
***Facultad de Ciencias***

# Metodología TopDown sobre GPUs NVIDIA
## TopDown Methodology on NVIDIA's GPUs

**Trabajo de Fin de Grado**
**para acceder al**

**GRADO EN INGENIERÍA INFORMÁTICA**

**Autor: Álvaro Saiz Fernández**

**Director: Pablo Prieto Torralbo**

**Julio - 2021**

# Índice de Contenidos

# Índice de Figuras

# Índice de Tablas

# Resumen

Las Unidades de Procesamiento Gráfico (GPU) han sido inicialmente diseñadas para gráficos, y frecuentemente utilizadas en diseño y videojuegos. Sin embargo, el auge de algoritmos de uso intensivo de datos, como los de Machine Learning, ha provocado un aumento en el uso de estos dispositivos en otros campos en lo que se conoce como GPGPU (General-purpose computing on graphics processing units). En este tipo de algoritmos se explota el Paralelismo a Nivel de Datos o Single Instruction Multiple Data (SIMD), propio del diseño de las GPU, que las hace candidatas ideales para obtener al máximo rendimiento, superior al obtenido por una CPU en este paradigma de ejecución

Sin embargo, la ejecución en este tipo de sistemas, aparentemente sencillos en su arquitectura, no es trivial. Si queremos optimizar el uso de sus recursos y mejorar su funcionamiento, se hace recomendable tener un modelo de análisis que pueda entrar en profundidad en el comportamiento de la microarquitectura.

Una evaluación de este tipo, en la que se analizan las pérdidas de rendimiento en los diferentes componentes hardware es el análisis TopDown. Inicialmente desarrollado por Intel para sus CPUs, el objetivo de este proyecto es el de intentar replicar esta metodología, adaptándola a las características hardware de una GPU de NVIDIA. Para ello, primero se hará un estudio de la metodología propuesta por Intel, identificando la forma en la que esta ha sido diseñada. A continuación, se tratará de replicar en una GPU de NVIDIA, explicando los conceptos microarquitecturales de este dispositivo, que permitirán posteriormente definir nuestro TopDown para NVIDIA, haciendo uso de los contadores hardware disponibles para obtener los resultados. Por último, al final del proyecto, se proponen una serie de pruebas que permiten analizar cómo se comportan diferentes arquitecturas modernas de NVIDIA sobre benchmarks recientes, evaluando en que componentes microarquitecturales se producen pérdidas de rendimiento.

**Palabras Clave**: GPUs, NVIDIA, TopDown, Contadores Hardware, Rendimiento.

# Abstract

Graphics Processing Units (GPU) have been initially designed for graphics and they are frequently used in design and videogames. However, the rise of data-intensive algorithms, such as Machine Learning, has meant an increase in utilization of these devices in other fields in what is known as GPGPU (General-purpose computing on graphics processing units). This kind of algorithms makes use of Data Level Parallelism or Single Instruction Multiple Data (SIMD), characteristic of GPU design, which makes them ideal candidates.

However, the execution on GPUs, seemingly simple in its architecture, is not trivial. If we want to optimize the resources used, and improve their functioning, it is advisable to have an analysis model that can go into depth in the microarchitecture behavior.

An evaluation of this type, in which the performance losses in the different hardware components are analyzed, is the TopDown analysis. Initially developed by Intel for its CPUs, the target of this proyect is trying to replicate this methodology, adapting it to NVIDIA's GPUs hardware features. For this purpose, first a study of the metholodgy proposed by Intel will be carried out, taking into account the way in which it has been designed. Then, we will try to replicate it on a NVIDIA GPU for what, the microarchitectural concepts of this device will be explained, which will later allow to define our TopDown for NVIDIA devices, making use of the available hardware counters. Finally, at the end of the proyect, a series of tests are proposed to analyze how different modern NVIDIA architectures behave in recent benchmarks, assessing in which microarchitecture components performance losses occur.

**Keywords**: GPUs, NVIDIA, TopDown, Hardware Counters, Profiling.

# Capítulo 1

# Introducción y Motivación

## 1.1. Motivación

La tecnología de los computadores ha evolucionado considerablemente con el paso de los años, desde el Intel 4004 hasta los modernos Intel Core I9 o similares. El concepto y la forma de ejecución ha ido cambiando con el tiempo, intentando siempre obtener el máximo rendimiento. La tendencia al comienzo marcaba un incremento constante en la frecuencia del reloj, soportado con la bajada de tensión umbral gracias al escalado de la tecnología, como indica la ley de Dennard [1]. El aumento de frecuencia sin embargo tiene su límite con el incremento de relevancia de la corriente de fugas, que hizo necesario buscar alternativas para incrementar el rendimiento de los procesadores. El paralelismo a nivel de instrucción (ILP) es una de las vías que las CPUs han explotado (superescalares, ejecución fuera de orden, especulación...), pero ha llegado a un límite, y cada vez el beneficio obtenido es más costoso (dependencias de datos, bloqueos...). Pero la industria ha seguido mejorando la capacidad de integración, y esta limitación en la mejora del rendimiento de un procesador ha sido complementada por el paralelismo a nivel de thread (TLP), multiplicando el número de procesadores en un chip, y aumentando el número de hilos que es posible ejecutar en un solo núcleo.

Adicionalmente, la tendencia de los algoritmos de hoy en día con gran volumen de datos, como el Machine Learning, ha orientado a las CPUs a explotar el paralelismo a nivel de datos (DLP). Para ello, estas han ido incorporando extensiones vectoriales con cada vez capacidad de trabajar mayor cantidad de datos en una sola instrucción (SIMD), utilizando unidades hardware dedicadas para realizar las operaciones y almacenar los resultados (registros vectoriales). A día de hoy constituyen una fuente de rendimiento importante. Por ejemplo, en **HPC** (**High Performance Computing**) las extensiones vectoriales son utilizadas para lograr el máximo rendimiento, utilizados en algunos de los computadores más potentes del mundo, según la lista TOP500 [2].

Sin embargo, existe un dispositivo que parece adecuado para este tipo de algoritmos, para el que el paralelismo a nivel de thread y el paralelismo a nivel de datos es nativo, y cuyo consumo de energía es relativamente bajo en comparación. Se trata de las **Unidades de Procesamiento Gráfico** o **GPUs**. El paradigma de estos co-procesadores es el SIMD (Single Instruction Multiple Data) o SIMT (Single Instruction Multiple Thread), es decir, ejecutar una única instrucción, pero sobre diferentes datos; utilizando gran cantidad de threads en paralelo. Inicialmente, las GPUs fueron diseñadas para aliviar el trabajo de las CPUs en cuanto al procesamiento de gráficos, por ejemplo, videojuegos o aplicaciones 3D, por lo que principalmente eran utilizadas en estos campos, y no tanto en otros como el HPC. Sin embargo, los algoritmos recientes, y en especial, el Machine Learning, han impulsado a las GPUs más allá de los videojuegos y la computación gráfica.

El **Machine Learning** constituye uno de los campos principales en donde se explota el paralelismo a nivel de datos. Esta rama de la Inteligencia Artificial permite crear sistemas de aprendizaje automático que permiten al computador detectar patrones de datos y tomar decisiones con estos sin necesidad de que nadie determine lo que debe de hacer. Para lograr tales fines, hay multitud de algoritmos diferentes, pero comparten el mismo propósito anterior. Por ejemplo, hay algoritmos que hacen uso de un aprendizaje previo para entrenar al algoritmo para que posteriormente pueda pasar a tomar sus decisiones. Del mismo modo, también hay algoritmos que no utilizan este aprendizaje previo si no que tienen un aprendizaje continuo. En general, estos algoritmos utilizan densas matrices de datos en donde se realizan operaciones sobre ellas, por lo que el paralelismo a nivel de datos y el paralelismo a nivel de aplicación cobran especial relevancia. Esta dependencia de algoritmos que explotan el paralelismo a nivel de datos ha promovido el impulso de la **GPGPU** (**General-Purpose computation on Graphics Processing Units**) o computación de propósito general en unidades de procesamiento gráfico. Este concepto hace referencia a la utilización de la GPU para fines diferentes a los que inicialmente fue diseñada, es decir, los gráficos por computador. El gran rendimiento que ofrece la GPU en relación con la potencia que consume ha hecho que las GPUs pasen a ser utilizadas también en áreas como las mencionadas anteriormente. Al uso de las GPGPU ha favorecido, en el caso de NVIDIA, la unificación de su modelo de programación en torno a una nueva arquitectura llamada **Compute Unified Device Architecture** (**CUDA**), originaria en el año 2007 [3]. Además, la tendencia reciente de NVIDIA ha sido la de diferenciar entre GPUs de propósito general, utilizadas para HPC, Machine Learning, Redes Neuronales... y GPUs específicas para procesamiento de gráficos. Un ejemplo de esta tendencia son las microarquitecturas Volta [4] y Turing [5], ambas sucesoras de la microarquitectura Pascal [6]. A raíz del rendimiento que producen las GPUs sobre este tipo de aplicaciones tan demandadas hoy en día, estas han sido incluidas como una opción altamente viable para la supercomputación de alto rendimiento o HPC. De hecho, los últimos supercomputadores tienden a utilizar estos aceleradores en su configuración [7], permitiendo obtener más operaciones de coma flotante por segundo o TFlop/s con un consumo de potencia muy bueno.

No obstante, el rendimiento de las GPUs puede decaer considerablemente si el modo de programación seguido no es el adecuado. Optimizar el rendimiento en una GPU no es trivial y requiere mucha experiencia. Para conseguirlo es necesario conocer con detalle cómo funciona la microarquitectura subyacente de la GPU, además del modelo y estilo de programación. Sin embargo, a pesar de ser una microarquitectura aparentemente más sencilla que la de las CPUs, con una ejecución en orden y sin especulación; la forma en la que se realiza la ejecución no es trivial, por lo que muchas veces resulta difícil determinar lo que está ocurriendo a nivel microarquitectural.

Para ayudarnos a optimizar el código y conocer el comportamiento hardware de la GPU, es importante disponer de herramientas adicionales que permitan al usuario recopilar la información que recibe sobre la ejecución de un determinado código fuente. En el caso concreto de NVIDIA, proporciona algunas herramientas para sus GPUs que permiten obtener información acerca de dónde se producen los cuellos de botella en los códigos de forma que puedan ser solucionados. Estas herramientas muestran, por ejemplo, los tiempos transcurridos en cada una de las partes del código fuente, lo que puede ayudar a detectar dónde está el problema desde un punto de vista software.

Normalmente la tendencia de NVIDIA ha sido la de arrojar información muy cercana al programador, con datos relativos a la planificación y características de ejecución. Esto permite al programador detectar algunos puntos de mejora, pero limitados en algunos aspectos. Por ejemplo, no ofrece gran cantidad de información relativa a la microarquitectura. Esto limita la información recibida, lo que impediría reestructurar el código para evitar pérdidas de rendimiento. Por ejemplo, se podría detectar que hay un problema de rendimiento en una porción concreta de la jerarquía de memoria, por lo que una acción correctora sería la de cambiar la utilización que se hace de esta.

Además de para la programación optimizada, desde el punto de vista de la Arquitectura de Computadores, este tipo de análisis permitirían también a los fabricantes como NVIDIA detectar dónde se

encuentran los problemas de rendimiento en los algoritmos utilizados a día de hoy, lo que ofrece la posibilidad de que puedan hacer énfasis en ellos y establecer técnicas correctoras para versiones de GPU posteriores.

Un ejemplo de mecanismo de detección de problemas en la microarquitectura de una CPU es el presentado en el año 2014 por el Grupo de Arquitectura de Intel, conocido como **metodología TopDown** [8]. Este análisis busca detectar los cuellos de botella en el pipeline de una CPU mediante contadores hardware, y para la cual, Intel añadió algunos contadores específicos a su arquitectura. Estos cuellos de botella se traducen en partes de la microarquitectura en donde la CPU está bloqueada, provocando una pérdida de rendimiento frente al **IPC** (**Instrucciones Por Ciclo**) máximo teórico. La metodología sigue una estructura jerárquica, dividiendo el pipeline con un grado de detalle creciente.

## 1.2. Objetivos

El objetivo de este trabajo consiste en intentar replicar esta metodología de análisis TopDown en las GPUs de NVIDIA, utilizando estrategias similares a las seguidas por Intel, con las diferencias que conlleva una CPU frente a una GPU. Nos apoyaremos en las herramientas facilitadas por NVIDIA para la obtención de métricas de rendimiento y los contadores disponibles, buscando los límites de una metodología de este estilo y las necesidades para una mejor implementación de la misma.

Adicionalmente, en el trabajo se pretende comprobar la viabilidad de la implementación de la metodología TopDown en las GPUs de NVIDIA, así como las posibles necesidades futuras que pueden ser requeridas para implementar esta forma de análisis de manera correcta.

Los objetivos marcados en el TFG son:

- Comprender la metodología Topdown para CPUs.
- Analizar la microarquitectura de una GPU NVIDIA.
- Analizar los contadores hardware disponibles en las distintas arquitecturas de NVIDIA disponibles, y comprender el significado de los mismos, de cara a un posible análisis Topdown.
- Proponer un análisis Topdown para GPUs NVIDIA, con los contadores adecuados.
- Desarrollar una herramienta capaz de automatizar el proceso de análisis TopDown, replicando un comportamiento semejante al de las herramientas equivalentes de CPU.
- Probar la metodología propuesta con diferentes arquitecturas NVIDIA, que usan herramientas distintas, y con distintos benchmarks.

## 1.3. Estructura

El resto del documento se estructura de la siguiente forma:

- En el capítulo 2 haremos un estudio del estado del arte, incluyendo una breve descripción de la metodología TopDown planteada para CPUs.
- En el capítulo 3 se detalla la arquitectura de una GPU, especialmente el Stream Multiprocessor (SM), que será el componente fundamental sobre el que centraremos el análisis TopDown.
- En el capítulo 4 se describe la propuesta desarrollada en este trabajo sobre una metodología TopDown en GPUs, con las decisiones tomadas al respecto.

- En el capítulo 5 haremos una prueba de concepto de la herramienta sobre dos benchmarks de GPU: Rodinia y Altis.
- Finalmente, en el capítulo 6 daremos unas conclusiones y propuestas de ampliación.

# Capítulo 2

# Estado del Arte

A medida que la tecnología va evolucionando, esta se va convirtiendo en más compleja, por lo que entender el funcionamiento interno o el modo de comportamiento, tanto en una CPU como en una GPU resulta tarea difícil. Es por esto que es recomendable proporcionar mecanismos para que sea posible entender o comprender qué está ocurriendo a bajo nivel cuando se trata de ejecutar una aplicación. El **profiling** o medición de rendimiento constituye una pieza fundamental para ello, y permite por ejemplo conseguir avances en la optimización de una aplicación. Además, también es de gran ayuda para los arquitectos hardware, para comprender el funcionamiento interno dado un análisis. Es por esto que, a medida que pasan los años, la información que proporciona el hardware sobre lo que se ejecuta es cada vez más específica y detallada, haciendo la arquitectura subyacente "visible" al usuario.

## 2.1. Unidades de Monitorización y eventos Hardware

Para poder determinar el comportamiento del hardware (tanto en CPU como GPU), se añaden unidades adicionales que se encargan de monitorizar el comportamiento de los distintos componentes de la microarquitectura. Estas unidades se encargan de recabar esta información, que posteriormente será utilizada por herramientas de alto nivel, que se la mostrarán al usuario. El funcionamiento de estas unidades en CPU y GPU es diferente, en tanto que el pipeline de estos es radicalmente distinto. Por un lado, la forma de recolectar los datos es diferente. Por otro lado, el tipo de información mostrada también es distinto, como se verá más adelante. No obstante, a pesar de estas diferencias, comparten un mismo objetivo y es el de mostrar, de una manera o de otra, cómo se comportan sus unidades lógicas subyacentes. En una CPU, la **PMU** (**Performance Monitoring Unit**) constituye la unidad básica de medición de rendimiento. Es una unidad hardware utilizada para monitorizar eventos microarquitecturales. Esta es dependiente de la microarquitectura, aunque si son cercanas en el tiempo pueden compartir algunas similitudes. Normalmente está compuesta de una colección de registros **MSRs** (**Model Specific Register**) [9]. Este tipo de registro suele ser utilizado para depurar, rastrear la ejecución de programas, monitorear el rendimiento de la computadora y alternar ciertas características de la CPU. Son accedidos a través de las instrucciones **RDMSR** y **WRMSR**, además de la instrucción RDMC (para algunos contadores de registros). La PMU incluye diferentes tipos de MSRs: registros contador, selección de eventos y control de eventos globales. Estos registros son configurables, y tendrán que ser parametrizados para poder realizar la función deseada. Por ejemplo, la Figura 2.1 representa un MSR para la selección de eventos.

