UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Integración de simuladores de tecnologías de memoria avanzadas con GPGPU-Sim

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autor: Carlos Baiget Orts

Tutor: Salvador Petit Martí
Julio Sahuquillo Borrás

Curso 2021/22

Dedicatoria

A las tres ilusiones de mi vida: a mis hijos Atila y Maia, y a mi esposa Mª José.

Agradecimientos

Este trabajo habría sido imposible sin la inestimable ayuda, paciencia y consejos que en todo momento me prestaron mis tutores Salva y Julio, la cual extendieron con creces mucho más allá de los límites a los que tenían derecho.

Así mismo, expreso mi eterna gratitud a Francisco Candel, por compartir desinteresadamente su imprescindible buen criterio a la hora de resolver los numerosos errores de programación que se presentaron a lo largo de este trabajo.

Resum

Els sistemes de computació heterogènia basats en un paradigma híbrid de computació escalar i vectorial són l'aposta actual com a mitjà d'aconseguir la computació a exaescala. El maquinari encarregat dels càlculs vectorials és normalment una GPU o unitat de processament gràfic, i és objecte d'una especial atenció en el camp de l'enginyeria de computadors.

Entre les múltiples complexitats en el disseny de les GPU, el subsistema de memòria és un punt crític i que concentra una gran part d'atenció. Tant per a la proposta de millores en aquest subsistema, o com per al disseny de nous subsistemes, les eines de simulació detallada aporten major velocitat i menors costos respecte al disseny de prototips funcionals.

En aquest TFG es proposa la integració d'un simulador detallat de GPU, GPGPU-Sim, àmpliament utilitzat per al modelatge de GPU amb tres simuladors de memòria principal diferents. Es descriuen els aspectes de disseny i implementació per a aquesta integració i es realitzen diversos experiments demostrant les capacitats de la proposta: i) permet la comparació de models de tecnologies de memòria entre diferents simuladors, ii) permet modelar una àmplia varietat de tecnologies de memòria, i iii) obri la possibilitat de realitzar anàlisi que no és possible efectuar únicament amb GPGPU-Sim.

Paraules clau: GPU, GTX480, GPGPU, Memòria principal, Rodinia, Simulació

Resumen

Los sistemas de computación heterogénea basados en un paradigma híbrido de computación escalar y vectorial son la apuesta actual como medio de alcanzar la computación a exaescala. El hardware encargado de los cálculos vectoriales es normalmente una GPU o unidad de procesamiento gráfico, y es objeto de una especial atención en el campo de la ingeniería de computadores.

Entre las múltiples complejidades en el diseño de las GPU, el subsistema de memoria es un punto crítico y que concentra una gran parte de atención. Tanto para la propuesta de mejoras en este subsistema, o como para el diseño de nuevos subsistemas, las herramientas de simulación detallada aportan mayor velocidad y menores costes con respecto al diseño de prototipos funcionales.

En este TFG se propone la integración de un simulador detallado de GPU, GPGPU-Sim, ampliamente utilizado para el modelado de GPU con tres simuladores de memoria principal diferentes. Se describen los aspectos de diseño e implementación para esta integración y se realizan varios experimentos demostrando las capacidades de la propuesta: i) permite la comparación de modelos de tecnologías de memoria entre diferentes simuladores, ii) permite modelar una amplia variedad de tecnologías de memoria, y iii) abre la posibilidad de realizar análisis que no es posible efectuar únicamente con GPGPU-Sim.

Palabras clave: GPU, GTX480, GPGPU, Memoria principal, Rodinia, Simulación

Abstract

Heterogeneous computing systems based on a hybrid paradigm of scalar and vector computing are the current bet as a means of achieving exascale computing. The hardware in charge of vector calculations is usually a GPU (Graphics Processing Unit), and it is subject of special attention in computer engineering.

Among the many complexities in the design of GPUs, the memory subsystem is a critical key and one that receives a large part of the attention. Both for the proposal of improvements in this subsystem, or for the design of new subsystems, detailed simulation tools provide greater speed and lower costs with respect to the design of functional prototypes.

In this TFG, we propose the integration of a detailed and widely used GPU simulator, GPGPU-Sim, with three different main memory simulators. The design and implementation aspects for this integration are described, and several experiments are carried out demonstrating the capabilities of the proposal: i) it allows comparison of memory technology models between different simulators, ii) it allows a wide variety of memory technologies to be modelled, and iii) it opens up the ability to perform experimental analyses that are not possible with GPGPU-Sim alone.

Key words: GPU, GTX480, GPGPU, Main memory, Rodinia, Simulation

Índice general

Índice de figuras

Índice de tablas

CAPÍTULO 1
Introducción

Desde la aparición de las primeras computadoras electrónicas, la búsqueda de una mayor eficiencia en el rendimiento de las sucesivas generaciones ha consistido principalmente, en el desarrollo de nuevas e ingeniosas soluciones arquitectónicas para conseguir realizar el mayor número de operaciones en el menor tiempo posible.

Este rediseño y mejora continua afecta a todos los elementos que integran un sistema informático, desde las CPU a los buses de interconexión, y también las memorias. Pero también ha dado lugar nuevas unidades funcionales que no existían en las arquitecturas clásicas, siendo las Unidades de Proceso Gráfico, o GPU, una de las más recientes soluciones de ingeniería de cara a permitir el procesado masivo de datos para cierto tipo de aplicaciones, las cuales se pueden paralelizar masivamente, ofreciendo altísimos rendimientos en comparación con la ejecución en CPU multinúcleo convencionales.

Entre los muchos desafíos que afronta el diseño arquitectónico de las GPU (energéticos, térmicos, interconexión de elementos, etc.) y como consecuencia directa del alto paralelismo que estas unidades alcanzan, se encuentra la necesidad de disponer de un subsistema de memoria cuyo tamaño y velocidad de acceso esté en armonía con la velocidad con la que estas GPU pueden procesar grandes volúmenes de datos, que han de ser leídos y escritos en una memoria principal capaz de atender elevados anchos de banda.

Tanto en la industria como en la academia, uno de los recursos empleados en la búsqueda de soluciones para estos compromisos son los simuladores, unas herramientas de software que permiten modelar y evaluar el rendimiento de una estrategia concreta de implementación de un diseño hardware a partir de su propuesta teórica, antes de su fabricación. En este contexto podemos encontrar simuladores completos de GPU, y también más específicos, por ejemplo, aquellos desarrollados únicamente para modelar subsistemas de memoria o la memoria principal.

Sin embargo, como desarrollos independientes que habitualmente son, es frecuente que entre las funcionalidades ofrecidas por distintos simuladores exista una discordancia de prestaciones, de forma que se convierta en una carencia de un simulador completo de GPU el no poder aprovechar alguna funcionalidad avanzada ofrecida por otro más específico para alguno de sus subsistemas, por ejemplo y de forma notoria, el correspondiente a la memoria principal. En otras palabras, el modelado de un amplio rango de tecnologías de memoria, algo deseable en el estudio del impacto de estas tecnologías en las prestaciones de las GPU, no es factible utilizando exclusivamente un simulador de sistema GPU, sino que requiere de la integración de simuladores especializados en el modelado de la memoria.

El objetivo de este TFG es atender a esta demanda, integrando la funcionalidad de tres simuladores de memoria principal en un simulador de GPU, GPGPU-Sim [5], para extender y actualizar la funcionalidad de este último y así permitir el estudio de nuevas

opciones de diseño y evaluación del rendimiento de sistemas computación de propósito general basados en GPU. El desarrollo realizado en este TFG añade nuevas opciones de configuración para seleccionar entre simuladores de memoria, soporte a múltiples transacciones con memoria principal por acceso a memoria y soporte a la gestión por GPGPU-Sim de los datos contenidos en memoria.

Para las pruebas realizadas con el desarrollo realizado en este TFG, se ha seleccionado la suite de benchmarks Rodinia [6]. Los resultados muestran que el objetivo perseguido se ha conseguido y que el desarrollo realizado permite, entre otras funcionalidades:

- Comparar diversos modelos de tecnologías de memoria principal implementados en distintos simuladores.
- Simular una amplia variedad de tecnologías de memoria.
- Realizar análisis que no es posible efectuar contando únicamente con un simulador de GPU que soporta una sola tecnología de memoria.

El resto de este trabajo se organiza como sigue. En el capítulo 2 se introduce la arquitectura de la GPU GTX480 de Nvidia, así como simuladores de memoria principal y de sistemas basados en GPU. En el capítulo 3 se describe la propuesta que da lugar a este trabajo, partiendo de su diseño básico y enumerando las decisiones más relevantes que permitieron llevarla a cabo. En el capítulo 4 se especifica paso a paso la implantación del software necesario para desarrollar y evaluar la propuesta. En el capítulo 5 se presenta el entorno experimental, incluyendo aplicaciones, configuraciones y métricas analizadas. En el capítulo 6 se presentan y analizan los resultados obtenidos. Finalmente, el capítulo 7 ofrece conclusiones y posibles trabajos futuros, además de un sumario de las competencias trabajadas, y una retrospectiva sobre la relación de ciertas asignaturas del Grado de Ingeniería Informática con este TFG.

CAPÍTULO 2
Antecedentes

En este capítulo, en primer lugar, se introduce la computación de propósito general en GPU (GPGPU). En segundo lugar, se presenta la arquitectura GPU Fermi de Nvidia y la GPU GTX480, implementada con esta arquitectura. Posteriormente, se introducen las principales tecnologías de memoria que son modeladas por simuladores de memoria del estado del arte. Se discuten algunos de estos simuladores de memoria y otros simuladores detallados de sistemas GPU. Finalmente, se describen algunos TFG relacionados.

2.1 Computación de propósito general en GPU

El término GPU o Unidad de Procesamiento Gráfico, acuñado hace apenas un par de décadas en contraposición al concepto clásico de CPU o Unidad Central de Procesamiento, hace referencia a un tipo de arquitectura específica para acelerar ciertas operaciones relativas a la generación de imágenes por parte de un ordenador.

Las GPU, se concibieron originalmente como un *Graphics Pipeline* (secuencia de etapas) de generación de imágenes (*rendering*). Las etapas del pipeline implementaban operaciones gráficas mediante procesadores de uso específico (p.e. *pixel processors* y *vertex processors*) [7]. Estos diseños específicos evolucionaron hacia procesadores de propósito general. La evolución fue motivada en un primer momento para que las prestaciones de las cargas de renderizado no se vieran limitadas por la cantidad de hardware específico en el pipeline, y, más tarde, para adaptar las GPU a otras tareas de cómputo de propósito general altamente paralelizables. Actualmente, las GPU tienen la capacidad de ejecutar algoritmos sobre grandes cantidades datos tanto enteros como de coma flotante, dando lugar al concepto de GPGPU, o Computación de Propósito General con Unidades de Procesamiento Gráfico.

La prioridad de esta arquitectura es maximizar el paralelismo en el procesamiento de grandes conjuntos de datos. Las GPU maximizan el paralelismo siguiendo un paradigma de cómputo SIMT (Single Instruction, Multiple Threads) [8]. Este paradigma combina el clásico paradigma SIMD (Single Instruction Multiple Data) [8] con la ejecución multihilo. Mientras que el número de hilos de ejecución que pueden manejar las CPU actuales se encuentra en el orden de decenas, en el caso de las GPU esta magnitud puede alcanzar las decenas de millar, lo que hace a las GPU mucho más adecuadas que las CPU para la ejecución de aplicaciones altamente paralelizables.

Los imperativos de diseño que afectan a las CPU referentes a latencias bajas en el acceso a memoria y la lógica necesaria para la ejecución especulativa fuera de orden no forman parte de los requerimientos de las GPU, cuyas tareas son tolerantes a mayores latencias de memoria, y pueden ser optimizadas para el procesamiento en paralelo alimentado por un alto ancho de banda de memoria.

Los principales fabricantes de GPU facilitan entornos y lenguajes de programación para explotar las capacidades de estos dispositivos, como son CUDA [9], OpenCL [10] y OpenACC [11]. Estos entornos de programación gozan de una gran aceptación en el ámbito científico por el salto cuantitativo en la escala de los problemas abordables con las GPU en comparación con las CPU.

Hoy en día la complejidad de programación de las GPU, es, en general, similar a la de las CPU. Esto se debe a que, por un lado, hoy en día todas las CPU son multihilo y multinúcleo, por lo que explotar sus capacidades eficientemente requiere de un esfuerzo equivalente al realizado en las GPU. Por otro lado, la programación de estos dispositivos se ha facilitado enormemente gracias a los mencionados entornos y lenguajes de programación.

2.2 Arquitectura de las GPU Fermi

En la actualidad, los principales fabricantes de GPU son Nvidia y AMD. Estos fabricantes mejoran la arquitectura de sus GPU en cada generación. Este trabajo se centra en la arquitectura Fermi de Nvidia [1], tal como la reproduce GPGPU-Sim funcionando como una GPU real basada en esta arquitectura, concretamente la Nvidia Gforce GTX480. Nvidia denomina al paradigma de cómputo subyacente esta arquitectura como *Stream*

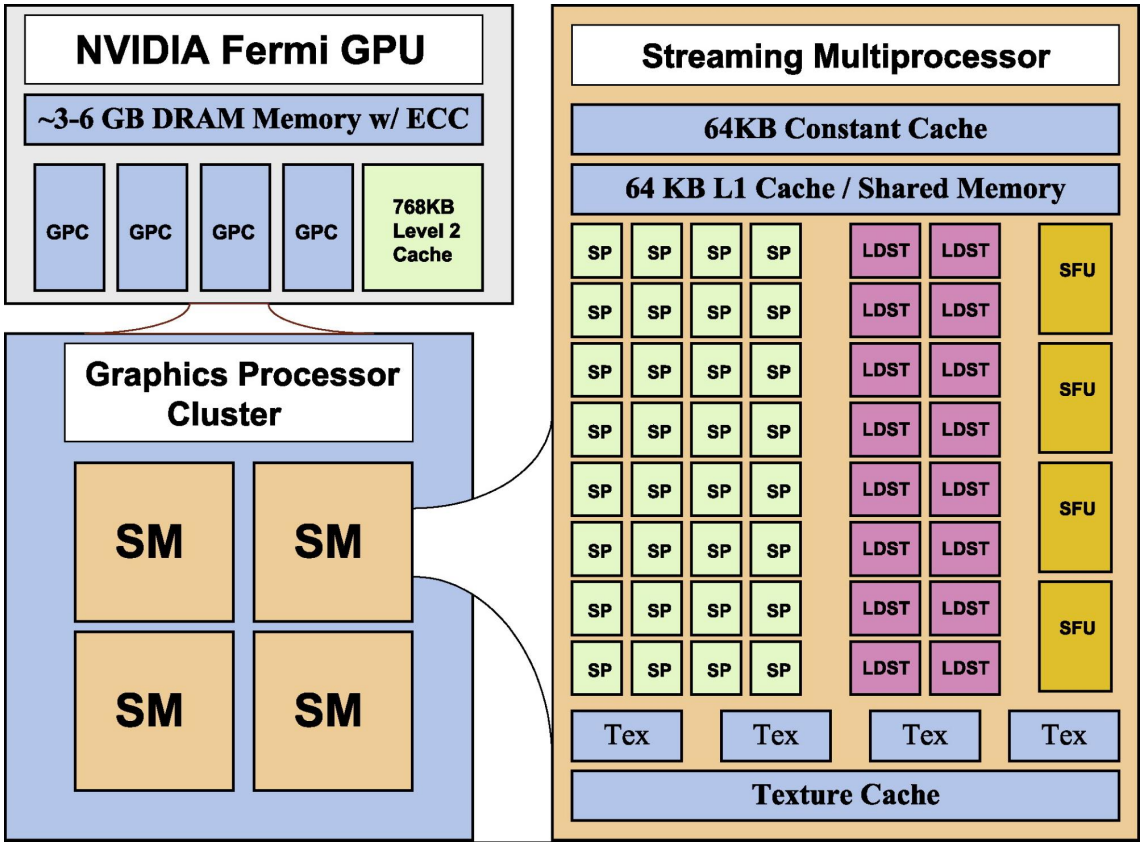

Figura 2.1: Diagrama de una GPU con arquitectura Nvidia Fermi. Fuente: [1]

.

Elemento Software	Componente Hardware
kernel grid (m thread blocks)	GPU (x SM)
thread block (n threads, $n/32$ warps)	SM (32 SP)
thread	SP

Tabla 2.1: Correspondencia entre los elementos software utilizados en la programación en CUDA y los componentes hardware en los que se ejecutan.

Processing [12], el cual es definido como el procesado continuo de nuevos eventos de datos conforme se reciben.

Tal como muestra la figura 2.1, una GPU de la arquitectura Fermi está compuesta de varios *Streaming Multiprocessors* (SM), cada uno de los cuales implementa $4 \times 8 = 32$ *Stream Processors* (SP), también denominados *CUDA cores*.

Los SP son los encargados de realizar los cálculos, y cada uno dispone de sus propias unidades lógicas, de coma flotante, de enteros, comparadores, y de saltos. En un momento dado, cada uno de los 32 SP en un SM procesa la misma instrucción de un hilo diferente. Esta es la razón por la que los hilos de una aplicación GPU se agrupan en conjuntos de 32 hilos denominados *warps*. A su vez, estos warps se agrupan en *thread blocks* en estructuras de una o más dimensiones. El conjunto de todos los thread blocks de una aplicación constituye el llamado *kernel grid*, que la GPU ejecuta. La tabla 2.1 muestra la correspondencia entre los elementos del software a ejecutar (threads, warps, thread blocks y kernel grid) y los componentes del hardware donde se ejecutan (SP, SM y GPU).

