencia a la cantidad de bytes que encontramos por fila en la matriz de datos.
- **Tamaño de una fila de la matriz auxiliar:** Hace referencia a la cantidad de bytes que encontramos por fila en la matriz auxiliar. No coincide con el tamaño de la matriz de datos, de ahí tener que mandarlo también.
- **Numero de elementos por fila:** Indica cual es el numero de elementos por fila actual. Este numero se va actualizando a medida que se aplica DWT en varios niveles.
- **Numero de filas:** Indica cual es el numero de filas que existen en las matrices.

Una vez que disponemos de todos estos parámetros podemos obtener cual es el identificador del thread que esta ejecutando. Con este identificador podemos obtener a que fila y columna de la matriz afecta. Para obtener la columna de la matriz sobre la que esta trabajando el thread tenemos que obtener el resto de la división del thread y del numero de elementos por fila. Para obtener la fila sobre la que tiene que operar el thread nos sirve con dividir el identificador entre el numero de elementos por fila.

Tras obtener la columna y la fila solo se proceden a procesar aquellas columnas que correspondan con números pares. Esto es debido a que aplicar DWT implica realizar cálculos de los datos entre las columnas pares de las filas de la matriz y la siguiente posición, por lo que solo se proceden a realizar los cálculos sobre las posiciones pares de las columnas.

Si el thread al final es uno de los que se tiene que procesar obtienen los punteros a las filas de la matriz de datos y la auxiliar correspondientes para el thread en ejecución. Esto se obtiene partiendo de los punteros que se han pasado como parámetros a los cuales se les suma la multiplicación de la fila a la que corresponde el thread, que hemos calculado en el primer paso, con el tamaño de cada fila. Esto se hace tanto para el puntero de la matriz de datos como para la auxiliar por lo que, a la hora de la multiplicación y la suma hay que tener en cuenta que para cada uno hay que usar su puntero correspondiente y su tamaño de fila correspondiente.

Con los punteros de las fila donde se tienen que obtener los datos de la matriz para realizar los cálculos y de la fila donde se tiene que escribir los resultados, hace falta obtener la columna sobre la que se tienen que guardar los datos. Esta columna se obtiene con la división de la columna obtenida en el primer paso entre 2. Con esto lo que obtenemos tras procesar los datos es que la cantidad de datos con la que tenemos que operar en la siguiente iteración, es la mitad ya que vamos reduciendo la matriz de datos con cada iteración.

Después de obtener donde tiene que ir resultado se procede a aplicar los cálculos necesarios para aplicar DWT a la matriz de datos y los resultados se recogen en la matriz auxiliar. Gracias a estos cambios conseguimos reducir el número kernels reduciendo el tiempo de ejecución así como no tener que hacer la copia de los elementos de vuelta a la matriz de datos que también reduce el tiempo de ejecución.

Con esto se consigue tener una implementación OpenCL similar a la original en CUDA, y por lo tanto sólo permite utilizar una única GPU por ejecución. En la siguiente sección se utilizará

la herramienta EngineCL para obtener una implementación que pueda aprovechar sistemas con múltiples GPUs.

3.5. Diseño de la aplicación en EngineCL

Esta sección se centra en el diseño e implementación llevada a cabo para la traducción del código a EngineCL desde el cambio necesario de la estructura de datos hasta la ejecución de los cálculos de la DWT en la señal de metilación que cambia por completo la preparación para poder ejecutar los kernels.

3.5.1. Encapsulación del código EngineCL

De la misma manera que en las otras implementaciones todo el código relacionado con la GPU esta encapsulado. Debido a la incorporación de la herramienta EngineCL en el flujo del programa, se ha creado una clase dedicada que se encarga de realizar todo el código de las anteriores implementaciones reduciendo todo a un solo método.

Mediante la encapsulación realizada para esta implementación se reducen las líneas de código a solo dos. En este caso el nombre de la clase y de la variable que contenía la clase ha sido modificada para diferenciarlo del resto de las implementaciones.

```
1 haar_enginecl_service.opencl_calculo_haar_L(opencl_data);
2 haar_enginecl_service.opencl_main(opencl_data);
```

Figura 3.8: Métodos dedicados a GPU para EngineCL.

Como se puede observar en la listado 3.8, el código es reducido a solo dos métodos de una clase concreta que ejecutan las mismas acciones explicadas en el apartado 3.3.1 pero la implementación es completamente distinta. Ahora el diseño para usar la GPU cambia por completo debido a EngineCL, aunque la implementación se ha realizado sobre la capa de nivel dos explicada en 2.3, la abstracción que permite EngineCL permite usar la herramienta de una forma sencilla sin tener que entender por completo como funciona a bajo nivel.

Aunque la herramienta de EngineCL ayuda y abstrae todas las implementaciones y los usos de OpenCL, exige ciertas limitaciones de uso que serán exploradas más a fondo posteriormente como puede ser el cambio de la estructura de datos en la que se guarda la señal de metilación a la que hay que aplicar DWT multinivel, pasando de ser un conjunto de punteros a cada fila de la matriz a un vector que contiene todo los elementos de la matriz seguidos.

3.5.2. Exploración del código EngineCL

El código de EngineCL esta recogido en una clase con nombre HaarEngineCL. Esta clase ha sido cambiada de forma radical con respecto al resto de implementaciones debido a la utilización de la herramienta de EngineCL, aunque las funciones que se realizan son las mismas. Esta clase contiene ahora mismo dos métodos, uno conocido desde la primera implementación que se dedica a obtener ciertos valores para saber la cantidad de datos que se obtiene por fila de la matriz al final de la aplicación de la DWT multinivel. El otro método se dedica a realizar todo el trabajo necesario para poder acabar haciendo uso de todos los dispositivos disponibles del sistema.

Dentro de esta nueva implementación y clase dedicada podemos dividir los métodos, de nuevo, en dos conjuntos:

- **Funciones ejecutadas por el host:** opencl_calculo_haar_L(), find_platform_devices(), opencl_main()
- **Funciones ejecutadas por el device:** transform_wavedec()

Funciones ejecutadas por el host

opencl_calculo_haar_L()

De la misma manera que ocurre en el apartado 3.4.2, este método recibe la misma explicación que la recibida en 3.3.2. En resumen, se encarga de encontrar el número de datos por fila tras aplicar la DWT multinivel.

find_platform_devices()

Este método esta dedicado a la búsqueda de las plataformas y de los dispositivos de cada plataforma. Primeramente busca si existe alguna plataforma en el sistema, si esta condición se cumple, se procede a obtener los dispositivos de la plataforma y se generan varias duplas entre el identificador de la plataforma con los identificadores de los dispositivos.

Esta identificación de dispositivos por plataforma puede realizarse con distintos objetivos. El método de búsqueda de los identificadores de los dispositivos permite el paso de parámetros, uno de ellos es una flag que indica el tipo de dispositivos que quieren que se busquen. En la implementación actual solo se buscan GPUs pero si se dispone de FPGAs o de CPU, de los que se dispone del driver desarrollado para OpenCL, cambiando la flag indicando que se quieren buscar cualquier tipo de dispositivo serán encontrados y añadidos a la lista.

opencl_main()

Este método esta dedicado a toda la preparación necesaria para poner la herramienta de EngineCL a funcionar con los datos que necesitamos procesar y el kernel que queremos que las GPUs ejecuten. Al reducir el número de métodos en esta implementación con respecto a las anteriores, el método se encarga de realizar más funciones que anteriormente con el siguiente orden:

1. **Lectura del kernel:**
   Al igual que en la anterior implementación de OpenCL, EngineCL hace uso del mismo kernel por lo que hay que realizar la lectura del kernel de la misma manera pero, gracias a EngineCL, ya no tenemos que usar el código que habíamos desarrollado previamente ya que se nos proporciona un método dedicado para poder realizar la lectura de un archivo.

2. **Cálculo del número de threads:**
   A la hora de utilizar la clase Runtime para poder usar las capacidades de EngineCL, es necesario indicarle en el constructor la distribución del número de threads totales y como van a ser agrupados. Por esto, es necesario realizar el cálculo de los threads necesarios para ejecutar el kernel basándonos en los datos que ofrece el programa, en concreto se calcula con el numero de filas y de elementos por filas de la matriz y teniendo en cuenta el número de threads por bloque que se quiere ejecutar.

3. **Reserva de memoria para matriz auxiliar:**
   Para poder intercambiar los resultados entre matrices y así mantener la idea de tener dos matrices que van intercambiando posiciones en para encontrar el resultado, es necesario tener una matriz auxiliar. En esta implementación la matriz auxiliar se guarda directamente en un vector de datos inicializado por completo a ceros. La matriz auxiliar tiene como tamaño el número de threads que se ha calculado anteriormente en el primer paso. Esta es otra limitación que se comentara posteriormente.

