Modelado de GPUs e implementación de características dentro de Accel-sim

Grado en Ingeniería Informática

Trabajo Fin de Grado

Autor:
Juan José Castillo Otón
Tutor:
Manuel Eugenio Acacio Sánchez

Facultad
Informática
Universidad
Murcia

22 de Enero de 2024

Resumen

El avance en la eficiencia y potencia de las GPU ha sido posible gracias a la fuerte investigación que hay detrás en el campo de arquitectura de computadores. Esta investigación que se realiza tanto de forma privada como de forma pública, es vital para el desarrollo tecnológico. Uno de los problemas de la investigación privada es el secreto industrial, que ha llevado a los investigadores a crear modelos de simulación que se aproximen a las arquitecturas actuales. Uno de estos simuladores es Accel-sim [1], el cual intenta recrear la arquitectura de las GPUs de NVIDIA mediante la simulación de trazas en ensamblador (tanto en PTX como en SASS). Este trabajo se va a centrar en la utilización de este simulador para generar trazas, caracterizar una GPU real dentro del simulador y llevar a cabo modificaciones en el simulador, en concreto se implementarán un planificador de hilos y una nueva política de cache dentro del simulador.

Utilizando el simulador dentro de Accel-sim, GPGPU-sim, se ha modificado para conseguir estadísticas con las que alimentar un programa que se ha implementado para generar Roofline models. Además, se ha implementado en GPGPU-sim un planificador de warps aleatorio y la política de caches NRU.

Se han realizado múltiples simulaciones con varios benchmarks de la suite Rodinia 2.0 [2] con los cuales se han obtenido resultados sobre diferentes planificadores de warps implementados en GPGPU-sim (El simulador de GPU que utiliza Accel-sim), además del planificador Random (selecciona aleatoriamente cada warp). Con estas simulaciones se ha llegado a la conclusión de que Oldest First junto Gready the Oldest son los mejores planificadores de warps para los kernels ejecutados. En adición, se ha demostrado que Restricted Round Robin es un simulador que tiene una efectividad inferior que un planificador de warps aleatorio.

En adición, se han realizado las mismas simulaciones con una política de cache diferente, NRU (Non Rencently Used). De forma generalizada se encuentra que NRU y LRU se comportan de forma muy similar, no habiendo mas de un 1% de mejora entre los diferentes planificadores. Sin embargo, cuando el benchmark simulado da mejores resultados con NRU que con LRU (un 2% en nuestro caso) parece existir indicios de una posible correlación entre el planificador de warps elegido y la política de cache seleccionada. El tamaño de los experimentos es pequeño, por tanto, sería necesario realizar más experimentos para poder corroborar una posible correlación.

Extended Abstract

Graphical Processing Units were created as a specific hardware accelerator to exploit the parallelism in multimedia and screen rendering workloads. It was found that it could be leveraged to exploit the parallelisation of other types of workloads, such as scientific computing or artificial intelligence, creating the concept of General Purpose Graphical Processing Units GP-GPUs. As a result of this, the evolution of this hardware accelerator was fast and has led to a revolution in neural networks. Today, its use is widespread in the viewing of videos on PC,laptop and other devices, videogames rendering, the use of popular social media. It is clear then that GPUs have a significant impact on the present and it will continue to do so in the future.

As in many fields, the main goal in hardware architecture is improving performance and executing kernels as quickly as possible, but instruction performance is not the only factor to be taken into account. Others include energy efficiency, compatibility and portability, all of which require a significant effort to keep pace with the evolution of the industry.