La ejecución de los warps en los SM se realiza siguiendo un algoritmo de planificación que determina en cada momento qué warp ejecutará la siguiente instrucción. Si la

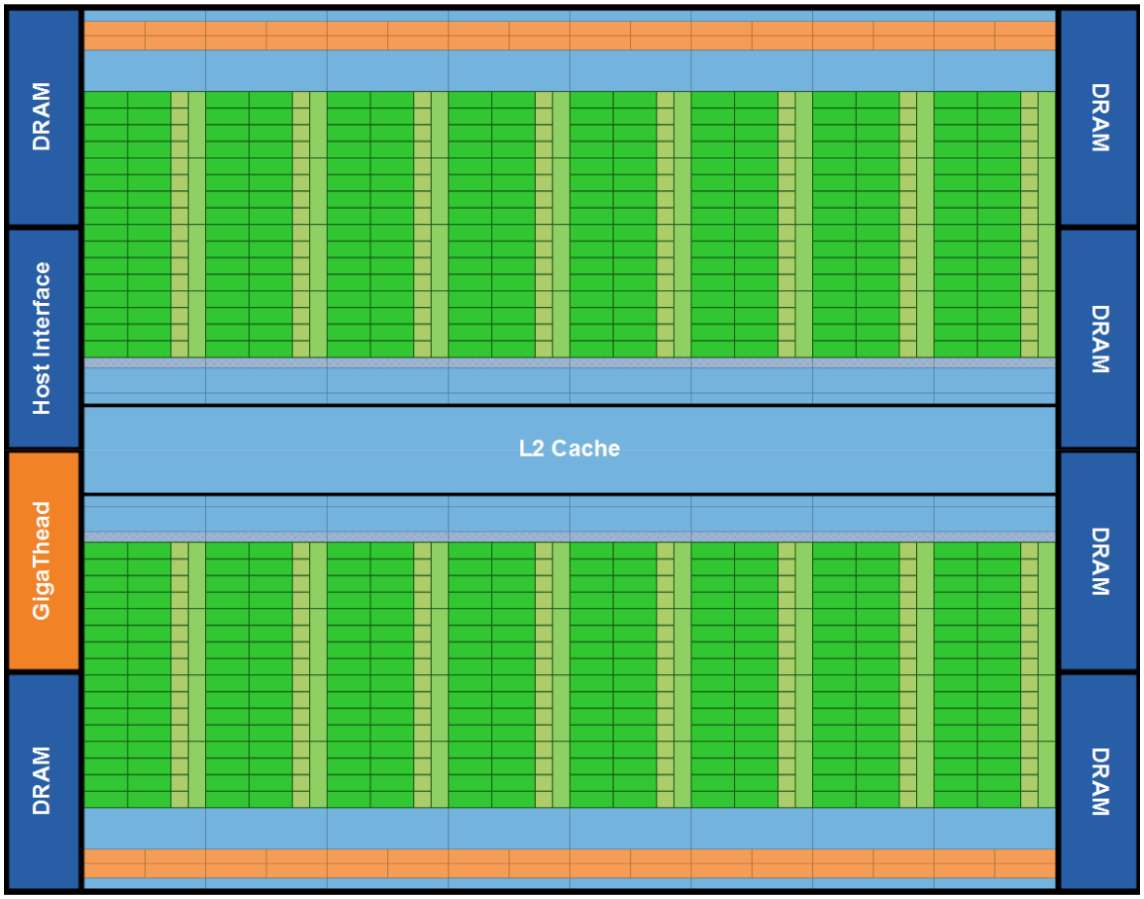

Figura 2.2: Distribución de componentes en la GPU GTX480. Fuente: [1].

siguiente instrucción a ejecutar por un warp determinado no dispone aún de sus operandos (por depender, por ejemplo, de un acceso a memoria), el algoritmo de planificación realiza cambios de contexto para que el SM pase a ejecutar un warp que sí tenga disponible los operandos. Por tanto, para maximizar el rendimiento del sistema, es necesario que el subsistema de memoria proporcione los datos con suficiente ancho de banda para mantener a los SM ocupados continuamente.

2.3 Nvidia GeForce GTX480

La GPU Nvidia GeForce GTX480, en la que se centra este trabajo, está implementada con la arquitectura Fermi. Las principales características de esta GPU [13] se describen a continuación.

La figura 2.2 muestra la distribución de componentes en la GPU GTX480 Fermi. Cada SM viene representado por un rectángulo vertical que contiene una porción naranja (planificador y etapa de *dispatch*), una porción verde (SP), así como una porción azul claro (registros y cache L1). Estas unidades funcionales se distribuyen alrededor de un banco de cache L2 común (en azul claro). Una característica distintiva de cómo implementa la GTX480 esta arquitectura es que uno de los SM está deshabilitado, siendo el número final de SM igual a 15, es decir, $15 \times 32 = 480$ SP.

La tecnología de la memoria principal de esta GPU es GDDR5 [13], funcionando a una frecuencia de reloj de 924MHz. Dado que esta tecnología emplea *Double Data Rate* (DDR), se pueden realizar 2 transferencias en el bus por ciclo de reloj, lo que da un ancho de banda de $924 \times 2 = 1848$ MT/s (Megatransferencias por segundo). La memoria principal

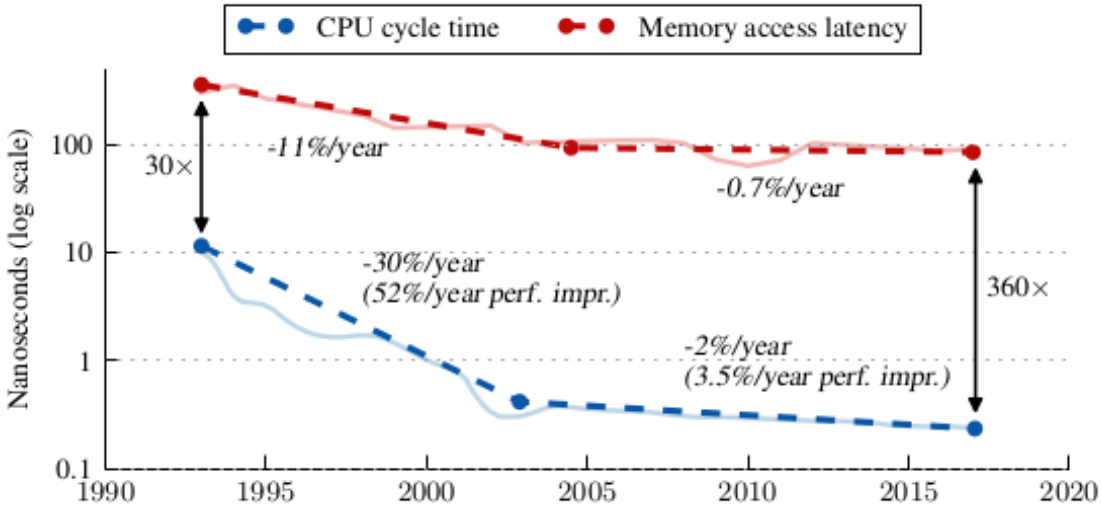

Figura 2.3: Evolución histórica de la latencia de memoria principal con respecto al periodo de reloj de las CPUs. Fuente: [2].

se divide en 6 particiones conectadas a la GPU con un bus de 64 bits (8 bytes) cada una, lo que da un ancho de bus total de 384 bits. Puesto que el ancho de canal en GDDR5 es 32 bits (4 bytes), cada partición de memoria está conectada a un controlador de memoria de doble canal [14].

A cada partición de memoria le corresponden 128 KB de la cache L2 unificada compartida por todos los SM, conformando una capacidad de L2 total de 768 KB. Los detalles sobre la implementación física de esta cache no han sido publicados por Nvidia. No obstante, el estudio realizado mediante *microbenchmarks* en [14] muestra que la L2 de las GPU con arquitectura Fermi se organiza en bancos de 128 KB, cada uno de los cuáles se compone de 2 *slices* o módulos independientes. En el caso de la GTX480 este diseño representa un total de 6 bancos.

2.4 Tecnologías de memoria principal

Aunque los objetivos de diseño en cuanto a latencias, ancho de banda y paralelismo divergen sustancialmente entre los subsistema de memoria de CPU y GPU, su rol es análogo en ambos tipos de procesadores. Actualmente, la memoria principal en ambos tipos de sistemas se implementa mediante tecnología DRAM. Tanto en sistemas CPU como GPU, el aumento de la escala de integración ha dado lugar asimismo al problema denominado *Memory Wall* [2], que consiste en el aumento de la congestión en el acceso a los chips de memoria. Este problema se debe a que la capacidad de cómputo en cada nueva generación de procesadores es mucho mayor que el del ancho de banda de memoria principal, el cual tiene limitaciones físicas propias (por ejemplo, en el número de pines, consumo energético, y disipación térmica). Aunque la constatación de este problema se produjo hace casi tres décadas (el término se acuña en 1995), las distintas mejoras de los subsistemas de memoria no han conseguido solucionarlo. Como se observa en la figura 2.3, la proporción entre el periodo de reloj de la CPU y la latencia en el acceso a memoria no ha hecho más que incrementarse en los últimos 25-30 años, pasando de ser unas 30 a veces superior a unas 360 veces hacia el final de este periodo.

En este contexto, resulta de interés el estudio del impacto y potencial de mejora que pueden tener diversas tecnologías de memoria sobre el rendimiento de las GPU. En este

Característica	DDR2	DDR3	DDR4	GDDR5	GDDR6	HBM	PCM
Burst length	4,8	8	8	8	8,16	2	4,8
Data rate	DDR	DDR	DDR	DDR	DDR,QDR	DDR	DDR
Channel width (bits)	64	64	64	32	32	64	64

Tabla 2.2: Características de las tecnologías de memoria principal estudiadas.

TFG se desarrolla un entorno de simulación que permite estudiar un amplio abanico de tecnologías: DDR2 [15], DDR3 [16], DDR4 [17], GDDR5 [18], GDDR6 [19], HBM [20] y PCM [21]. Para entender las velocidades que cada tecnología puede alcanzar, es necesario entender las características denominadas *channel width*, *data rate* y *burst length*:

- **Channel width**. Es el ancho del canal (porción del bus de memoria) por el que se realizan las transferencias desde/a memoria. Se especifica en bits o bytes.
- **Data rate**. Es la cantidad de transferencias que se pueden realizar por el canal en un ciclo de reloj en el bus. Actualmente suele ser 2 (DDR) o 4 (QDR).
- **Burst length**. Un acceso a memoria representa más de 1 transferencia para leer o escribir un bloque o parte de un bloque de cache. *Burst length* es el número de transferencias soportadas por acceso en cada tecnología.

Los posibles valores para cada una de estas características en las memorias estudiadas se muestran en la tabla 2.2. A continuación, se discute brevemente cada una de las tecnologías presentadas en la tabla.

2.4.1. DDR2

Las memorias DDR (acrónimo de Double Data Rate) son un tipo particular de memoria SDRAM o memoria de acceso aleatorio síncrona y dinámica. Estas memorias se caracterizan principalmente por ser volátiles y tener un funcionamiento interno coordinado mediante una señal de reloj suministrada de forma externa.

Las memorias de la familia DDR consiguen que la frecuencia de transmisión de datos sea el doble de la frecuencia de la señal de reloj a la cual opera la memoria, al transmitir datos tanto en el flanco de subida como de bajada de esta señal de reloj.

La especificación denominada DDR2 es una mejora de la versión anterior DDR. Alcanza alternativamente mayor velocidad o menor latencia, al duplicar las prestaciones que alcanza trabajando a la misma frecuencia que una DDR, al tiempo que tiene menores requerimientos de energía.

2.4.2. DDR3

La memoria DDR3 es a su vez una mejora sobre DDR2, consiguiendo velocidades de transferencia máximas equivalentes al doble de las conseguidas por DDR2, operando a la misma frecuencia de reloj. Esta nueva iteración también mejora la eficiencia energética de la versión anterior. Con respecto a DDR2, las memorias DDR3 consiguen mayor ancho de banda pero no mejoran la latencia, la cual es incluso teóricamente superior por un pequeño margen.

2.4.3. DDR4

La memoria DDR4 es una revisión de DDR3, sobre la cual mejora las características de densidad, requerimientos de voltaje, así como de ancho de banda. De las distintas iteraciones de mejora de la tecnología base DDR, es la primera que no basa su mejora del rendimiento aumentando el factor de *prefetch* [22] o número de lecturas que se realizan previamente a ser solicitadas.

Las mejoras en velocidad surgen principalmente de poder soportar un mayor número de operaciones de lectura/escritura por segundo, mediante una subdivisión de los bancos en grupos accesibles de forma concurrente.

2.4.4. GDDR5

La memoria GDDR5 (Graphics DDR5) es una especificación orientada a los requerimientos particulares de las tarjetas gráficas. Es un estándar basado en la memoria DDR3, optimizando las capacidades relativas al ancho de banda frente a la latencia. Asimismo consigue también menores requerimientos energéticos y consecuentemente también de disipación térmica.

2.4.5. GDDR6

GDDR6 es una revisión de la especificación GDDR5, la cual supera en ancho de banda (hasta 16Gb/s por pin) al tiempo que requiere menores voltajes. Una diferencia importante respecto a GDDR5 es que en la memoria GDDR6 la frecuencia de transmisión de datos no es doble sino el cuádruple que la frecuencia de reloj, por lo que se trata realmente de una memoria QDR (Quad Data Rate).

2.4.6. HBM

La memoria principal de tipo *High Bandwidth Memory* (HBM) es una memoria de altas prestaciones que consigue mayor ancho de banda que DDR4 o GDDR6 con menores requerimientos energéticos y de área. Esto es debido a que es una tecnología de memoria 3D en la que se pueden apilar hasta 4 capas de memoria en un mismo circuito integrado.

El bus de la memoria HBM es mucho más ancho que otras tecnologías DRAM como puedan ser DDR4 o GDDR6. Este amplio número de conexiones complica el acoplamiento entre GPU y HBM, requiriendo en algunos casos el uso de chips intermediarios, que incrementan el coste final de la memoria.

2.4.7. PCM

Phase Change Memory (PCM) es un tipo de memoria no volátil, que aprovecha las propiedades del vidrio calcógeno, el cual puede permanecer estable en los estados cristalino o amorfo, y cuyos cambios pueden ser inducidos por el calor generado por una corriente eléctrica. El estado cristalino tiene poca resistencia eléctrica, mientras que en el estado amorfo es alta. El hecho de ser una memoria no volátil la hace atractiva como competidor de las memorias flash, si bien presenta algunas carencias respecto a estas, principalmente una latencia de acceso varias veces superior, y una menor vida útil debido a la degradación que sufre el sustrato de las celdas tras los sucesivos ciclos de dilatación-contracción. Mientras que las otras tecnologías de memoria expuestas en esta sección disponen de

un estándar industrial (normalmente proporcionado por el *Joint Electron Device Engineering Council* o JEDEC), en este TFG, PCM referencia al modelo de este tipo de memoria implementado en el simulador Ramulator.

2.5 Simuladores

La fase de diseño de cualquier microarquitectura implica la toma de decisiones sobre las características funcionales de los distintos componentes que integran una GPU, y que incluyen no sólo a las unidades cómputo sino también, y a veces con un rol más crítico de cara al rendimiento, las configuraciones de memoria y las redes de conexión.

Los simuladores son una herramienta de diseño que permite poner a prueba, previamente a la implementación real, una determinada propuesta de microarquitectura con las cargas a las que se espera que se vea sometida. Estas cargas suelen consistir en aplicaciones que realizan tareas propias de los distintos ámbitos en las que las GPU son de aplicación. También pueden ser pequeños programas orientados a provocar situaciones de alta demanda o congestión en ciertos componentes de la microarquitectura, para evaluar el impacto de la sobrecarga en las prestaciones globales del sistema.

Los simuladores se dividen en dos subgrupos según su modo de funcionamiento: dirigidos por ejecución (*execution-driven*), y dirigidos por trazas (*trace-driven*). Los simuladores dirigidos por ejecución son capaces de ejecutar las cargas de prueba, mientras que los dirigidos por trazas utilizan como entrada ficheros de trazas que contienen secuencias detalladas del efecto de la ejecución de aplicaciones en el hardware modelado por el simulador. En cualquier caso, la utilidad de un simulador proviene de su capacidad para proporcionar, durante la ejecución simulada de las cargas o el procesado de las trazas, métricas referentes al estado de los componentes en cada ciclo de ejecución, así como información sobre prestaciones y consumos energéticos alcanzados.

En esta sección se presentan varios simuladores de GPU y de memoria principal considerados para la realización de este trabajo.

2.5.1. Simuladores de GPU

GPGPU-Sim

GPGPU-Sim [5] Es un simulador desarrollado en C++ enfocado a la computación de propósito general con GPU, o GPGPU. Incorpora una implementación propia de todos los subsistemas necesarios para poder realizar una simulación precisa a nivel de ciclo de las arquitecturas GT200 y Fermi de Nvidia, la cual ha sido validada con hardware real [4], mostrando una exactitud superior al 97 % durante la ejecución del conjunto de aplicaciones de la suite de *benchmarks* Rodinia [23] [24].

Debido a la mayor exactitud de este simulador en la simulación de la arquitectura Fermi y a que también incorpora un modelo validado de la memoria GDDR5 implementada en la GPU GTX480, este ha sido el simulador seleccionado para el desarrollo de este TFG.