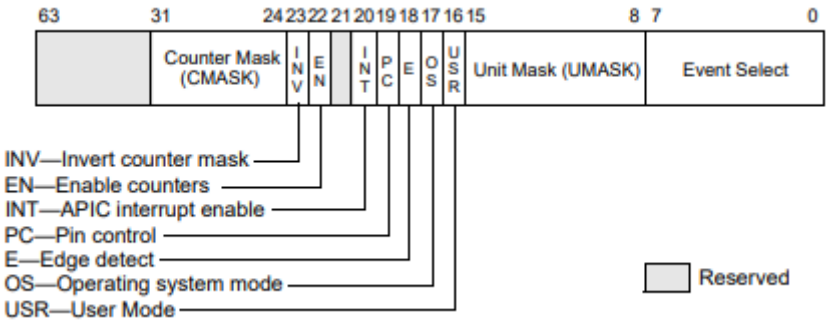


**Figura 2.1:** Registro MSR de selección de eventos, obtenido de la referencia [10].

El registro está dividido en diferentes campos, cada uno de ellos con un significado y longitud diferente. De ellos, el **Event Select** (bits 0-7) se encarga de determinar el evento a ser contado y el **Unit Mask** (**UMASK**, bits 8-15) se encargada de la clasificación de las condiciones. Además, también hay campos de parametrización (bits 16-23), encargados de la configuración del monitoreo en base a distintos parámetros como el filtrado por privilegios (eventos en modo usuario USR o privilegiado OS) o la gestión de las interrupciones y excepciones, entre otros aspectos. Por último, el campo **Counter Mask** (**CMASK**, bits 24-31), que permite incrementar el registro contador en caso de que el valor de este sea mayor que el valor almacenado en el campo CMASK.

Los registros MSR de **selección de eventos** se emparejan con un **registro contador**, que es el que lleva la cuenta de los eventos propiamente. Para saber qué eventos están disponibles en un procesador determinado es necesario acceder al manual de profiling de la CPU en cuestión, en donde se indica el nombre del evento y una descripción del mismo. Un problema que presentan estos registros es que son limitados y en caso de que haya más eventos a medir que registros, es necesario realizar **multiplexación en el tiempo** para poder realizar todas las mediciones, con la consiguiente pérdida de precisión.

Desde su introducción, la PMU no ha dejado de evolucionar y aumentar el nivel de detalle de los resultados que proporciona, centrándose en partes más concretas de la microarquitectura. La Figura 2.2 recoge la evolución de los contadores hardware en las diferentes microarquitecturas de la arquitectura Intel Core, ordenadas de izquierda a derecha por orden temporal. En ella se diferencian tres tipos de contadores. Por un lado, los clasificados como **Core** reflejan los contadores hardware que miden eventos relacionados con un core **lógico**. Por su parte, los mencionados como **Uncore** reflejan los eventos que miden aspectos compartidos entre los diferentes cores **físicos** del procesador. Por último, los **Offcore** se encargan de aspectos que están fuera de los cores de la CPU. En la figura se puede comprobar que el número de eventos va aumentando paulatinamente hasta llegar a Broadwell, en el que el incremento fue sustancialmente mayor. Sin embargo, a partir de este, el número de eventos desciende, probablemente porque algunos de los eventos no serían utilizados, o estarían obsoletos.

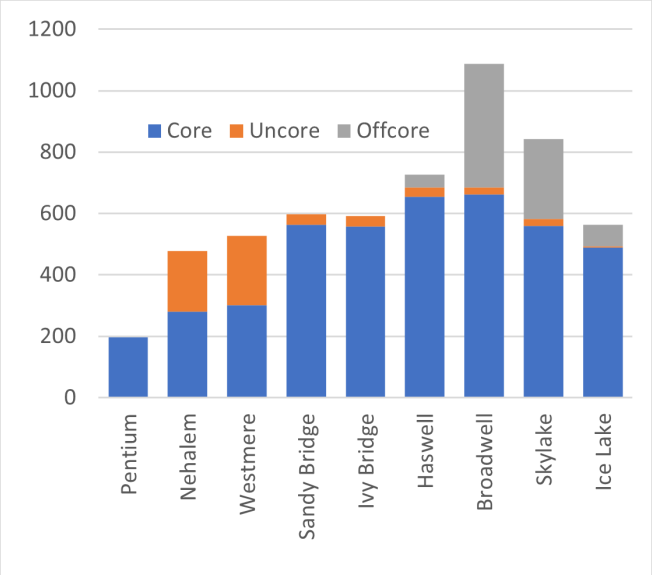


**Figura 2.2:** Evolución número de eventos en procesadores Intel Core.

De una forma similar a lo que ocurre en las CPU, en la GPU se reproduce esta forma de análisis y muestreo de resultados, con ligeras diferencias. En el caso de las GPUs de NVIDIA, los Performance Counters o contadores de rendimiento son los encargados de coleccionar valores de contador de rendimiento durante la ejecución de un determinado kernel (código que se ejecuta en la GPU, descrito en el capítulo 3). Estos contadores son actualizados en tiempo de ejecución y pueden ser consultados por el usuario, a través de las herramientas proporcionadas por NVIDIA, una vez se haya terminado el análisis de todos y cada uno de los kernels que van a ser medidos.

Las GPUs de NVIDIA recogen los datos de diversas maneras [11], utilizando para ello diferentes mecanismos de recolección, con sus ventajas y desventajas:

1. **HWPM**: Mecanismo de profiling que permite recolectar datos de cualquier unidad hardware de la GPU, pero con la desventaja de que solo puede recopilar la información de un subconjunto de unidades hardware de un mismo chip al mismo tiempo.
2. **SMPC**: Mecanismo que permite recolectar datos únicamente de **Streams Multiprocessors** (**SMs**), pero que presenta la ventaja de que puede monitorizar todos los SMs de la GPU al mismo tiempo.

A diferencia de lo que ocurre en las CPUs en donde se es posible acceder a la unidad mínima de almacenamiento y gestión de la PMU (registro), en el caso de NVIDIA es distinto, pues solo permite trabajar con la librería **CUDA Profiling Tools Interface** (**CUPTI**) que permite realizar el profiling de aplicaciones CUDA [12].

Los contadores hardware disponibles en las GPUs de NVIDIA son limitados, tal y como ocurre en las CPUs, y además, su uso depende del mecanismo utilizado para coleccionar sus valores. En estos casos, las herramientas de profiling consiguen medir todos los contadores necesarios volviendo a ejecutar el mismo kernel, repitiéndose sucesivas veces el mismo procedimiento hasta obtener todos los valores [31]. Esto supone algunos problemas, como la dificultad de mantener el estado del hardware lo más similar posible entre ejecuciones, y las posibles diferencias en las ejecuciones que puedan inducir a errores en los resultados. Por otro lado, el hecho de tener que repetir varias veces cada kernel en caso

de no poder completar las medidas, supone un sobrecoste a medida que el número de replicaciones aumentan. Aunque el objetivo de estos análisis no es el rendimiento en el proceso de medida, sí que puede convertirse en un problema en el caso de que los kernels sean muy lentos, o haya excesivo número de ellos.

Además de las diferencias anteriores, los tipos de datos arrojados por las unidades de la CPU y la GPU son ligeramente diferentes. Las primeras arquitecturas de GPUs de NVIDIA ofrecen dos tipos de mediciones: **eventos** y **métricas**. El primero de ellos es similar al caso de las CPUs, es decir, valores computados en base a una medición respecto a una unidad lógica determinada o alguna característica del kernel. Las métricas por otro lado se obtienen a través de la combinación de uno o más eventos, y ofrecen resultados que pueden ser computados como ratios, throughputs u otros valores característicos utilizados para medir el rendimiento de un determinado kernel (como por ejemplo el IPC), eficiencias en la jerarquía de memoria, en saltos etc.

La utilización por parte de NVIDIA de métricas ofrece al usuario la posibilidad de simplificar las tareas de profiling. Sin embargo, este tipo de medición implica indirectamente trabajar, generalmente, con dos eventos por métrica (aunque en algunos casos con métricas sencillas, solo es necesario un evento). Esto puede suponer un problema de rendimiento debido a que la utilización de métricas puede provocar que el número de repeticiones o replays sea elevado. Esto es debido a la alta utilización de eventos y registros que hacen las métricas.

Esta forma de análisis de rendimiento, diferenciando entre eventos y métricas, ha sido utilizada por NVIDIA durante años. No obstante, este modelo de análisis fue unificado en la arquitectura Turing (2018). Esta eliminó los eventos y los integró en las métricas, eliminando el concepto de evento y añadiendo una gran variedad de características.

El número de métricas en las GPUs ha aumentando considerablemente desde la unificación de eventos y métricas. La Figura 2.3 recoge la evolución del número de métricas en las diferentes arquitecturas de NVIDIA. En el caso de Turing, solo se tienen en cuenta las métricas relativas al SM y a la subpartición (del SM).

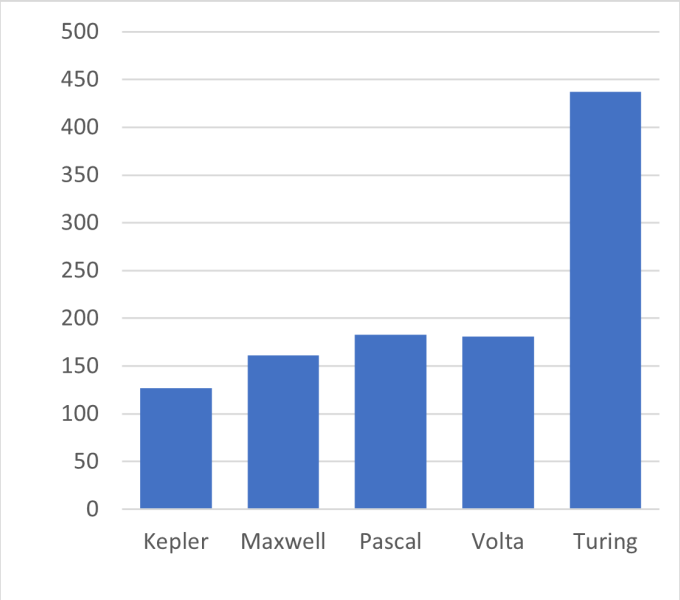


**Figura 2.3:** Evolución métricas en arquitecturas NVIDIA.

Tal y como se puede apreciar, la tendencia muestra valores similares entre las arquitecturas Kepler y Volta. Sin embargo, esta tendencia cambió radicalmente a partir de Turing, aumentando considerablemente el número de métricas, permitiendo al usuario obtener una mayor información sobre cómo se está comportando la GPU.

## 2.2. Herramientas de Alto Nivel

En la práctica, la utilización de los contadores hardware a bajo nivel resulta costosa a la hora de realizar el profiling de un determinado proceso, tanto en el caso de las CPUs como en el caso de las GPUs. En el primero de los casos, la gestión se realiza a nivel de registro, como se ha mencionado en el apartado anterior. Por otro lado, en las GPUs de NVIDIA lo máximo que se dispone es la librería CUPTI, algo más cercana al programador que el caso anterior. No obstante, ambos modelos son costosos a la hora de configurar y gestionar, por lo que es conveniente añadir una **capa de abstracción** entre las unidades hardware de monitorización y las aplicaciones a ser medidas. El objetivo es el de ocultar la mayor parte de los aspectos de configuración, además de unificar diferentes microarquitecturas.

En el caso de la CPU, una aplicación que implementa esta abstracción es **perf** [13]. La herramienta (que utiliza perf_events integrados en el kernel de Linux) está basada en una interfaz por línea de comandos (CLI) y permite realizar el profiling de aplicaciones utilizando los contadores hardware. Dispone de dos modos de análisis, el **sampling** y el **counting**. El primero de ellos consiste en la generación de una muestra de información tras la ocurrencia de un número determinado de eventos. Por su parte, el segundo es utilizado para coleccionar cuentas de eventos, que son especificados mediante los valores vistos en los MSR de selección de eventos (Event Select y UMASK). Adicionalmente, perf incluye alias para ciertos eventos comunes, por ejemplo, el evento de medición de las instrucciones ejecutadas queda recogido bajo el alias “instructions”, lo que permite utilizar este nombre para las diferentes microarquitecturas que perf soporta, sin necesidad de conocer los valores del MSR para ese evento concreto.

Por otro lado, en las GPUs de NVIDIA se utiliza la herramienta **nvprof** [14] (disponible en Linux, Windows, y OS X) que proporciona una interfaz por línea de comandos (CLI) para realizar el análisis de rendimiento de las aplicaciones ejecutadas por la GPU. Dispone de un modo por defecto en el que se muestra una visión general de los kernels de la GPU y las copias de memoria CPU – GPU (y vicerversa) realizadas. Adicionalmente, también se puede realizar un análisis utilizando los contadores hardware vía eventos y métricas. Esta herramienta es utilizada por las GPUs que disponen de ambas opciones (eventos y métricas) para realizar el profiling, hasta la arquitectura Volta. Para arquitecturas posteriores (a partir de Turing), se utiliza la herramienta **ncu** (**Nsight Compute CLI** [15]), que proporciona una interfaz por línea de comandos similar a la anterior. Esta herramienta realiza un análisis por defecto de cada kernel en el que se arroja información muy orientada al programador dividida en tres apartados. En primer lugar, se muestran resultados generales de la GPU en cuanto a frecuencias y utilización de las unidades de la misma (memorias cache, SM, memoria...). A estos resultados se les pueden añadir comentarios que arrojan más información sobre cómo ha sido la ejecución en términos de **throughput** y **ancho de banda de memoria**, respecto del rendimiento de pico de la GPU. En segundo lugar, se pasa a mostrar las características de la ejecución en cuanto a cómo ha sido programada la GPU durante la ejecución (tamaños de grid y bloque, número de threads, registros por threads; utilizaciones de la memoria por bloque...). Por último, se incluye un análisis de resultados basados en la ocupación (límites de bloque de SM, registros, memoria compartida, warps; ocupaciones conseguidas por warp; ocupaciones teóricas máximas por SM...).

Además de las versiones con interfaz de línea de comandos, existen otras basadas en interfaces gráficas de usuario (GUIs), que permiten visualizar y entender los resultados de forma más sencilla.

En el caso de las CPUs, una herramienta (de Intel) es **VTune** [16], utilizada para la optimización y configuración del sistema en multitud de campos como HPC (High Performance Computing), entornos de Cloud, almacenamiento... El objetivo de esta herramienta es el de analizar los cuellos de botella y realizar optimizaciones de rendimiento de un código fuente, mostrando al usuario el tiempo de CPU de las diferentes partes del código. De forma similar, NVIDIA también proporciona herramientas basadas en interfaces gráficas de usuario. Por un lado, las arquitecturas de NVIDIA que disponen de métricas y eventos emplean la herramienta **Visual Profiler** [17]. El objetivo de esta es optimizar el código fuente, y permite al usuario detectar rápidamente los cuellos de botella temporales en su aplicación, a través de elementos gráficos intuitivos. Además, muestra información detallada de cómo ha resultado la ejecución, especificando las llamadas de API que han sido utilizadas durante la ejecución, además de las transferencias de memoria entre la CPU y GPU y viceversa. Por último, también ofrece datos relativos a métricas de rendimiento, además de valores relativos a la potencia consumida y los valores de las frecuencias de reloj durante la ejecución de la aplicación.

Finalmente, las arquitecturas más recientes utilizan las herramientas **NVIDIA Nsight Systems** [18] y **NVIDIA Nsight Compute** [19] (versión gráfica). La primera es utilizada para que los desarrolladores puedan optimizar su software. Para ello, permite a estos identificar los cuellos de botella y propone optimizaciones que pueden mejorar el rendimiento de su software. Para realizar esto, la herramienta tiene en cuenta aspectos no solo de la GPU, sino que también tiene en cuenta la/s CPU/s interconectada/s. En cuanto a la segunda, el objetivo es el de realizar un análisis de rendimiento de cada uno de los kernels que componen la aplicación, así como sus respectivos procesos hijo. Para ello, la herramienta proporciona resultados detallados de **métricas de rendimiento** y **depuraciones del código**. Los resultados ofrecidos son similares a los de la versión por línea de comandos, pero mucho más intuitiva en su uso y visualización.

No obstante, a pesar de que las herramientas anteriores suponen un gran avance a la hora de realizar el análisis de un código fuente, estas suelen estar ligadas al **programador** y a cómo mejorar su código, "obviando" la microarquitectura. Es por esto que es necesario diseñar modelos de análisis que estén más enfocados en cómo es el comportamiento del hardware, es decir, cómo se comporta la microarquitectura con respecto a la ejecución de un determinado código fuente. El objetivo de las herramientas anteriores es justo el contrario, es decir, comprobar cómo se adapta el código al hardware.