4. **Cambio de tamaño a la matriz de datos:**
   La matriz que contiene los datos ahora consiste en un vector que contiene de forma continua todos los datos por lo que realizar mejoras en la estructura de datos no llevaría a una mejora distinguible para que merezca la pena implementarla. Debido a esto no se aplica ningún tipo de padding o desplazamiento a los datos para que se encuentren alineados con la memoria de la GPU. En cambio es necesario cambiar el tamaño de la matriz de datos y aumentarlo de manera que sea el mismo que el número de threads que se ha calculado anteriormente en el primer paso. Como se ha comentado antes, esto sera explicado en el siguiente apartado.

5. **Preparación de las matrices para EngineCL:**
   Para poder pasar datos a EngineCL y que sean enviado a las GPUs se necesita que las estructuras de datos estén en un formato concreto por lo que es necesario adaptar las estructuras de datos para poder usar la herramienta.

6. **Exploración de plataformas y dispositivos:**
   Debido a que la implementación con EngineCL busca utilizar múltiples GPUs, es necesario realizar la búsqueda de plataformas y dispositivos para conocer los dispositivos que pueden ser utilizados. Para poder hacer esto se realiza una búsqueda entre todas las plataformas detectadas por OpenCL y de ellas se obtienen los dispositivos que son identificados como GPU.

7. **Ejecución mediante EngineCL:**
   Para la ejecución de los cálculos mediante la GPU hay que pasar por el uso de la herramienta de EngineCL y hay que tener en cuenta varios factores a la hora de realizar la implementación.

   Se mantiene el bucle de control en relación con los niveles que se han realizado en el lado del host. Esto implica que por cada ejecución se tienen que realizar los preparatorios para poder hacer uso de EngineCL. Por un lado se mantienen los beneficios que otorgaba la implementación en OpenCL, pero es necesario hacer más cosas que antes.

   Debido al uso de EngineCL en cada ejecución del bucle es necesaria la creación de los objetos conocidos como **Devices** definidos en la segundo capa de EngineCL. También es necesario la creación del objeto **Scheduler** dedicado a hacer de planificador para la ejecución al que hay que indicar como tiene que hacer el balanceo de carga. Todo esto conlleva la creación de un nuevo objeto **Runtime** al que se le tiene que indicar: la lista de **Devices** que tiene que utilizar, el **Scheduler** del que tiene que hacer uso para planificar las tareas entre distintas GPUs y el kernel que tiene que utilizar que se ha leído en el primer paso.

   Todo esto es obligatorio debido a varias restricciones al usar la herramienta de EngineCL, que serán indicados en el siguiente apartado. Esto implica que cada vez que se realiza un nivel de DWT sobre la señal de metilación es necesario crear todas estas clases y no es posible hacer una reutilización. Esto implica que hay que pasar por todo el proceso de

realizar el balanceo de carga, la creación de los objetos con los buffers correspondientes y el envío de datos necesarios para cada GPU.

Tras esto se procede con la implementación realizada para OpenCL que implica tener un control sobre la iteración que estamos realizando del bucle para saber qué matriz de datos tenemos que mandar como la principal y cual tenemos que mandar como la secundaria.

8. **Recuperación de los datos:**
Tras la ejecución de los cálculos es necesario recuperar los resultados de la matriz correspondiente. Al haber tenido que cambiar el tipo de estructura de datos para poder implementar EngineCL, la copia de datos de la matriz ahora es tan simple como realizar un cambio de punteros, haciendo que la variable que tiene que contener el resultado apunte al vector de datos obtenido.

Tras todos los cambios ejecutados la secuencia de la ejecución principal de la GPU queda de la siguiente manera:

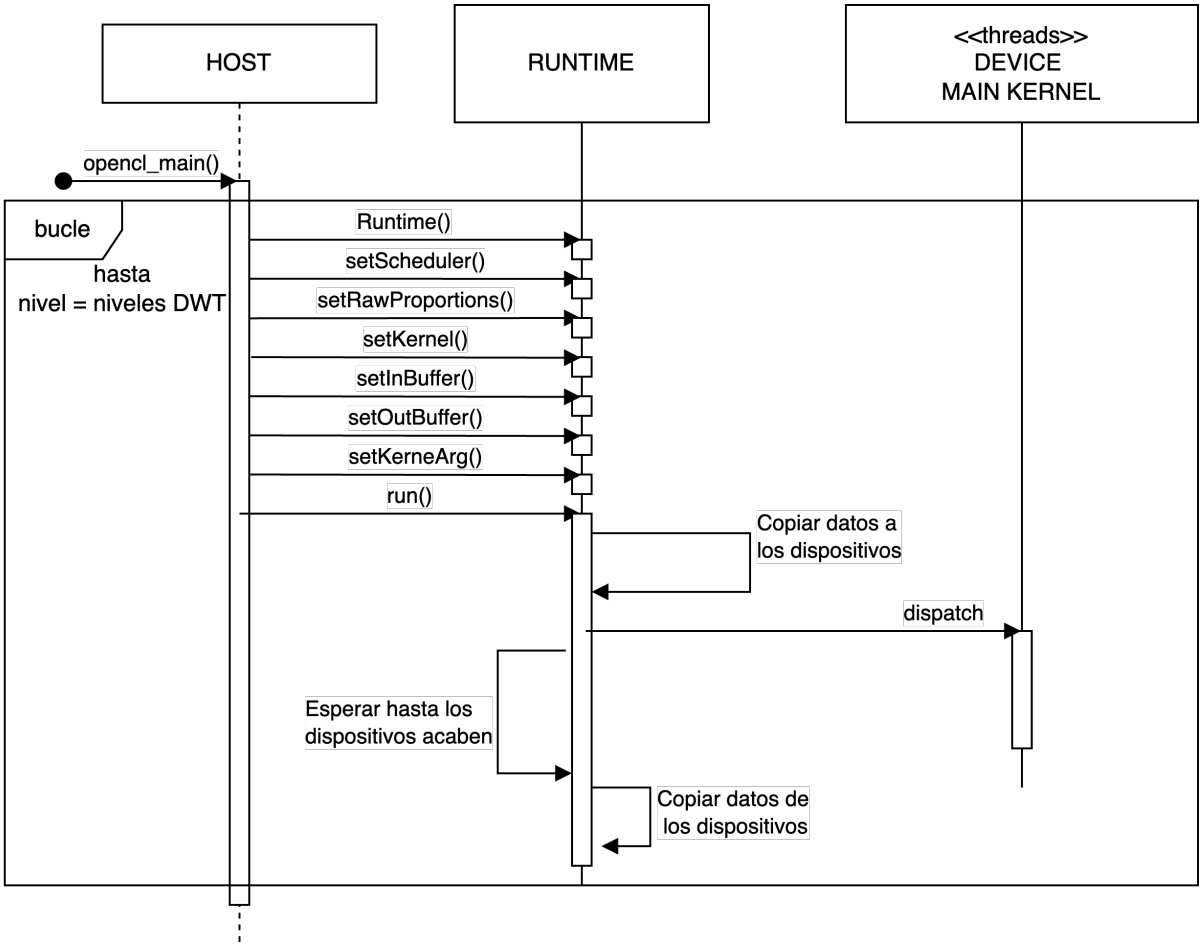

Figura 3.9: Diagrama de secuencia aplicado a la paralelización en GPU usando EngineCL.

Como se puede observar en la figura 3.9 todo lo que implica hacer uso de OpenCL y requisitos que habría que cubrir previamente son manejados por EngineCL facilitando la implementación y solo teniendo que indicar las variables de control necesarias y los buffers de entrada y de salida junto a los argumentos del kernel. Con esto hay que resaltar que la clase de **Runtime** hace de punto neurálgico para la utilización de las GPUs haciendo uso de los recursos de los que dispone

EnginceCL. Hay que tener en cuenta que, aunque en la figura solo sale un **DEVICE** puede haber múltiples, cada uno representando a cada GPU conectada al sistema.

Funciones ejecutadas por el device

transform_wavedec()

El kernel dedicado para GPU se mantiene de la implementación de OpenCL explicado en el apartado 3.4.2. Por resumir, el kernel calcula si el thread que esta ejecutando entra dentro de los límites de la matriz para seguir su ejecución partiendo del identificador del thread y acabando la ejecución si no es así. Si este thread es par entonces calcula la fila a la que tiene que acceder dependiendo de su identificador y ejecuta los cálculos para aplicar DWT y guardar el resultado en la matriz auxiliar.

3.5.3. Limitaciones de EngineCL

Como se ha comentado en sección anterior existen ciertas limitaciones con la herramienta de EngineCL. Estas restricciones se deben a la decisiones de diseño y arquitectura que se tuvieron que tomar para poder desarrollar la herramienta.