There is a large amount of research on this topic, which is divided between their public and private spheres. Private research is the main source of innovation, as it is the only one capable of manufacturing chips, which are costly to design and produce, thus not many businesses are able to do so. As a result, the exact details of industrial designs are not disclosed which hinders the public investigation of both computer architecture and, subsequently, GPUs. The public sector has to research and develop its own tools in parallel to the private, due to the secrecy of the latter in order to maintain a competitive advantage. This is best demonstrated by the mISA (machine Instruction Set Architecture) from NVIDIA SASS (modern Source and ASSembly). This mISA is not publicly available and its purpose is to maintain flexibility when changing from one generation of micro-structures to another. This allows changes that are not easily detectable by competitors and having the freedom to change the microarchitecture without the restrictions of backwards compatibility, since the virtual ISA exposed is a well documented one named PTX.

In the current world of GPU architecture, Accel-sim [1], whose public repository can be found in github.com/accel-sim, is attempting to accurately model the way in which GPUs work at present.

More specifically, the simulator used by Accel-sim is called GPGPU-sim. This simulator has a performance model that outputs a set of counters that have 1:1 equivalents with hardware data emitted by NVIDIA profilers, asserting high detail and accuracy in the simulation. This simulator, which works specifically with GPUs from NVIDIA, simulates the execution of kernels written in PTX, in a GPU whose characteristics are defined in a configuration file whose features

are highly modifiable. Accel-sim broadens the functionality of GPGPU-sim. Real GPUs can be used to execute a benchmark and generate a trace file written in SASS. Then, SASS parser feeds the simulator with instruction to simulate a specific kernel. Furthermore, Accel-sim has a tuner tool which is able to extract the information of a GPU using a collection of benchmarks and, through this obtaining a configuration file that defines a "virtual" GPU very similar to the real GPU.

In addition to creating these tools, designed to simulate new solutions implemented by the research community, it is necessary to present the information in a coherent and concise manner. To this end, data is usually represented in the form of graphics. When it comes to the field of computer architecture, and more specifically accelerators, Roofline Model is the preferred option.

This chart outlines the theoretical limit of the maximum performance level that can be obtained by an application when being executed by this specific processor. Performance is usually represented in FLOPs (Floating Operation per second). To obtain the peak performance, an application has to have a specific characteristic called Arithmetic Intensity (AI). AI is an inherent property of each algorithm, which defines the number of floating instructions that the program executed by the bytes read. Joining all these concepts, we can draw a Roofline Model. The uses of this chart are, comparing different architecture, visualise possible problems in the processor pipeline and bandwidth problems.

Our research proposal is creating an application that automatically creates Rooflines Models. This app would be integrated in the GPGPU-sim simulator and would be created with the goal of facilitating the viewing of information about the execution of kernels allowed by this simulator to the researchers who employ this tool. As an input, it would have statistics files that generate GPGPU-sim and, as an output, the creation of a Roofline Model.

Furthermore, one of the aims of this project is to verify the proper functioning of the tools provided in Accel-sim, using the Tuning tool to obtain a configuration file that approximates the characteristics of a real GPU. Afterwards, the functionality to obtain SASS traces of microbenchmarks executed on the GPU will be leveraged, and they will be used for subsequent simulations.

In addition, a new warp scheduler has been implemented in GPGPU-sim. A warp is a grouping of threads that are executed in the real GPU hardware. Warps are what are executed within each SM (Streaming Multiprocessor in NVIDIA terminology). Like any processor, for the execution of a set of threads, a scheduler is needed to indicate the execution order of these threads. Currently, there are different schedulers such as GTO (Greedy the oldest), which keeps selecting the warp running as long as it is not blocked, with the next one chosen being the oldest warp. LRR (Loose Round Robin) chooses the next thread in a circular list each time a thread needs to be selected, RRR (Restricted Round Robin) applies the same idea as GTO to Round Robin, OF (Oldest First) selects the thread that has been waiting the longest each time a thread is scheduled. Given this variety of schedulers, we have devised a new scheduler, the Random Scheduler, which randomly selects the warp. The creation of this scheduler is justified in testing the effectiveness of all the aforementioned schedulers because, if there is no substantial improvement compared to this scheduler, we can reason that it is not a scheduler worth implementing.