## 2.3. Metodología TopDown CPU

Las CPUs actuales son sistemas complejos: superescalares, con ejecución fuera de orden, especulación... Entender el impacto de los distintos componentes de la microarquitectura en la ejecución de un código es complicado, y han surgido distintas soluciones entre la que destaca la propuesta de Intel, conocida como **metodología TopDown** [8].

La metodología TopDown describe un mecanismo que identifica los cuellos de botella en los procesadores **fuera de orden**. Para realizar esta acometida, se hace una división arquitectural de la CPU en **frontend** y **backend**. El primero es el encargado de realizar el fetch de las instrucciones desde memoria y traducirlas en micro-instrucciones que son enviadas al backend para su posterior ejecución. Por su parte, el backend recibe las instrucciones del frontend y es el responsable de planificar qué instrucciones deben de ser ejecutadas en un momento determinado, ejecutarlas y restablecer el orden original y lógico del programa. A su vez, frontend y backend pueden ser divididos en diferentes etapas, que serán desglosadas a medida que se aumenta el nivel de detalle en el TopDown.

El objetivo de esta metodología es analizar dónde se ha producido una pérdida de rendimiento en cada una de estas etapas, a través de contadores hardware. Una vez que una instrucción es seleccionada para ser ejecutada, esta puede ser completada o no. Los errores en la predicción de saltos pueden conllevar a que varias instrucciones no sean completadas, ya que al determinarse que el salto es

incorrecto, las instrucciones ejecutadas especulativamente son eliminadas del pipeline. Esto implica que de estas instrucciones no se hará el **retire**, es decir, no se completará su ejecución y no se actualizarán las estructuras que fueran necesarias (memoria, registros...). Todas las instrucciones, hagan el retire o no, consumen **slots** o recursos del pipeline una vez que son seleccionadas para ser ejecutadas. Lo que se mide como pérdida de rendimiento son los ciclos perdidos o **stalls** donde la CPU no ha estado ejecutando ninguna instrucción, alejándose del IPC ideal, y se trata de localizar su procedencia (por ejemplo, error en la predicción de saltos, fallos en la cache de instrucciones; ir a buscar un dato a memoria...). En la Figura 2.4 se puede ver la división en niveles implementada en la metodología TopDown.

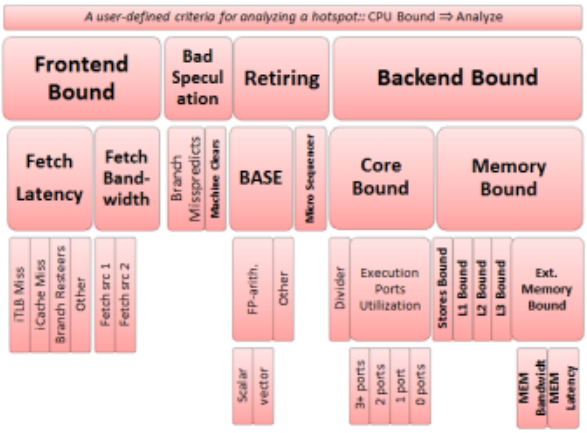


**Figura 2.4:** Jerarquía de análisis TopDown obtenida de la referencia [8].

Para el primer nivel de la jerarquía TopDown (superior en la Figura 2.4), tenemos los siguientes componentes:

1. **Frontend Bound**: El Frontend es la primera parte del pipeline de la CPU, encargada de determinar la siguiente instrucción a ejecutar, hacer el fetch de la instrucción elegida y traducirla a micro-operaciones que serán posteriormente ejecutadas por el Backend. En la metodología Topdown, el comportamiento del predictor de saltos entra en otra categoría. En esta porción de la jerarquía, se tienen en cuenta problemas de retraso en el fetch de instrucciones, como por ejemplo fallos en la cache de instrucciones (i-cache), denotado en niveles inferiores como **Frontend Latency Bound**; o ineficiencias en la decodificación de la instrucción (denotado en niveles inferiores como **Frontend Bandwidth Bound**). En definitiva, el Frontend Bound denota las partes en las que se está produciendo un cuello de botella para siguientes partes del pipeline, como por ejemplo las unidades funcionales, que no realizarán la ejecución de ninguna micro-operación, lo que se traduce en una pérdida de IPC.

2. **Bad Speculation**: En esta parte se recoge la pérdida de rendimiento provocada por la especulación incorrecta. Este perjuicio puede ser producido principalmente por dos razones: los ciclos utilizados por parte de la CPU en una instrucción en la que no se va a hacer el retire, es decir, no se va a completar; y los ciclos dedicados a restablecer el estado del procesador previo al fallo de predicción. Un ejemplo podría ser un error en el predictor de saltos, en el que entrarían todas las instrucciones tras el salto. No solo son instrucciones en distintas fases de ejecución sobre las que no se va a hacer retire, si no que el proceso de recuperación del estado de la CPU es complejo.

3. **Retiring**: Esta categoría representa las instrucciones en las que se va a hacer el retire, es decir, su ejecución se va a completar. A medida que este valor se va aproximando a su máximo (100 %),

el rendimiento se aproxima al ideal por ciclo del sistema analizado. En consecuencia, el objetivo será el de obtener un valor lo más alto posible.

4. **Backend Bound**: Ciclos perdidos en los que el backend no procesa, es decir, no realiza la operación de ninguna instrucción debido a no disponer de recursos para ello. En esta categoría se encuentran por ejemplo los bloqueos provocados por fallos en la cache de datos (que provoca que haya que ir a buscar el dato a memoria, o a niveles de cache inferior), denotado en niveles inferiores por **Memory Bound**; o por no disponer de unidades funcionales para realizar una determina operación (por ejemplo, operaciones multiciclo que mantienen una unidad funcional ocupada), denotado en niveles inferiores como **Core Bound**.

A medida que el nivel de TopDown va descendiendo también se aumenta el detalle de los resultados, focalizándose en partes específicas de la microarquitectura (como los distintos niveles de la jerarquía cache, o las unidades de ejecución del procesador). El objetivo es ir de lo general a lo específico, dividiendo las partes más generales en partes más concretas que se acerquen a las unidades que constituyen pérdidas de rendimiento en una CPU.

El objetivo de este documento es el de intentar replicar el modelo de análisis propuesto por Intel para sus CPUs, pero utilizando GPUs de la marca NVIDIA. Para ello se intentará seguir una estructura similar a lo propuesto anteriormente. La microarquitectura de una GPU de NVIDIA sigue también una estructura en **pipeline**, por lo que, teniendo en cuenta el funcionamiento de esta, se podrán identificar las partes microarquitecturales que producen pérdidas de rendimiento, respecto de una **ejecución teórica ideal**. A partir de estas partes se construye el modelo jerárquico, basado en niveles, similar al de Intel.

# Capítulo 3

# Arquitectura de una GPU NVIDIA

Con el auge de las GPUs de propósito general (GPGPU), siguiendo un modelo de programación parecido al de la CPU, utilizando lenguajes de programación de alto nivel con funciones de librería que soportaban GPGPU (OpenCL, OpenACC, Accelerated OpenMP, SyCL...); NVIDIA unificó en el año 2007 el modelo de programación de sus GPGPU en torno a una nueva arquitectura desconocida hasta la fecha, denominada **CUDA** (**Compute Unified Device Architecture**). Desarrollada y gestionada por NVIDIA, específica para GPUs de NVIDIA (no siendo compatible con ningún otra GPU de otra marca), es una plataforma de computación paralela y modelo de programación, cuyo objetivo es el de dotar a los programadores de un modelo de programación que permita acelerar de forma considerablemente las aplicaciones, aprovechándose de la potencia de cómputo que ofrece la GPU.

CUDA es una extensión del lenguaje de programación C, que desde el punto de vista del programador es una **librería** y un conjunto de **tokens adicionales** que le permiten escribir código con relativa facilidad que se ejecuta en la GPU de NVIDIA. Una de las características de CUDA es la de dividir el código en dos partes: **host** y **device**. El primero de ellos se refiere al código que es ejecutado sobre la CPU, mientras que el segundo es ejecutado directamente sobre la GPU, denominado **kernel**. Este es una función de C, que recibe una serie de argumentos que son pasados desde el host. Cada kernel es compilado al ensamblador de NVIDIA, **PTX** (**Parallel Thread Execution**) [20]. Para invocar a cualquier kernel, tan solo basta con llamar la función que va a ser ejecutada en la GPU. Estas funciones están bien definidas, a través de los tokens correspondientes, para indicar que son ejecutadas sobre la GPU. Estos reciben el nombre de **cualificadores de función** (__host__, __device__...). Del mismo modo, además de la definición apropiada de la función, la invocación del kernel también debe de incluir **la forma en la que el kernel es ejecutado** (dimensiones y forma del grid).

Desde el punto de vista del programador, la GPU está dividida en un **grid**, con una serie de **bloques de threads**, con un conjunto de **threads** por cada bloque (generalmente 1024 o 2048); tal y como se recoge en la Figura 3.1.

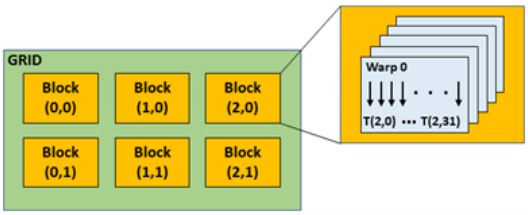


**Figura 3.1:** Ejemplo de un GRID.

El programador debe de tener especial cuidado a la hora de determinar el número de bloques y threads por cada bloque debido a que estos se ejecutan sin un orden fijo. Además, son concurrentes y asíncronos, multiplexándose en el tiempo.

De igual manera, el número de threads por bloque utilizados debe de ser seleccionado teniendo en cuenta que los threads de cada bloque posteriormente se ejecutan agrupados en **warps**, que es un conjunto de threads (**32** en NVIDIA por defecto) que ejecutan la misma instrucción sobre diferentes datos; por lo que un valor inferior a 32, o que no sea múltiplo de este provocará que no se aprovechen los warps de la forma correcta (por ejemplo, puede haber warps en los que se ejecuten menos de 32 threads).

Para facilitar el modelo de programación, el programador puede acceder a cada uno de los threads a través de sus **identificadores**, que son calculados utilizando las variables proporcionadas por CUDA (threadIdx.x/y/z, blockIdx.x/y/z; blockDim.x/y/z y gridDim.x/y/z).


**Figura 3.2:** Organización de NVIDIA PASCAL GP100, obtenido de la referencia [6].

## 3.1. GPU: Estructura Hardware

El objetivo de una GPU es el de obtener el máximo paralelismo de datos que sea posible. La Figura 3.2 muestra cómo es la organización hardware de una GPU moderna.

La GPU está dividida en **GPCs** o **Graphics Processing Clusters** (6 en el caso de la figura anterior). Sobre estos clústeres se ejecutan las instrucciones, compartiendo entre todos ellos una **memoria cache de nivel L2**, común a todos los GPCs. A su vez, cada GPC está dividido en TCPs (Texture Processing Clusters), que a su vez están divididos en **SMs** o **Stream Multiprocessor** (generalmente 2 SMs por cada TCP), que es la unidad hardware básica de la GPU, con una **L1 de instrucciones** y **datos** para cada SM. Finalmente, toda la GPU está conectada al computador a través del bus de alta velocidad **PCI Express** (PCIe o PCI-e). Este se comunica con los (8) **controladores de memoria**, encargados de las transferencias de memoria con la CPU y entre la misma GPU.

## 3.2. Stream Multiprocessor

Una vez que el kernel es definido por el programador, el hardware de la GPU es el encargado de planificar la ejecución de acuerdo con las dimensiones de threads y bloques establecidos por el programador. Para ello, la GPU calcula los **recursos requeridos** por cada bloque de threads y los asigna a los diferentes SMs que componen el GPC, siguiendo generalmente un algoritmo de **round robin**.

El **SM** o **Stream Multiprocessor** constituye la unidad hardware encargada de realizar la selección, planificación y ejecución de los warps (threads). Este a su vez puede estar dividido en **subparticiones** de iguales características y componentes. Cada SM recibe un conjunto de bloques de threads que deben de ser ejecutados (con un máximo de 1024/2048 threads por bloque). Para ejecutarlos, los threads del bloque son agrupados en **warps**, con 32 threads por warp en el caso de NVIDIA (concepto arquitectural definido por defecto). Un ejemplo de SM es el recogido en la siguiente figura:


**Figura 3.3:** Stream Multiprocessor NVIDIA PASCAL GP100, obtenido de la referencia [6].

Una vez que el SM dispone de los warps para ser ejecutados, se debe de determinar qué warp es ejecutado en cada una de las subparticiones del SM. Para ello se dispone del **Warp scheduler** o **planificador de warp**. Para realizar esta función, habitualmente los planificadores siguen un algoritmo **Greedy Then Oldest** (**GTO**) [21], que consiste en elegir el warp que más tiempo lleva sin haber sido ejecutado, si este está disponible (hay recursos disponibles para él).

Escogido el warp a ser ejecutado, este pasa a la **Dispatch Unit** en la que se realiza el dispatch de una instrucción del warp. Algunos SMs pueden proporcionar dos unidades por cada planificador de warp, lo que permite realizar el dispatch de **dos instrucciones** del warp.

A continuación, se realiza la asignación de los registros fuente y destino en el **banco de registros**. En cada instrucción del warp se generan 32 operaciones, por lo que el banco de registros debe de tener un tamaño considerable para poder satisfacer todas estas operaciones. Los 32 threads del warp necesitan disponer de registros suficientes para poder realizar las operaciones. Es por esto por lo que los bancos de registros de una GPU (uno por cada subpartición, según lo visto en la Figura 3.3) tienen un tamaño muy superior al de un banco de registros de una CPU.

Finalmente, la instrucción del warp es ejecutada y pasada a las unidades funcionales. Estas son las encargadas de realizar la operación determinada. En condiciones normales, la instrucción del warp es ejecutada por los 32 threads simultáneamente. Cada una de las SM está formada por una serie de unidades funcionales, siendo iguales para todas las subparticiones. Las unidades por SM pueden ser las siguientes:

1. **FP64** (**DP** Unit en la figura): Unidad para realizar operaciones de coma flotante en doble precisión (32 en la figura).
2. **INT** (**Core** en la figura): Unidad para realizar operaciones con enteros (64 en la figura).
3. **FP32** (**Core** en la figura): Unidad para realizar operaciones de coma flotante en simple precisión (64 en la figura).
4. **LD/ST** (Load/Store): Unidades para realizar lecturas/escrituras en memoria (16 en la figura).
5. **SFU** (Special Function Units): Unidad para realizar operaciones matemáticas específicas, tales como operaciones trigonométricas (cálculo de seno, coseno y tangente), raíces cuadradas, exponentes, logaritmos... Del mismo modo, esta unidad también es utilizada para entrenar largas redes neuronales (16 en la figura).
6. **Texture Unit**: Unidades especializadas para realizar operaciones con texturas. Optimizado para gráficos (4 en la figura).

En arquitecturas más modernas, como la Turing, ya es posible observar nuevas unidades funcionales, como los **Tensor Cores**, utilizados para acelerar operaciones con matrices como las utilizadas en Machine Learning. Adicionalmente, las unidades llamadas Cores (Figura 3.3) se dividen en unidades de manejo de enteros (INT) y flotantes de precisión simple (FP32).

Cada subpartición del SM dispone de un **buffer de instrucciones** (o **L0 Instruction cache**) privado. El objetivo de esta cache es el de agilizar el proceso de dispatch de la instrucción del warp. Adicionalmente, cada SM dispone de una **cache L1 de datos** (***shared memory***) y de **instrucciones**, privada al SM, pero común a todas las subparticiones. Esta memoria es utilizada para intercambiar datos entre los threads de un mismo bloque.

Por último, la Figura 3.4 recoge un resumen (simplificado) del pipeline del SM. La estructura de esta es similar a la descrita anteriormente, utilizando una única unidad de dispatch por cada planificador del warp.