La primer limitación que encontramos es en los tipos de datos que aceptan los buffers de entrada y de salida. La herramienta admite un puntero compartido de un vector de datos y solo de ese tipo, por lo que ha sido necesario modificar la estructura de datos del programa para poder usar correctamente la herramienta ya que si no, el trasladar el contenido de una estructura a otra, ralentizaba la aplicación varios segundos con una carga de trabajo de prueba. Con esto en cuenta se decidió que seria mejor realizar este cambio que tenia pocas implicaciones debido a que solo se usa la matriz de datos para guardar la señal de metilación, que es la que se utiliza posteriormente para realizar los cálculos. Para guardar la señal de metilación lo único que se ha tenido que cambiar es como se accedía a la matriz para guardar los resultados.

Siguiendo con la memoria encontramos la limitación de que la cantidad de elementos de memoria que tiene que ser utilizado tiene que coincidir con el número de threads total que va a ser lanzado por la herramienta. Esto se debe a que la distribución de la memoria entre distintas GPUs esta realizada usando el número de threads. Esto implica que si el número de elementos de la memoria no esta alineado con esta cantidad la memoria utilizada para realizar los cálculos no será la correcta y a la hora de copiar el resultado no se realizará de forma correcta la operación.

Otra de las limitaciones encontradas es la necesidad de tener que pasar por inicializar un nuevo **Runtime** con un nuevo **Scheduler** y con una nueva lista de **Devices**. Esto se debe a que estas clases por debajo trabajan con threads. Esto se podría arreglar reutilizando los threads recogiéndolos nada más acaben la tarea indicada, el problema que genera esta solución es que usar solo el thread para esa tarea garantiza la ejecución correcta y en orden de todo el código mientras que si se aplica la idea explicada no existe ninguna limitación para que el código sea ejecutado en el orden querido y se obtendrían solapamientos entre ejecuciones.

Problemas implementando el paralelismo

A la hora de aplicar EngineCL sobre una implementación estándar, que tiene de entrada varios buffers y de salida donde se escriben los resultados de manera lineal sin tener que aplicar desplazamientos, es sencillo y no implica ningún cambio sobre el código. En nuestro caso, tras realizar los cálculos relativos con DWT necesitamos guardar el resultado, pero no en el mismo

índice desde el que se han realizado los cálculos. Como se explica en el apartado 3.4.2 el resultado se escribe en el índice de la columna dividido entre dos.

Viendo lo que ocurre con la escritura de los resultados a la hora de realizar la división de la carga de trabajo, no se puede realizar una división estándar. Imaginemos que tenemos una matriz de dos filas con 14 columnas cada una y se produce una división del trabajo entre dos GPUs. La primera GPU recibe de trabajo solo realizar los cálculos de las primeras 10 columnas y el resto de cálculos tienen que ser realizados por la segunda GPU. Si revisamos que hace la segunda GPU, se encargará de hacer cálculos desde la columna 11, por lo tanto, su siguiente cálculo sera sobre la columna 12 y se guardará en la columna 6. Aquí es donde tenemos el problema ya que la implementación base de EngineCL se empieza a copiar de vuelta desde la columna 10, y la columna 6 donde se ha almacenado el resultado no se copia.

Para poder arreglar esto se ha tenido que modificar la versión base de EngineCL para que a la hora de copiar los contenidos de vuelta, se tenga en cuenta desde que posición se empiezan a guardar datos desde el primer thread que se lanza a cada GPU gracias al offset que se aplica. Y lo mismo ocurre con el número de elementos a copiar que también hay que tener en cuenta que la cantidad de resultados va a ser la mitad de los threads que se lanza a cada GPU.

Capítulo 4

Experimentos y resultados

El objetivo de este capítulo es mostrar los experimentos realizados sobre los distintos desarrollos, mostrar una verificación de la funcionalidad de las implementaciones y plasmar el rendimiento que alcanzan. Se han realizado varios experimentos para obtener los datos necesarios para llevar a cabo estos objetivos:

- Se han llevado a cabo evaluaciones sobre las **3 implementaciones sobre una sola GPU**
- Se han llevado a cabo **evaluaciones sobre la implementación de EngineCL haciendo uso de 2 y 4 GPUs**
- Se **comparan los distintos planificadores que se pueden usar en EngineCL** con la implementación correspondiente.

4.1. Metodología

Para las implementaciones se ha hecho uso de dos sistemas, el primero para poder realizar el desarrollo y validación de las nuevas implementaciones de forma sencilla y el segundo dedicado a realizar las pruebas en una GPU y multi-GPU:

- El primer sistema se compone de un procesador Intel Xeon E5620 con 4 núcleos y 8 subprocesos. La frecuencia base del procesador es de 2.4 GHz. También dispone de una tarjeta gráfica NVIDIA GeForce RTX 2070 SUPER que dispone de 2560 CUDA cores a una frecuencia de 1.61 GHz. Todo esto sobre el sistema operativo Ubuntu 22.04.2 LTS.
- El segundo sistema se compone de un procesador Intel Xeon Gold 6230 con 20 núcleos y 40 subprocesos. La frecuencia base del procesador es de 2.1 GHz. También dispone de 4 tarjetas gráficas NVIDIA Tesla V100 que dispone de 5120 CUDA cores a una frecuencia de 1.465 GHz. Todo sobre el sistema operativo Rocky Linux en su versión 8.8.

Los datos que se han recogido para realizar las evaluaciones y comparaciones son: el tiempo de comunicación del sistema con las GPUs, el tiempo de ejecución sobre el código ejecutado en GPU, la cantidad de memoria utilizada y el tiempo de ejecución total.

Para poder obtener estos resultados se han realizado varios scripts en Python que permiten llevar acabo los experimentos y controlar que las pruebas se hayan realizado con éxito. El script principal en pPython recibe como primer argumento el tipo de implementación que se tiene que

ejecutar, como segundo recibe el tipo de resultado que se quiere obtener en las pruebas, siendo los datos explicados anteriormente las opciones a introducir y por último en el tercer parámetro se indica el número de veces que se tienen que ejecutar las pruebas. Tras las ejecuciones pertinentes el script genera un archivo CSV con los datos obtenidos de manera que se puedan generar luego gráficos con ellos.

Debido a que el script levanta tanto el servidor como el cliente en distintos subprocesos se pueden obtener las mediciones directamente desde la terminal. Recogemos los logs del cliente y del servidor en memoria y cuando el cliente finaliza paramos el servicio y el hilo correspondiente al servidor. A partir de aquí se recorren los logs con patrones de búsqueda para encontrar dónde ha guardado los resultados y se dispone todo en unas listas de resultados, cada uno correspondiente con su tipo de resultado correspondiente. Después de esto se generan los CSV para poder generar las gráficas con otro script.

Las métricas utilizadas para evaluar el rendimiento de las implementaciones son:

- Los **tiempos de comunicación** hacen referencia al tiempo empleado para poder enviar y recibir los datos de la CPU a la GPU y viceversa. En cuanto al tiempo de comunicación hay que tener en cuenta que implica todas aquellas interacciones que existen entre CPU y GPU para el envío de datos entre ambos. En este caso ese tiempo se centra en el envío de la matriz de datos inicial a la GPU y el envío de vuelta a CPU.

- Aunque la aplicación realiza varias acciones para obtener los resultados, una parte muy importante de cara a la evaluación es el **tiempo que dedica de la GPU para la ejecución** ya que es lo que se busca mejorar. El tiempo de ejecución se refiere a la cantidad de tiempo que pasa desde que se lanza el kernel hasta que se recupera la ejecución en el host tras la sincronización de la GPU.

- Para saber cómo afectan los cambios realizados a las estructuras de datos se ha analizado el **consumo de memoria física de la aplicación**. Esta memoria hace referencia a la que se está utilizando realmente y no tiene en cuenta la memoria virtual del sistema operativo.

- Por último, el **tiempo de ejecución total** se refiere al tiempo que ha sido necesario desde que se recibe la petición del cliente hasta su respuesta final.

4.1.1. Cargas de trabajo

Para realizar las pruebas se han utilizado distintas cargas de trabajo que son indicadas mediante la estructura de datos que es enviada por el cliente que indican las variables de control utilizadas por la aplicación y los datos que tiene que utilizar para llevar a cabo la ejecución. En cuanto a las cargas de trabajo utilizadas se han utilizado dos en concreto: una de carga baja y otra de carga alta.

La carga de trabajo baja se realizan partiendo de solo un cromosoma simplificado siendo la menor carga de trabajo posible que se puede utilizar. Debido a que los cromosomas son diferentes entre si, no se puede establecer una carga de trabajo igual si se decide cambiar el cromosoma para realizar las pruebas, por lo que siempre será usado el mismo para las pruebas de trabajo bajas. Para poder hacer esto, en la estructura de datos, hay que indicar la muestra que queremos usar para poder realizar la ejecución y la muestra de control. Los archivos que contienen este cromosoma tienen un tamaño de 30/40 Mega-bytes. Para la carga de trabajo

baja que vamos a usar solo se van a ejecutar cinco niveles para aplicar DWT multinivel.