Multi2Sim

Multi2Sim [25] es un simulador de arquitecturas heterogéneas CPU-GPU. Por tanto, no es un simulador específico de GPU. Consiste en un emulador funcional a nivel de aplicación que es capaz de ejecutar binarios diseñados para la arquitectura que simula.

Por su modularidad y flexibilidad, puede simular una amplia variedad de juegos de instrucciones. Incluye soporte para las ISA de CPU x86, MIPS-32 y ARM. Respecto a la GPU modela arquitecturas de AMD Evergreen, AMD Southern Islands, y Nvidia Fermi. Respecto a otros componentes del sistema, como la jerarquía de memoria y redes de interconexión, Multi2Sim incluye modelos propios.

Gem5-GPU

Gem5-GPU [26] es otra solución para la simulación de arquitecturas heterogéneas CPU-GPU. Al contrario que Multi2Sim, es el resultado de la integración de herramientas procedentes de otros desarrollos independientes, cada una de las cuales ha probado su efectividad en la simulación de diferentes partes del sistema. De esta manera, la simulación de CPU se realiza mediante Gem5, mientras que la GPU se simula mediante GPGPU-Sim. Es asimismo un simulador funcional.

GPUTejas

GPUTejas [27] es un simulador desarrollado en Java cuyo diseño está condicionado por estrategias para obtener rendimientos elevados de ejecución, mediante el uso de sincronizaciones tolerantes, estructuras de datos no bloqueantes, así como esquemas de particionado y planificación orientados a la paralelización de la simulación de una GPU. Es capaz de emular las arquitecturas Tesla, Fermi y Kepler de Nvidia.

2.5.2. Simuladores de tecnologías de memoria principal

Los simuladores de memoria principal no son específicos para una tecnología concreta, sino que son parametrizables y permiten simular diferentes tecnologías. Además, tampoco se suelen enfocar exclusivamente a un tipo de sistema (p.e. basado en CPU ó GPU), sino que se centran en modelar con la máxima precisión la memoria principal que se puede implementar en una amplia variedad de sistemas, desde los de altas prestaciones hasta los sistemas móviles y empotrados. Por otro lado, los simuladores de memoria suelen ser dirigidos por trazas, aunque en algunos casos ofrecen la posibilidad de integrarse con simuladores de CPU o GPU como librerías dinámicas, cuya funcionalidad es accesible mediante una API.

En este TFG se utilizan tres simuladores diferentes de memoria principal, lo que permite simular una amplia variedad de tecnologías. Los simuladores se presentan a continuación, incluyendo una breve descripción y justificación de su elección para el presente TFG.

DRAMSim2

DRAMSim2 [28] es un simulador de memoria ciclo a ciclo. Su objetivo es la precisión, portabilidad y reducido tamaño. Los archivos de configuración incluidos en la distribución permiten modelar sistemas de memoria DDR2/3 así como STT-MRAM, y admite dos modos de utilización: i) como ejecutable alimentado por trazas o ii) como librería dinámica.

Este simulador es el más veterano de los tres seleccionados. Propuestas posteriores han ido relegando la funcionalidad de este simulador a un segundo plano. No obstante, debido a su mayor madurez y precisión, y a que su diseño facilita la integración con otros

simuladores, es un simulador atractivo para una primera aproximación a los objetivos de este trabajo.

DRAMSim3

DRAMSim3 [29] es una revisión más reciente de DRAMSim2, que incluye soporte para la simulación del comportamiento energético y termal de distintas tecnologías. DRAMSim3 incluye ficheros de configuración para simular más de 80 configuraciones de memoria principal, incluyendo las tecnologías DDR3, DDR4, GDDR5, GDDR5X, GDDR6, HBM, HBM2, HMC, HMC2, LPDDR3, LPDDR4 y STT-MRAM. DRAMSim3 aporta sobre DRAMSim2 mejoras de rendimiento así como de flexibilidad a la hora de modelar distintas tecnologías de memoria, por lo que también ha sido considerado un simulador adecuado para su utilización en este TFG.

Ramulator

Ramulator [30] se define como un simulador rápido y extensible. Permite el modelado de gran variedad de tecnologías DRAM incluyendo ALDRAM, DDR3, DDR4, DSARP, GDDR5, HBM, LPDDR3, LPDDR4, PCM, SALP, STTMRAM, TLDRAM, WideIO y WideIO2.

Por otra parte, la configuración de las distintas memorias por medio de ficheros de configuración es muy limitada, y la mayoría de los parámetros están codificados en los correspondientes ficheros con el código fuente en C++ que modelan las distintas tecnologías. Esto hace que sea más complicada su configuración respecto a los otros simuladores mencionados, los cuales se configuran exclusivamente mediante ficheros que son leídos en tiempo de ejecución, sin necesidad de escribir nuevo código y recompilar.

Además de los simuladores DRAMSim2 y DRAMSim3, Ramulator ha sido seleccionado por su velocidad y su extensibilidad, así como por incluir preconfiguraciones para simular diversas tecnologías de memoria de uso industrial y también de propuestas académicas.

2.6 TFG relacionados

A continuación se mencionan otros trabajos cuyo objeto de estudio esta estrechamente relacionado con el que da lugar a este TFG.

- Checa Muelas, A., *Evaluación de aplicaciones en una GPU Nvidia Titan X y modelado de la GPU*, 2018, [31].

  Este trabajo modela un determinado hardware de computación GPU usando el simulador GPGPU-Sim, y evalua su rendimiento bajo un subconjunto de la suite Rodinia. Nuestro TFG extiende la funcionalidad del simulador GPGPU-Sim con el uso de simuladores externos para memorias avanzadas tanto para simular un hardware concreto como algunas variaciones del mismo, centrando su atención en el comportamiento del subsistema de memoria.

- Baselga Masiá, D., *Caracterización de aplicaciones GPU y estudio de interferencias en una GPU Nvidia GeForce Titan X*, 2017, [32].

  Se trata de un trabajo centrado en la caracterización de aplicaciones GPU en un hardware concreto. Este estudio no hace uso de simuladores pero si de aplicacio-

nes optimizadas para arquitecturas de procesamiento paralelo, al igual que nuestro TFG.

- Avargues Gutiérrez, MA., *Análisis de requerimientos y diseño de un controlador de memoria principal no volátil*, 2021, [33].

  Como puntos de similitud, este trabajo hace uso de simuladores para realizar estudios de sistemas de memoria no volátil. Por otra parte, los simuladores elegidos son distintos a los utilizados en este TFG, y las cargas son relativas a los ámbitos de almacenamiento y recuperación de datos en lugar de las correspondientes a la computación paralela, como las usadas para la realización de nuestro TFG.

- Tárrega Sánchez, H., *Diseño de Caches L1 utilizando la tecnología emergente Domain Wall Memory*, 2020, [34].

  Este trabajo hace uso de simuladores para el estudio de componentes de la memoria, y del mismo modo recurre a distintos *benchmarks* para analizar su comportamiento. En este trabajo se hace uso de un simulador distinto al utilizado en el presente TFG, y se centra en el primer nivel de la jerarquía de memoria, la caché L1, analizando el rendimiento de aplicaciones optimizadas para cómputo escalar.

Ninguno de los TFG discutidos se centra en el desarrollo de un simulador detallado para GPU con capacidad de modelar múltiples tecnologías de memoria principal. Nuestro TFG trata de aportar un nuevo recurso en este campo de cara a facilitar nuevas posibilidades en el diseño de sistemas de computación de propósito general con GPU.

CAPÍTULO 3
Propuesta

En este capítulo se plantea el diseño inicial del desarrollo realizado en este TFG, explicando brevemente las funcionalidades ofrecidas por los simuladores de memoria principal y los requisitos de la integración con GPGPU-Sim, para finalmente decidir las partes del subsistema de memoria de GPGU-Sim que es necesario modificar. Posteriormente, se detallan las modificaciones realizadas, incluyendo: i) nuevas opciones de configuración, ii) interceptación de los accesos a memoria principal, iii) soporte a múltiples transacciones con memoria por acceso, iv) gestión de los datos contenidos en memoria, y v) otras particularidades de la implementación.

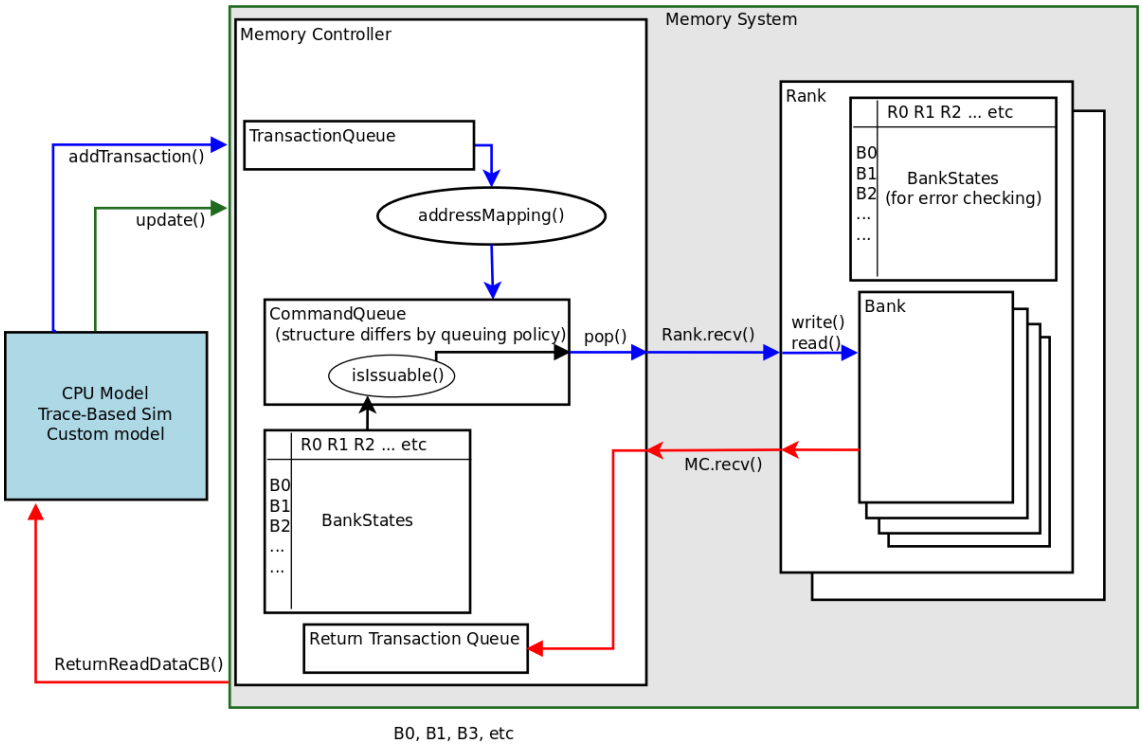

Figura 3.1: Jerarquía de clases del simulador DRAMSim2. Fuente: [3].

3.1 Diseño

Este trabajo consiste en la integración de varios simuladores de memoria (DRAMSim2, DRAMSim3, y Ramulator) en la versión 3.2.2. del simulador GPGPU-Sim, la cual modela únicamente memorias DRAM GDDR5. Por otra parte, el subsistema de memoria de este simulador solamente incluye un planificador básico basado en una política Round-Robin, adecuado para la simulación de la tarjeta GTX480 de Nvidia, pero limitado en contraste con las opciones que puede proporcionar simuladores específicos para tecnologías de memorias.

Aunque el código fuente de GPGPU-Sim es de libre acceso y modificación, no está diseñado para ser integrado con simuladores de memoria externos. Por tanto, se requiere la inspección del código fuente para determinar qué estrategia es la más adecuada para llevarlo a cabo.

Como primer objetivo de integración, se considera el simulador DRAMSim2, el cual incorpora de forma nativa una opción de compilación que permite generar una librería dinámica que expone la funcionalidad mediante una jerarquía de clases y unos métodos públicos, tal como representa el diagrama en la Figura 3.1.

En esta figura podemos ver que es posible interactuar con el sistema DRAMSim2 principalmente mediante dos llamadas: *AddTransaction()* la cual se utiliza para realizar un nuevo acceso de lectura/escritura en memoria, y *update()*, que informa al subsistema de que ha pasado un nuevo ciclo de reloj. La función *ReturnReadDataCB()* representa una función *callback*, que es el método mediante el cual el simulador de memoria puede informar que el acceso a memoria ha finalizado. La compilación de DRAMSim2 como librería dinámica permite acceder a todas estas funcionalidades por medio de la clase *MemorySystem*, la cual engloba el resto de componentes del sistema.

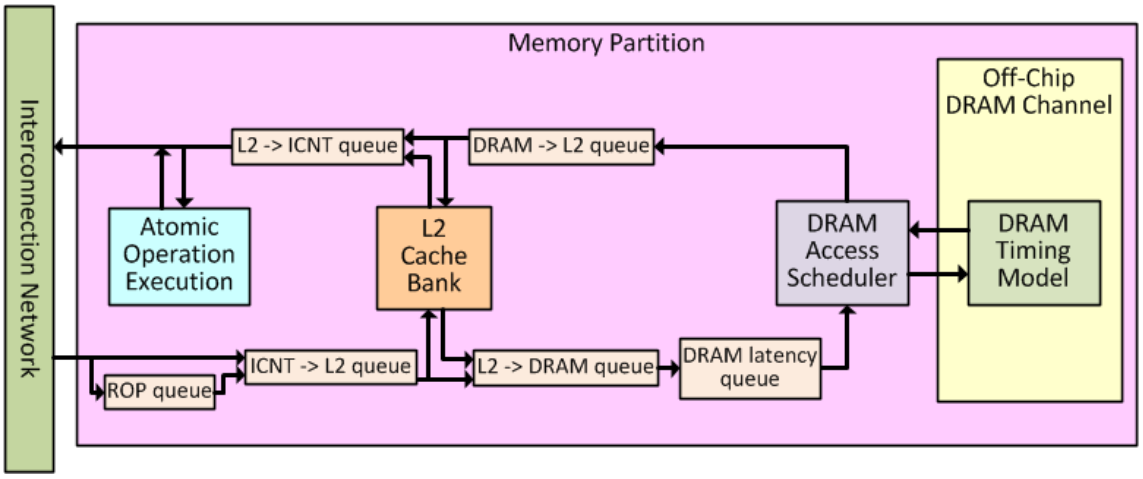

Figura 3.2: Subsistema de memoria (cache L2, planificador de acceso y memoria principal) en el simulador GPGPU-Sim. Fuente: [4].

El resto de simuladores tienen su propia implementación interna y una jerarquía de clases totalmente distinta, pero presentan una interfaz análoga a la DRAMSim2. Los simuladores proporcionan además otras llamadas requeridas por GPGPU-Sim, como por ejemplo aquella que consulta sobre la disposición del controlador de memoria a aceptar un nuevo acceso, requisito previo que es necesario comprobar antes de iniciarlo.

Teniendo en cuenta el diagrama de la estructura lógica de una partición de memoria en el simulador GPGPU-Sim (ver figura 3.2), se observa que, en principio, se presentan varias opciones en cuanto a los posibles lugares donde reconducir a DRAMSim2 los accesos a memoria principal (DRAM). La opción que parece más inmediata es hacerlo en la entrada y salida hacia la DRAM desde el controlador de memoria, denominado planificador (*DRAM Access Scheduler*) en GPGPU-Sim. Sin embargo, se optó por otra alternativa, en base a las consideraciones que se exponen a continuación.

En la figura se observan cuatro colas o *buffers* conectadas al banco de L2 correspondiente de la partición de memoria (ver sección 2.3). Estos buffers son FIFO y comunican la cache L2 con el planificador y con las caches L1 a través de la red de interconexión. Los buffers permiten comunicar áreas trabajando a diferentes frecuencias de reloj (dominios de temporización), tal como ocurre en la GTX480, donde la red de interconexión y la DRAM funcionan a diferente frecuencia que el resto de la GPU. Por lo tanto, la entrada y salida de los buffers funciona a diferentes frecuencias. Asimismo, existe una quinta cola, denominada *DRAM latency queue*, que no representa ningún dispositivo real, sino que modela un retardo medido en la GPU por los desarrolladores de GPGPU-Sim [4].

Atendiendo al mencionado diseño, en este TFG, se ha estimado conveniente capturar los accesos a memoria principal tras la cola *DRAM latency* y justo antes de que sean procesados por el controlador de memoria (*DRAM Access Scheduler*). Esto se ha hecho así por dos razones. En primer lugar, interceptar los accesos a memoria en este punto permite utilizar los controladores de memoria más avanzados disponibles en los simuladores de memoria principal considerados. En segundo lugar, en este punto ya se tiene en cuenta el retardo modelado por la *DRAM latency queue*, el cual no se incluye en esos simuladores. La figura 3.3 muestra dónde se integran los simuladores Dramsim2, Dramsim3 y Ramulator en el subsistema de memoria de GPGPU-Sim. Nótese que las respuestas a los accesos a memoria se envían a la cola *DRAM→L2 queue*.