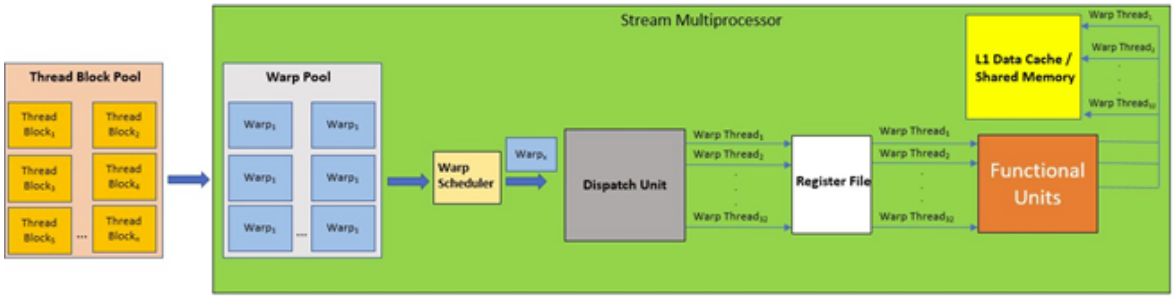


**Figura 3.4:** Estructura (simplificada) de un SM.

# Capítulo 4

# TopDown sobre GPUs NVIDIA

La metodología TopDown está desarrollada para CPUs (de Intel). El objetivo de este documento es el de intentar replicar esta forma de análisis en las GPUs de NVIDIA. Para ello, se seguirá el modelo de análisis establecido por Intel, intentando adaptarlo a las GPUs de NVIDIA, con las diferencias que conlleva una CPU de una GPU.

La forma de análisis es similar a la de la CPU. Se establece una metodología basada en niveles que van de lo general a lo específico. A medida que se aumenta el nivel de detalle, la precisión de los resultados también aumenta. Para poder realizar el análisis, al igual que ocurre en la metodología de la CPU, se hará uso de los **contadores hardware** de la GPU, tal y como se especifican en la guía de profiling de NVIDIA [14, 15]. El objetivo es el de identificar las pérdidas de rendimiento que se producen en las diferentes partes arquitecturales de la GPU, respecto de una **ejecución teórica ideal**, en la que se obtendría el máximo rendimiento en la GPU, es decir, cuando el IPC (Instrucciones por Ciclo) es el máximo soportado por la GPU. Para que esta situación se llegue a producir, todas las operaciones de cada uno de los threads que componen los warp de todos y cada uno de los SMs (y sus correspondientes subparticiones) deben de ejecutarse en un ciclo de reloj. En consecuencia, este valor será proporcional al número de operaciones del warp ejecutadas en los SMs.

A la hora de la ejecución de un determinado proceso sobre la GPU, hay dos fuentes principales de pérdida de rendimiento, y por ende, pérdida de IPC. Por un lado, una fuente de pérdida de rendimiento son los **stalls**. Estos se producen cuando una parte determinada (o varias) del pipeline de la GPU no puede realizar ninguna operación (está bloqueada) debido a que no dispone de recursos necesarios para realizarla. Por ejemplo, un stall podría ser no disponer de unidades funcionales para realizar una determinada operación por estar todos ocupados realizando otras. En este caso, el pipeline debe bloquearse (hay un stall) y la operación no puede empezar a ejecutarse hasta que no se dispongan de las unidades funcionales necesarias para poder llevarla a cabo. Por otro lado, la otra fuente de pérdida de rendimiento (IPC) es la **divergencia**. Esta se produce cuando no hay un aprovechamiento de todos los threads de un warp, por ejemplo, debido a un salto condicional. O cuando existe una repetición del código (por un salto IF-ELSE o por el uso de una unidad funcional de la que no se tienen suficientes copias). Se diferencian de los stalls en que en este caso el pipeline de la GPU no está bloqueado, sino que no se aprovecha como debe debido a la ejecución del código.

El objetivo de la metodología TopDown será, por tanto, identificar los stalls en las diferentes partes del pipeline de la GPU, así como las pérdidas de rendimiento ocasionadas por la divergencia. Para ello, se puede considerar el máximo rendimiento (IPC) como:

$$IPC_{MAX} = IPC_{STALL} + IPC_{DIVERGENCE} + IPC_{RETIRE} \tag{4.1}$$

En donde $IPC_{STALL}$ denota el IPC perdido debido a los stalls (o bloqueos), $IPC_{DIVERGENCE}$ se refiere al perdido a causa de la divergencia e $IPC_{RETIRE}$ se trata del rendimiento o IPC real, obtenido por la aplicación/proceso ejecutado. El objetivo de la metodología es obtener los valores anteriores, identificando en qué partes arquitecturales se producen esas pérdidas. Para ello, a continuación, se van a detallar y especificar los diferentes niveles de análisis propuestos.

## 4.1. La jerarquía

La jerarquía de la metodología TopDown para GPUs escogida en este trabajo se encuentra recogida en la Figura 4.1. Las dos fuentes de pérdida de rendimiento denotadas en el apartado anterior deben de quedar recogidas en el primer nivel de análisis (nivel más alto de la jerarquía). Con niveles de análisis más profundos, se ofrece más detalle en el origen de la pérdida de rendimiento.

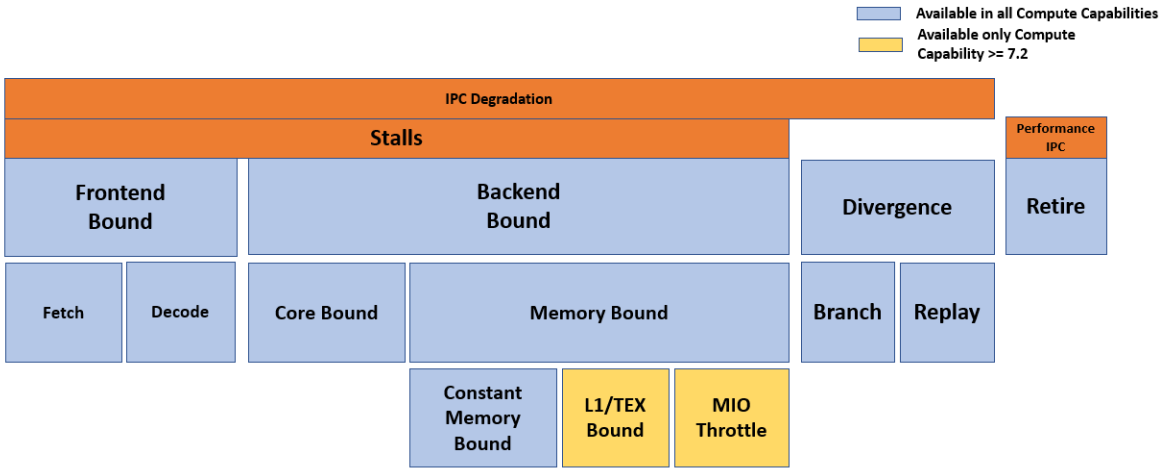


**Figura 4.1:** Jerarquía TopDown GPUs NVIDIA.

Los niveles son jerárquicos, es decir, los resultados de un nivel inferior también incluyen los resultados de niveles superiores, más generales y con menos detalle. Los límites en cuanto al número de resultados que pueden obtenerse, así como el nivel de detalle que se puede alcanzar, dependen de los contadores hardware que proporciona NVIDIA. Esto sobre todo es notado a medida que se va aumentando el nivel de detalle, donde los valores se focalizan en partes hardware concretas, lo que conlleva a contadores específicos sobre esa parte. Esto es comprobable en la jerarquía propuesta, en donde hay partes que solo pueden ser calculadas utilizando arquitecturas más recientes (a partir de Volta), denotadas en amarillo en la figura anterior.

## 4.2. Retire

La metodología TopDown es una forma de análisis que permite determinar los cuellos de botella microarquitecturales, en nuestro caso, de una GPU. No obstante, también es necesario calcular el rendimiento que obtienen las aplicaciónes, ya que, comparando el rendimiento máximo que puede soportar una GPU con el obtenido, se pueden calcular las pérdidas que se han producido. Durante este apartado se tratará de determinar cómo se obtienen ambos valores (rendimiento máximo y rendimiento real).

En un primer lugar, es necesario obtener el valor máximo teórico de rendimiento de una GPU, es decir, el rendimiento de una **ejecución teórica ideal**. Este valor vendrá determinado por el uso más eficiente posible de cada Stream Multiproccessor (SM), cuando se puedan aprovechar eficientemente todas las subparticiones que componen esta unidad. El máximo rendimiento viene determinado por el IPC en su máximo exponente, que se producirá cuando se consigan ejecutar, por cada ciclo de reloj, tantas instrucciones como un SM sea capaz. Si es posible, cada unidad de dispatch del SM enviará a las unidades funcionales una instrucción del warp. Entonces, el IPC máximo se producirá cuando se puedan enviar a las unidades funcionales, tantas instrucciones de warp como número de unidades de dispatch tenga el SM. Por ejemplo, dado el SM de la Figura 3.3 visto en el apartado anterior, hay dos subparticiones en las que en cada una de ellas hay dos unidades de dispatch por cada planificador de warp. Teniendo en cuenta lo mencionado anteriormente, el máximo rendimiento o IPC máximo sería 4.0.

Por otra parte, el **retire** determina el rendimiento real obtenido por la aplicación sobre el total, es decir, el IPC, definido como el número de instrucciones completas (que usan los 32 threads del warp) que se ejecutan por ciclo en una SM. Hay que tener en cuenta que la métrica que nos proporciona el IPC de la herramienta indica el número de instrucciones ejecutadas en un SM por ciclo, pero no tiene en cuenta si se hace uso de todos los threads del warp, o si por el contrario sólo usan unos pocos. Es por esto que hay que tener en cuenta, para cada instrucción, los threads del warp que realmente hacen uso de las unidades funcionales. Dicho de otro modo, el retire puede ser calculado de acuerdo con la siguiente expresión

$$Retire = IPC \cdot Eficiencia_{Warp} \tag{4.2}$$

Para determinar los elementos de la expresión anterior es necesario hacer uso de los contadores hardware, en donde se deben de identificar los eventos y/o métricas que calculan los valores anteriores. Estos dependen de la microarquitectura de la GPU utilizada en el análisis. Diferentes arquitecturas pueden emplear diferentes nombres en sus eventos/métricas. En el caso de las GPUs de NVIDIA la diferenciación de los valores medibles depende de las características de la arquitectura, denotadas a través de la denominada Compute Capability (CC) [22]. Este valor numérico es utilizado para denotar las características y especificaciones generales de la GPU. Las GPUs que comparten este valor utilizan las mismas métricas/eventos. No obstante, muchos de los valores anteriores son compartidos entre diferentes CCs, lo que facilita su identificación para la mayor parte de las arquitecturas. Las Tablas 4.1 y 4.2 recogen las métricas necesarias para obtener estos valores, de acuerdo con la arquitectura de la GPU.

| **Métrica** | **Abreviatura** | **Descripción** |
|---|---|---|
| ipc | IPC | Promedio de instrucciones por ciclo ejecutadas, por SM. |
| warp_execution_efficiency | $Eficiciencia_{Warp}$ | Relación entre el promedio de threads activos por warp con el número máximo de threads por warp. |

**Tabla 4.1:** Definición métricas Retire (CC menor que 7.2).

| **Métrica** | **Abreviatura** | **Descripción** |
|---|---|---|
| sm__inst_executed.avg.per_cycle_active | IPC | Promedio de instrucciones por ciclo ejecutadas, por SM. |
| sm__thread_inst_executed_per_inst_executed.ratio | $Eficiciencia_{Warp}$ | Relación entre el promedio de threads activos por warp con el número máximo de threads por warp. |

**Tabla 4.2:** Definición métricas Retire (CC mayor o igual que 7.2).

Como se ha mencionado anteriormente, el retire constituye el rendimiento obtenido por la aplicación. Una vez obtenido este valor, el siguiente paso es determinar las pérdidas de rendimiento que se han producido. Durante las siguientes secciones se hace una división de las posibles pérdidas de rendimiento que puede experimentar una GPU, de acuerdo a la división del TopDown realizada en el apartado anterior.

## 4.3. Divergencia

Existen dos causas de pérdida de rendimiento que se contemplan dentro del apartado de Divergencia: la ejecución de saltos condicionales, y la repetición de instrucciones. Cuando en la ejecución del código se llega a un **salto condicional**, se pueden dar dos situaciones diferentes. Por un lado, en una ejecución IF sin ELSE, en el warp hay threads con flujos diferentes, es decir, unos threads ejecutan instrucciones que otros threads dentro del mismo warp no tienen que ejecutar (los que entran dentro del IF frente a los que no lo hacen). Esta situación implica que puedan desaprovecharse algunas de las unidades funcionales de la GPU, debido a que en un warp todos los threads tienen que ejecutar la misma instrucción (sobre diferentes datos). Por otro lado, en una ejecución de un salto IF con ELSE, en un warp hay threads que ejecutan una instrucción (o varias) mientras que otros ejecutan otra/s. Este caso es similar al anterior, ya que no se aprovecharán todos los cores de la GPU, pero además le añade otra fuente de pérdida de rendimiento y es que se tardará el doble de ciclos en ejecutar, puesto que hay que ejecutar las dos partes.

Un ejemplo ilustrativo de una situación de divergencia es el recogido en la Figura 4.2. En este caso, la mitad (16) de los threads del warp ejecutan la parte del IF, y la otra mitad (16) la parte del ELSE. En consecuencia, la **eficiencia del warp** es del 0.5 (50 %).

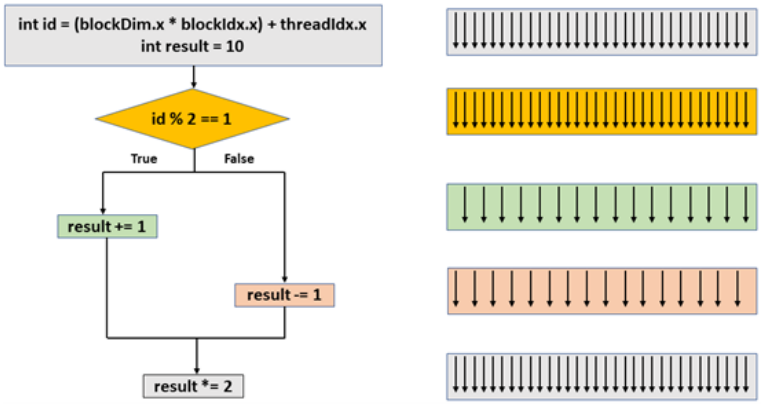


**Figura 4.2:** Ejecución de un código con divergencia.

Aunque este problema ha estado presente en las GPUs desde sus comienzos y arquitecturas recientes todavía lo siguen acarreando, la tendencia de las arquitecturas de NVIDIA más recientes es la de solucionar este problema, aprovechando los threads que no hacen trabajo para ejecutar otra instrucción en paralelo.

De cara al análisis TopDown, es necesario traducir esta pérdida de rendimiento en pérdida de IPC. Por tanto, hay que tener en cuenta cuáles han sido los efectos de la ejecución de código divergente. Para ello, al igual que en el apartado anterior, se hará uso de los contadores hardware. En este caso, para determinar estos efectos será necesario obtener cuál es el efecto de la divergencia en los warps, es decir, de todas las ejecuciones de código que pueden producir divergencias, cuáles son las que han provocado

que no se aproveche completamente todos los threads del warp. Además, también será necesario saber cuál es el IPC real de la aplicación, para saber qué parte de ese IPC es debido a la divergencia. Las Tablas 4.1 y 4.2 recogen las métricas necesarias, de acuerdo con la Compute Capability de la GPU. La pérdida de IPC producida a causa de los saltos (condicionales) puede ser calculada de la siguiente manera:

$$IPC_{Branch} = IPC \cdot (1 - Eficiciencia_{Warp}) \tag{4.3}$$

Adicionalmente, existe otra fuente de repetición de instrucciones. Las Unidades Funcionales de cada una de las subparticiones del SM son limitadas y en muchas ocasiones no hay tantas como para satisfacer la ejecución completa de un warp. Por ejemplo, en la sección anterior se vio un ejemplo de un SM en el que cada subpartición contenía 16 unidades FP64. Al ser el tamaño del warp de 32 threads no hay unidades funcionales suficientes para realizar esta operación en una vez. Es por esto por lo que es necesario realizar el dispatch del warp más de una vez para poder realizar la operación completa, lo que se conoce como un **Instruction Replay**. Además de las unidades funcionales, esta situación también puede ser producida en caso de conflictos en el banco de registros, en las operaciones de Lectura/Escritura en los espacios de memoria, en los que las peticiones se producen sobre el mismo banco de memoria, además de accesos de más de una línea en las memorias cache, entre otros.