La segunda carga de trabajo seria la carga de trabajo pesada que consistiría en datos reales sobre un cromosoma completo. **Las muestras de los casos y de las variables de control de este cromosoma tienen un tamaño de alrededor de 1 Giga-byte** lo que supone un trabajo adicional para aplicación ya que la carga de datos es muy pesada y no se pueden usar los datos puros si no que hay que procesar e hidratar una estructura de datos con los archivos. Para esta carga de trabajo se van a aplicar seis niveles de DWT a señal de metilación.

4.2. Pruebas con una sola GPU y Multi-GPU

El objetivo de esta sección es mostrar los resultados que se obtienen al ejecutar la aplicación de HPG_Dhunter en un sistema que solo contenga un dispositivo y comprobar cuál es la diferencia entre las distintas implementaciones.

4.2.1. Carga de trabajo baja

En esta primera parte de la evaluación se usarán cargas de trabajo bajas las cuales pueden estar solucionadas en menos de 25 segundos por la implementación inicial de CUDA.

Tiempo de comunicación

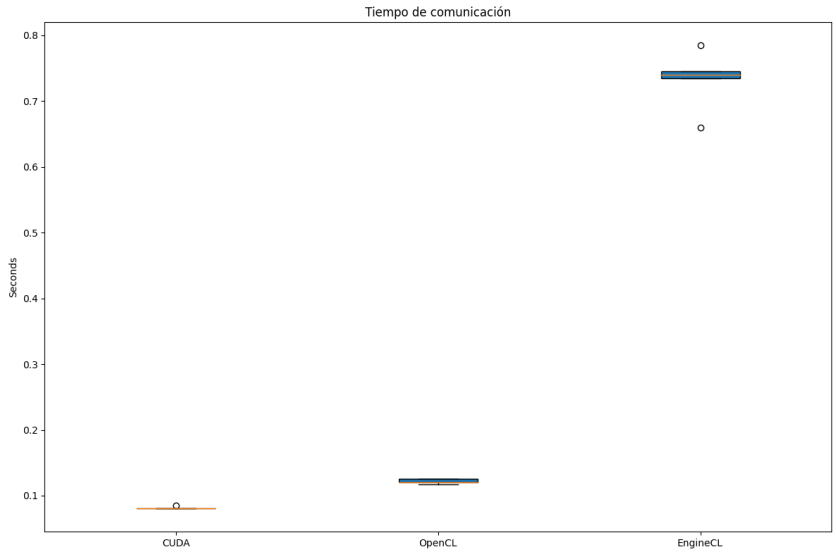

Figura 4.1: Tiempos de comunicación en las distintas implementaciones

Como se observa en la figura 4.1 la implementación original CUDA es la que obtiene los mejores tiempos de comunicación (cuanto menor sea el tiempo mejor) seguido de cerca por la segunda implementación de OpenCL y por último está EngineCL.

Aún siendo tiempos similares entre CUDA y OpenCL, la implementación en CUDA tarda menos en enviar y recibir datos de la GPU. Esta diferencia puede explicarse por la mayor optimización del driver de CUDA, compatible solo con tarjetas NVIDIA, frente a OpenCL, que

necesita a la mayoría de tarjetas/dispositivos del mercado.

En el caso de EngineCL, el tiempo de comunicación llega a ser 5 veces más elevado que la implementación OpenCL en la que se basa. Recordando la explicación de 3.5.2, EngineCL necesita enviar y recuperar los datos de la GPU tras la ejecución debido a la necesidad de tener que crear todos los elementos necesarios para cada ejecución de la herramienta. Esto conlleva a que el tiempo de comunicación se vea afectado drásticamente ya que la herramienta tiene que enviar a la GPU todos los datos por cada nivel y no puede reutilizar los datos que ya tiene en memoria. Esto también implica tener que recibir de la GPU la matriz de datos de nuevo tras la ejecución por lo que el tiempo de comunicación se dispara y escala linealmente con respecto al número de niveles que se tienen que llevar a cabo. Para verificar este resultado se realizaron las mismas pruebas pero ejecutando solo en un solo nivel el DWT. Como se puede observar en la figura 4.2 el tiempo de EngineCL baja drásticamente acercándose bastante al tiempo del resto de implementaciones aunque sigue pudiendo apreciarse cierta diferencia de tiempos. Esto se debe a las tareas que realiza EngineCL de forma interna, relacionadas con con la distribución de la memoria y el manejo de los bufferes.

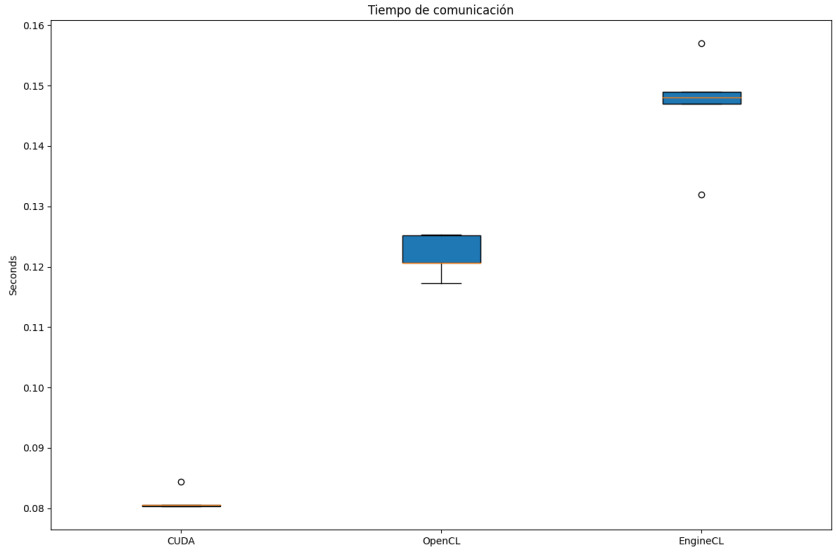

Figura 4.2: Tiempos de comunicación con un solo nivel para DWT

Mirando más a fondo los tiempos de OpenCL y en EngineCL vemos como son más estables de lo esperado pero en EngineCL los picos en los tiempos de ejecución mínimos y máximos difieren de la media, aunque la diferencia en este caso es prácticamente inexistente.

Memoria consumida

Como se observa en la figura 4.3 la implementación de CUDA y OpenCL consumen prácticamente la misma memoria. Aún habiendo tenido que generar nuevas estructuras de datos para OpenCL debido a que tenemos que mantener ciertas variables del contexto, no ha habido un incremento en el uso de memoria notable. Esto se debe a que OpenCL ofrece los punteros para toda la información que se necesita sobre los dispositivos y gracias a eso el incremento de memoria es prácticamente inexistente.

En cambio si nos fijamos en lo que ocurre con EngineCL vemos que existe un aumento de entre 200 y 250 MiB. Para la implementación de EngineCL hubo que realizar cambios en la

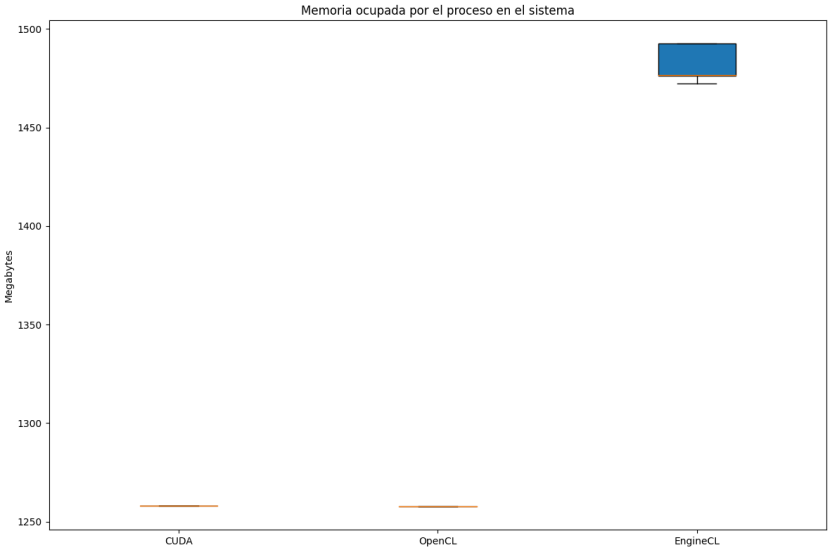

Figura 4.3: Memoria consumida en las distintas implementaciones

estructura de datos de la aplicación. Este cambio ha provocado que la memoria aumente sustancialmente, así como tener que instanciar toda la herramienta de EngineCL, lo cual implica que toda la búsqueda de dispositivos, uso de variables y el uso de las clases relacionadas con la herramienta lo que también provoca un aumento en el consumo de la memoria.