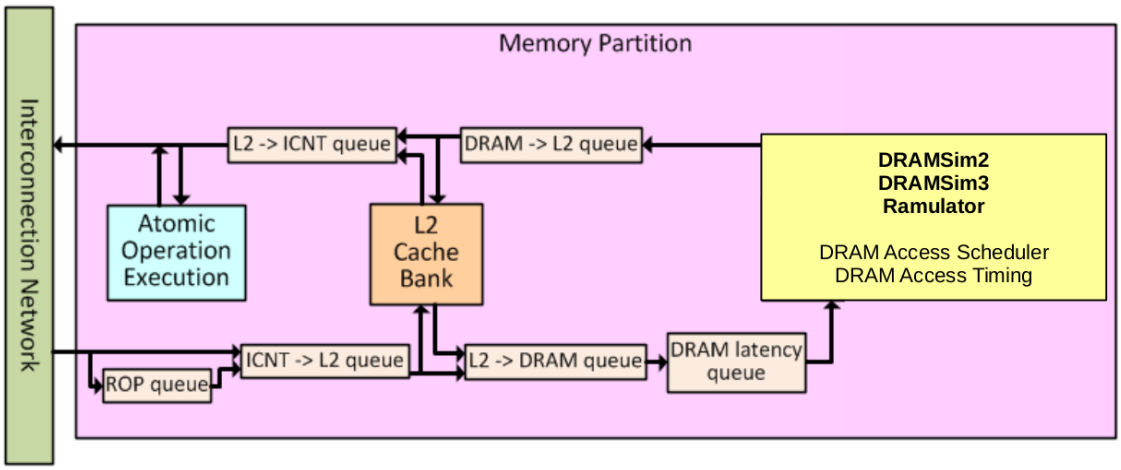

Figura 3.3: Integración de los simuladores considerados en el subsistema de memoria (cache L2 y memoria principal) de GPGPU-Sim. Figura modificada. Fuente: [4].

3.2 Implementación

3.2.1. Nuevas opciones de configuración

Anticipando la necesidad de poder utilizar tanto el subsistema de memoria principal nativo de GPGPU-Sim, como los simuladores de memoria considerados, se han implementado nuevas opciones de configuración en GPGPU-Sim. Estas opciones permiten seleccionar el simulador de memoria y su configuración, la cuál depende a su vez del simulador de memoria finalmente seleccionado. Las nuevas opciones son:

- **dram_simulator** (0 nativo, 1 DRAMSim2, 2 Ramulator, 3 DRAMSim3).
- **dramsim2_controller_ini**: Ruta al archivo de configuración del controlador de memoria utilizado por DRAMSim2.
- **dramsim2_dram_ini**: Ruta al archivo de configuración que modela la DRAM en DRAMSim2.
- **dramsim2_vis_file**: Ruta al fichero de salida de DRAMSim2.
- **dramsim2_total_memory_megs**: Tamaño en MB de la memoria en DRAMSim2.
- **ramulator_config_file**: Fichero de configuración de Ramulator.
- **dramsim3_dram_ini**: Ruta al archivo de configuración para el simulador DRAMSim3.
- **dramsim3_output_dir**: Directorio donde DRAMSim3 escribe sus ficheros de salida. Esta opción se puede omitir, en cuyo caso será el mismo directorio donde se encuentra el ejecutable de la aplicación que hace uso del simulador.

Dichas opciones se han incorporado a los ficheros *gpu-sim.h* y *gpu-sim.cc* de GPGPU-Sim, de forma que puedan ser establecidas como cualquier otra de las opciones reconocidas por el simulador, y definidas durante la inicialización.

Código original:

```
m_dram = new dram_t(m_id,m_config,m_stats,this);
```

Código modificado:

```
if (m_config->dram_simulator==0){ //SIMULADOR = NATIVO
    m_dram = new dram_t(m_id,m_config,m_stats,this);
    printf("\nUsando simulador nativo de GPGPU-Sim\n");}
else if (m_config->dram_simulator==1){ //SIMULADOR == DRAMSIM2
    m_dram = new dram_ds2_t(m_id,m_config,m_stats,this);
    printf("\nUsando Dramsim2\n");}
else if (m_config->dram_simulator==2){ //SIMULADOR == RAMULATOR
    m_dram = new dram_ramulator_t(m_id,m_config,m_stats,this);
    printf("\nUsando Ramulator\n");}
else if (m_config->dram_simulator==3){ //SIMULADOR == DRAMSIM3
    m_dram = new dram_ds3_t(m_id,m_config,m_stats,this);
    printf("\nUsando Dramsim3\n");}
else exit(1);
```

Figura 3.4: Modificación del código para reemplazar la instanciación de la clase *dram_t*.

3.2.2. Interceptación de los accesos a memoria principal

Una vez implementadas las opciones de configuración, siguiendo lo discutido en la sección 3.1, se ha establecido el punto del código desde donde se envían los accesos al simulador de memoria principal y se reciben las respuestas de este último. Estos accesos se modelan en la clase *mem_fetch*, la cual se implementa en los ficheros *mem_fetch.cc* y *mem_fetch.h*. Por otro lado, los componentes del subsistema de memoria de GPGPU-Sim afectados por las modificaciones realizadas en este TFG son el controlador de memoria, implementado en los ficheros *dram_sched.cc* y *dram_sched.h*, y los dispositivos DRAM, implementados en *dram.cc* y *dram.h*.

Los simuladores de memoria principal considerados sustituyen a ambos componentes. En principio, esto dificulta la implementación, ya que implica que las llamadas a ambos componentes sean sustituidas por llamadas al simulador correspondiente. Este problema se ha evitado utilizando una funcionalidad existente en GPGPU-Sim (disponible mediante una opción de configuración), que permite que los accesos a memoria se realicen internamente mediante una cola FIFO. Activando esta opción se simplifica la integración de otros simuladores, ya que solo es necesario reemplazar el acceso a esa cola FIFO por el acceso al simulador de memoria.

La clase *dram_t* implementada en los ficheros *dram.h* y *dram.cc* de GPGPU-Sim, sirve tanto para emular el modelo de temporización de la DRAM, como para servir de interfaz con otras partes del simulador. Para integrar los simuladores de memoria, se han definido tres nuevas clases, *dram_ds2_t*, *dram_ds3_t* y *dram_ramulator_t*, las cuales heredan de *dram_t*, de tal manera que pueden instanciarse en lugar de ella en cualquier contexto. Estas clases sobrecargan *dram_t* de manera que utilizan las funcionalidades de los simuladores de memoria correspondientes en lugar del modelo de la DRAM disponible en *dram_t*. De este modo, se ha sustituido la instanciación de *dram_t* en el código del fichero *l2cache.cc* por una de las tres clases mencionadas dependiendo del simulador de memoria que se elige en la configuración, tal como se muestra en la figura 3.4.

La figura 3.5 muestra un ejemplo de cómo una de las clases mencionadas (*dram_ds3_t*) reemplaza la funcionalidad disponible en *dram_t*. En este caso, se muestra el código que sobrecarga el método *push*. Este método realiza un acceso a memoria (*mem_fetch*) en la memoria principal. Puesto que el método *push* en *dram_t* se corresponde con el método

```
1    void dram_ds3_t::push(class mem_fetch *data){
2        uint64_t mf = reinterpret_cast<uint64_t>(data);
3        objDramSim3->AddTransaction(data->get_addr(),data->get_is_write(), mf);
```

Figura 3.5: Sobrecarga del método *push* de *dram_t*.

AddTransaction() de DRAMSim3, la sobrecarga que implementa la clase *dram_ds_t* llama a este método con la información – tipo de acceso (lectura o escritura) y dirección – proveniente del objeto *mem_fetch*. Nótese que la clase *dram_ds3_t* contiene un objeto *objDramSim3*, cuyo método *AddTransaction()* es accesible al utilizar DRAMSim3 como una librería dinámica. La estrategia mostrada en el ejemplo es similar en todos los simuladores considerados, variando en cada caso la llamada específica realizada a cada simulador.

3.2.3. Múltiples transacciones con memoria principal por acceso

En un acceso a memoria principal, la unidad de datos que se accede es un bloque de cache. En general, un bloque de cache tiene un tamaño de 64B o 128B el cual, dependiendo de la tecnología de memoria utilizada, puede ser menor que la cantidad de bytes accedidos en una sola transacción con memoria principal. En ese caso, es necesario realizar múltiples transacciones para acceder (leer o escribir) a un bloque de cache.

Al incluir en GPGPU-Sim el soporte a varios simuladores de memoria principal que modelan diferentes tecnologías, ha sido necesario dar soporte a múltiples transacciones por acceso. Este soporte implica: i) añadir a GPGPU-Sim el código para realizar múltiples transacciones, y ii) añadir a los simuladores de memoria la funcionalidad que permite indicar que son capaces de encolar múltiples transacciones correspondientes a un acceso.

Las figuras 3.6 y 3.7 muestran el código de GPGPU-Sim involucrado en la generación y finalización, respectivamente, de un acceso compuesto por múltiples transacciones. Usando una memoria GDDR5, la cual tiene ancho de canal de 32 bits (4 bytes), y un *burst length* de 8, son necesarias cuatro transacciones para recuperar los 128 bytes que se almacenan en un bloque de la cache L2 en la GTX480. En el caso de usar una memoria DDR2, DDR3, o DDR4, puesto que tienen un ancho de canal de 8 bytes, el número de transacciones se reduce a la mitad. En la sección 5.2 se discute cuantas transacciones requiere por acceso cada una de las tecnologías estudiadas. Por otro lado, también es necesario ajustar el desplazamiento para que las transacciones accedan a direcciones de memoria consecutivas.

Para no dar por finalizada una operación hasta que hayan terminado todas las transacciones que generó, se incorpora un contador al *mem_fetch*, el cual sirve tanto para lanzar el número de operaciones necesarias, como para darlas por finalizadas. Este contador ha de tenerse en cuenta al recibir cada confirmación de transacción realizada por parte del simulador de memoria, ya que sólo tras completarse con éxito todas las transacciones que se generaron a partir de un único *mem_fetch*, puede devolverse el control del mismo a GPGPU-Sim. Es por ello que tras cada confirmación se decrementa el contador asociado, dándose por completado el acceso tras llegar el contador a 0.

3.2.4. Gestión de los datos contenidos en memoria

Los simuladores de memoria principal utilizados en este TFG sólo modelan el *timing* de los accesos, pero no realizan ninguna gestión con los contenidos almacenados en memoria. Es decir, un acceso a memoria se modela únicamente con la dirección solicitada y el tipo de acceso (lectura o escritura). Tras aceptar un acceso, el simulador reproduce

```
1  void dram_ds3_t::push( class mem_fetch *data ){
2  int m=4;     //multiplicador de operaciones (para GDDR5)
3  int s=32;    //incremento sobre la direccion base del mf.
4  data->set_status(IN_PARTITION_MC_INTERFACE_QUEUE,gpu_sim_cycle+
       gpu_tot_sim_cycle);
5
6  data->set_cont(m); //este mem_fetch ha generado 4 operaciones = 4 Requests
7  uint64_t mf = reinterpret_cast<uint64_t>(data);
8  //mandamos un lote de requests:
9  for (int j=0;j < m;j++){ //tienen el mismo mf, pero van a direcciones
       contiguas
10   bool b=objDramSim3->AddTransaction(data->get_addr()+s*j, data->
         get_is_write() , mf);
11 }
```

Figura 3.6: Generación de múltiples transacciones para acceder a un bloque de cache.

```
1  void dram_ds3_t::read_complete(uint64_t mf_return){
2  mem_fetch *data=reinterpret_cast<mem_fetch*>(mf_return);
3
4  if (data->dec_cont()==0){ //ya ha vuelto el ultimo Request de todos los que
       se generaron:
5    data->set_status(IN_PARTITION_MC_RETURNQ,gpu_sim_cycle+gpu_tot_sim_cycle)
         ;
6    if( data->get_access_type() != L1_WRBK_ACC && data->get_access_type() !=
       L2_WRBK_ACC ) {
7       data->set_reply();
8       returnq->push(data);//Entrega el control del mem_fetch a GPGPU-Sim
9     ..
10    ..
11    }
12 }
```

Figura 3.7: Recepción y recuento de las múltiples transacciones que generó un acceso.

iterativamente, en base a una señal de reloj, la secuencia de pasos que el hardware real realiza para llevarlo a cabo. Cuando el acceso finaliza, se genera un evento en GPGPU-Sim para que pueda continuar las operaciones dependientes del acceso. Sin embargo, GPGPU-Sim no sólo modela el *timing* de la arquitectura, sino que gestiona la información almacenada en la jerarquía de memoria, tal como lo hace el hardware real.

La gestión de los datos en GPGPU-Sim se implementa mediante la clase *mem_fetch*, la cual incorpora todos los métodos y propiedades que el simulador necesita internamente para este fin. Cada acceso a memoria genera una instancia de esta clase, que contiene toda la información asociada a ella, incluyendo el dato que se lee o se escribe, pero también otra información adicional. Por ejemplo, la instancia indica en qué nivel de la jerarquía de memoria (buffers, red, caches, memoria principal, etc.) se encuentra el acceso.

Para mantener esta gestión de los datos pese a que los simuladores de memoria no la soportan, se ha realizado una modificación mínima en estos últimos. Esta modificación consiste en el añadido de un nuevo atributo a las clases equivalentes a *mem_fetch* que modelan los accesos a memoria en los simuladores de memoria considerados. Este atributo es un puntero a la instancia de *mem_fetch* correspondiente al acceso. Para evitar introducir dependencias innecesarias entre el código de diferentes simuladores, este puntero es de tipo *uint64*. Durante la ejecución, cuando es necesario, se recurre al mecanismo de *casting* de C++ para pasar de este tipo de datos al tipo puntero a *mem_fetch* y viceversa.

```
1  void dram_ds3_t::read_complete(uint64_t mf_return){
2      mem_fetch *mf=reinterpret_cast<mem_fetch*>(mf_return);
3      //..
4      //(Continua codigo GPGPU-Sim y las operaciones sobre el objeto mf)
5      //..
```

Figura 3.8: Sobrecarga del método *read_complete* de *dram_t*.

Se ilustra ese proceso en la figura 3.5 mediante el ejemplo de la integración con DRAMSim3, siendo similar para el resto de simuladores. Además, de incluir la dirección y el tipo de acceso, la llamada al método *AddTransaction()* tambien incluye el *cast* a la instancia de *mem_fetch*. Este *cast* forma parte del objeto *Transaction* de DRAMSim3, sin interferir en su funcionamiento. Tras finalizar el acceso a memoria, el simulador de memoria invoca a GPGPU-Sim mediante una función *callback* (*read_complete*), a la cual le pasa el *uint64* que contiene el puntero al *mem_fetch* asociado al acceso, haciendo posible reasignarlo de nuevo, como muestra la figura 3.8.

3.2.5. Particularidades en la integración de los distintos simuladores

Con una metodología similar a la mostrada se sobrecargaron el resto de métodos necesarios para que todos los simuladores externos, encapsulados en una clase compatible con GPGPU-Sim, pudieran ajustarse a la interfaz definida por la clase *dram_t*, permitiendo su sustitución efectiva.

En una primera aproximación, se depuró y comprobó la funcionalidad de la integración con DRAMSim2 por estar más documentada su funcionalidad como librería dinámica. Posteriormente y gracias a la experiencia obtenida tras integrarlo con éxito en GPGPU-Sim, se procedió a una integración con Ramulator y DRAMSim3 de una forma similar, si bien estos últimos presentaron dos inconvenientes.

El primero de ellos consiste en que Ramulator no ofrece la posibilidad de funcionar como una librería dinámica. Si bien soporta la integración con el simulador Gem5, esta se consigue por medio de la aplicación de un parche sobre el código fuente durante la compilación. Esta circunstancia se solventó mediante la alteración del proceso de compilación de Ramulator para que genere una librería dinámica que pueda integrarse tal como DRAMSim2.

El segundo inconveniente se debe a que tanto Ramulator como DRAMSim3, trabajan con dos colas de acceso a memoria, una para lecturas y otras para escrituras, mientras que GPGPU-Sim y DRAMSim2 funcionan con una única cola. Así, en estos dos últimos simuladores, cuando es necesario encolar un acceso a memoria (tipo *mem_fetch*), basta con comprobar mediante el método *dram_t::full()* (sin argumentos) si hay espacio en la cola, mientras que en los dos primeros es necesario averiguar de qué tipo es el acceso para encolarlo en la cola correspondiente.

Para solucionar este problema, se añadió el método *dram_t::full(*mem_fetch)*, el cual comprueba la cola correspondiente al tipo de acceso en los simuladores Ramulator y DRAMSim3. Así, *dram_t::full()* se dedica a comprobar la única cola en GPGPU-Sim y DRAMSim2, mientras que *dram_t::full(*mem_fetch)* comprueba la cola correspondiente (de lectura o de escritura) en Ramulator y DRAMSim3.

Si cualquiera de estos métodos se utiliza en un simulador que no le corresponde, el método devuelve siempre *false*. Este funcionamiento minimiza las modificaciones a realizar en el código del subsistema de memoria de GPGPU-Sim, tal como muestra la figura 3.9, donde se observa que solo es necesario añadir una línea al código original. La línea

Código original:

```
1  if( !m_dram->full() ) {
2
3      // etapa: L2->DRAM
4      // ..
5      // Codigo de arbitraje entre multiples subparticiones L2
6      // ..
7      mem_fetch *mf = m_sub_partition[spid]->L2_dram_queue_top();
8      m_sub_partition[spid]->L2_dram_queue_pop();
9      MEMPART_DPRINTF("Issue mem_fetch request %p from sub partition %d to dram\n
           ", mf, spid);
10   ...
```

Código modificado:

```
1  if( !m_dram->full() ){ // Comprobacion para GPGPU-Sim y DRAMSim2,
2                         // Ramulator y DRAMSim3 siempre continuan.
3
4      // etapa: L2->DRAM
5      // ..
6      // Codigo de arbitraje entre multiples subparticiones L2
7      // ..
8      mem_fetch *mf = m_sub_partition[spid]->L2_dram_queue_top();
9      if (m_dram->full(mf)) break; // Comprobacion para Ramulator y DRAMSim3,
10                                  // GPGPU-Sim y DRAMSim2 siempre continuan.
11     m_sub_partition[spid]->L2_dram_queue_pop();
12     MEMPART_DPRINTF("Issue mem_fetch request %p from sub partition %d to dram\n
           ", mf, spid);
13   ...
```

Figura 3.9: Modificación en GPGPU-Sim para soportar dos colas (lectura y escritura) en los simuladores de memoria principal.