Este efecto de tener que realizar varias veces el dispatch de una misma instrucción del warp provoca una pérdida de rendimiento en cuanto a que son necesarios más ciclos para poder completar la operación. Esto se traduce en una degradación del IPC de la aplicación. Para poder obtener el efecto que los replays producen en el rendimiento se hará uso de los contadores Hardware. Las Tablas 4.3 y 4.4 recogen las métricas necesarias para poder computar esta pérdida.

| **Métrica** | **Abreviatura** | **Descripción** |
|---|---|---|
| issued_ipc | $IPC_{Issued}$ | Promedio de instrucciones por ciclo emitidas, teniendo en cuenta la repetición de las instrucciones. |
| ipc | IPC | Promedio de instrucciones por ciclo ejecutadas, por SM. |

**Tabla 4.3:** Definición métricas Replay (CC menor que 7.2).

| **Métrica** | **Abreviatura** | **Descripción** |
|---|---|---|
| sm__inst_issued.avg.per_cycle_active | $IPC_{Issued}$ | Promedio de instrucciones por ciclo emitidas, teniendo en cuenta la repetición de las instrucciones. |
| sm__inst_executed.avg.per_cycle_active | IPC | Promedio de instrucciones por ciclo ejecutadas, por SM. |

**Tabla 4.4:** Definición métricas Replay (CC mayor o igual que 7.2).

El $IPC_{Issued}$ denota el IPC obtenido teniendo en cuenta todas las instrucciones que son emitidas, es decir, considerando los replays. Entonces, sabiendo que el IPC solo contabiliza las instrucciones que son ejecutadas, es decir, no tiene en cuenta los replays, se puede obtener cual es el IPC perdido, de acuerdo con la siguiente expresión

$$IPC_{Replay} = IPC_{Issued} - IPC \tag{4.4}$$

Finalmente, juntando las dos fuentes de pérdida de rendimiento anteriores, a través de las expresiones 4.3 y 4.4 se puede obtener la pérdida de IPC total debida a la divergencia:

$$IPC_{Divergence} = IPC_{Branch} + IPC_{Replay} = IPC \cdot (1 - Eficiciencia_{Warp}) + (IPC_{Issued} - IPC) \tag{4.5}$$

donde $IPC_{Branch}$ denota la pérdida de IPC a causa de los saltos e $IPC_{Replay}$ la provocada por los replays.

Los dos elementos analizados en este apartado, recogidos bajo la divergencia (los saltos condicionales y la repetición de instrucciones), suponen una pérdida de rendimiento, pero no un bloqueo del pipeline de la GPU. A continuación, se definen estos casos, en los que se produce un bloqueo de pipeline en la GPU, que llamaremos stalls.

## 4.4. Frontend

En esta categoría se recogen las pérdidas de rendimiento (bloqueos o stalls) ocasionadas por el Frontend del pipeline de la GPU. Esta parte incluye el **fetch** (**Fetch Bound**), en el que se determina el warp escogido a ser ejecutado en cada una de las subparticiones de cada uno de los SMs, gestionando la lectura de las instrucciones de cada thread de los warps. Por ejemplo, en esta categoría se encuentran las pérdidas debidas a fallos en la cache de instrucciones, sincronización entre threads etc.

Además, también se incluye la parte de **decodificación** (**Decode Bound**) en el que se realiza la lectura y asignación de registros a los operandos de las operaciones de cada thread de cada warp. Por ejemplo, conflictos en el banco de registros o latencias de la unidad de dispatch.

El objetivo es el de determinar cuál es el porcentaje de **bloqueos** o **stalls** que ha sufrido la GPU debido a operaciones del Frontend. De acuerdo con las connotaciones de stalls o bloqueos mencionados en el apartado anterior, estos pueden ser categorizados de acuerdo con la siguiente expresión:

$$STALL_{Frontend} = STALL_{Fetch} + STALL_{Decode} \tag{4.6}$$

Al igual que se ha realizado en los casos anteriores, para obtener los valores de la expresión anterior se hará uso de los contadores hardware.

| **Métrica** | **Descripción** |
|---|---|
| stall_inst_fetch | Porcentaje de stalls debidos al fetch de las instrucciones. |
| stall_sync | Porcentaje de stalls provocados por sincronizaciones de los threads (llamada función _syncthreads() o similar). |
| stall_other | Porcentaje de stalls del frontend provocados por otras causas. Por ejemplo, conflictos en el banco de registros. |

**Tabla 4.5:** Definición métricas Frontend (CC menor que 7.2).

| Métrica | Descripción |
|---|---|
| smsp__warp_issue_stalled_no_instruction_per_warp_active.pct | Porcentaje de stalls provocados por el bloqueo de warps esperando a ser seleccionados para realizar el fetch o por un fallo en la cache de instrucciones. |
| smsp__warp_issue_stalled_barrier_per_warp_active.pct | Porcentaje de stalls debidos a barreras de sincronización en los warps de un bloque. |
| smsp__warp_issue_stalled_membar_per_warp_active.pct | Porcentaje de stalls provocados por barreras de memoria. |
| smsp__warp_issue_stalled_dispatch_stall_per_warp_active.pct | Porcentaje de stalls provocados por la unidad de dispatch. |
| smsp__warp_issue_stalled_misc_per_warp_active.pct | Porcentaje de stalls provocados por diversos motivos hardware. |
| smsp__warp_issue_stalled_branch_resolving_per_warp_active.pct | Porcentaje de stalls provocados esperando al cálculo del branch target y el program counter sea actualizado. |
| smsp__warp_issue_stalled_sleeping_per_warp_active.pct | Porcentaje de stall provocados por estar todos los threads del warp en el estado bloqueado, cedido o dormido. |

**Tabla 4.6:** Definición métricas Frontend (CC mayor o igual que 7.2).

Como se puede apreciar de las Tablas 4.5 y 4.6, en ambos casos las métricas computan el resultado como porcentajes respecto del **total de stalls**. Por esta razón, tan solo es necesario sumar el valor de las métricas para poder obtener los stalls totales del Frontend.

Además, dividiendo las métricas anteriores en las dos partes del Frontend (Fetch y Decode), hace que se pueda obtener el STALLFetch y STALLDecode. Las métricas de cada parte, así como su cálculo se recogen en las Tablas 4.7 y 4.8, dependiendo de la Compute Capability de la GPU.

| Parte analizada | Abreviatura | Métricas de análisis empleadas |
|---|---|---|
| Fetch | $STALL_{Fetch}$ | stall_inst_fetch + stall_sync |
| Decode | $STALL_{Decode}$ | stall_other |

**Tabla 4.7:** Cálculo stalls partes del Frontend (CC mayor o igual que 7.2).

| Parte analizada | Abreviatura | Métricas de análisis empleadas |
|---|---|---|
| Fetch | $STALL_{Fetch}$ | smsp__warp_issue_stalled_no_instruction_per_warp_active.pct + smsp__warp_issue_stalled_barrier_per_warp_active.pct + smsp__warp_issue_stalled_membar_per_warp_active.pct + smsp__warp_issue_stalled_sleeping_per_warp_active.pct + smsp__warp_issue_stalled_branch_resolving_per_warp_active.pct |
| Decode | $STALL_{Decode}$ | smsp__warp_issue_stalled_misc_per_warp_active.pct + smsp__warp_issue_stalled_dispatch_stall_per_warp_active.pct |

**Tabla 4.8:** Cálculo stalls partes del Frontend (CC mayor o igual que 7.2).

Una vez obtenido el porcentaje de stalls en el Frontend, el siguiente paso es traducir esos en pérdida de rendimiento "real". Para ello se hará uso de la ecuación 4.1. El primer caso consiste en obtener la pérdida de IPC debido a stalls, que puede ser obtenida de la siguiente manera

$$IPC_{STALL} = IPC_{MAX} - (IPC_{DIVERGENCE} + IPC_{RETIRE}) \tag{4.7}$$

Sobre el que es necesario previamente obtener el $IPC_{RETIRE}$ e $IPC_{DIVERGENCE}$ (véase Secciones 4.2 e 4.3 para poder calcularlo, respectivamente); así como el $IPC_{MAX}$, que se corresponde con el IPC máximo teórico de la GPU, obtenido a partir de lo mencionado en el apartado 4.2. Finalmente,

teniendo en cuenta las expresiones 4.6 y 4.7 se puede obtener la pérdida de rendimiento (IPC) causada por bloqueos debidos al Frontend de la GPU.

$$IPC_{STALL_{Frontend}} = \frac{STALL_{Frontend}}{100} \cdot IPC_{STALL} \tag{4.8}$$

Por último, teniendo en cuenta los valores de $STALL_{Fetch}$ y $STALL_{Decode}$ y el valor global del $IPC_{STALL_{Frontend}}$, se puede obtener el IPC perdido en las dos subpartes del Frontend, de la siguiente manera

$$IPC_{STALL_{Fetch}} = \frac{STALL_{Fetch}}{100} \cdot IPC_{STALL} \tag{4.9}$$

$$IPC_{STALL_{Decode}} = \frac{STALL_{Decode}}{100} \cdot IPC_{STALL} \tag{4.10}$$

## 4.5. Backend

Como continuación del frontend, bajo esta categoría se encuentran las pérdidas de rendimiento (bloqueos o stalls) debidos al Backend del pipeline de la GPU. Esto incluye la **ejecución** o **execute** de las instrucciones de cada thread, así como las posibles dependencias que se pueden producir.

Como se ha comentado anteriormente, las ejecuciones se realizan utilizando las unidades funcionales de cada una de las subparticiones de los diferentes SMs. Sin embargo, una instrucción podría no ejecutarse si una unidad funcional está ocupada resolviendo una instrucción anterior. En este caso, habrá threads del warp que deberán quedarse **bloqueados** esperando a que otros threads hayan terminado y hayan dejado libre la unidad funcional. Este tiempo de espera puede ser elevado, teniendo en cuenta que hay operaciones que llevan más tiempo que otras (por ejemplo, una división respecto de una suma), y se recoge en el **Core Bound**. Diversos problemas respecto a dependencias y problemas en la jerarquía de memoria son recogidas en el **Memory Bound**. Por ejemplo, los fallos en las caches, tanto en operaciones de lectura como de escritura, hacen que se necesite más tiempo en completar la operación, ya que hay que descender en la jerarquía de memoria para completar la operación.

De forma análoga al Frontend, la pérdida de rendimiento en este caso queda determinada por los stalls o bloqueos, de acuerdo a la siguiente expresión

$$STALL_{Backend} = STALL_{MemoryBound} + STALL_{CoreBound} \tag{4.11}$$

Para obtener estos valores se hará uso de los contadores hardware, en el que, al igual que en el caso del Frontend, estos serán diferentes en función de la Compute Capability de la GPU. Las Tablas 4.9 y 4.10 recogen las métricas empleadas. En ambos casos los valores son computados como porcentajes sobre el total de stalls de la GPU.

| **Métrica** | **Descripción** |
|---|---|
| stall_memory_dependency | Porcentaje de stalls provocados por dependencias de datos en la jerarquía de memoria. |
| stall_exec_dependency | Porcentaje de stalls debidos a dependencias de datos. |
| stall_constant_memory_dependency | Porcentaje de stalls provocados por fallos en la memoria constante. |
| stall_pipe_busy | Porcentaje de stalls provocados por no disponer de unidades funcionales. |
| stall_memory_throttle | Porcentaje de stalls provocador por no disponer de recursos en la LSU (unidad de LD/ST) |

**Tabla 4.9:** Definición métricas BackEnd (CC menor que 7.2).

| **Métrica** | **Descripción** |
|---|---|
| smsp__warp_issue_stalled_ long_scoreboard_per_warp_active.pct | Porcentaje de stalls provocados por una dependencia en una operación L1/TEX. |
| smsp__warp_issue_stalled_imc_miss_per_warp_active.pct | Porcentaje de stalls provocados por fallos en la memoria constante. |
| smsp__warp_issue_stalled_math_pipe_throttle_per_warp_active.pct | Porcentaje de stalls provocados por no disponer de unidad de ejecución disponible. |
| smsp__warp_issue_stalled_mio_throttle_per_warp_active.pct | Porcentaje de stalls provocados por no disponer de una entrada en la cola de instrucciones de memoria de E/S. |
| smsp__warp_issue_stalled_drain_per_warp_active.pct | Porcentaje de stalls provocados por esperas a que los recursos (del warp) estén libres en operaciones de memoria. |
| smsp__warp_issue_stalled_lg_throttle_per_warp_active.pct | Porcentaje de stalls provocados por esperas a que la cola de (instrucciones de) L1 no esté llena, en operaciones de memoria local y global. |
| smsp__warp_issue_stalled_short_scoreboard_per_warp_active.pct | Porcentaje de stalls provocados en dependencias de operaciones de Memoria de E/S (MIO o Memory Input/Output). |
| smsp__warp_issue_stalled_wait_per_warp_active.pct | Porcentaje de stalls provocados por dependencias de ejecución con latencias fijas. |
| smsp__warp_issue_stalled_tex_throttle_per_warp_active.pct | Porcentaje de stalls provocados por esperas a que la cola de (instrucciones de) L1 no esté llena, en operaciones de texturas. |

**Tabla 4.10:** Definición métricas BackEnd (CC mayor o igual que 7.2).

A continuación, una vez definidas las métricas, estas deben de ser asignadas a cada una de las partes que componen el Backend (Memory Bound y Core Bound). De acuerdo con las definiciones anteriores, las Tablas 4.11 y 4.12 denotan las correspondencias y formas de cálculo, dependiendo de la CC de la GPU.

| **Parte analizada** | **Abreviatura** | **Métricas de análisis empleadas** |
|---|---|---|
| Memory Bound | $STALL_{MemoryBound}$ | stall_memory_dependency + stall_constant_memory_dependency + stall_memory_throttle |
| Core Bound | $STALL_{CoreBound}$ | stall_pipe_busy + stall_exec_dependecy |

**Tabla 4.11:** Cálculo stalls partes del Backend (CC menor que 7.2).

| Parte analizada | Abreviatura | Métricas de análisis empleadas |
|---|---|---|
| Memory Bound | $STALL_{MemoryBound}$ | smsp__warp_issue_stalled_long_scoreboard_per_warp_active.pct<br>+ smsp__warp_issue_stalled_imc_miss_per_warp_active.pct<br>+ smsp__warp_issue_stalled_drain_per_warp_active.pct<br>+ smsp__warp_issue_stalled_lg_throttle_per_warp_active.pct<br>+ smsp__warp_issue_stalled_tex_throttle_per_warp_active.pct |
| Core Bound | $STALL_{CoreBound}$ | smsp__warp_issue_stalled_math_pipe_throttle_per_warp_active.pct |

**Tabla 4.12:** Cálculo stalls partes del Backend (CC mayor o igual que 7.2).

A través de las métricas se pueden obtener $STALL_{MemoryBound}$ y $STALL_{CoreBound}$. Después, se determina la pérdida de rendimiento provocada por estos stalls o bloqueos. De igual manera que en el caso del Frontend, primero se obtiene el IPC perdido a causa de los bloqueos, utilizando la expresión 4.7. Posteriormente, teniendo en cuenta la expresión 4.11 se puede obtener el IPC perdido por los stalls del Backend, de acuerdo con la siguiente expresión

$$IPC_{STALL_{BackEnd}} = \frac{STALL_{Backend}}{100} \cdot IPC_{STALL} \tag{4.12}$$

$$IPC_{STALL_{MemoryBound}} = \frac{STALL_{MemoryBound}}{100} \cdot IPC_{STALL} \tag{4.13}$$

$$IPC_{STALL_{CoreBound}} = \frac{STALL_{CoreBound}}{100} \cdot IPC_{STALL} \tag{4.14}$$

## 4.6. Disponibilidad

Con objeto de realizar pruebas y verificaciones de la metodología TopDown descrita anteriormente, se ha realizado la implementación de una herramienta que permite realizar el análisis TopDown sobre los procesos ejecutados sobre la GPU. El código fuente de la herramienta se encuentra recogido en la siguiente dirección

`https://github.com/asf174/TopDownNvidia`

Para trabajar con la herramienta, tan solo hay que obtener el código fuente y seguir los pasos del README para realizar la instalación completa. Además, se puede seguir la evolución temporal de cómo ha sido el desarrollo de la herramienta, a través de los más de 220 commits que dispone el repositorio, enfocados en la parte de diseño y posteriores pruebas.

La implementación de la herramienta ha estado basada en un diseño modular, con posibilidad por parte del usuario de personalizar algunos parámetros. El repositorio está distribuido en diferentes directorios y ficheros, cada uno de ellos con diferente significado y función:

- **Topdown.py**: Fichero que contiene la ejecución principal de la herramienta. Hace uso de los diferentes directorios que componen la herramienta.
- **Args**: Incluye la gestión de los argumentos de la CLI del fichero anterior.
- **Errors**: Directorio donde se encuentran la definición de alguno de los posibles errores que puede generar la herramienta.
- **Graph**: Directorio que incluyen los ficheros que definen las gráficas utilizadas por la herramienta.