An important part of the architecture of any chip is the cache replacement policy. Caches are a small set of memory that are fed by the main memory. When caches are full you need to

replace one cache block. Which block you choose to replace can seriously affect the performance of the chip. The most commonly used policy is the Least Recently Used (LRU), which evicts the block of cache which hasn't been used for the longest time. LRU doesn't work well with distant re-reference interval [3]. As a result, we implemented a new cache replacement policy in GPGPU-sim, an NRU (Not Recently Used). This replacement policy is more suitable than LRU when the progam has more distant re-reference intervals. It works with one bit, the NRU-bit, when the bit is 0 it means that there is a distant re-reference in the block but when the bit is 1 it means that there is a re-reference in the near-immediate future. This way, we can model the re-referenceation of the cache blocks and not evict them so easily.

With regards to the methodology employed, the creation of the Roofline Model Generator employed the Python programming language along with the matplotlib library. This was chosen due to the flexibility and intuitiveness of Python, alongside its extensive use in data analysis and data collection. Additionally, the matplotlib library is specifically designed for plotting. Furthermore, it was necessary to modify the GPGPU-sim code to obtain essential information from each benchmark, such as floating-point instructions per second.

For the use of the Accel-sim tools, a computer with the following specifications was employed: processor - AMD Ryzen 7 5800X 3.8GHz; secondary memory - 1TB SSD PCIe 4.0; main memory - 32 GB RAM DDR4 3600MHz; GPU - RTX 3060 TI. The tools of Accel-sim were employed following the steps outlined in the usage guide available on the Accel-sim GitHub. The implementation of the Random Scheduler was performed within GPGPU-sim by modifying the simulator's code to introduce a new warp scheduler. This scheduler can be selected within the GPGPU-sim configuration file to define the characteristics of each graph.

To compare this scheduler with others, the configuration file obtained from the execution of the Accel-sim tuner tool and the SASS traces from the GPU of several applications from Rodinia 2.0 benchmark suite were utilized. Executing these traces while changing the scheduler in the configuration file is how the results were obtained.

The results of the first part of this work have been mostly positive. The SASS traces were successfully extracted using the tracer_nvbit tool according to the instructions. Accel-Sim Tuner was used to obtain a configuration file approximating the real GPU. However, this configuration file was not generated coherently, and GPGPU-sim was unable to process it. These inconsistencies were related to the arrangement of memory banks in L1 and L2 caches, which were resolved by using the L1 and L2 memory bank configuration of the GPU from the same generation, RTX 3070, provided by Accel-sim. Therefore, several parts of the configuration file had to be modified to make it usable.

The next inconsistency is related to hashing within the simulator. The configuration generated by the Tuner requires the implementation of new entries for the IPOLY hashing function within the simulator. This includes special cases where the entry has 2 subdivided memory banks, resulting in a hashed entry divided into 4. With these modifications, the proper functioning of this configuration file has been achieved.

The execution of the Rodinia 2.0 benchmarks with different schedulers, including GTO, LRR, RRR, OF, and Random shows that the better schedulers among the benchmarks executed are OF and GTO. OF and GTO follow a similar principle they use a oldest first list. The differences are that GTO are greedy and executes the same warp until it blocks. The results shows that being greedy or not is not that important. Another noteworthy result is the inefficiency of the RRR scheduler compared to all others, being significantly worse than the Random Scheduler.

This makes it a suboptimal scheduler for implementation in real hardware. Additionally, we found that depending on the kernel, the Random scheduler is on the same level of performance as the LRR and OF schedulers. Besides RRR are 5% slower than LRR. This is interesting because follows a similar principle as OF and GTO does. Therefore, RRR is slower that means that the greedy approach is not meanless and sometime can give to a drawback in performance.