1 del código modificado comprueba, sólo para los simuladores con una cola, si hay sitio en esta. En los demás simuladores *m_dram->full()* devuelve *false*, permitiendo continuar hasta la comprobación para simuladores con dos colas (*m_dram->full(mf)*), situada en la línea 8.

CAPÍTULO 4
Implantación

En este capítulo se describe y la implantación del entorno de desarrollo de este TFG, incluyendo consideraciones sobre el software a instalar y la explicación paso a paso de la instalación de sistema operativo, compilador, librerías, simuladores y *benchmarks* para la evaluación experimental.

4.1 Consideraciones sobre el entorno de desarrollo

La implantación del entorno experimental realizada en este TFG es compleja, ya que requiere de múltiples componentes software no relacionados que deben interactuar. Cada uno de esos componentes presenta diversos requerimientos en cuanto a sistema operativo, compiladores y librerías necesarias para su correcto funcionamiento.

Si bien la integración de GPGPU-Sim con Ramulator, DRAMSim2 y DRAMSim3 es flexible en cuanto a los requerimientos de sistema operativo y compiladores, el modelado y simulación de la arquitectura Fermi de Nvidia en la versión 3.2.2 de GPGPU-Sim exige que el entorno de desarrollo CUDA [9] de Nvidia no sea superior a la versión 4.2. A su vez, esto implica que el compilador GNU GCC [35] utilizado no tenga una versión igual o superior a la 4.7. Por otro lado, en este TFG se utiliza la suite de *benchmarks* Rodinia 3.1 [6] para evaluar la integración de los simuladores. Las aplicaciones de la suite sólo pueden ser compiladas con determinadas versiones de GCC.

Por todas estas razones, ha sido necesario seleccionar un entorno de desarrollo que fuera compatible con todos los mencionados componentes. El entorno elegido ha sido el sistema operativo Ubuntu 14.04, cuya empresa distribuidora, Canonical, ofrece soporte extendido hasta el año 2024. Ubuntu 14.04 dispone en sus repositorios de la versión 4.6 del compilador GCC, compatible con todos los componentes.

El entorno de desarrollo se ha instalado en una máquina virtual ejecutándose en un PC de escritorio con Sistema Operativo Ubuntu 20.04 y el software de virtualización VirtualBox 6.1. La interacción con el entorno experimental se realizó por medio de una conexión *ssh* desde el S.O. anfitrión, así como una conexión *sftp* para operaciones sobre el sistema de archivos, principalmente la recolección de resultados. A continuación se describe el proceso para replicar el entorno de desarrollo utilizado en este TFG.

4.2 Instalación del sistema operativo, compilador y librerías necesarias

El sistema operativo, compilador y librerías se han instalado siguiendo los siguientes pasos:

1. Instalación en una máquina virtual del S.O. Ubuntu 14.04 64 bit, disponible en
   https://releases.ubuntu.com/14.04/ubuntu-14.04.6-desktop-amd64.iso.

2. Instalación del servidor OpenSSH. Posteriormente a este paso y con la redirección de puertos correspondiente en el software de virtualización, es posible acceder a una terminal desde el equipo anfitrión.

```
1  $ sudo apt install ssh
```

3. Instalación de todos los paquetes necesarios para la posterior compilación de los distintos módulos:

```
1  $ sudo apt install build-essential gcc-4.6 g++-4.6 xutils-dev
       bison flex zlib1g-dev libXi-dev libXmu-dev libglu1-mesa-dev
       freeglut3-dev git clang libtool
```

4. Crear los enlaces necesarios para que la versión por defecto de GCC en el sistema se corresponda con la versión 4.6:

```
$ sudo ln -sf /usr/bin/gcc-4.6 /usr/bin/gcc
$ sudo ln -sf /usr/bin/g++-4.6 /usr/bin/g++
```

5. A continuación descargar el SDK y toolkit CUDA4.2 de los repositorios de Nvidia:

```
$ wget https://developer.download.nvidia.com/compute/cuda/4_2/rel
    /toolkit/cudatoolkit_4.2.9_linux_64_ubuntu11.04.run
$ wget https://developer.download.nvidia.com/compute/cuda/4_2/rel
    /sdk/gpucomputingsdk_4.2.9_linux.run
```

6. Otorgar permisos de ejecución a los instaladores de Nvidia:

```
$ chmod +x cudatoolkit_4.2.9_linux_64_ubuntu11.04.run
$ chmod +x gpucomputingsdk_4.2.9_linux.run
```

7. Instalación de toolkit de CUDA, y establecimiento de las variables de entorno correspondientes, aprovechando para incluir las rutas a las librerías que posteriormente requerirá nuestra versión modificada de GPGPU-Sim:

```
$ sudo ./cudatoolkit_4.2.9_linux_64_ubuntu11.04.run
$ echo 'export PATH=$PATH:/usr/local/cuda/bin' >> .bashrc
$ echo 'export CUDA_INSTALL_PATH=/usr/local/cuda' >> .bashrc
$ echo 'export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/cuda/
    lib64:/usr/local/cuda/lib:$HOME/gpgpu-sim_distribution/src/
    ramulator-master:$HOME/gpgpu-sim_distribution/src/DRAMSim2' >>
     .bashrc
```

8. Tras salir del sistema y volver a acceder, para que dichas variables se carguen en el entorno al inicio, instalamos el SDK, modificando el fichero *Makefile* para que sólo se compilen las librerías compartidas, así como definir una nueva variable de entorno:

```
$ ./gpucomputingsdk_4.2.9_linux.run
$ cd NVIDIA_GPU_Computing_SDK
$ nano Makefile # Comentamos las lineas correspondientes en
    seccion 'all', para evitar la compilacion de los ejemplos y
    sus librerias, y OpenCL:

    all:
    +@\$(MAKE) -C ./shared
    #+@\$(MAKE) -C ./C
    #+@\$(MAKE) -C ./CUDALibraries
    #+@\$(MAKE) -C ./OpenCL

$ make
$ echo 'export NVIDIA_CUDA_SDK_LOCATION=$HOME/
    NVIDIA_GPU_Computing_SDK' >> .bashrc
```

4.3 Instalación de GPGPU-Sim y los simuladores de memoria considerados

1. Tras salir y volver a ingresar en el sistema, ya es posible clonar el repositorio GitHub que contiene la versión modificada de GPGPU-Sim resultado de este TFG, la cual incluye las correspondientes versiones modificadas de DRAMSim2, DRAMSim3 y Ramulator que se integran en él:

```
$ git clone https://github.com/carbaior/tfg gpgpu-
    sim_distribution
```

2. Compilamos en primer lugar DRAMSim2, DRAMSim3 y Ramulator en forma de librerias:

```
$ cd $HOME/gpgpu-sim_distribution/src/DRAMSim2
$ make libdramsim.so
$ cd ../DRAMsim3
$ make libdramsim3
$ cd ../ramulator-master
$ make libramulator.a
$ gcc -shared -fPIC -o libramulator.so ./obj/*
```

3. Por último, pasamos a compilar GPGPU-Sim:

```
$ cd $HOME/gpgpu-sim_distribution
$ source setup_environment
$ make
```

Tras este último paso, la sesión en curso queda configurada para que los binarios que requieran de las librerías de CUDA para ejecutarse, (como los correspondientes a la suite Rodinia), usen en su lugar el simulador GPGPU-Sim, el cual mostrará por la salida estándar todos los informes relativos a su ejecución.

En el caso de querer lanzar nuevas ejecuciones en otras sesiones, previamente es necesario ejecutar la orden `source setup_enviroment` en el directorio *gpgpu-sim_distribution* para que se establezcan las variables de entorno necesarias para interceptar las llamadas a las librerias nativas de CUDA. En un entorno dedicado exclusivamente al uso de GPGPU-Sim, se recomienda la ejecución automática de esta orden al inicio de sesión, incluyendo la linea `source $HOME/gpgpu-sim_distribution/setup_environment` al final del archivo *.bashrc* en el directorio $HOME.

4.4 Instalación de la suite de benchmarks Rodinia.

Los *benchmarks* que forman parte de la suite Rodinia se distribuyen como un paquete que contiene el código fuente de las distintas aplicaciones que lo forman. La modificación y redistribución de los códigos fuente están permitidos sin mayor restricción que el mantenimiento de los mismos términos y condiciones estipulados en el paquete original. El proceso de construcción *build* admite tres objetivos de compilación, los cuales pueden ser CUDA [9], OpenMP [36], u OpenCL [10]. En este trabajo tan sólo se hace uso del objetivo de compilación CUDA, el cual tiene como dependencias el entorno de desarrollo CUDA correspondiente. Sin embargo, si esta compilación se realiza en el mismo equipo en el que instaló GPGPU-Sim, estas dependencias ya está satisfechas y tan sólo es necesario invocar el proceso de compilación y asegurarse de usar un compilador adecuado a las convenciones del código usado en las aplicaciones. En nuestro caso, el proceso de compilación se llevó a cabo usando la versión 4.6 de GCC.

El proceso de descarga e instalación se detalla a continuación:

```
$wget http://www.cs.virginia.edu/~skadron/lava/Rodinia/Packages/
    rodinia_3.1.tar.bz2
$tar xjvf rodinia_3.1.tar.bz2
$cd rodinia_3.1
$make CUDA
```

Durante la compilación, en el directorio *rodinia_3.1/cuda* se crea un directorio para cada uno de los *benchmarks*, y que contiene un *script* llamado *run* para lanzar su ejecución. Dependiendo de las necesidades, es posible modificar los parámetros de ejecución de cada programa mediante la edición de este *script*.

CAPÍTULO 5
Entorno experimental

En este capítulo se describe el entorno experimental para la evaluación de la propuesta. Primero se introduce la suite de *benchmarks* Rodinia, que proporciona las aplicaciones ejecutadas en los experimentos. Después, se discute la configuración de la GPU simulada, haciendo especial énfasis en el subsistema de memoria y las configuración de las tecnologías de memoria. Finalmente se presentan las métricas de prestaciones utilizadas para evaluar los resultados.

5.1 La suite de benchmarks Rodinia

La suite Rodinia [6] recopila *benchmarks* GPGPU diseñados para explotar las posibilidades de las arquitecturas GPU. Estos *benchmarks* pertenecen a distintos ámbitos de la ingeniería y la industria, como el procesamiento de imágenes médicas, bioinformática, dinámica de fluidos, minería de datos, etc.

Debido a que en este TFG se simula detalladamente (a nivel de la microarquitectura) la ejecución de aplicaciones GPGPU en un simulador ejecutándose sobre una CPU, la realización de experimentos puede ser muy lenta en algunos casos. Por ello, en este TFG se ha seleccionado un subconjunto de la suite Rodinia que se puede ejecutar en un tiempo razonable (alrededor de 24 horas), con el objetivo de que sea viable un proceso iterativo de lanzamiento de experimentos, análisis y lanzamiento de nuevos experimentos a partir del análisis.

La selección consiste en 11 aplicaciones de la suite, las cuales se presentan ordenadas en base al criterio de impacto en sus prestaciones por parte del subsistema de memoria. Como se verá en la sección 6.1, este impacto está altamente correlacionado con la tasa de fallos de la cache de L2. A continuación se describen brevemente las aplicaciones seleccionadas:

- **nw** [37]: implementación del algoritmo Needleman-Wunsch, utilizado en el campo de la bioinformática para el alineamiento de secuencias proteínas o de nucleotidos. Puesto que este algoritmo trabaja sobre una matriz, y sobre ella pueden realizarse de forma concurrente varias de las operaciones requeridas, una implementación basada en GPU puede alcanzar un rendimiento hasta 6 veces superior a la implementación secuencial en CPU.

- **dwt** [38]: implementación de la transformada wavelet discreta (DWT) en 2 dimensiones. Consiste en un procesamiento de datos ampliamente utilizado en el campo de la imagen digital para incrementar las redundancias una señal, a menudo con el fin de aumentar la eficacia de posteriores etapas de compresión de datos. La estructura fundamental del algoritmo también son las matrices, las cuales se pueden operar con muy alta eficiencia con el uso de GPU.

- **pathfinder** [39]: Algoritmo de búsqueda de rutas, o el camino más corto entre dos puntos. Es un recurso habitual en la programación de videojuegos de cara a modelar el comportamiento de agentes inteligentes. Los algoritmos de pathfinding están estrechamente relacionados con los de recorridos de grafos, y se pueden optimizar para entornos GPU. Esta eficiencia mejorada es a veces un requisito para la jugabilidad, pues en ocasiones el algoritmo ha de explorar muchos posibles caminos que finalmente resultan no existir, lo que puede agotar los recursos de una CPU convencional.

- **srad_v2** [40]: Se trata de un algoritmo utilizado en procesamiento de imágenes, y cuyo nombre es acrónimo de *Speckle Reducing Anisotropic Diffusion*, o reducción de ruido granular mediante difusión anisotrópica. La difusión anisotrópica es una técnica que permite la eliminación de ruido en la imagen sin difuminación de los bordes. El algoritmo contiene distintas etapas, algunas de las cuales se pueden realizar de forma paralela, pudiendo optimizarse mediante el uso de GPU.

- **hotspot** [41, 42]: Es un algoritmo ampliamente utilizado para estimar la temperatura del procesador a partir de los planos de su arquitectura y las medidas de potencia simuladas. Su funcionamiento resuelve iterativamente una serie ecuaciones

diferenciales por bloque, correspondientes a un área. La aceleración GPU permite concurrencia en el cálculo de la temperatura media de las regiones asociadas a estos bloques. Pertenece al dominio de las simulaciones físicas.

- **lud** [43]: En Álgebra Lineal, la descomposición LU (*Lower-Upper*) es un algoritmo para calcular las soluciones de un conjunto de ecuaciones lineales. Esta descomposición consiste en el cálculo de la equivalencia de una matriz de entrada con la del producto de dos matrices triangulares, una superior y otra inferior.

- **bfs** [44]: Dentro de la familia de algoritmos para grafos, *Breadth-First Search* (BFS) se utiliza para recorrer o buscar los elementos en un grafo. La implementación en GPU lanza un hilo de ejecución para cada vértice inicial a partir del cual se aplica el algoritmo, consiguiendo una elevada paralelización.

- **myocite** [45]: Pertenece al dominio de la simulación biológica. El software *myocyte* modela una célula cardíaca o miocito y simula su comportamiento, según la caracterización propuesta por investigadores de la Universidad de Virginia.

- **kmeans** [46]: Es un algoritmo de *clustering* o agrupamiento, extensamente utilizado en el ámbito de minería de datos. Se utiliza en tareas de agrupamiento para obterner información acerca de la estructura del conjunto de datos, agrupando los puntos de datos en distintos subgrupos sin producir solapamientos.

- **lavaMD** [47]: Pertenece al dominio la dinámica molecular. Este código calcula el potencial y reubicación de partículas en un espacio tridimensional debido a las fuerzas que surgen por su interacción. Este espacio se divide en cubos que se pueden asignar concurrentemente a distintos nodos del cluster de computación. En base a observaciones sobre la velocidad obtenida frente a una implementación iterativa en una CPU monohilo, la aceleración alcanzada es del orden de 16 veces superior.

- **gaussian** [48]: La Eliminación Gausianna se aplica en el ámbito del Álgebra Lineal para resolver sistemas de ecuaciones lineales. La implementación en GPU calcula el resultado fila por fila para todas las variables del sistema de ecuaciones. Las iteraciones deben sincronizarse, pero los cálculos en cada iteración puede ser realizados en paralelo, acelerando la ejecución.

5.2 Configuración de los experimentos

Para realizar los experimentos, se ha modelado la GPU GTX480 con diversas tecnologías de memoria. La tabla 5.1 presenta la configuración base de la GPU GTX480, exceptuando la configuración de la memoria principal.

Como se indica en la tabla, el subsistema de memoria de la tarjeta dispone de 6 particiones de memoria principal, cada una de las cuales se conecta a dos módulos de cache L2 a través de un bus de memoria de 8 bytes. A su vez, ese bus de memoria se distribuye en varios canales. La tabla 5.2 muestra la configuración de cada partición para todas las tecnologías de memoria principal estudiadas, incluyendo el ancho de canal en cada tecnología. Por ejemplo, el bus de memoria de 8 bytes en una partición se distribuye en 2 canales de 4 bytes con la tecnología GDDR5, mientras que con las memorias DDR2, DDR3 y DDR4 solo hay 1 canal de 8 bytes por partición.

La tabla también muestra el número de transacciones con memoria principal que deben realizarse con cada tecnología para acceder a un bloque de cache (128B), valor que depende del ancho de canal de memoria (4 u 8 bytes) y del número de transferencias

Parámetro	Valor
f_{cpu}	700MHz
Streaming Multiprocessors (SM)	15
Planificadores de *warp* por SM	2
Núcleos CUDA por SM	32
Special Function Units (SFM) por SM	4
Registros por SM	32K
Unidades de carga/almacenamiento por SM	16
Tamaño de bloque de cache	128B
Cache L1	16KB, asociativa de 4 vías
Cache L2	64KB por módulo, asociativa de 8 vías
Particiones de memoria principal	6
Módulos L2 por partición de memoria principal	2
Ancho de bus de memoria por partición	8 bytes

Tabla 5.1: Configuración de la GPU GTX480 simulada.