- **Measure_levels**: Directorio que incluye los ficheros con la definición de los diferentes niveles de la metodología TopDown.
- **Measure_parts**: Directorio que incluye los ficheros de las diferentes partes que componen cada uno de los niveles de la jerarquía.
- **Parameters**: Directorio con los ficheros que contienen la parametrización de la herramienta.
- **Shell**: Directorio que incluye el fichero necesario para poder lanzar comandos sobre una Shell Linux.
- **Show_messages**: Directorio que incluye el fichero necesario para los estilos a la hora de mostrar los resultados generados por la herramienta

Además, la herramienta dispone de una CLI que permite ampliar las características, haciendo que el usuario pueda interactuar con esta a través de las diferentes opciones que dispone.

# Capítulo 5

# Evaluación de TopDown GPU

En este apartado, vamos a poner a prueba la metodología propuesta, evaluando dos arquitecturas diferentes de GPUs de NVIDIA, haciendo uso de un conocido benchmark para GPUs. En esta prueba, evaluaremos la versatilidad de la herramienta, comprobando donde se encuentra el cuello de botella en una arquitectura relativamente actual. Comprobaremos cómo ha evolucionado la arquitectura, analizando dos arquitecturas recientes, y finalmente, analizaremos el coste de realizar un análisis de este estilo y su viabilidad.

Se va a trabajar sobre dos arquitecturas distintas: una NVIDIA GTX 1070 [23] instalada en un equipo portátil y una NVIDIA Quadro RTX 4000 [24] instalada en un servidor dentro del CPD 3MARES de la Universidad de Cantabria. La GPU GTX 1070 es una arquitectura Pascal, de forma que la herramienta usará nvprof por debajo, y el análisis llevará asociado tanto el uso de métricas como eventos. Por otro lado, la GPU Quadro RTX 4000 tiene arquitectura Turing, lo que supone el uso de Nsight para el análisis, donde sólo se utilizarán métricas. En la Tabla 5.1 están resumidas las principales características de las dos GPUs que se usarán para el análisis en este apartado.

| **Características** | **Descripción** | |
|---|---|---|
| Nombre | NVIDIA GTX 1070 | NVIDIA Quadro RTX 4000 |
| Microarquitectura | Pascal | Turing |
| Memoria | 8GB GDDR5 | 8GB GDDR6 |
| Compute Capability | 6.1 | 7.5 |
| Frecuencia de reloj | 1506MHz | 1005MHz |
| Frecuencia de reloj máxima | 1683MHz | 1545MHz |
| CUDA Cores | 1920 | 2304 |
| SMs | 15 | 36 |
| Potencia | 150W | 160W |
| Máximo número de threads por bloque | 1024 | 1024 |
| Subparticiones por SM | 4 | 2 |
| Tensor Units | 0 | 288 |
| Eventos y/o Métricas Medibles | 495 | 1627 |

**Tabla 5.1:** Descripción GPUs empleadas en análisis.

## 5.1. Evaluación del benchmark Rodinia

Rodinia es un benchmark desarrollado por la Universidad de Virginia [25] en el año 2009. Implementa varias aplicaciones de diferentes campos y ramas científicas para determinar el rendimiento tanto en CPUs como en GPUs, observando las diferencias arquitecturales entre ambas. Las aplicaciones han sido cuidadosamente escogidas para explotar diferentes comportamientos, haciendo uso de diferentes paralelismos, patrones de acceso y compartición de datos. Estos han sido implementados sobre aplicaciones que hacen uso de álgebra lineal, programación dinámica, grids estructurados y no estructurados, búsquedas en grafos...

| **Nombre** | **Tipo** | **Descripción** |
|---|---|---|
| B + tree | Búsqueda | Realiza búsquedas en arboles de tipo B + tree. |
| Back Propagation | Reconocimiento de patrones | Aprendizaje automático basado en el entrenamiento de los pesos de los nodos en una red neuronal. |
| BFS(Breadth-First Search) | Algoritmo basado en grafos | Búsqueda a partir de un vértice inicial para descubrir todos los vértices (posibles). |
| CFD | Dinámica de Fluidos | Resolución de las ecuaciones de Euler para fluido compresible. |
| Gaussian | Álgebra Lineal | Resolución de sistemas de ecuaciones lineales. |
| Heartwall | Imagen Médica | Análisis del movimiento del corazón en una secuencia de imágenes de ultrasonido para determinar la respuesta a este. |
| Hotspot | Simulación Física | Estimación de la temperatura del procesador en base a mediciones de potencia simuladas. |
| Hotspot3D | Simulación Física | Simulación térmica empleando la temperatura del procesador. |
| Huffman | Comprensión de datos sin pérdidas | Algoritmo de compresión de datos. |
| Hybridsort | Algoritmos de ordenación | Ordenación de estructuras de datos basado en la combinación de otros algoritmos de ordenación. |
| Kmeans | Minería de datos | Agrupación de datos en grupos para minimizar la distancia entre estos. |
| lavaMD | Dinámica modular | Cálculo de elementos característicos de partículas en base a fuerzas mutuas entre estas. |
| Myocyte | Simulación Biológica | Modelado del miocito cardiaco, simulando el comportamiento de esta célula en base a una serie de parámetros. |
| NN (Nearest Neighbor) | Minería de datos | Encuentra los vecinos más próximos de un conjunto de datos no estructurados entre sí. |
| Needleman-Wsunsh | Bioinformática | Optimización de secuencias de ADN. |
| Particlefilter | Imagen Médica | Estimador estadístico de la ubicación de un dato en base a unos parámetros característicos. |
| Pathfinder | Cruce de cuadrícula | Encuentra una ruta en 2D desde la parte (fila) inferior a la superior utilizando combinación de los pesos más pequeños. |
| SRAD (Speckle Reducing Anisotropic Diffusion) | Procesado de imágenes | Método de difusión utilizado en imágenes de radar y ultrasonido, empleando ecuaciones diferenciales para resolverlo. |
| Streamcluster | Minería de datos | Agrupación de datos en stream, tratando de reducir la distancia entre estos. |

**Tabla 5.2:** Aplicaciones utilizadas Benchmark Rodinia.

Desde su publicación en el año 2009, el Benchmark ha ido creciendo en cuanto a número de aplicaciones (duplicando el número de estas), mejorando en el rendimiento y corrigiendo algunos errores o bugs. A lo largo de este apartado se hará uso solo de la última versión utilizada en GPUs, 3.1, para realizar el análisis de la metodología TopDown sobre las diferentes aplicaciones. El objetivo

consistirá en determinar los cuellos de botella y dónde se producen las pérdidas de rendimiento, respecto a una ejecución teórica ideal en la que se obtendría el máximo rendimiento. La Tabla 5.2 recoge las aplicaciones de Rodinia empleadas.

El análisis se centrará en analizar el rendimiento de las aplicaciones anteriores, utilizando la herramienta descrita en el apartado anterior. La Figura 5.1 recoge el análisis TopDown, empleando las dos arquitecturas definidas anteriormente sobre los mismos algoritmos.

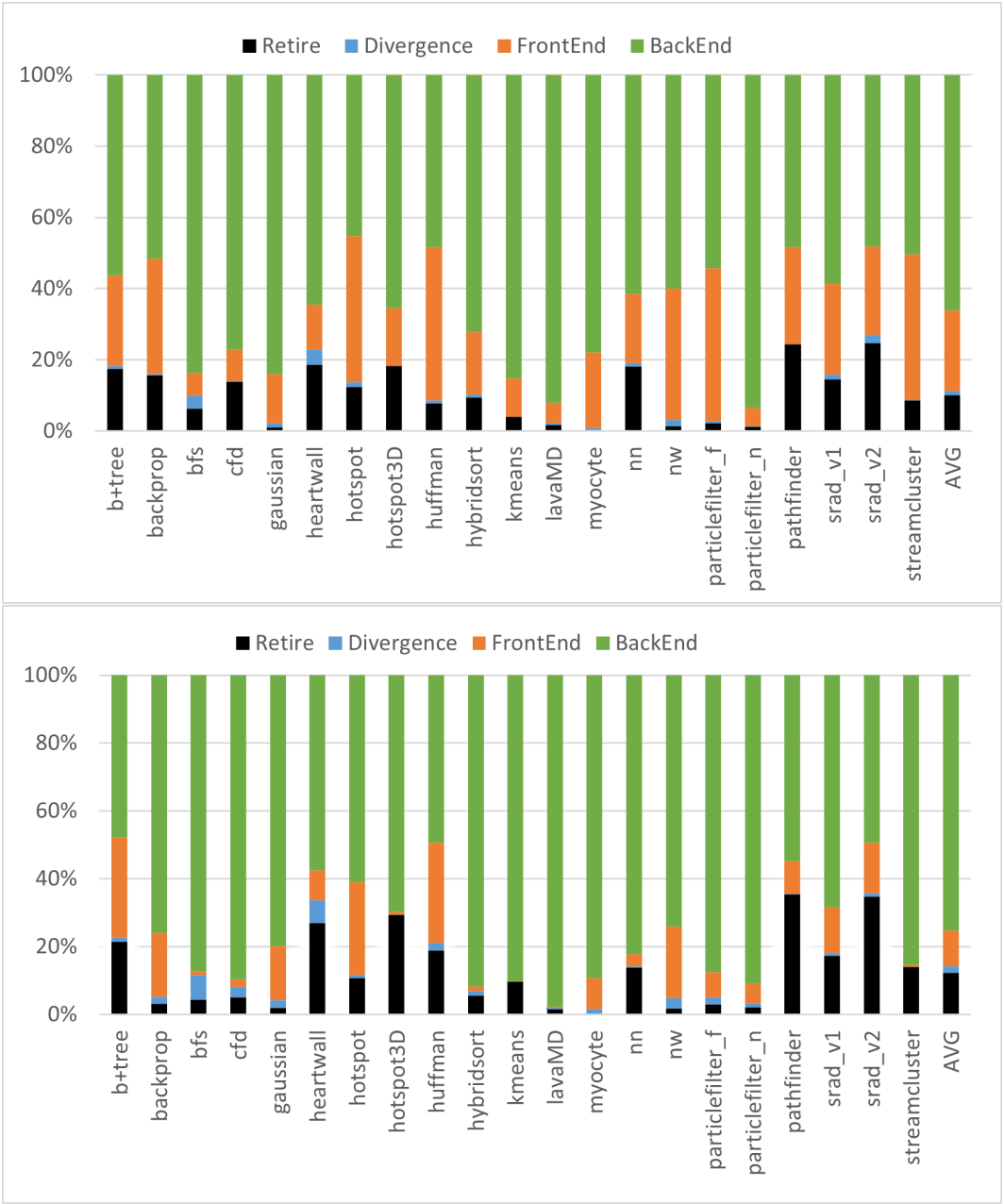


**Figura 5.1:** Ejecución Rodinia sobre Pascal (arriba) y Turing (abajo).

En ambos diagramas se muestra la pérdida de rendimiento en cada una de las partes de la metodología TopDown desarrollada. Como se mencionó en el apartado anterior (4.2), el retire no constituye una pérdida de rendimiento, sino el rendimiento real de la aplicación.

Las figuras anteriores ofrecen resultados similares en cuanto a las partes en las que hay cuellos de botellas. En ambos casos, la mayor parte del rendimiento se pierde en el Backend, siendo similar en algunas aplicaciones en ambos dispositivos. No obstante, en algunas otras aplicaciones, el valor del Backend tiene más trascendencia en Turing. Por su parte, el Frontend constituye también una fuente de pérdida de rendimiento, con un menor impacto en Turing que Pascal, en líneas generales. Por otro lado, la divergencia tiene un impacto prácticamente despreciable en ambas arquitecturas (algo más relevante en Turing), con valores muy cercanos al 0 %.

Por último, el rendimiento o retire obtenido es relativamente bajo en ambos casos, siendo difícil alcanzar el máximo teórico. No obstante, en algunas aplicaciones el valor puede considerarse aceptable en ambos casos, como por ejemplo srad_v2, heartwall, hotspot3D o pathfinder. En otras, en cambio, el rendimiento es realmente pobre en ambos dispositivos, por ejemplo en las aplicaciones particlefilter (en su versión de float) y nw. En general, es posible observar cómo el porcentaje de retire es ligeramente superior en la arquitectura más moderna (Turing), siendo considerablemente menor el impacto de los stalls en el Frontend. Por otra parte, el impacto del Backend, aunque semejante en ambos casos, es más acusado en Turing, siendo algo menor en Pascal.

El nivel 1 analizado ofrece una visión general del comportamiento de la GPU. Sin embargo, para poder clarificar las partes concretas en las que se han producido los cuellos de botellas es necesario descender en la jerarquía. Vamos a ampliar el detalle de nuestro análisis en aquellas aplicaciones en las que los resultados son especialmente llamativos. Dado que la potencia del análisis es superior en la arquitectura más moderna, lo haremos sobre ésta. La Figura 5.2 recoge el análisis de nivel 2 TopDown realizado sobre la arquitectura Turing para unas aplicaciones seleccionadas por tener comportamientos diferenciados.

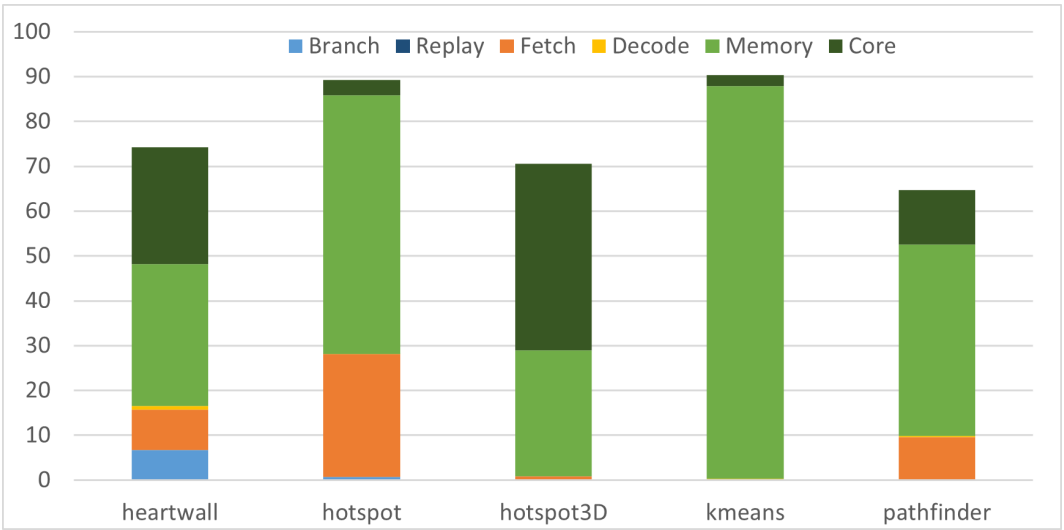


**Figura 5.2:** Análisis Nivel 2 Rodinia sobre Turing.

Se observa cómo el efecto general de las pérdidas de rendimiento en el Backend es principalmente producido por la jerarquía de memoria. Es posible, en cualquier caso, observar aplicaciones en las que las unidades funcionales están especialmente ocupadas (Core bound), en las aplicaciones heartwall y, especialmente, hotspot3D. En los casos en los que la divergencia tiene relevancia, ésta es motivada

principalmente por la ineficiencia del warp, posiblemente debido a los saltos condicionales. Finalmente, en las aplicaciones donde el Frontend tiene impacto negativo en el rendimiento, parece que es porque no se puede realizar el fetch de la instrucción (quizá por problemas en la cache de instrucciones), y no tanto por conflictos como podrían darse en el banco de registros.

Dado que parece que el impacto de la jerarquía de memoria en el rendimiento es generalizado, y siendo donde nuestra herramienta tiene más nivel de detalle, vamos a realizar un análisis de nivel 3. La Figura 5.3 recoge el análisis TopDown de nivel 3 en la porción de Memory Bound, para las aplicaciones de la Figura 5.2.