Besides the execution of the Rodinia 2.0 with LRU policies, we also executed the same benchmarks with the same schedulers from the NRU cache replacement policy. In doing so, we found that in general the differences between NRU and LRU are very subtle. Although when a specific benchmark is faster in NRU than in LRU (in this case the benchmark Backprop is 2.2% faster) then there are more differences between LRU and NRU also between different schedulers inside the NRU policies. We found a 4.4% increase in speed between GTO and LRR with NRU replacement. Moreover, the best warp scheduler in Backprop with NRU is LRR. Therefore, it is likely that a correlation exists between the scheduler and the cache replacement which is used. In our case this correlation is minimal and only happens when the NRU is better than the LRU.

For future work, it would be advisable to expand the number of GPU configuration files and identify common errors that the Accel-sim Tuner tool may have, thus enabling the development of a systematic solution. Moreover, increasing the number of benchmarks used to assess the varied performances of different schedulers and cache police replacement, including the Random scheduler, could provide more accurate conclusions about the usefulness of implementing specific schedulers.

A research avenue that should be pursued involves the creation of new schedulers aimed at improving existing ones. This involves providing more information within the SM to achieve better warp scheduling for an enhanced performance.

Índice general

Índice de figuras

1 Introducción

Las primeras generaciones de GPU (Graphical Processing Unit) fueron creadas con la intención de acelerar el cómputo de aplicaciones multimedia y en la rasterización de polígonos tanto para videojuegos como para aplicaciones de edición gráfica. Estos chips se centraban en la explotación del paralelismo que existía en las operaciones de las aplicaciones gráficas. Esto hizo que las GPU dedicaran más silicio a las unidades de cómputo que a la jerarquía de memoria y tratamiento de lógicas complejas. La arquitectura de GPUs se basa en núcleos que se encargan de ejecutar distintos hilos con instrucciones SIMD. GPGPU-sim llama a estos núcleos SIMT (Single Instruction Multiple Thread) similares a los que CUDA llama Streaming Multiprocessor, que no hay que confundir con la denominación de NVIDIA CUDA cores, que se refiere en realidad a las unidades funcionales dentro de los Streaming Multiprocesors. Dentro de cada núcleo SIMT la ejecución de instrucciones se produce con una agrupación lógica de hilos llamada warp, esta agrupación es donde convergen el modelo de programación de CUDA (infinidad de hilos independientes) con el modelo de ejecución (SIMD) trasformando un grupo de hilos en una instrucción con múltiples datos. La memoria dentro de una GPU se divide en diferentes tipos de memoria, la memoria global que se aloja dentro de la VRAM y es accesible por todos los hilos, la memoria compartida que es accesible por un grupo de hilo llamado *thread block* y dentro de esta agrupación nos encontramos la memoria privada solo accesible por el warp y donde nos encontramos los registros utilizados para poder cambiar de contexto entre warps rápidamente dentro de un núcleo SIMT.

Esta característica de centrar las GPUs en el cómputo dio lugar a las GPGPUs (General Purpose Graphics Processing Unit). Ya que resulta que las GPUs son muy efectivas a la hora de ejecutar cálculo científico, esto es debido a que el cálculo científico necesita la realización de multitud de operaciones que generalmente se pueden realizar en paralelo, por ejemplo, en simulaciones físicas. Otro campo muy importante actualmente es el entrenamiento y ejecución de redes neuronales profundas, denominadas DNNs (Deep Neural Network). Podríamos pensar que la idea de una red neuronal es algo actual dado a la gran relevancia que tiene en la comunidad investigadora actualmente. Sin embargo, esto es falso ya que la idea de red neuronal ya fue concebida en 1943 por Warren S. Mcculloch y Walter Pitts [7]. La razón clave de por qué las redes neuronales han vivido esta gran expansión en estos años es el avance de la computación y en específico de las GPU. Un ejemplo del gran avance de las redes neuronales, son los grandes modelos del lenguaje como GPT4 [8]. Este gran modelo del lenguaje de OpenAI ha causado un gran impacto debido a sus capacidades y a la popularización de chatGPT, que utiliza por debajo GPT3.5 y GPT4. No hay información pública sobre los detalles del entrenamiento de GPT4. Pero sí hay información de los detalles de GPT3, siendo entrenado con 175 mil millones de parámetros, requiriendo miles de Petaflop/s-days [9]. Las GPUs utilizadas para el entrenamiento de GPT3 fueron las V100 de NVIDIA, la arquitectura de estas GPUs, al igual que todas las GPUs de NVIDIA, está oculta públicamente, tanto es así que la ISA utilizada dentro de las GPUs de NVIDIA está oculta detrás de un ISA virtual, PTX.