Tecnología	f_{bus} (MHz)	DDR/QDR	Ancho de canal (bytes)	Burst Length	Trans./bloque
GDDR5	924	DDR	4	8	4
GDDR6	1515,15	QDR	4	16	2
DDR2	333,33	DDR	8	8	2
DDR3	800	DDR	8	8	2
DDR4	3200	DDR	8	8	2
PCM	400	DDR	8	8	2
HBM	400	DDR	8	8	2

Tabla 5.2: Configuración de cada partición de memoria principal dependiendo de la tecnología considerada.

realizadas en el canal por acceso o *burst length* (8 ó 16). Así, para transferir un bloque de cache con la tecnología GDDR5 hacen falta $128/(4 \times 8) = 4$ transacciones.

En el estudio de escalabilidad realizado sobre memorias con tecnología HBM (ver sección 6.3), ha sido necesario modificar el ancho de bus por partición y el ancho de canal a 32 y 16 bytes, respectivamente, para cumplir con ese estándar. Por esa misma razón, se ha establecido para HBM un *burst length* de 2. En este caso, el número de transacciones por bloque es $128/(2 \times 16) = 4$.

Por último, indicar que la ejecución de todos los *benchmarks* para todas las configuraciones estudiadas se realiza mediante un *script* en *BASH*. Del mismo modo, se procesan los archivos resultados de la ejecución, generando un sumario de resultados que se importa en una hoja de cálculo para obtener los resultados presentados en el capítulo 6.

5.3 Métricas de prestaciones

En cuanto a las métricas elegidas para evaluar los resultados en este TFG, se ha considerado el IPC (instrucciones por ciclo) promedio que se obtiene al ejecutar cada *benchmark*, la aceleración con respecto a un sistema base y los ciclos de parada en la GPU causados por el subsistema de memoria. La figura 5.1 muestra los distintos lugares dentro de la arquitectura de la GPU donde pueden causarse ciclos de parada, junto con el nombre de la causa proporcionado por GPGPU-Sim. Estas causas se describen a continuación:

- **gpgpu_n_stall_shd_mem**: ciclos de parada causados por i) conflicto en un banco compartido de memoria, ii) acceso a memoria sin coalescencia, o iii) acceso a memoria de constantes serializado.

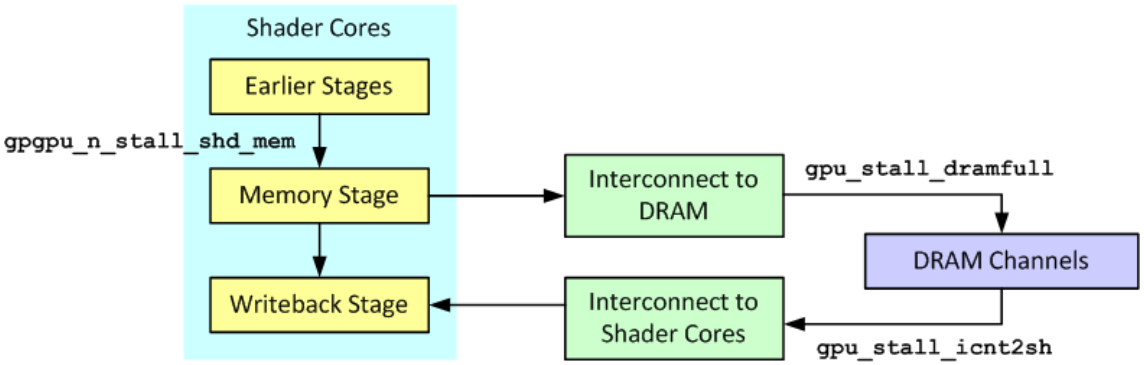

Figura 5.1: Flujo de los accesos a memoria en GPGPU-Sim y causas de introducción de ciclos de parada. Fuente: [4].

- **gpu_stall_dramfull**: ciclos de parada debidos a que las colas de entrada de la DRAM están llenas.
- **gpu_stall_icnt2sh**: ciclos de parada debidos a que la red de interconexión de la DRAM con los SM está congestionada y no puede aceptar más respuestas.

CAPÍTULO 6
Resultados

En este capítulo se discuten tres experimentos realizados con el entorno propuesto integrado por GPGPU-Sim y los distintos simuladores de memoria principal. El primer experimento compara los resultados del sistema base GPU GTX480 utilizando los diferentes simuladores disponibles de memoria principal GDDR5. El segundo experimento demuestra la potencia de la propuesta, al mostrar resultados de prestaciones del sistema base con una amplia variedad de tecnologías de memoria principal. Finalmente, en el último experimento se realiza un análisis de escalabilidad del sistema variando el número de canales de una memoria implementada con tecnología HBM. Este experimento permite vislumbrar las posibilidades que da el desarrollo propuesto en el análisis de nuevas tecnologías de memoria aplicadas a las GPU.

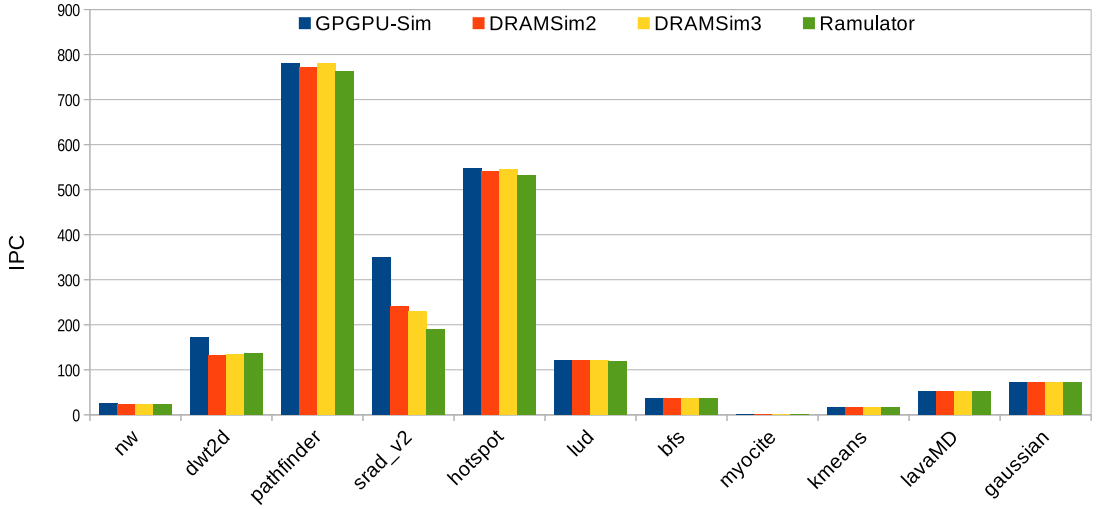

Figura 6.1: IPC variando el simulador de memoria GDDR5.

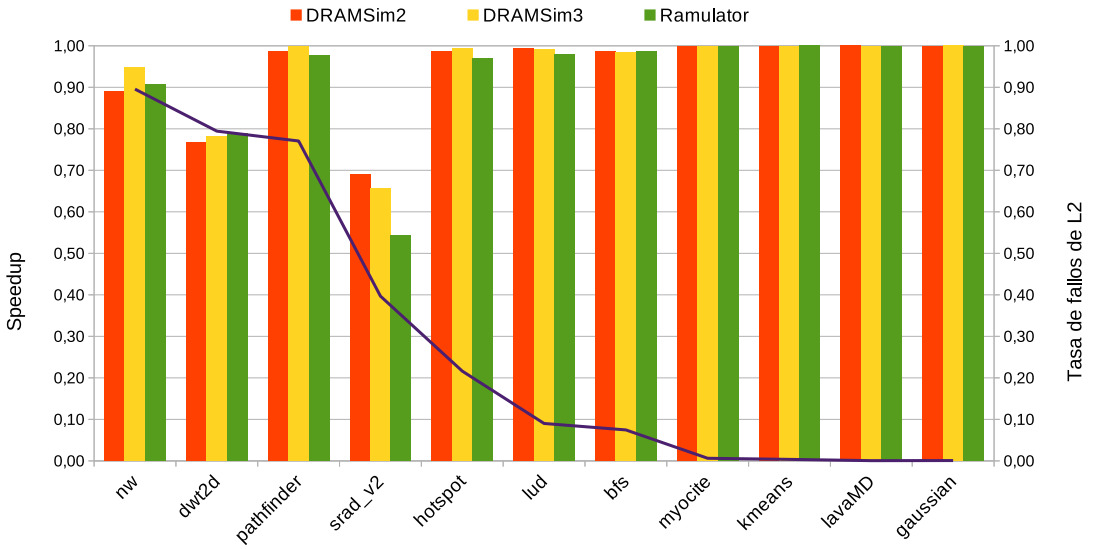

Figura 6.2: Aceleración con respecto a GPGPU-Sim y tasa de fallos de L2 variando el simulador de memoria GDDR5.

6.1 Sistema base con diferentes simuladores de GDDR5

En primer lugar, se evalúa la integración de GPGPU-Sim con cada simulador de memoria principal utilizando la tecnología GDDR5. Esta tecnología es la que implementa la GPU GTX480 y es posible modelarla en todos los simuladores considerados.

La figura 6.1 muestra el IPC promedio alcanzado por cada aplicación en la GPU GTX480 variando el simulador de memoria GDDR5. En general, el IPC es similar entre simuladores excepto para 2 de las aplicaciones estudiadas (*dwt2d* y *srad_v2*). Si se comprueba la aceleración (*speedup*) alcanzada por cada configuración con simulador de memoria principal (DRAMSim2, DRAMSim3, o Ramulator) con respecto a GPGPU-Sim se observa que, en general, las variaciones en las prestaciones están correlacionadas con la tasa de fallos en L2, tal como muestra la figura 6.2. Ello es debido a que una menor

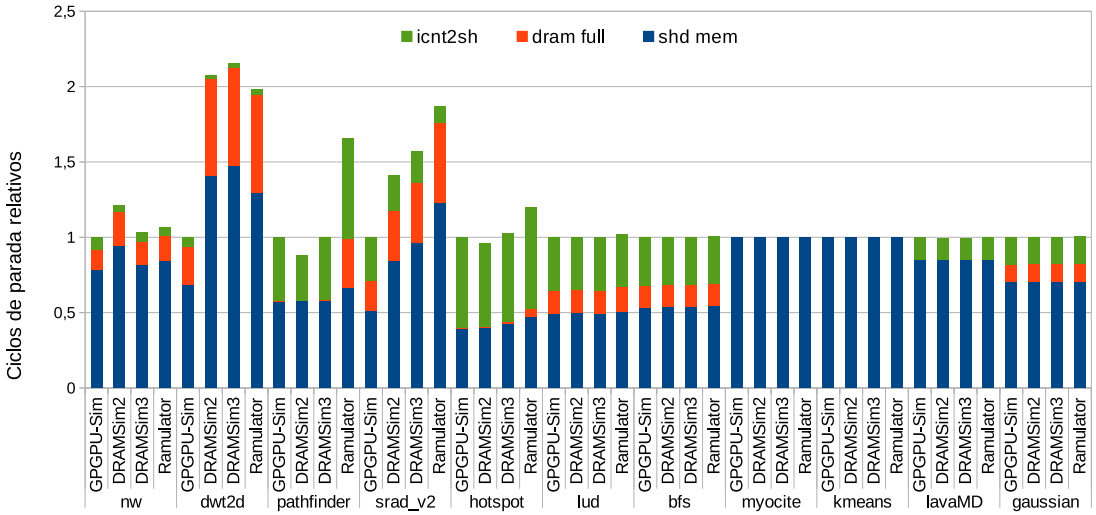

Figura 6.3: Distribución de ciclos de parada con respecto a GPGPU-Sim variando el simulador de memoria GDDR5.

tasa de fallos implica un menor número de accesos a memoria principal, lo que reduce el impacto de la simulación de memoria principal en los resultados.

La variación en las prestaciones se explica mediante características de las aplicaciones, tales como el número de accesos a memoria principal por unidad de tiempo, o el número de ciclos de parada en los que se incurre durante la ejecución. La figura 6.3 muestra la distribución de ciclos de parada con respecto a los presentados por GPGPU-Sim para cada simulador de memoria. En general, se observa que un aumento de los ciclos de parada implica una reducción en las prestaciones, excepto en algún caso (por ejemplo, en *pathfinder* con Ramulator). Aunque la mayoría de los ciclos de parada ocurren por ineficiencias en el acceso a memoria compartida por parte de los núcleos (categoría *shd mem*, ver sección 5.3), más de la mitad de aplicaciones presentan un porcentaje significativo de ciclos de parada debidos a contención en el acceso a memoria principal (*dram full*) y en la red de interconexión (*icnt2sh*). La variación entre los ciclos reportados para cada categoría por los distintos simuladores depende tanto del comportamiento de cada aplicación como el detalle del modelo de memoria principal correspondiente. El análisis de estas variaciones se encuentra fuera de los objetivos de este TFG.

6.2 Simulación de distintas tecnologías de memoria

En este experimento se aprovechan las capacidades únicas de cada simulador de memoria para mostrar como la integración realizada permite simular una amplia variedad de tecnologías diferentes de la GDDR5 disponible en GPGPU-Sim. Concretamente se muestran las tecnologías DDR2, DDR3, DDR4, GDDR6, y PCM. DDR2 y DDR3 se simulan utilizando DRAMSim2; DDR4 y GDDR6 con DRAMSim3; PCM con Ramulator.

La figura 6.4 muestra la aceleración presentada por cada tecnología de memoria con respecto a DDR2 para las aplicaciones estudiadas. DDR2 ha sido seleccionado como referencia por tener, en general, las peores prestaciones entre las tecnologías de memoria estudiadas. Como es de esperar, las tecnologías de memoria más recientes, DDR4 y GDDR6 presentan los mejores resultados debido principalmente a su mayor ancho de banda. Por otro lado, la tecnología PCM presenta unas prestaciones similares a DDR3, aunque en algún caso (p.e. *pathfinder*) ofrece unas pobres prestaciones (menores que las de DDR2).

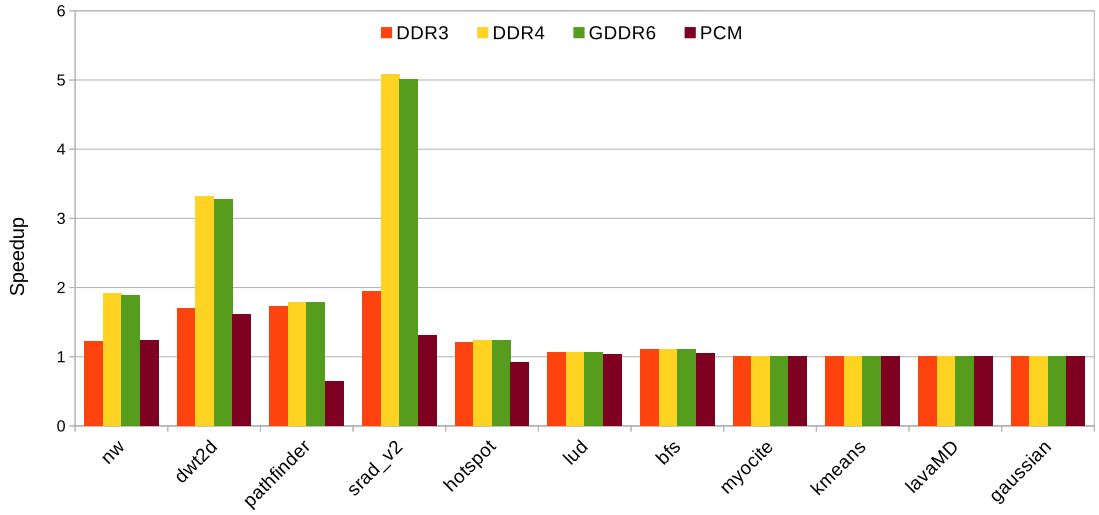

Figura 6.4: Aceleración con respecto a DDR2 variando la tecnología de memoria.

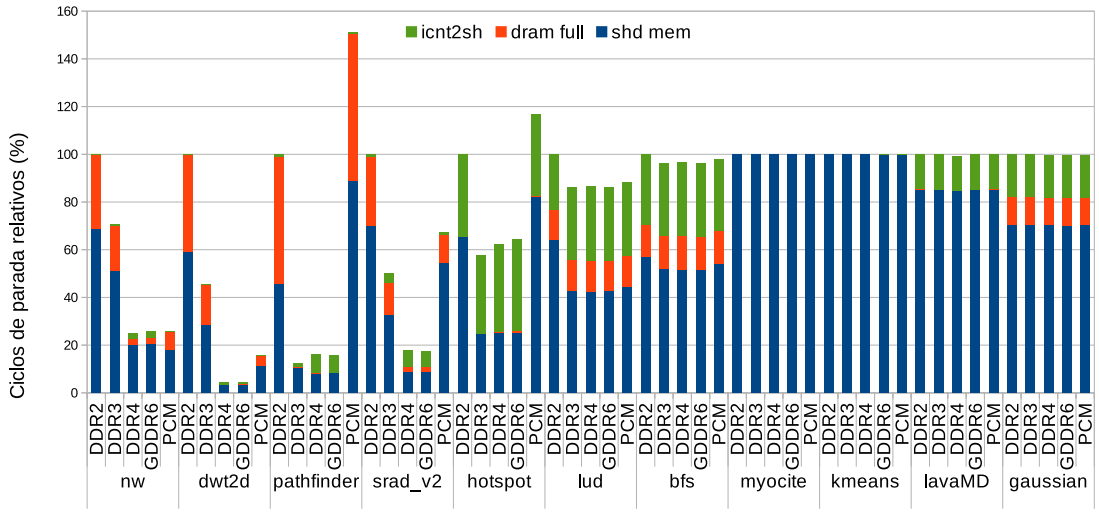

Figura 6.5: Distribución de ciclos de parada con respecto a DDR2 variando la tecnología de memoria.