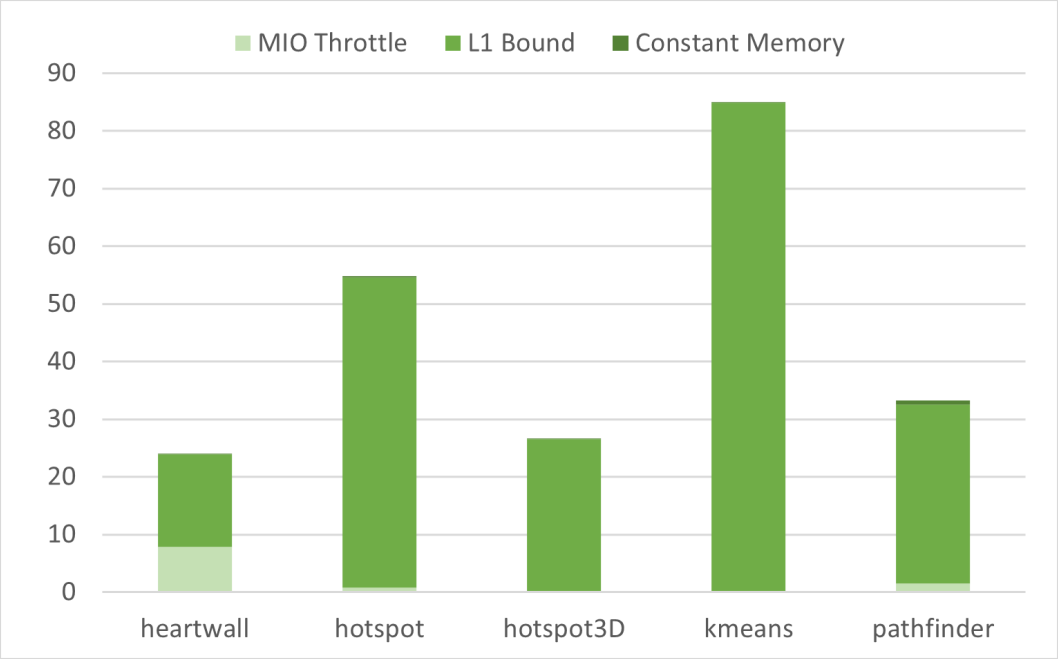


**Figura 5.3:** Análisis Nivel 2 Rodinia sobre Turing.

La figura anterior muestra una pérdida de rendimiento clara en la memoria cache L1. Estos bloqueos pueden estar motivados por dependencias de datos en memoria local, global, texturas... Además, también puede haber esperas por no disponer de recursos en la cola de instrucciones LSU (Load/Store Unit) en operaciones sobre la memoria global y local. Por su parte, en una menor medida, aplicaciones como heartwall sufren cuellos de botellas en el sistema de memoria de entrada/salida (MIO o Memory Input/Ouput Throttle), que denota las esperas por no disponer de entradas libres en la cola (FIFO) de memoria, para el resto no tiene prácticamente impacto en el rendimiento. Estos stalls se deben generalmente a usos intensivos de operaciones de Load en la memoria local, global y shared. Por último, los fallos y dependencias en la memoria constante (Constant Memory) apenas afectan al rendimiento, al no ser trascendental en ninguna de las aplicaciones, con valores muy cercanos a 0 %.

## 5.2. Evaluación del Benchmark Altis

El benchmark Rodinia ofrece la posibilidad de analizar el comportamiento de una gran variedad de aplicaciones sobre una GPU. Sin embargo, este benchmark es relativamente antiguo y a pesar de haber evolucionado, las aplicaciones están basadas en las necesidades de la época en la que fue

diseñado y no en algunos campos en los que la GPU es altamente utilizada hoy en día. Por ejemplo, el álgebra relacional o redes neuronales profundas (DNN). Rodinia no incluye kernels basados en redes neuronales, por lo que no puede analizarse el impacto que tienen este tipo de aplicaciones. Es por esto que se decide usar un benchmark más reciente, Altis, que permita obtener una visión del comportamiento de GPUs actuales ejecutando aplicaciones de relevancia hoy en día.

Altis [26] es un benchmark reciente, evolución de dos suites previas, Rodinia y SHOC [27]. En particular, Altis incorpora diversas características utilizadas en modernas y populares DNNs. Además, los benchmark han sido readaptados con las nuevas características de GPUs con versiones CUDA más recientes. Por ejemplo, se hace uso de la Memoria Unificada [28], que permite compartir el espacio de direcciones entre la CPU y la GPU, con el objetivo de evitar las transferencias de memoria entre ambos dispositivos. Del mismo modo, alguna aplicación incorpora un Paralelismo Dinámico [29], que hace que los kernels invoquen a su vez a otros kernels. Además, también se hace uso de la tecnología HyperQ, que permite ejecutar diferentes kernels en paralelo en la misma GPU si es posible. Por su parte, se incorpora GridSync [30], que permite sincronizar todos los threads de todos los bloques, además de mejorar las técnicas de recolección de tiempos de ejecución, a través de los eventos de CUDA. Por último, se incorporan los grafos de CUDA, que permite trabajar con nodos, que agrupan kernels y operaciones de memoria. Esto provoca que se puedan juntar las operaciones de la GPU, consiguiendo un menor número de estas. La Tabla 5.3 recoge las aplicaciones empleadas durante el análisis.

| **Nombre** | **Descripción** |
|---|---|
| DNNs | Implementa diferentes fases que sigue una red neuronal, desglosándolas en diferentes aplicaciones (activation, avgpool, connected, dropout, normalización, y softmax) |
| Breadth First Search | Implementa la búsqueda primero en anchura, un algoritmo de búsqueda en grafos. |
| CFD | Dinámica de fluidos dinámica. Resuelve ecuaciones de Euler tridimensionales. |
| GPUDWT | Implementa la transformada ondícula discretas (comprensión de imagen y vídeo). |
| General Matrix Multiply | Multiplicación de matrices con diferentes características. |
| GUPS | Medición de frecuencia con la que se pueden emitir actualizaciones a zonas de RAM generadas de manera aleatoria. |
| LavaMD | Interacción entre partículas, obteniendo el potencial de partículas y reubicación en base a fuerzas entre ellas. |
| Mandelbrot | Obtiene una imagen basada en un fractal de Mandelbrot. |
| Needleman-Wunsch | Optimización de secuencias de ADN, organizadas en matrices de dos dimensiones. |
| ParticleFilter | Evaluación estadística de la ubicación de un objetivo en base a mediciones de ubicaciones y rutas. |
| Raytracing | Renderizado para obtener imágenes en base al seguimiento de la trayectoria de la luz. |
| Sort | Implementación de un algoritmo de ordenado sobre un array de números enteros. |
| SRAD | Reducción de ruido y motas en imágenes, sin alterar las características de la imagen. |
| Where | Algebra relacional que, dado unos registros de entrada, obtiene un subconjunto de estos que cumplen una serie de premisas. |

**Tabla 5.3:** Aplicaciones utilizadas Benchmark Altis.

Al igual que en el caso de Rodinia, se realiza un análisis TopDown, sobre las aplicaciones anteriores. La Figura 5.4 muestra los resultados obtenidos. En ella se puede comprobar cómo hay una prevalencia de pérdidas de rendimiento en el Backend, seguido del Frontend, con una diferencia notable entre ambos. Por su parte, la divergencia no constituye prácticamente una fuente de pérdida de rendimiento, al ser en todas las aplicaciones un valor relativamente bajo. Por último, el retire o rendimiento de la aplicación también es relativamente bajo en términos generales, aunque algunas aplicaciones ofrecen rendimientos notables.

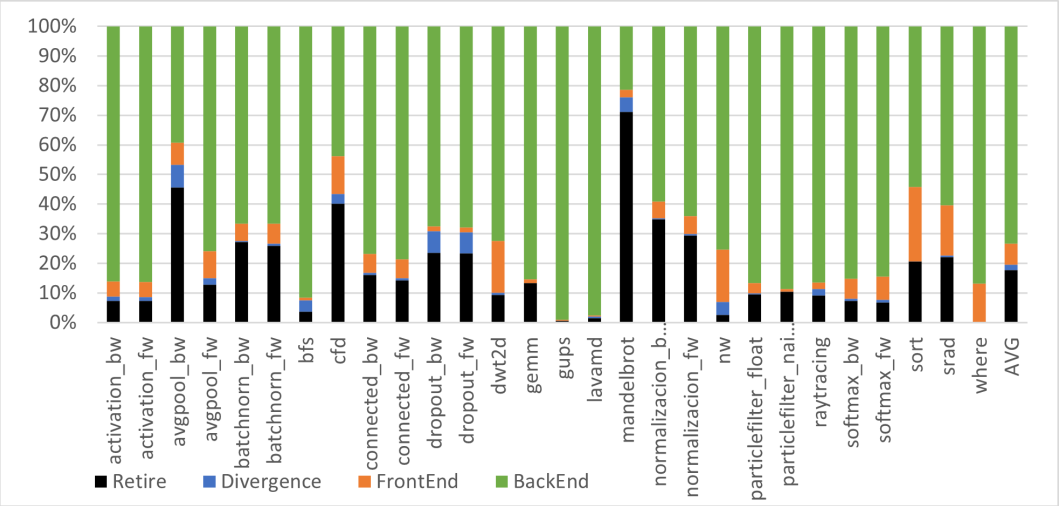


**Figura 5.4:** Análisis TopDown de las aplicaciones de Altis en Turing.

Comparando con el benchmark anterior en la misma arquitectura (Figura 5.1, abajo), los resultados son similares. La prevalencia de pérdida de rendimiento se produce claramente en el Backend, al igual que ocurria en Rodinia. En ambos casos el frontend constituye la segunda fuente de pérdida de rendimiento, aunque el rendimiento promedio (Retire) parece superior en las aplicaciones de Altis, con aplicaciones que rondando el 40 % de eficiencia, y alguna llegando al 70 % del rendimiento máximo teórico de la máquina (mandelbrot). Por último, la tendencia se repite en los valores de la divergencia, con valores muy bajos, cercanos en ambos análisis.

Con respecto a las aplicaciones iguales en ambos benchmarks, en algunos casos Altis ofrece unos algoritmos que son parecidos a los de Rodinia (los mismos, pero adaptados a tiempos más modernos). Sin embargo, los resultados que ofrecen son prácticamente iguales. Por ejemplo, los algoritmos BFS o NW obtienen un rendimiento muy similar en ambos benchmarks, sobre la misma arquitectura (Figuras 5.1 abajo y 5.4). Sin embargo, es posible apreciar diferencias de comportamiento en aplicaciones como CFD, que obtiene mejor rendimiento.

## 5.3. Análisis periódico

Las pruebas realizadas hasta el momento tienen como objetivo analizar el rendimiento de la aplicación en su conjunto. Esto implica tener en consideración todos y cada uno de los kernels que componen la misma, como un conjunto. A la hora de realizar el análisis de la metodología TopDown sobre una aplicación con una gran variedad de kernels, hay que tener en cuenta también el tiempo de ejecución de cada uno de estos, con el objetivo de que los kernels que hayan estado ejecutando más tiempo tengan más peso en el resultado final. Por ejemplo, si uno de ellos con una degradación alta de Backend

ha estado ejecutando el doble de tiempo que otro con degradación de rendimiento en el Frontend, el resultado del primer kernel aporta el doble al cómputo del resultado final de ambos kernels.

No obstante, la herramienta muestra resultados para todos los kernels ejecutados, y es posible hacer un análisis individualizado, pudiendo observar su comportamiento a lo largo del tiempo. Para realizar este análisis, se escoge una aplicación con un número reducido de kernels, pero que sean ejecutados varias veces, de forma que sea posible ver su evolución. En la Figura 5.5 se recoge el análisis TopDown de dos kernels (srad_cuda_1 y srad_cuda_2), pertenecientes a la aplicación SRAD, mostrando la evolución temporal de estos a medida que son invocados.

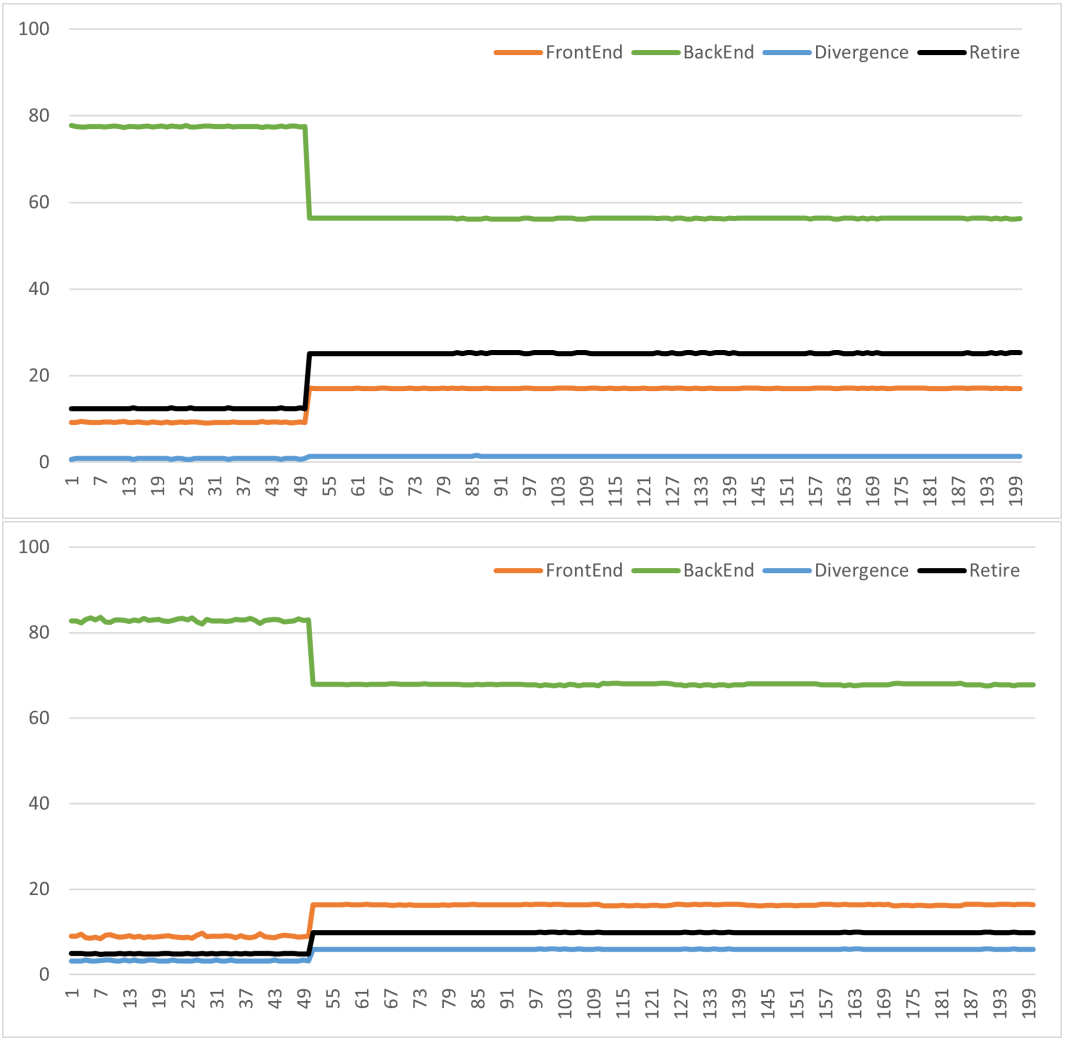


**Figura 5.5:** Evolución temporal, en Turing, de los kernels de SRAD de Altis: srad_cuda_1 (arriba) y srad_cuda_2 (abajo).

En ambos kernels se pueden apreciar dos fases. Una primera, más corta, que va desde el principio hasta la invocación 50, y la otra desde este momento al final. Los stalls en el Backend dominan claramente la primera fase de la ejecución en ambos kernels. En la segunda fase, la presión sobre el Backend parece reducirse, más pronunciado en srad_cuda_1, donde el rendimiento mejora de forma más notable, aunque sigue siendo el componente que más influye. En ambos casos, al llegar la segunda

fase aumenta la presión sobre el Frontend. Por su parte, la divergencia experimenta un aumento en el cambio de fase, siendo muy pequeño en ambos casos, en especial en srad_cuda_1. Por último, el rendimiento o retire en el cambio de fase sufre una mejora, provocado por el descenso del Backend; siendo más notable en el caso de srad_cuda_1.

## 5.4. Overhead

El análisis TopDown realizado anteriormente hace uso de los contadores hardware para obtener los resultados. Esto implica que, además de realizar la ejecución de la aplicación, también debe computar las métricas y eventos (si los hubiere) necesarios para poder realizar el análisis. Como se ha mencionado en la sección 2.1, el hardware encargado de la monitorización de los contadores hardware es limitado, y si la cantidad de valores a medir excede esta capacidad, es necesario realizar varias ejecuciones de los kernels para poder obtener los resultados. Esta replicación se produce **por cada kernel** por lo que en muchas ocasiones puede añadir un overhead considerable que ralentice la obtención de los resultados.

Sin embargo, el overhead no se produce tan sólo por el tiempo que lleva ejecutar múltiples veces. En cada una de las repeticiones, para mejorar la precisión de los datos obtenidos, las modificaciones que hace un kernel al ejecutarse deben de ser eliminadas para el siguiente replay, de tal forma que este no tenga ventaja sobre el anterior y la ejecución sea lo más semejante posible. En nuestro caso, sabemos que para un análisis TopDown, se hace un total de 8 ejecuciones de los kernels para poder obtener todos los resultados.