Para poder realizar investigación pública de la arquitectura de GPUs de NVIDIA se desarrolló Accel-sim y GPGPU-sim. Accel-sim [1] como una herramienta de usuario que añade funcionalidades al simulador GPGPU-sim. Gracias al uso de las herramientas de Accel-sim se ha realizado el modelado de una GPU real y la generación de trazas de la suite de benchmarks Rodinia [2]. Además, se han realizado modificaciones dentro del GPGPU-sim tanto para conseguir mayor información de la simulación como para implementar nuevos políticas dentro del mismo.

Las aportaciones que se realizan en este TFG son las siguientes:

- Creación de un programa que dibuja roofline models alimentado por estadísticas personalizadas dentro de GPGPU-sim.
- Modelado de una GPU y generación de trazas utilizando las herramientas de Accel-sim.
- Implementación de un planificador de warps aleatorio dentro de GPGPU-sim.
- Implementación de la política de reemplazo de cache NRU dentro de GPGPU-sim.

El resto de la memoria se organiza como sigue:

Capítulo Contexto. En este capítulo se exponen conceptos tratados dentro del TFG relacionados con las GPUs, con la intencionalidad de marcar un lenguaje común para cualquier lector que decida leer este trabajo.

Capítulo Generación de Roofline Models con Accel-sim. Se realiza una explicación del código necesario para implementar las estadísticas necesarias para construir la herramienta para dibujar roofline models, junto con una explicación de la propia herramienta y un manual de usuario.

Capítulo Modelado de una GPU con Accel-sim. Se han utilizado las herramientas proporcionadas por Accel-sim para generar un fichero de configuración que modela una GPU real y la generación de trazas de Rodinia. Además se indican los errores generados por estas herramientas y una solución parcial.

Planificación de Warps en GPGPU-sim. Se expone la implementación de un nuevo planificador de warps aleatorio y experimentos realizados por el mismo para llegar a la conclusión de la ineficiencia de Restricted Round Robin.

Política de Remplazo de Caches en GPGPU-sim. Implementación de NRU dentro de GPGPU-sim, además la presentación de resultados de experimentos del capítulo anterior pero con la política de remplazo NRU.

2 Contexto

2.1 Descripción de una arquitectura de GPU

Las GPUs como su nombre indican (**Graphic Processing Unit**) pertenecen a una arquitectura de propósito especifico, o al menos esto es lo que se pretendió en sus inicios. Actualmente las GPUs se les denomina también GP-GPU de **General Propouse-Graphic Processing Unit** ya que estas son útiles en muchos otro tipo de tareas a parte de los gráficos.

La característica que se puede explotar de los programas gráficos es la reiteración de muchas operaciones simples que se pueden paralelizar. Por tanto, el modelo de ejecución que siguen las GPUs es un modelo de instrucciones SIMD según la taxonomía de Flynn[10] que explota la paralelización de los datos realizando una misma operación a múltiples datos. Otra característica que diferencia las GPUs a una CPU es que ocupa más transistores para las unidades funcionales y menos en cachés y control de flujo de instrucciones, pudiendo así dar una capacidad de cómputo mucho mayor. Esto lo podemos ver en la figura 2.1.