Como se comenta en el apartado 3.5.3, el cambio de la estructura de datos se debe a la mayor facilidad de manera un vector que contenga la información, aunque esto implique un mayor uso de memoria En cuanto al cambio de estructuras de datos, como se comenta en el apartado 3.5.3, ha sido preferible realizarlo debido a la facilidad de manejar un vector de datos con respecto a la estructura anterior, aunque esto implique un aumento en el consumo. Gracias a esto, se ha conseguido reducir el tiempo de ejecución de la aplicación. El tiempo previo al cambio de estructuras con la implementación de EngineCL era en media de 32 segundos mientras que el tiempo con el cambio se reduce a 25 segundos implicando un reducción de 7 segundos. A esto hay que añadir que en esta sección se esta teniendo en cuenta una carga de trabajo baja con una matriz de datos pequeña por lo que cuanto más aumenten los datos a procesar más aumentará la diferencia entre las dos versiones.

Tiempo de ejecución en GPU

Como se muestra en la figura 4.4 los tiempos de ejecución de la GPU en CUDA y OpenCL son muy parecidos con ventaja de OpenCL debido a los cambios realizados sobre los kernels. En cambio EngineCL el tiempo de ejecución es significativamente más alto. Ocurre lo mismo que con el tiempo de comunicación, es necesario ejecutar el kernel el número de niveles que se quiere aplicar DWT multinivel por lo tanto la diferencia debería de ser de al menos 5 veces más lenta. Aún así la diferencia que se muestra es aún más grande. Al igual que en el tiempo de comunicación se ha realizado una prueba que solo realice un solo nivel de DWT.

Mirando la figura 4.5 se puede observar como existe una diferencia obvia entre los tiempos de ejecución de CUDA y OpenCL con respecto a la de EngineCL. Tener en cuenta que esta diferencia de tiempo no tiene que ver solo con el tiempo de ejecución de la GPU, sino que también hay que considerar la división sobre la carga de trabajo entre GPUs y la sincronización.

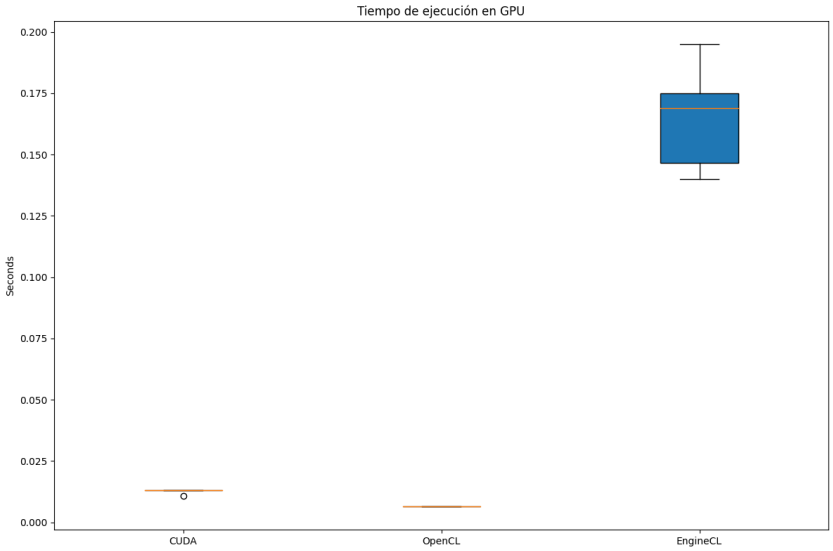

Figura 4.4: Tiempos de ejecución en GPU en las distintas implementaciones

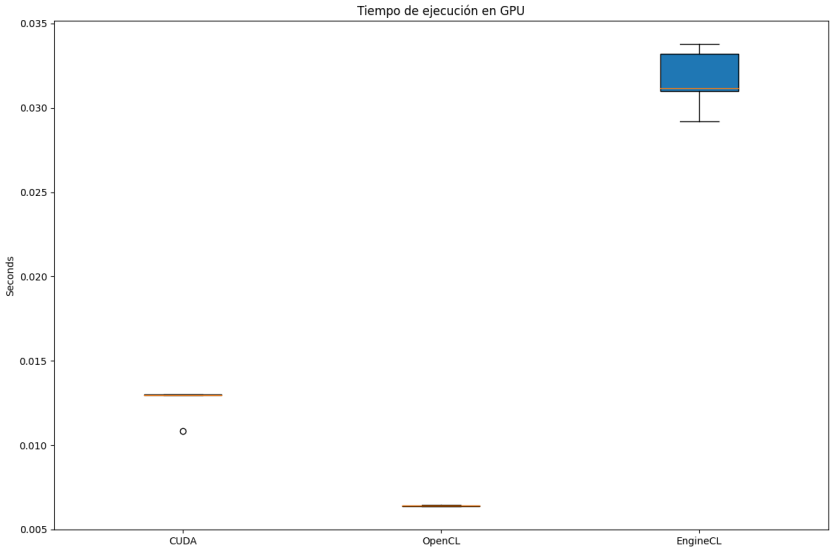

Figura 4.5: Tiempos de ejecución en GPU con un solo nivel para DWT

Tiempo de ejecución total

En la figura 4.1 vemos cómo existe una gran diferencia en los tiempos entre CUDA y OpenCL con EngineCL. Como se ha comentado anteriormente, esto se debe a las veces que se tiene que ejecutar el kernel debido a la necesidad de tener que generar toda la información de nuevo, de tener que enviar y recibir los datos, de tener que realizar la sincronización con las GPUs y la gestión del planificador, todo esto por cada ejecución del kernel lo que implica un aumento notorio en el tiempo total.

Aún así si miramos la figura 4.7 podemos observar como todos los tiempos se acercan, incluyendo los de EngineCL. En esta gráfica se observan las ejecuciones de la aplicación solo aplicando un nivel para DWT. Los tiempos de ejecución total se acercan con respecto a la figura anterior donde había cinco niveles que ejecutar, pero si miramos el tiempo de ejecución

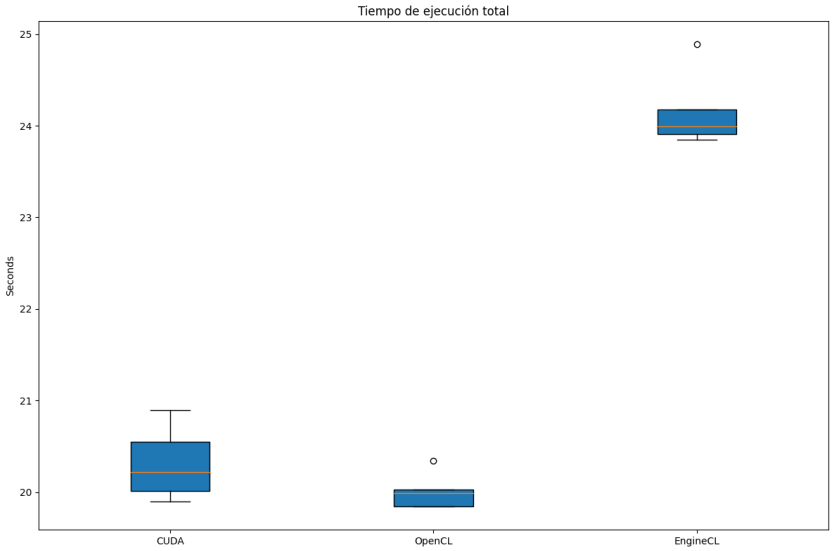

Figura 4.6: Tiempos total de las distintas implementaciones

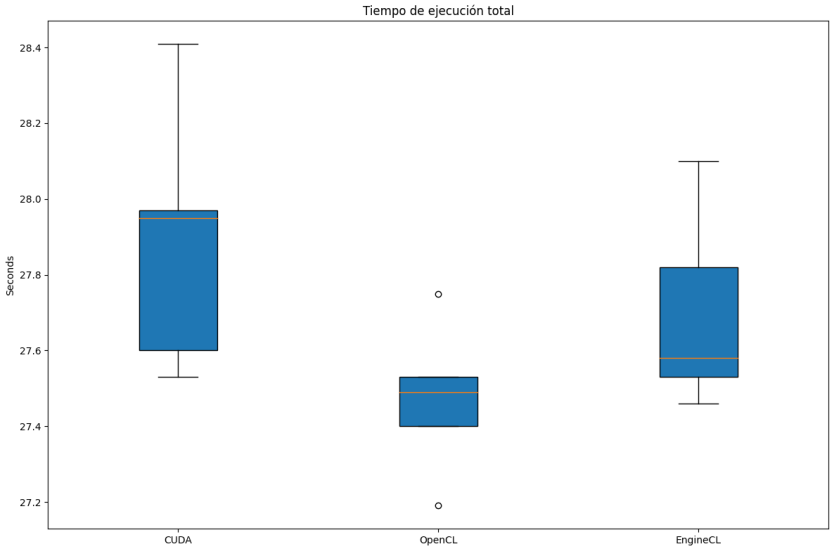

Figura 4.7: Tiempos total de con un solo nivel para DWT

ha aumentado en todas las implementaciones. Debido a que DWT se dedica a dividir el tamaño de la matriz de datos, a cada nivel que se calcula la matriz se reduce su tamaño en la mitad. Dependiendo del número de niveles a aplicar DWT, la matriz de datos que resulta y es tratada por el host disminuye en sus dimensiones. Cuando el número de niveles a aplicar es bajo, la matriz de datos sera de una mayor dimensión y al continuar la aplicación ejecutando desde la CPU implica un aumento considerable de los tiempos de ejecución.