Esto puede deberse a que este tipo de memorias presenta bajas prestaciones en accesos de escritura y con baja localidad espacial. Por último, cabe destacar que las aplicaciones con una menor tasa de fallos de L2 no se ven afectadas por la implementación de la memoria principal, tal como se observaba en la anterior sección.

Como en la sección anterior, las diferencias en prestaciones se pueden explicar estudiando los ciclos de parada. La figura 6.5 presenta la distribución de ciclos de parada en cada tecnología relativos a la distribución en DDR2. Se observa una clara relación inversa entre los ciclos de parada y las prestaciones reportadas en la figura 6.4. En las aplicaciones donde el impacto de la tecnología es significativo (es decir, aquellas con una mayor tasa de accesos a L2) las tecnologías más rápidas reducen la contención en el acceso a memoria principal (*dram full*) y las ineficiencias en el acceso a memoria compartida desde los núcleos (*shd mem*). Esto último es debido, entre otras razones, a que memorias principales más rápidas y avanzadas reducen la probabilidad de conflictos en bancos de memoria. Nótese que la reducción de ciclos de parada debidos a las categorías mencionadas con-

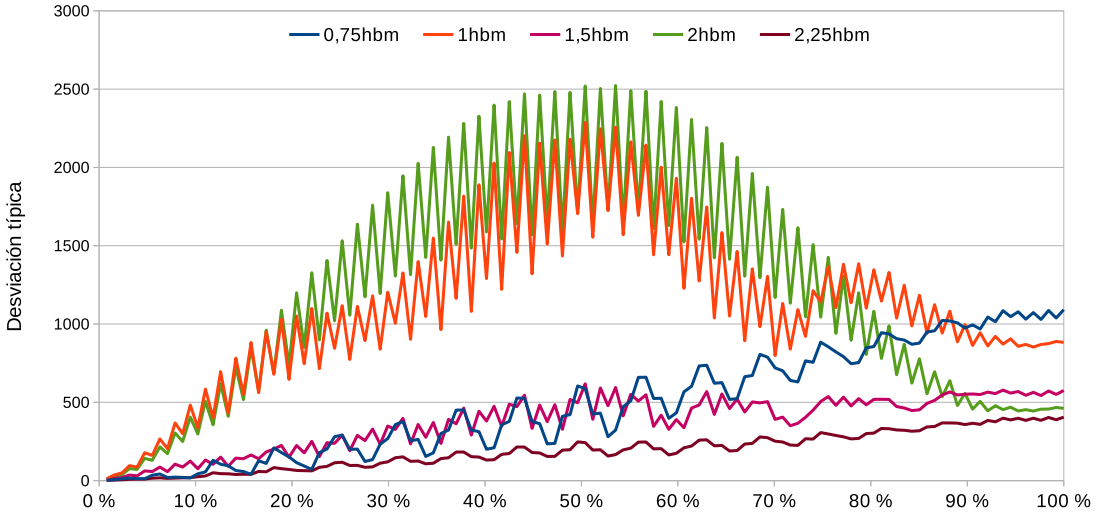

Figura 6.6: Desviación típica del número de accesos a los módulos de L2 durante la ejecución de *nw* variando la configuración HBM.

lleva la aparición de nuevos ciclos de parada debidos a la red de interconexión (*icnt2sh*) al cambiar el principal cuello de botella del sistema. Este efecto es más visible en *pathfinder* y *srad_v2* con las tecnologías de memoria DDR4 y GDDR6. Finalmente, PCM, la única tecnología no DRAM considerada, muestra un comportamiento muy heterogéneo entre las aplicaciones estudiadas. Esto indica que su rendimiento es altamente sensible al patrón de accesos de cada aplicación.

6.3 Estudio de escalabilidad del número de canales de memoria con tecnología HBM

En este experimento demostramos el potencial de la integración desarrollada en este TFG a través de un estudio de escalabilidad del número de canales de memoria con la tecnología HBM, modelada con el simulador Ramulator. La tecnología HBM es una tecnología de memoria 3D DRAM mucho más avanzada que la tecnología GDDR5 integrada en la GPU GTX480, lo que demuestra la bondad del desarrollo realizado.

El estándar de tecnología HBM no es adaptable directamente al ancho de bus de memoria por partición que presenta la GPU GTX480, ya que este último es de 8 bytes y HBM requiere un mínimo de 2 canales de 16 bytes por partición. Es por ello que en este experimento, cuyo objetivo es estudiar la escalabilidad de la tecnología HBM se ha adaptado el ancho de bus por partición a 32 bytes distribuido en 2 canales de 16 bytes.

Durante el diseño del experimento se consideraron diferentes números de particiones. Estos números de particiones se han etiquetado con la nomenclatura *Xhbm* donde X representa el número de pilas 3D HBM modeladas. Una pila HBM contiene 8 matrices de memoria, cada una de las cuales responde a uno de los canales conectados a la GPU. Así por ejemplo, 1hbm representa 8 canales, mientras que 0.75hbm y 1.5hbm son 6 ($0{,}75 \times 8$) y 12 ($1{,}5 \times 8$) canales, respectivamente.

Debido a la distribución de la carga entre los módulos L2 del sistema, una configuración con más canales no siempre implica unas mayores prestaciones. En algunos casos el acceso a los módulos L2 puede quedar desequilibrado, lo que afecta negativamente a las prestaciones. Como ejemplo, la figura 6.6 muestra la desviación típica en el número de

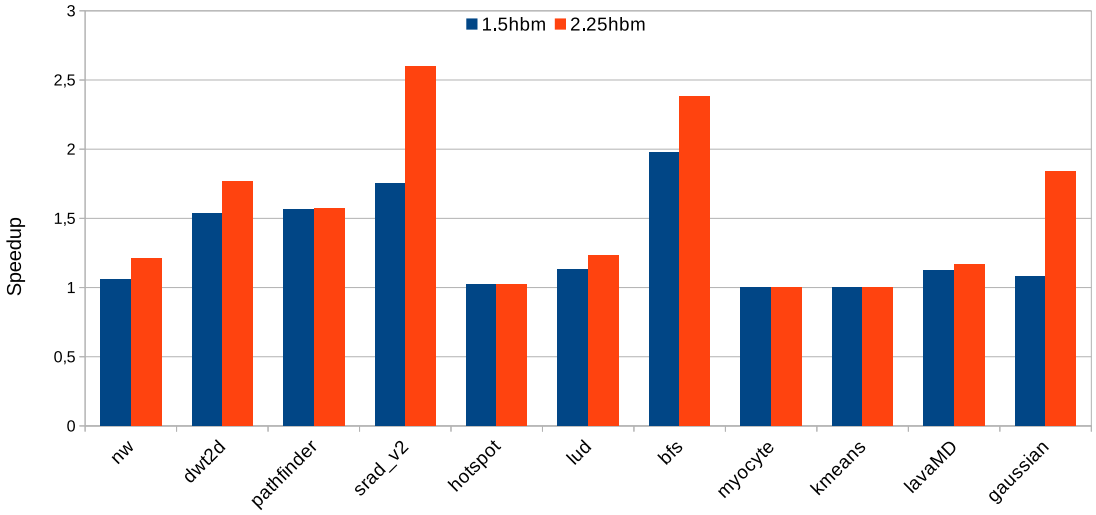

Figura 6.7: Aceleración con respecto a 0.75hbm variando la configuración HBM.

accesos a los diferentes módulos de L2 a lo largo de la ejecución de la aplicación *nw* para diferentes configuraciones HBM. Como se puede observar configuraciones HBM con un número de canales no múltiplo de 6 (1hbm y 2hbm) presentan una mayor variabilidad. Esto se traduce en peores prestaciones pese a disponer de un mayor número de canales que otras configuraciones. Así, con 2hbm, nw obtiene un IPC de 13,6, mientras que con 1.5hbm el IPC es de 22,9. Debido a este efecto, el estudio de escalabilidad se ha realizado con las configuraciones 0.75hbm (6 canales), 1.5hbm (12), y 2.25hbm (18).

La figura 6.7 muestra la aceleración de las configuraciones 1.5hbm y 2.25hbm con respecto a 0.75hbm. Aumentar el número de particiones de memoria tiene beneficios en prácticamente todos las aplicaciones, incluso en algunas con reducida tasa de fallos de L2. Esto es debido a que un aumento de particiones también implica un aumento de la capacidad global de la cache L2, ya que cada partición añade 2 módulos L2. Algunas de las aplicaciones (p.e. *srad_v2*, *gaussian*) muestran grandes beneficios al pasar a 18 canales. Sin embargo, en otras (p.e. *lavaMD*) el incremento de prestaciones es muy reducido o casi inexistente (*pathfinder*) lo que indica, o bien una saturación en la escalabilidad con el número de canales o que sea necesario aumentar la carga de trabajo para aprovechar el mayor ancho de banda.

CAPÍTULO 7
Conclusiones y trabajos futuros

7.1 Conclusiones y trabajos futuros

En este TFG se han integrado un entorno de simulación de GPU con soporte para una sola tecnología de memoria (GDDR5) con múltiples simuladores de tecnologías de memoria principal. El objetivo de este desarrollo es facilitar el proceso de diseño de GPU, donde las características de la memoria principal utilizada son claves para las prestaciones.

El desarrollo se ha realizado teniendo en cuenta un simulador de GPU del estado del arte (GPGPU-Sim) y muchas de las tecnologías de memoria más recientes. El diseño ha sido probado con una suite de aplicaciones de cómputo de propósito general para GPU (Rodinia), demostrando la utilidad del proyecto para el diseño y evaluación de nuevas propuestas de GPU y de subsistemas de memoria para GPU. En particular se ha demostrado que el desarrollo permite: i) comparar modelos de tecnologías de memoria entre diferentes simuladores, ii) modelar una amplia variedad de tecnologías de memoria, y iii) realizar análisis que no es posible efectuar únicamente con GPGPU-Sim. Por todo esto, se puede decir que el objetivo principal de este TFG se ha conseguido con creces.

Al comienzo de este TFG, la última versión disponible de GPGPU-Sim era la versión 3.2.2. Durante el desarrollo se liberó la versión 4.0.1 de este simulador, a la cual se migraron las modificaciones fruto de este trabajo. Sin embargo, debido a numerosos errores de regresión, las aplicaciones de la suite Rodinia no funcionaban correctamente, por lo que se decidió continuar con la versión previa. Pese a ser uno de los simuladores de GPU más utilizados, GPGPU-Sim necesita una adaptación de su código fuente a sistemas operativos y compiladores actuales. La documentación oficial indica que su ejecución está verificada en Suse Linux 11.3 (año 2010), y la versión 4.5 del compilador GCC. En un situación similar se encuentra la suite de *benchmarks* Rodinia. A este respecto, se proponen a continuación posibles trabajos futuros para continuar el trabajo iniciado en este TFG:

- Actualización del código de GPGPU-Sim 4 y documentación asociada, para actualizarse a compiladores y sistemas operativos modernos.
- Actualización de Rodinia a entornos de desarrollo modernos, recopilando aplicaciones científicas modernas de computación paralela apropiadas para *benchmarking*.
- Validación de los resultados obtenidos por GPGU-Sim con hardware real.
- Evaluación de otros paradigmas de computación alternativos a CUDA (p.e. OpenCL), y estudio del simulador con este tipo de cargas.

- Evaluación de las funcionalidades exclusivas proporcionadas por el simulador DRAMSim3, como el modelado del comportamiento térmico y simulación de memorias *Hybrid Memory Cube* (HMC).

7.2 Competencias Transversales adquiridas durante el desarrollo del TFG

La realización de este trabajo ha requerido el ejercicio y mejora continua de distintas Competencias Transversales, sobre las cuales se ofrece a continuación un breve comentario relativo a la forma en que se trabajaron:

- **CT1 - Comprensión e integración**: Tanto el objetivo general del trabajo como los medios para alcanzarlo no tienen por qué ser evidentes o acertados para un público de otras especialidades. Se realizó por tanto un esfuerzo para que la descripción de los objetivos incluyera el contexto necesario para facilitar una valoración informada, y una discriminación de los aspectos más relevantes frente a los secundarios para describir de forma amena el recorrido que lo lleva a cabo. Estas descripciones fueron en muchas ocasiones, resultado de la propia necesidad del autor de esquematizar para su propia comprensión toda la información con la que debía trabajar.

- **CT2 – Aplicación y pensamiento práctico**: Por la propia naturaleza del técnica del trabajo, esta competencia fue realmente una condición para poder llevarlo a cabo. En la medida de lo posible se recurrió a las fuentes autorizadas para la consulta de la información sobre las distintas tecnologías, y cuando ello no fue posible por ser información no publicada debido a cuestiones de confidencialidad o secreto comercial, se referenciaron las hipótesis proporcionadas por los trabajos académicos que las describen tras análisis de ingeniería inversa. En todos los casos se citaron y reconocieron los autores de estos trabajos en la forma que se solicitaba hacerlo. Al mismo tiempo, el propio carácter de los datos obtenidos, consistentes en métricas sobre el rendimiento de componentes hardware, resulta ser también un indicador válido para observar el éxito del proyecto y la validez de la solución.

- **CT3 – Análisis y resolución de problemas**: Tras el planteamiento del objetivo se procedió a alcanzarlo mediante una estrategia consistente en progresar hacia los hitos más inmediatos que llevaban a él. Para ello, se partió del mismo planteamiento conceptual, y para pasar después a la integración de las funcionalidades más básicas del simulador externo cuyo acoplamiento era más simple, tras ello las distintas funcionalidades, y posteriormente, los simuladores más complejos. Se describieron los obstáculos que se encontraron en este camino, así como la forma de solventarlos y su justificación.

- **CT4 - Creatividad, innovación y emprendimiento**: El TFG se enmarca en un ámbito de la ingeniería de computadores correspondiente a la computación paralela, el cual se encuentra en pleno auge y presenta muchas oportunidades de innovación a corto y medio plazo. Como herramienta de ingeniería, los simuladores ocupan un lugar destacado en este campo por su capacidad de permitir una evaluación previa de múltiples propuestas de diseño.

- **CT7 - Responsabilidad ética, medioambiental y profesional**: Las tecnologías de memoria son especificaciones complejas que requieren una determinación estricta de todos su parámetros para poder funcionar correctamente. Debido a ello se

presenta como totalmente imprescindible la consulta de estas especificaciones y estándares industriales para establecer sus distintos parámetros. Al mismo tiempo, se hace uso de las referencias que los principales actores industriales facilitan para la documentación técnica, así como la nomenclatura estandarizada.

- **CT8 - Comunicación efectiva**: La necesidad de expresar en términos asequibles las distintas etapas de realización del proyecto, así como de enlazarlas coherentemente para conseguir que su presentación no corra el riesgo de ser percibida como inconexa, requirió de un diálogo constante con los tutores que en ocasiones y de manera muy oportuna, cobró el carácter del debate académico.
- **CT9 - Pensamiento crítico**: Como consecuencia de afrontar un proyecto que se percibía ambicioso para un recién titulado, fue necesario un sondeo del actual contexto tecnológico en su campo, para tener una visión general de los desarrollos existentes en el mismo ámbito en el que se deseaba proyectar una aportación propia, y poder valorar de manera realista el rol que en ese escenario podía ocupar.
- **CT11 – Aprendizaje permanente**: La realización del TFG requirió la familiarización con nuevos términos, conceptos, paradigmas, y particularidades que sólo aparecen al afrontar ciertos aspectos de subsistemas muy concretos de las arquitecturas trabajadas, lo que obligó no sólo a un aprendizaje basado en la obtención de nuevos conocimientos, sino también a un proceso práctico de ensayo y error que derivó en la adquisición de ciertas aptitudes que permanecen innacesibles por otras vías.
- **CT13 – Instrumentación específica**: Este trabajo implicó el uso y modificación de software de aplicación, el cual tiene a su vez sus propios requisitos de hardware y software para funcionar. De esta manera, fue necesaria la interacción cotidiana con las más diversas herramientas y utilidades además de la preparación de la configuración Sistema Operativo donde se ejecutaban, alcanzando a distintos ámbitos como son las conexiones remotas, la automatización de tareas, el filtrado y formateado de ficheros de salida, software de virtualización, compresores de datos, software ofimático, compiladores, lenguajes de marcas, y otros.

7.3 Asignaturas de la carrera relacionadas con el TFG

A continuación se mencionan las asignaturas cursadas en el Grado en Informática más directamente relacionadas con este TFG, aprovechando para ello la descripción de las mismas que se encuentra en la guía docente.

- **Arquitectura e Ingeniería de Computadores [49]**

  Objetivos de la asignatura más directamente relacionados:

  - Definir el concepto de arquitectura. Distinguir los parámetros que influyen sobre las prestaciones de una arquitectura.
  - Conocer y comprender las técnicas utilizadas para diseñar subsistemas de memoria de altas prestaciones.
  - Distinguir los tipos de computadores orientados al procesamiento de vectores y gráficos.