Por otro lado, a medida que el número de kernels aumenta, la herramienta desarrollada tiene que recoger más datos para realizar el análisis. Sobre estos valores es necesario determinar el tiempo que ha estado ejecutando cada kernel sobre la GPU, por lo que los cálculos adicionales, además de los propios del TopDown, también suponen un overhead adicional que, a medida que hay un mayor número de kernels, es mayor, y que se suma a la necesidad de repetición.

No obstante, el número de kernels no es el único factor que influye en el overhead. Hay situaciones en las que basta con un número "reducido" de estos para que el overhead ya sea elevado. Esto es provocado por la gran cantidad de datos utilizados por el kernel, lo que provoca que el borrado de la memoria sea costoso, como se ha explicado antes.

Para determinar el overhead que implica realizar un análisis TopDown es necesario compararlo con su versión **nativa**, es decir, la ejecución de la aplicación sin hacer el análisis TopDown. La Figura 5.6 muestra el análisis de overhead de nuestra herramienta en las aplicaciones de Rodinia y Altis ejecutadas sobre Turing. El diagrama muestra la relación de la ejecución nativa con un análisis TopDown de nivel 3.

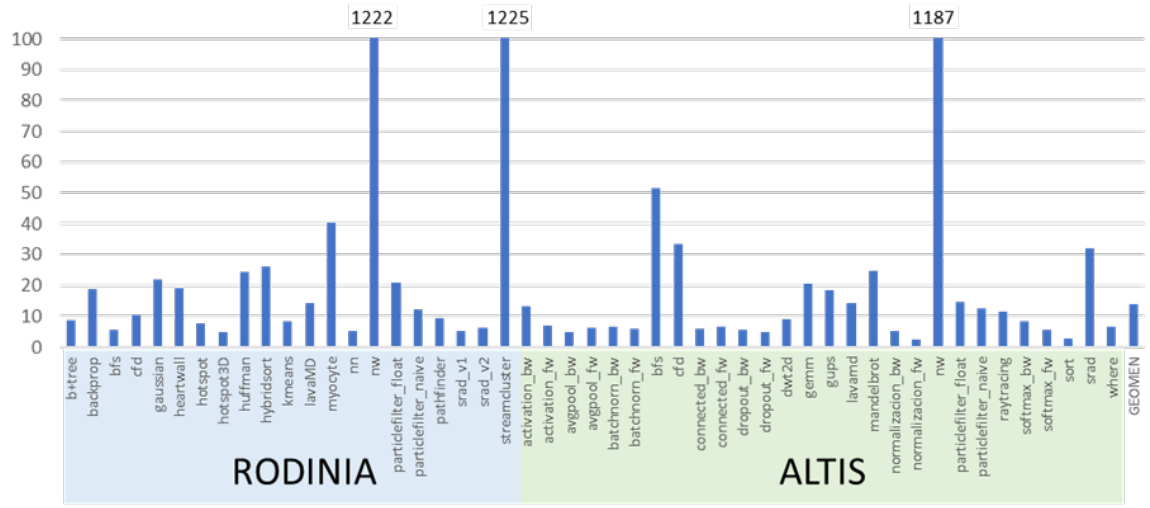


**Figura 5.6:** Overhead de la herramienta en Rodinia y Altis sobre Turing.

La tendencia de las aplicaciones indica un overhead por 16 entre la ejecución nativa y TopDown, que parece razonable teniendo en cuenta que cada kernel debe ejecutarse 8 veces. La mayor parte de las aplicaciones se encuentran muy próximas a este valor, algunas superándolo ligeramente, pero con factores menores de 30.

No obstante, también hay aplicaciones en las que el overhead es excesivamente grande. Estas son NW, tanto en Rodinia como Altis, y streamcluster, perteneciente a Rodinia. Como se ha explicado anteriormente, el número elevado de kernels, así como un alto volumen de datos en cada kernel, que se combina en estas aplicaciones, motiva un overhead excesivamente elevado.

Por último, las aplicaciones ejecutadas en Rodinia sobre Pascal, utilizando la herramienta nvprof, experimentan un overhead similar, ligeramente más elevado que en el caso de Turing (26). Hay que tener en cuenta que se trata de una herramienta y jerarquía distinta, aunque en ambos casos la herramienta necesita 8 ejecuciones para completar el análisis.

## 5.5. Análisis de TensorFlow

Para finalizar, se ha realizado un análisis empleando aplicaciones de Machine Learning. En concreto, se ha hecho uso de la herramienta **Tensorflow**, desarrollada por Google, basada en algoritmos de aprendizaje automático [32]. Se trata de un framework que trabaja con funciones de alto nivel para construir redes neuronales. En concreto usa la API **Keras** [33], que abstrae muchas de las características de la red.

Para realizar el análisis, se escoge una aplicación dentro del conjunto de aplicaciones de **TensorFlow Model Garden** [34]. Para la prueba, se seleccionó una red para clasificación de imágenes, usando el dataset de **Fashion MNIST**.

A continuación, se realizó una ejecución del benchmark tanto en la arquitectura Pascal como en la Turing. En ambos casos, hubo que parar la ejecución tras varias horas sin que terminase tan siquiera una época. Se hicieron pruebas de medidas con un menor número de métricas, y el overhead seguía siendo considerable. Al analizar lo ocurrido, se observó que el número de kernels lanzados por TensorFlow superaba los 100.000, haciendo impracticable realizar el análisis en un tiempo razonable.

# Capítulo 6

# Conclusiones y Trabajo Futuro

A lo largo del documento se han ido proponiendo diferentes técnicas para poder implementar una metodología de análisis originaria de CPUs Intel, TopDown, en GPUs de la marca NVIDIA. Los desarrolladores proporcionan herramientas cercanas al programador, en lugar de la microarquitectura, lo que ha motivado la necesidad de implementar esta metodología de análisis.

A pesar de las diferencias existentes entre la arquitectura de una CPU y una GPU, siguiendo el modelo implementado en las CPUs, se ha podido replicar esta metodología de análisis, estableciendo un pipeline e identificando las diferentes fuentes de pérdida de rendimiento. El proceso de selección de las diferentes partes que componen la metodología ha sido similar al de la CPU, con la salvedad de algunas características concretas de la GPU, que no están en la CPU y viceversa. La ejecución especulativa o la divergencia son algunos ejemplos de conceptos específicos de CPUs y GPUs, respectivamente.

El análisis Topdown original fue diseñado por Intel, y vino acompañado de cambios en el hardware, proporcionando contadores específicos que permitieron realizar un estudio muy concreto sobre las diferentes partes de la microarquitectura. En la realización de este trabajo, solo se ha podido utilizar la información que proporciona NVIDIA, y en muchos casos esta no ha sido suficiente o lo más eficiente.

Durante el proceso de selección de los contadores hardware adecuados, se han realizado múltiples pruebas que verifican el comportamiento de estos, por ejemplo, para la detección del IPC máximo, o el comportamiento de la divergencia. Además, se ha trabajado con diferentes arquitecturas para permitir que un amplio número de GPUs puedan hacer uso de la propuesta, lo que ha implicado trabajar con herramientas y contadores distintos. Se ha tenido especial cuidado en que los resultados arrojados sean lo más parecidos posible entre las diferentes arquitecturas, permitiendo al usuario establecer diferencias entre estas.

El trabajo ha estado limitado a los contadores hardware que NVIDIA proporciona, y hay márgenes de mejora en este aspecto. Como trabajo futuro, resultaría interesante poder ofrecer métricas más concretas relativas a la jerarquía de memoria, estableciendo con más detalle los stalls que provocan en el rendimiento los diferentes componentes hardware de la jerarquía de memoria. Igualmente, sería interesante un conocimiento más en detalle del uso de las distintas unidades funcionales, especialmente las que se encuentran en número reducido. Sin embargo, es una limitación que necesita cambios en el hardware por parte de NVIDIA.

Además de las limitaciones que ofrece en cuanto a la información que proporciona, la forma de obtención de los resultados también ha presentado un problema y es que añade un overhead que en muchas situaciones provoca que la herramienta pueda ser poco práctica, pues el tiempo para realizar el análisis aumenta considerablemente a medida que se aumenta el número de métricas solicitadas. La técnica de replicación de kernels para poder medir todas las métricas y eventos (si los hubiere) puede

ser un cuello de botella considerable. En muchos casos, hasta con un número pequeño de métricas computadas, la recolección de estos valores produce overheads considerables, de uno o dos órdenes de magnitud, debido al elevado número de kernels de la aplicación. Una solución podría ser la de dotar a las unidades hardware de recolección de más mecanismos para poder realizar las mediciones de más contadores hardware sin necesidad de aumentar el número de repeticiones de los kernels.

Las pruebas de la metodología realizadas han sido escogidas a partir de un grupo de benchmarks representativos en GPUs. Estas pruebas pueden servir como punto de ayuda para detectar donde están las debilidades microarquitecturales de las GPUs, lo que permite aplicar técnicas correctoras para poder evitar estos problemas en versiones futuras. Además, se ha trabajado con diferentes tipos de GPUs. Por un lado, se han realizado pruebas con una GPU convencional enfocada para computación gráfica y otra utilizada para investigación y aplicaciones de gran escala. A través de estas pruebas se pueden también observar las diferencias entre tipos estilos de GPUs, viendo como una GPU para gráficos está más limitada en cuanto es sacada del modelo de cómputo para el que idealmente ha sido diseñada.

Como posible trabajo futuro, la fase de pruebas también puede ser ampliada, utilizando la herramienta sobre un gran número de arquitecturas, lo que permitiría obtener más información acerca de las GPUs, observando las diferencias de rendimiento entre estas. Por último, también se podrían seleccionar benchmarks más concretos en caso de que se quiera explorar el comportamiento de una GPU determinada sobre un tipo de aplicación concreta, por ejemplo mediante el empleo de redes neuronales, o diferentes algoritmos de aprendizaje automático.

# Referencias


[1] Robert H Dennard, Fritz H Gaensslen, Hwa-Nien Yu, V Leo Rideout, Ernest Bassous y Andre R LeBlanc. "Design of ion-implanted MOSFET's with very small physical dimensions". *IEEE Journal of Solid-State Circuits* 9.5 (1974), págs. 256-268.

[2] TOP500. *Listas TOP500*. 2021. URL: `https://www.top500.org/lists/hpcg/` (visitado 10-07-2021).

[3] Shane Cook. *CUDA programming: a developer's guide to parallel computing with GPUs*. Newnes, 2012.

[4] NVIDIA. "NVIDIA TESLA V100 GPU ARCHITECTURE: THE WORLD'S MOST ADVANCED DATA CENTER GPU" (2017).

[5] NVIDIA. "NVIDIA Turing GPU Architecture: Graphics Reinvented" (2018).

[6] NVIDIA. "NVIDIA Tesla P100: The Most Advanced Datacenter Accelerator Ever Built Featuring Pascal GP100, the World's Fastest GPU" (2016).

[7] TOP500. *https://www.top500.org/news/new-gpu-accelerated-supercomputers-change-the-balance-of-power-on-the-top500/*. Jun. de 2018. URL: `https://www.top500.org/news/new-gpu-accelerated-supercomputers-change-the-balance-of-power-on-the-top500/` (visitado 14-07-2021).

[8] Ahmad Yasin. "A top-down method for performance analysis and counters architecture". En: *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE. 2014, págs. 35-44.

[9] Peggy Irelan y Shihjong Kuo. "Performance Monitoring Unit Sharing Guide". *Intel White Paper, http://software.intel.com/file/30388,(30388-PMU-Sharing-Guidelines.pdf)* (2019).

[10] Intel. "Intel® 64 and ia-32 architectures Software Developer's Manual". *Volume 3B: System programming Guide, Part* 3B.2 (sep. de 2016).

[11] NVIDIA. *Performance Counters*. URL: `https://docs.nvidia.com/nsight-visual-studio-edition/4.6/Content/Analysis/Report/CudaExperiments/KernelLevel/PerformanceCounters.html` (visitado 11-07-2021).

[12] NVIDIA. "CUPTI: User's Guide" (jul. de 2020).

[13] Arnaldo Carvalho De Melo. "The new linux 'perf' tools". En: *Slides from Linux Kongress*. Vol. 18. 2010, págs. 1-42.

[14] NVIDIA. *CUDA Profiler*. Jun. de 2021.

[15] NVIDIA. "Nsight Compute Command Line Interface: User Manual" (sep. de 2020).

[16] Intel. *Intel VTune Profiler*. Nov. de 2020. URL: `https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/vtune-profiler.html#gs.6gu7g5` (visitado 12-07-2021).

[17] NVIDIA. *NVIDIA Visual Profiler*. 2021. URL: https://developer.nvidia.com/nvidia-visual-profiler (visitado 14-07-2021).

[18] NVIDIA. *NVIDIA Nsight Systems*. 2021. URL: https://developer.nvidia.com/nsight-systems (visitado 14-07-2021).

[19] NVIDIA. "NSIGHT COMPUTE: User Manual" (jun. de 2021).

[20] *PARALLEL THREAD EXECUTION ISA: Application Guide*. Jun. de 2017.

[21] Mihir Awatramani, Xian Zhu, Joseph Zambreno y Diane Rover. "Phase aware warp scheduling: Mitigating effects of phase behavior in gpgpu applications". En: *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE. 2015, págs. 1-12.

[22] NVIDIA. "CUDA C++ Programming Guide". 11.3 (jun. de 2021).

[23] NVIDIA. *Tarjetas Gráficas GeForce GTX 1070Ti y GeForce GTX 1070*. 2016. URL: https://www.nvidia.com/es-la/geforce/products/10series/geforce-gtx-1070-ti/ (visitado 14-07-2021).

[24] NVIDIA. "REAL TIME MEANS REAL CHANGE: NVIDIA QUADRO RTX 4000" (jul. de 2020).

[25] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee y Kevin Skadron. "Rodinia: A benchmark suite for heterogeneous computing". En: *2009 IEEE international symposium on workload characterization (IISWC)*. Ieee. 2009, págs. 44-54.

[26] Bodun Hu y Christopher J Rossbach. "Altis: Modernizing GPGPU Benchmarks". En: *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE Computer Society. 2020, págs. 1-11.

[27] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S Meredith, Philip C Roth, Kyle Spafford, Vinod Tipparaju y Jeffrey S Vetter. "The scalable heterogeneous computing (SHOC) benchmark suite". En: *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. 2010, págs. 63-74.

[28] Dan Negrut, Radu Serban, Ang Li y Andrew Seidl. "Unified Memory in CUDA 6: A Brief Overview and Related Data Access/Transfer Issues" (ago. de 2018).

[29] NVIDIA. *Adaptive Parallel Computation with CUDA Dynamic Parallelism*. Mayo de 2014. URL: https://developer.nvidia.com/blog/cooperative-groups/ (visitado 14-07-2021).

[30] NVIDIA. *Cooperative Groups: Flexible CUDA Thread Programming*. Oct. de 2017. URL: https://developer.nvidia.com/blog/cooperative-groups/ (visitado 14-07-2021).

[31] NVIDIA. *Kernel Profiling Guide*. Mayo de 2021. URL: https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html (visitado 14-07-2021).

[32] Abadi Martin, Agarwal Ashish, Barham Paul, Brevdo Eugene, Chen Zhifeng, Citro Craig, Corrado Greg S., Davis Andy, Dean Jeffrey, Devin Matthieu, Ghemawat Sanjay, Goodfellow Ian, Harp Andrew, Irving Geoffrey, Isard Michael, Yangqing Jia, Jozefowicz Rafal, Kaiser Lukasz, Kudlur Manjunath, Levenberg Josh, Mane Dandelion, Monga Rajat, Moore Sherry, Murray Derek, Olah Chris, Schuster Mike, Shlens Jonathon, Steiner Benoit, Sutskever Ilya, Talwar Kunal, Tucker Paul, Vanhoucke Vincent, Vasudevan Vijay, Viegas Fernanda, Vinyals Oriol, Warden Pete, Wattenberg Martin, Wicke Martin, Yu Yuan y Zheng Xiaoqiang. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/ (visitado 14-07-2021).

[33] Francois Chollet y col. *Keras*. 2015. URL: https://keras.io (visitado 14-07-2021).

[34] Hongkun Yu, Chen Chen, Xianzhi Du, Yeqing Li, Abdullah Rashwan, Le Hou, Pengchong Jin, Fan Yang, Frederick Liu, Jaeyoun Kim y Jing Li. *TensorFlow Model Garden*. https://github.com/tensorflow/models. 2020.