4.2.2. Carga de trabajo alta

En esta segunda parte de la evaluación se usarán cargas de trabajo pesadas cuyo tiempo de ejecución ronda los 195 segundos desde la implementación inicial de CUDA.

Para esta carga de trabajo alta se ha añadido a las pruebas la implementación de EngineCL con el uso de 1, 2 y 4 GPUs en las ejecuciones y usando los distintos tipos de planificadores de EngineCL (estático y dinámico). Para el planificador estático se ha intentado dar la misma carga de trabajo a todas las GPUs por igual, mientras que con el planificador dinámico se ha probado a dividir el trabajo entre cantidades sobre el 20 % y 30 %, acabando por usar cantidades de cargas de 20 % ya que reflejaban mejores resultados.

Tiempo de comunicación

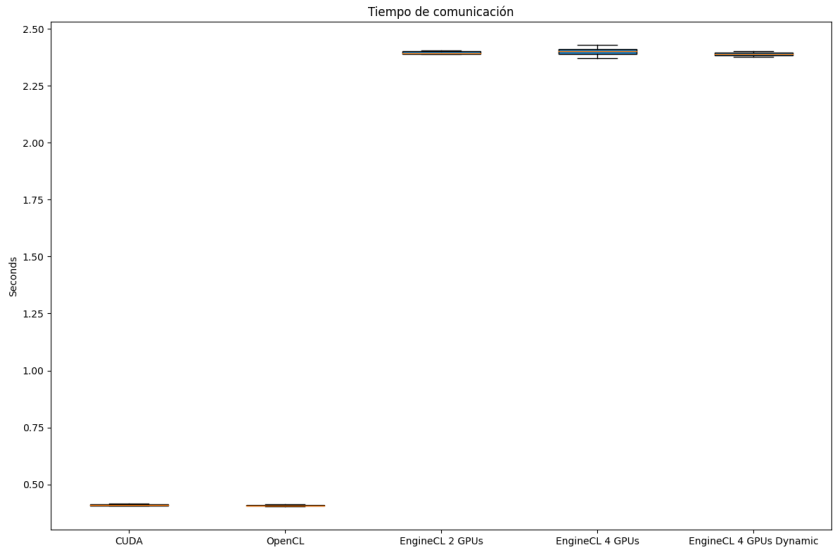

Figura 4.8: Tiempos de comunicación de las implementaciones con una carga pesada

Sobre el tiempo de comunicación mostrado en la figura 4.8 podemos observar como las implementaciones que no hacen uso de EngineCL (CUDA y OpenCL) se mantienen con un tiempo similar. Al igual que ocurría con los tiempos de comunicación en la carga de prueba baja, el tiempo de comunicación en EngineCL se dispara. Esto se debe a la necesidad de tener que usar la herramienta de EngineCL por cada nivel que se aplica de DWT. Como se ha mostrado en el apartado anterior en cuanto los niveles a aplicar se reducen a 1, los tiempos se acercan a los de las implementaciones de CUDA y OpenCL existiendo cierto overhead por la gestión y control realizado por la herramienta.

Memoria consumida

Sobre la memoria consumida de la aplicación podemos ver como existe un aumento enorme respecto a la memoria comparándolo con los resultados obtenidos de la carga de trabajo baja, pasando de usar 1.5 Giga-bytes a 18 Giga-bytes. Aún con este incremento del uso de la memoria, se observa como ocurre lo mismo con respecto a las implementaciones de EngineCL donde existe un aumento considerable de la memoria con respecto a las otras dos implementaciones. Como se comentó previamente, este aumento de la memoria existe debido al cambio de estructura de datos necesario para poder usar la herramienta y al propio uso de la misma. Comparando con la diferencia que ocurre en la carga baja, el aumento es mucho menor en relación al volumen de datos utilizados.

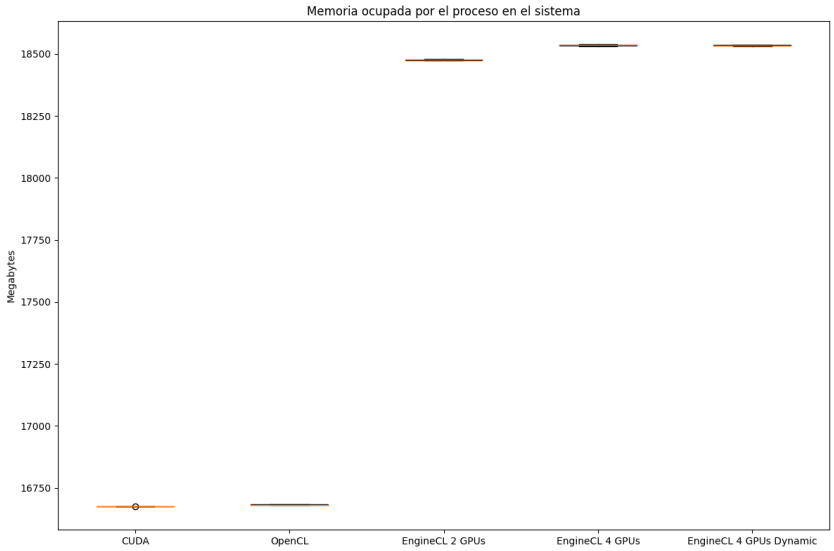

Figura 4.9: Memoria consumida por las implementaciones con una carga pesada

Tiempo de ejecución en GPU

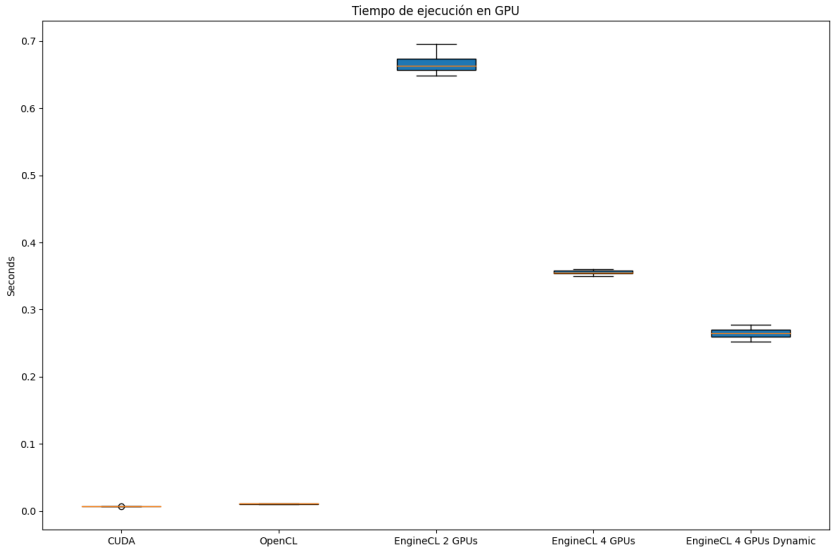

Figura 4.10: Tiempos de ejecución en GPU de las implementaciones con una carga pesada

En la figura 4.10 se muestran los tiempos de ejecución de la GPU donde vemos como existe la misma situación que estamos comentando en el resto de medidas pero con una diferencia sustancial. Tanto CUDA como OpenCL se encuentran en tiempos de uso de la GPU similares mientras que la implementación de EngineCL con dos GPUs usando el planificador estático, tiene un aumento significativo teniendo en cuenta el tiempo obtenido de CUDA y OpenCL inicialmente. Este aumento hay de tiempo en las versiones EngineCL tiene que ver con el aumento de la carga de trabajo enviada a la GPU y la necesidad de la sincronización de las GPUs por parte del host para poder seguir con el flujo del programa.

En cambio, para el caso de la ejecución que usa 4 GPUs, tanto con el planificador estático

como con el planificador dinámico, se observa como el tiempo se reduce de forma drástica. Al enviar una carga de trabajo más pequeña a las GPUs su tiempo de ejecución se acorta significativamente y el tiempo de espera necesario para sincronizar las GPUs se reduce de forma importante.