  Este TFG se centra en el estudio de una arquitectura de computación vectorial GPU, más concretamente sobre el comportamiento de subsistema de memoria. Por lo tanto resultaron fundamentales los conocimientos adquiridos en esta asignatura respecto a su diseño y compromisos que afectan sus prestaciones.

- **Arquitecturas Avanzadas [50]**

  Objetivos de la asignatura más directamente relacionados:

  - Comprensión de dónde y porqué se pueden producir pérdidas de prestaciones en los procesadores.
  - Comprender como aprovechar las capacidades de cómputo paralelo de los procesadores actuales.
  - Conocer arquitecturas heterogéneas de cómputo masivo actuales.
  - Comprender el impacto que pueden tener las redes de interconexión en las prestaciones del sistema.

  A la hora de evaluar su comportamiento bajo distintas cargas, fue necesario sopesar los distintos motivos que podían lugar a las variaciones de rendimiento. Así mismo, dado que el subsistema de memoria conecta con los núcleos de cómputo por medio de una red de interconexión de altas prestaciones, ha sido imprescindible conocer los detalles básicos sobre su funcionamiento, principalmente al modificar sus parámetros de cara a probar variaciones sobre la arquitectura base.

- **Lenguajes y entornos de programación paralela [51]**

  Objetivos de la asignatura más directamente relacionados:

  - Esta asignatura profundiza en el desarrollo de las habilidades adquiridas en el manejo de las herramientas de Programación Paralela introducidas previamente en la asignatura de Computación Paralela (CPA).
  - Se tratan diversos esquemas algorítmicos paralelos aplicables a problemas típicos utilizando conceptos avanzados de OpenMP y MPI con objeto de abordar problemas más complejos de forma más sencilla y eficiente.
  - La metodología docente está basada en el trabajo sobre casos de estudio, así como actividades de carácter práctico sobre plataformas reales, lo que permitirá adquirir las competencias específicas y transversales previstas en las materias Computación Paralela y Programación Paralela.

  Si bien los *benchmarks* Rodinia utilizados en es TFG se compilan con el objetivo CUDA como paradigma de computación paralela, estos admiten también el objetivo OpenMP, debido a que conceptualmente ambos estándares comparten el enfoque fundamental de computación paralela, variando únicamente en la implementación.

Bibliografía

[1] Fermi Computer Architecture Whitepaper. https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf. [Online; Accessed March-2022].

[2] Milan B. Radulovic. Memory bandwidth and latency in hpc: system requirements and performance impact. 2019.

[3] DRAMSim2: A cycle accurate DRAM simulator. https://github.com/umd-memsys/DRAMSim2. [Online; Accessed March-2022].

[4] GPGPU-Sim Manual. http://gpgpu-sim.org/manual/index.php/Main_Page. [Online; Accessed March-2022].

[5] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 163–174, 2009.

[6] Rodinia: Accelerating Compute-Intensive Applications with Accelerators. http://www.cs.virginia.edu/rodinia/doku.php?id=start. [Online; Accessed March-2022].

[7] Tomas Akenine-Moller, Eric Haines, and Naty Hoffman. *Real-Time Rendering, Fourth Edition*. A. K. Peters, Ltd., USA, 4th edition, 2018.

[8] John L. Hennessy and David A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 6th edition, 2017.

[9] nVidia Developer: About CUDA. https://developer.nvidia.com/about-cuda. [Online; Accessed March-2022].

[10] The OpenCL Specification. https://www.khronos.org/registry/OpenCL/specs/3.0-unified/pdf/OpenCL_API.pdf. [Online; Accessed March-2022].

[11] The OpenACC Application Programming Interface. https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC-3.1-final.pdf. [Online; Accessed March-2022].

[12] Nvidia - What Is Stream Processing? https://www.nvidia.com/en-us/glossary/data-science/stream-processing/. [Online; Accessed March-2022].

[13] NVIDIA GeForce GTX 480. https://www.techpowerup.com/gpu-specs/geforce-gtx-480.c268. [Online; Accessed March-2022].

[14] André Lopes, Frederico Pratas, Leonel Sousa, and Aleksandar Ilic. Exploring gpu performance, power and energy-efficiency bounds with cache-aware roofline modeling. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 259–268, 2017.

[15] JEDEC STANDARD DDR2 SDRAM SPECIFICATION JESD79-2F.

[16] JEDEC STANDARD DDR3 SDRAM SPECIFICATION JESD79-3F.

[17] JEDEC STANDARD DDR4 SDRAM SPECIFICATION JESD79-4D.

[18] JEDEC STANDARD GDDR5 SDRAM SPECIFICATION JESD212C.

[19] JEDEC STANDARD GDDR6 SDRAM SPECIFICATION JESD250C.

[20] JEDEC STANDARD HBM SDRAM SPECIFICATION JESD235D.

[21] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, page 2–13, New York, NY, USA, 2009. Association for Computing Machinery.

[22] Memory Prefetching. https://compas.cs.stonybrook.edu/~nhonarmand/courses/sp16/cse502/slides/13-prefetch.pdf. [Online; Accessed March-2022].

[23] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, 2009.

[24] Shuai Che, Jeremy W. Sheaffer, Michael Boyer, Lukasz G. Szafaryn, Liang Wang, and Kevin Skadron. A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads. In *IEEE International Symposium on Workload Characterization (IISWC'10)*, pages 1–11, 2010.

[25] Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. Multi2sim: A simulation framework for cpu-gpu computing. In *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 335–344, 2012.

[26] Jason Power, Joel Hestness, Marc S. Orr, Mark D. Hill, and David A. Wood. gem5-gpu: A heterogeneous cpu-gpu simulator. *IEEE Computer Architecture Letters*, 14(1):34–36, 2015.

[27] Geetika Malhotra, Seep Goel, and Smruti R. Sarangi. Gputejas: A parallel simulator for gpu architectures. In *2014 21st International Conference on High Performance Computing (HiPC)*, pages 1–10, 2014.

[28] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. Dramsim2: A cycle accurate memory system simulator. *IEEE Computer Architecture Letters*, 10(1):16–19, 2011.

[29] Shang Li, Zhiyuan Yang, Dhiraj Reddy, Ankur Srivastava, and Bruce Jacob. Dramsim3: A cycle-accurate, thermal-capable dram simulator. *IEEE Computer Architecture Letters*, 19(2):106–109, 2020.

[30] Yoongu Kim, Weikun Yang, and Onur Mutlu. Ramulator: A fast and extensible dram simulator. *IEEE Computer Architecture Letters*, 15(1):45–49, 2016.

[31] Alberto Checa Muelas. *Evaluación de aplicaciones en una GPU Nvidia Titan X y modelado de la GPU*. PhD thesis, Universitat Politècnica de València, 2018.

[32] David Baselga Masiá. *Caracterización de aplicaciones GPU y estudio de interferencias en una GPU Nvidia GeForce Titan X*. PhD thesis, Universitat Politècnica de València, 2017.

[33] Miguel Antonio Avargues Gutiérrez. *Análisis de requerimientos y diseño de un controlador de memoria principal no volátil*. PhD thesis, Universitat Politècnica de València, 2021.

[34] Hugo Tárrega Sánchez. *Diseño de Caches L1 utilizando la tecnología emergente Domain Wall Memory*. PhD thesis, Universitat Politècnica de València, 2020.

[35] GCC, the GNU Compiler Collection. https://gcc.gnu.org/. [Online; Accessed March-2022].

[36] The OpenMP API specification for parallel programming. https://www.openmp.org/. [Online; Accessed March-2022].

[37] Anuj Chaudhary, Deepkumar Kagathara, and Vibha Patel. A gpu based implementation of needleman-wunsch algorithm using skewing transformation. In *2015 Eighth International Conference on Contemporary Computing (IC3)*, pages 498–502, 2015.

[38] Vicente Estellés, Ricardo Colom-Palero, Rafael Gadea, Ángel Sebastiá, Marcos Martínez, Vicente Herrero, and Vicente Arnau. Transformada discreta wavelet 2-d para procesamiento de video en tiempo real. 09 2001.

[39] Ross Graham, Hugh McCabe, and Stephen Sheridan. Pathfinding in computer games. *The ITB Journal*, 4:6, 2003.

[40] Yongjian Yu and S.T. Acton. Speckle reducing anisotropic diffusion. *IEEE Transactions on Image Processing*, 11(11):1260–1270, 2002.

[41] Wei Huang, S. Ghosh, S. Velusamy, K. Sankaranarayanan, K. Skadron, and M.R. Stan. Hotspot: a compact thermal modeling methodology for early-stage vlsi design. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(5):501–513, 2006.

[42] Jiayuan Meng and Kevin Skadron. Performance modeling and automatic ghost zone optimization for iterative stencil loops on gpus. pages 256–265, 01 2009.

[43] Wolfram MathWorld, LU Decomposition. https://mathworld.wolfram.com/LUDecomposition.html. [Online; Accessed March-2022].

[44] Pawan Harish and P. J. Narayanan. Accelerating large graph algorithms on the gpu using cuda. In Srinivas Aluru, Manish Parashar, Ramamurthy Badrinath, and Viktor K. Prasanna, editors, *High Performance Computing – HiPC 2007*, pages 197–208, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[45] Lukasz Szafaryn, Kevin Skadron, and Jeffrey Saucerman. Experiences accelerating matlab systems biology applications. 09 2010.

[46] K-means Clustering: Algorithm, Applications, Evaluation Methods, and Drawbacks. https://towardsdatascience.com/k-means-clustering-algorithm-applications-evaluation-methods-and-drawbacks-aa03e644b48a. [Online; Accessed March-2022].

[47] Lukasz G. Szafaryn, Todd Gamblin, Bronis R. de Supinski, and Kevin Skadron. Experiences with achieving portability across heterogeneous architectures. 2011.

[48] Wolfram MathWorld, Gaussian Elimination. https://mathworld.wolfram.com/GaussianElimination.html. [Online; Accessed March-2022].

[49] Arquitectura e Ingeniería de Computadores, Grado en Ingeniería Informática UPV. https://www.upv.es/pls/oalu/sic_gdoc.get_content?P_OCW=&P_ASI=11553&P_IDIOMA=c&P_VISTA=MSE&P_TIT=156&P_CACA=2021&P_CONTENT=descripcion. [Online; Accessed March-2022].

[50] Arquitecturas Avanzadas, Grado en Ingeniería Informática UPV. https://www.upv.es/pls/oalu/sic_gdoc.get_content?P_OCW=&P_ASI=11582&P_IDIOMA=c&P_VISTA=MSE&P_TIT=156&P_CACA=2021&P_CONTENT=descripcion. [Online; Accessed March-2022].

[51] Lenguajes y Entornos de Programación Paralela, Grado en Ingeniería Informática UPV. https://www.upv.es/pls/oalu/sic_gdoc.get_content?P_OCW=&P_ASI=11575&P_IDIOMA=c&P_VISTA=MSE&P_TIT=156&P_CACA=2021&P_CONTENT=descripcion. [Online; Accessed March-2022].

ANEXO

OBJETIVOS DE DESARROLLO SOSTENIBLE

Grado de relación del trabajo con los Objetivos de Desarrollo Sostenible (ODS).

Objetivos de Desarrollo Sostenible	**Alto**	**Medio**	**Bajo**	**No procede**
ODS 1. **Fin de la pobreza.**			X	
ODS 2. **Hambre cero.**				X
ODS 3. **Salud y bienestar.**		X		
ODS 4. **Educación de calidad.**		X		
ODS 5. **Igualdad de género.**			X	
ODS 6. **Agua limpia y saneamiento.**				X
ODS 7. **Energía asequible y no contaminante.**	X			
ODS 8. **Trabajo decente y crecimiento económico.**			X	
ODS 9. **Industria, innovación e infraestructuras.**			X	
ODS 10. **Reducción de las desigualdades.**				X
ODS 11. **Ciudades y comunidades sostenibles.**				X
ODS 12. **Producción y consumo responsables.**				X
ODS 13. **Acción por el clima.**	X			
ODS 14. **Vida submarina.**				X
ODS 15. **Vida de ecosistemas terrestres.**				X
ODS 16. **Paz, justicia e instituciones sólidas.**				X
ODS 17. **Alianzas para lograr objetivos.**			X	

Reflexión sobre la relación del TFG/TFM con los ODS y con el/los ODS más relacionados.

La definición de la Agenda 2030 como un sumario de objetivos en lugar de un sumario de estrategias para lograrlos, convierte la cuestión sobre qué factores ayudan a alcanzarlos en una valoración con un componente de subjetividad. Se puede considerar que esta circunstancia es una forma de flexibilizar el debate sobre cómo alcanzarlos, haciendo énfasis en que lo más importante es conseguirlos, y confiando en que el planteamiento de estrategias erróneas para ello serían autoexcluyentes.

Respecto al vínculo que este TFG puede tener con estos objetivos, cabe contemplar tanto los directos como los indirectos. Estando centrado en una herramienta de ingeniería para el diseño de sistemas de cómputo paralelo más eficientes, existe un vínculo directo con la eficiencia energética y por tanto con la disminución de las contrapartidas indeseables que derivan de la producción de esta energía, como la emisión de gases de efecto invernadero o la polución, pero también con el alcance geográfico de estas tecnologías allá donde esta producción es escasa, algo relacionado con los objetivos de igualdad.

Al mismo tiempo existe también un vínculo indirecto con ciertos objetivos, por ser un TFG que se enmarca el ámbito de la mejora de las Tecnologías de la Información y las Comunicaciones. Estas tecnologías son catalogadas como centrales en la descripción de los procesos que permiten alcanzar algunos objetivos, varios de los cuales no aparentarían tener relación con ellos a primera vista. En estos casos se ha considerado que existía un vínculo de nivel bajo.

A continuación se exponen las reflexiones que dieron lugar a declarar el nivel de vinculación percibido con las distintos objetivos en la tabla superior.

- **ODS 1. Fin de la pobreza.**

  El punto 1.4 menciona la ampliación en el acceso a nuevas tecnologías. Se percibe un nivel bajo de vinculación por el rol que pueden tener los diseños específicos para este fin y facilitados por nuevas herramientas de ingeniería.

- **ODS 3. Salud y bienestar.**

  La exposición de este objetivo y su necesidad está actualmente naturalmente basada en la pandemia del virus Covid-19. En este sentido, se ha manifestado el importante papel de la supercomputación en la lucha contra la enfermedad y la investigación del genoma del coronavirus. Estas capacidades de computación están cada vez más basadas en el paradigma heterogéneo CPU-GPU. Por lo tanto se percibe un nivel medio de vinculación.

- **ODS 4. Educación de calidad.**

  De nuevo la pandemia ha manifestado el rol que pueden tener las T.I.C. en este objetivo, convirtiendo la educación a distancia en una necesidad bajo ciertas circunstancias. Además el punto 4b describe que entre las materias objeto de esta educación también han de incluir la ingeniería y las T.I.C., para lo cual son imprescindibles ciertas herramientas de apoyo. Se percibe un vínculo de nivel medio.

- **ODS 5. Igualdad de género.**

  Se percibe un posible vínculo quizás de nivel bajo pero existente, por mencionarse en el apartado 5b la mejora de la tecnología de la información y las comunicaciones para promover el empoderamiento de las mujeres.

- **ODS 7. Energía asequible y no contaminante.**

  Se concluye un vínculo de nivel alto con este objetivo, por ser explícito el punto 7.3 de duplicar la tasa mundial de mejora de la eficiencia energética para el año 2030, con la implicación de un enfoque prioritario sobre la eficiencia en todos los aparatos y dispositivos electrónicos, incluyendo las computadoras.

- **ODS 8. Trabajo decente y crecimiento económico.**

  Con el mismo rol que tiene la educación a distancia para alcanzar el ODS 4, el teletrabajo usando tecnologías de información y comunicación puede facilitar la consecución de los puntos 8.5 y 8.8, relativos al pleno empleo y la promoción de entornos de trabajo seguros. Por ello se percibe que existe un nivel de vinculación de nivel bajo.

- **ODS 9. Industria, innovación e infraestructuras.**

  Se percibe la existencia de un posible vínculo de nivel bajo, por encontrar relación con lo declarado en el objetivo 9c, "Aumentar significativamente el acceso a la tecnología de la información y las comunicaciones"

- **ODS 13. Acción por el clima.**

  Por las razones comentadas para el ODS 7, el estudio y mejora de la eficiencia de todos los dispositivos electrónicos puede tener repercusiones positivas en este aspecto. Concretamente, las GPU son el hardware fundamental de nuevas tendencias con gran impacto medioambiental como son las criptomonedas. El TFG facilita el uso del simulador de memoria DRAMSim3 el cual incluye la capacidad de analizar la disipación térmica de las memorias. Ello permite no sólo estudiar como se optimiza la eficiencia energética si no también ayudar a mejorar los sistemas de refrigeración que en ocasiones son necesarios, y también tienen un elevado consumo energético. Se percibe un vínculo de nivel alto con este objetivo.

- **ODS 17. Alianzas para lograr objetivos.**

  Se percibe la existencia de un vínculo de nivel bajo con este objetivo, por mencionarse en el punto 17.8 la intención de aumentar las tecnologías instrumentales, el particular la tecnología de la información y las comunicaciones.

ETS Enginyeria Informàtica
Camí de Vera, s/n, 46022, València
T +34 963 877 210
F +34 963 877 219
etsinf@upvnet.upv.es - www.inf.upv.es