Tiempo de ejecución total

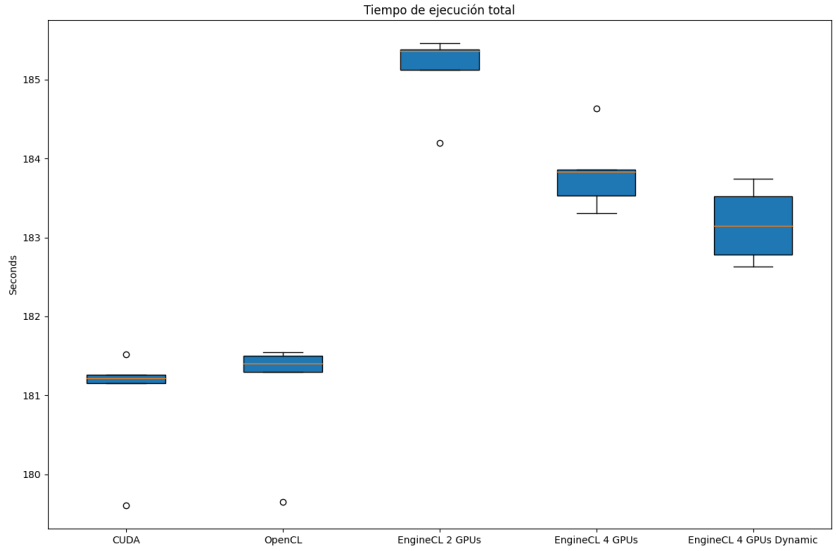

Figura 4.11: Tiempos total de las implementaciones con una carga pesada

Con respecto del tiempo total de ejecución de la aplicación, de la figura 4.11, vemos como existen amplias diferencias entre los tiempos de las distintas versiones. Se puede observar como la implementación de CUDA es la que tiene un tiempo de ejecución más bajo, teniendo un tiempo prácticamente igual la versión de OpenCL.

Viendo los tiempos de ejecución de la implementación de EngineCL se ve la misma tendencia que con el tiempo de ejecución en GPU. A la hora de usar más GPUs y haciendo uso del planificador dinámico, se reduce la cantidad de trabajo enviada a cada GPU. Esto conlleva que finalicen antes las ejecuciones y se puede llegar a un sincronización más rápida. Con estos resultados nos damos cuenta de que el algoritmo no escala bien con múltiples GPUs (al menos usando EngineCL), pues estamos obteniendo un mejor rendimiento con la implementación OpenCL de una sola GPU que con 4. Dentro de los tiempos podemos ver que hay casos mas extremos con respecto a la mediana que se deben a la lectura de los datos iniciales.

Capítulo 5

Conclusiones y trabajos futuros

En el capítulo final, se detallarán las conclusiones alcanzadas del proceso de desarrollo y de la posterior experimentación. Estas conclusiones abordarán aspectos como el logro de los objetivos propuestos, las reflexiones personales, los obstáculos encontrados, los conocimientos adquiridos y la valoración general del trabajo realizado.

También se incluirán sugerencias sobre posibles futuros trabajos y evoluciones relacionados con el proyecto y el trabajo realizado.

5.1. Conclusiones

Como se describe en la Sección 1.5, los objetivos del proyecto se centran en varios aspectos de la implementación y las experimentaciones asociadas. A continuación, se analizará lo que se ha logrado hasta el momento, detallando los aspectos que han influido en la complejidad y han marcado diferencias significativas en su consecución:

- El primer objetivo tenía como meta el estudio de la aplicación y llegar a realizar una ejecución en la GPU. Dentro de la aplicación existen varias capas de profundidad relacionadas con cálculos centrados en los cromosomas y las referencias de los genomas sobre las que no se han profundizado. Sobre la aplicación solo hacía falta centrarse en la implementación realizada de DWT multinivel. Todo esto viene desarrollado y explicado en el capítulo 2 y en las primeras secciones del capítulo 3. También se consiguió realizar varias ejecuciones de distintos cromosomas gracias al profesor Carlos Reaño de la Universidad de Valencia, que nos proporcionó acceso al sistema, preparó la aplicación para ser usada de forma sencilla y nos ha resuelto ciertas dudas a lo largo de toda la realización del trabajo.

- El segundo objetivo consistía en proponer una nueva implementación sobre la plataforma de OpenCL. El capítulo 3, específicamente la sección 3.4, aborda la modificación de la implementación inicial en CUDA para poder trasladarlo OpenCL y cómo esto ha dado lugar a cambios en el diseño, cambiando todos los kernels reduciendo solo a uno y moviendo el control de la ejecución de los niveles al host en vez de mantenerlo en la GPU. Sobre esta implementación hay dos diseños, siendo el segundo el que se ha llegado a implementar. El primero se basaba en replicar la estructura y diseño de la implementación inicial con una arquitectura padre-hijo dentro del kernel. El cambio de diseño vino al localizar que no era necesaria la copia de vuelta de datos ya que se podía utilizar solo una matriz auxiliar para mantener los datos e intercambiar las matrices de datos en los siguientes niveles. En cuanto a dificultad, el primer diseño implicaba una implementación más costosa y

farragosa debido a tener que manejar varias matrices de datos de forma paralela y siendo más complicado tener esa arquitectura de padre-hijo en OpenCL. Sobre el segundo diseño, una vez que se definió cómo tenía que actuar el host para poder mandar toda la carga de datos a la GPU fue una implementación muy directa. Se ha conseguido entender cuál es el tipo de diseño que hay que aplicar para diseñar aplicaciones que hagan uso de las GPUs y se ha llevado a cabo con éxito. También se ha desarrollado una nueva versión de la aplicación que utiliza el lenguaje de OpenCL para poder aplicar el algoritmo de DWT.

- El tercer y cuarto objetivos tenían como final realizar una implementación multi-GPU mediante la herramienta de EngineCL para aumentar el rendimiento y la escalabilidad de la aplicación. Dentro del capítulo 3, en la sección 3.5 dedicada a EngineCL se muestra todo el diseño y estructura seguida para llegar a realizar la implementación. La utilización de EngineCL impone una serie de restricciones como se expone en la sección 3.5.3 que han complicado la implementación inicial. A pesar de estas limitaciones, utilizar EngineCL es bastante sencillo y aplicarlo en este contexto implica algunas consideraciones adicionales debido a como esta enfocado el uso de EngineCL ya que se centra en aquellas aplicaciones que hacen uso de una sola ejecución de un kernel. En cambio, debido al algoritmo multinivel que se ha aplicado en la aplicación y existiendo una dependencia en los datos con el nivel previo, añadir EngineCL a la aplicación implica usar la herramienta en una situación no ideal para ella. Aún teniendo en cuenta estas consideraciones, la tarea de añadir EngineCL ha sido sencilla gracias a su diseño y estructura. Con todo esto se ha llegado a crear una nueva versión de la aplicación que hace uso de EngineCL para gestionar todo lo que tiene que ver con la comunicación y el uso de las GPUs

- El quinto objetivo se basaba en la evaluación de las nuevas implementaciones y desarrollos realizados, presentada en el capítulo 4, en el que se pudiese ver reflejado si existen mejorías con respecto a la implementación inicial. En este capítulo se explican los experimentos llevados a cabo y los análisis sobre los resultados obtenidos gracias al uso de varias gráficas. También habría que incluir aquí las limitaciones mencionadas en el punto previo que han implicado un cambio sobre la propia herramienta de EngineCL y cuyos cambios solo sirven para esta aplicación. Los resultados obtenidos muestran como la implementación de OpenCL es prácticamente la misma que la de CUDA pero permitiendo ser ejecutada por cualquier dispositivo. Con los resultados obtenidos de EngineCL se muestra como la herramienta no esta diseñada para este tipo de implementaciones y los resultados no son beneficiosos para esta implementación.

Hay que destacar que en el caso de las pruebas realizadas con multi-GPU no es necesario el uso de solo GPUs para poder utilizar la herramienta de EngineCL. Al igual que OpenCL, EngineCL entiende que un dispositivo es todo aquel tipo de procesador que disponga de un driver de OpenCL y esté operativo en el sistema.

5.2. Valoración personal

A nivel personal y profesional creo que este proyecto me ha ayudado a conocer ciertos elementos de la programación que no conocía, como es la programación y el uso de GPUs. Me ha ayudado a comprender la complejidad que existe a la hora de hacer un uso correcto de GPUs teniendo en cuenta el mantenimiento y la dedicación que implica la paralelización de una aplicación al hacer uso de una cantidad tan grande de datos.

Gracias a la optimización del rendimiento con el uso de GPUs y la máxima implementación de paralelismo en el código, se ha experimentado un cambio significativo en el enfoque de la resolución del problema.

Es importante destacar que en el sector tecnológico, es común no ser consciente de los beneficios y problemas si se abordasen los problemas desde perspectivas diferentes. Esta falta de conocimiento se debe en parte a la constante de evolución en el sector, que evoluciona de manera continua. Estos nuevos conocimientos simplemente se presentan como parte natural de la progresión en este campo.

5.3. Trabajos futuros

Viendo los resultados obtenidos al implementar OpenCL y EngineCL sobre la aplicación, existen varias mejoras que se pueden realizar para poder mejorar el rendimiento obtenido.

- **Permitir reutilizar el Runtime y Devices:** A la hora de usar EngineCL estaría bien poder reutilizar la herramienta de manera consecutiva sin tener que redefinir todos los elementos de nuevo.
- **Implementación alternativa del kernel:** Puede que se pueda aplicar alguna otra alternativa sobre la matriz de datos en vez DWT de manera que no exista una dependencia continua entre los resultados y los valores de la siguiente iteración.
- **Hacer uso de distintos aceleradores:** Debido a que EngineCL permite el uso de cualquier tipo de procesadores y acelerados se podría llega a usar la aplicación sobre FPGAs y probar si el rendimiento aumenta.
- **Usar distintos planificadores:** En las pruebas y por defecto EngineCL ofrece un planificador estático y dinámico. Se podría implementar distintos algoritmos de equilibrio de carga para ser usados sobre EngineCL y comprobar los resultados.

Bibliografía

[1] Tor M. Aamodt, Wilson Wai Lun Fung, and Timothy G. Rogers. *General-Purpose Graphics Processor Architectures*. Morgan & Claypool Publishers, 2018.

[2] Emilio Castillo, Cristobal Camarero, Ana Borrego, and José Luis Bosque. Financial applications on multi-cpu and multi-gpu architectures. *J. Supercomput.*, 71(2):729–739, 2015.

[3] J. Cheng, M. Grossman, and T. McKercher. *Professional CUDA C Programming*. EBL-Schweitzer. Wiley, 2014.

[4] William J. Dally, Stephen W. Keckler, and David B. Kirk. Evolution of the graphics processing unit (gpu). *IEEE Micro*, 41(6):42–51, 2021.

[5] Jie Feng, Wenjing Zhang, Qingqi Pei, Jinsong Wu, and Xiaodong Lin. Heterogeneous computation and resource allocation for wireless powered federated edge learning systems. *IEEE Transactions on Communications*, 70(5):3220–3233, 2022.

[6] Lisardo Fernández, Ricardo Olanda, Mariano Pérez, and Juan M Orduña. A web-based tool for automatic detection and visualization of dna differentially methylated regions. *Electronics*, 10(9):1083, 2021.

[7] Lisardo Fernández, Mariano Pérez, Ricardo Olanda, Juan M. Orduña, and Joan Marquez-Molins. Hpg-dhunter: an ultrafast, friendly tool for dmr detection and visualization. *BMC Bioinformatics*, 21(1):287, Jul 2020.

[8] Lisardo Fernández, Mariano Pérez, and Juan M. Orduña. Visualization of dna methylation results through a gpu-based parallelization of the wavelet transform. *The Journal of Supercomputing*, 75(3):1496–1509, Mar 2019.

[9] Maria Angelica Davila Guzman, Raúl Nozal, Ruben Gran Tejero, María Villarroya-Gaudó, Darío Suárez Gracia, and José Luis Bosque. Cooperative cpu, gpu, and FPGA heterogeneous execution with enginecl. *J. Supercomput.*, 75(3):1732–1746, 2019.

[10] Asad Hayat, Yasir Noman Khalid, Muhammad Siraj Rathore, and Muhammad Nadeem Nadir. A machine learning-based resource-efficient task scheduler for heterogeneous computer systems. *The Journal of Supercomputing*, pages 1–29, 2023.

[11] Stijn Heldens, Pieter Hijma, Ben Van Werkhoven, Jason Maassen, and Rob V Van Nieuwpoort. Lightning: Scaling the gpu programming model beyond a single gpu. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 492–503. IEEE, 2022.

[12] Adrian Horga, Sudipta Chattopadhyay, Petru Eles, and Zebo Peng. Systematic detection of memory related performance bottlenecks in gpgpu programs. *Journal of Systems Architecture*, 71:73–87, 2016.

[13] National Human Genome Research Institute. Definición de metilación. https://www.genome.gov/es/genetics-glossary/Methylation, May 2023.

[14] D. R. Kaeli, P. Mistry, D. Schaa, and D. P. Zhang. *Heterogeneous Computing with OpenCL 2.0.* Morgan Kaufmann Publishers Inc., 1st edition, 2015.

[15] D. B. Kirk and W. W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach.* Morgan Kaufmann, 4st edition, 2022.

[16] Shin-Ying Lee, Akhil Arunkumar, and Carole-Jean Wu. Cawa: Coordinated warp scheduling and cache prioritization for critical warp acceleration of gpgpu workloads. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 515–527, 2015.

[17] Xinxin Mei and Xiaowen Chu. Dissecting gpu memory hierarchy through microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems*, 28(1):72–86, 2017.

[18] Sparsh Mittal and Jeffrey S. Vetter. A survey of cpu-gpu heterogeneous computing techniques. *ACM Comput. Surv.*, 47(4), jul 2015.

[19] Celeste Moya-Valera, Francisco Lara-Hernández, Elena Quiroz, Veronica Lendinez, Rebeca Melero-Valverde, Wladimiro Díaz, Vicente Arnau, Lisardo Fernández, Juan M Orduña, Sergio Valdés, et al. Analysis of differentially methylated regions in the exome of patients with type ii diabetes. *Regulation*, 2(1.3):12.

[20] Raúl Nozal and José Luis Bosque. Exploiting co-execution with oneapi: Heterogeneity from a modern perspective. In Leonel Sousa, Nuno Roma, and Pedro Tomás, editors, *Euro-Par 2021: Parallel Processing - 27th International Conference on Parallel and Distributed Computing, Lisbon, Portugal, September 1-3, 2021, Proceedings*, volume 12820 of *Lecture Notes in Computer Science*, pages 501–516. Springer, 2021.

[21] Raúl Nozal and Jose Luis Bosque. Mashing load balancing algorithm to boost hybrid kernels in molecular dynamics simulations. *The Journal of Supercomputing*, 79(1):1065–1080, 2023.

[22] Raúl Nozal. *Optimizing performance and energy efficiency in massively parallel systems.* PhD thesis, 1 2022.

[23] Raúl Nozal, Jose Luis Bosque, and Ramon Beivide. Enginecl: Usability and performance in heterogeneous computing. *Future Generation Computer Systems*, 107:522–537, 2020.

[24] Juan M Orduña, Lisardo Fernández, and Mariano Pérez. On the use of parallel architectures in dna methylation analysis. In *International Conference on Information Technology & Systems*, pages 3–12. Springer, 2023.

[25] B. Pérez, J. L. Bosque, and R. Beivide. Simplifying programming and load balancing of data parallel applications on heterogeneous systems. In *9th Annual Workshop on General Purpose Processing Using Graphics Processing Unit*, GPGPU '16, pages 42–51. ACM, 2016.

[26] B. Pérez, E. Stafford, J. L. Bosque, and R. Beivide. Energy efficiency of load balancing for data-parallel applications in heterogeneous systems. *The Journal of Supercomputing*, 73(1):330–342, 2017.

[27] Borja Pérez, Esteban Stafford, José Luis Bosque, and Ramón Beivide. Sigmoid: An auto-tuned load balancing algorithm for heterogeneous systems. *J. Parallel Distributed Comput.*, 157:30–42, 2021.

[28] Borja Pérez, Esteban Stafford, José Luis Bosque, Ramón Beivide, Sergi Mateo, Xavier Teruel, Xavier Martorell, and Eduard Ayguadé. Auto-tuned opencl kernel co-execution in ompss for heterogeneous systems. *J. Parallel Distributed Comput.*, 125:45–57, 2019.

[29] James Reinders. Sycl, dpc++, xpus, oneapi. In Simon McIntosh-Smith, editor, *IWOCL'21: International Workshop on OpenCL, Munich Germany, April, 2021*, page 19:1. ACM, 2021.

[30] Gabriel Rodriguez-Canal, Yuri Torres, Francisco J Andújar, and Arturo Gonzalez-Escribano. Efficient heterogeneous programming with fpgas using the controller model. *The Journal of Supercomputing*, 77(12):13995–14010, 2021.

[31] Mauricio Rodríguez Dorantes, Nelly Téllez Ascencio, Marco A Cerbón, Marisol López, and Alicia Cervantes. Metilación del ADN: un fenómeno epigenético de importancia médica. *Rev. Invest. Clin.*, 56(1):56–71, 2004.

[32] Esteban Stafford, Borja Pérez, José Luis Bosque, Ramón Beivide, and Mateo Valero. To distribute or not to distribute: The question of load balancing for performance or energy. In Francisco F. Rivera, Tomás F. Pena, and José Carlos Cabaleiro, editors, *Euro-Par 2017: Parallel Processing - 23rd International Conference on Parallel and Distributed Computing, Santiago de Compostela, Spain, August 28 - September 1, 2017, Proceedings*, volume 10417 of *Lecture Notes in Computer Science*, pages 710–722. Springer, 2017.

[33] Pablo Toharia, Oscar David Robles, José Luis Bosque, and Angel Rodríguez. Scalable shot boundary detection. *J. Supercomput.*, 64(1):89–99, 2013.

[34] Mohamed Zahran. Heterogeneous computing: Here to stay: Hardware and software perspectives. *Queue*, 14(6):31–42, dec 2016.