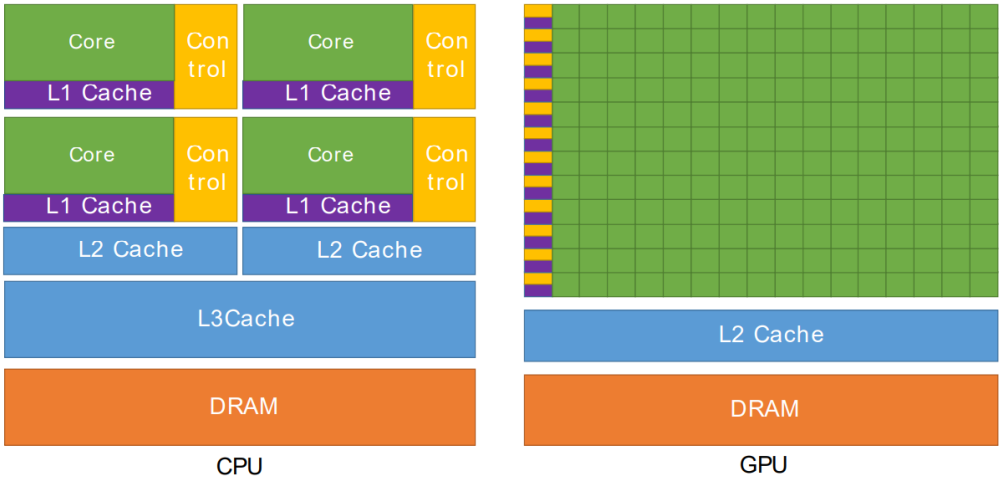

Figura 2.1: Diagrama de una CPU y una GPU [4]

2.1.1 Modelo de ejecución CUDA

Para la programación de propósito general en GPUs, existen dos modelos de programación OpenCL y CUDA. Una observación interesante [11] en el contexto de OpenCL es que el código que ha sido optimizado para funcionar en una arquitectura como la de una GPU se comporta de forma mucho peor en otra como la de una CPU. Accel-sim se especializa en ejecutar simulaciones de gráficas de NVIDIA por lo tanto vamos a centrarnos en CUDA.

Según *CUDA C++ Programming Guide* [4]. El objetivo del modelo de lenguaje de CUDA (al igual que de otros como OpenCL) es poder crear software que de forma trasparente escale

en paralelismo al igual que lo hacen las aplicaciones de gráficos 3D. Además, busca que sea amigable con programadores que estén familiarizados con lenguajes como el de C/C++.

Para lograr esto se definen dos abstracciones clave: la jerarquía de grupos de hilos y la jerarquía de memoria.

Jerarquía de hilos

Los hilos en CUDA (también en OpenCL) son vistos como hilos independientes que ejecutan el mismo programa, sin embargo, los hilos pueden seguir un flujo de ejecución diferente dependiendo de las estructuras condicionales dentro del código.

Cada hilo puede ser accedido por un *thread index*. Esta identificación es un vector que puede ser de 1, 2 o 3 dimensiones para que así se pueda formar un grupo de hilos de 1, 2 o 3 dimensiones llamados *thread block*. Esta disposición de hilos permite que se consiga modelar de forma natural operaciones en vectores, matrices o volúmenes.

Existe un límite en la cantidad de hilos que se pueden ejecutar en un *thread block* ya que cada hilo de un *thread block* se espera que se ejecute dentro de la misma unidad de procesamiento. En GPUs actuales de NVIDIA el límite es de 1024 hilos. Dentro del hardware real existe una organización de hilos intermedia llamada *warp*. Un warp consiste en un grupo de 32 hilos que se ejecutan en la misma unidad de cómputo, por lo tanto podríamos decir que cada *thread block* contiene 32 warps.

Para conseguir que la distribución de *thread block* sea efectiva existe una estructura que está encima de los *thread blocks* llamada *grid*, que se puede ver en la figura 2.2. Esta estructura también puede ser organizada de 1 a 3 dimensiones. El número de *thread blocks* dentro de un *grid* tiende a estar definido por el tamaño de datos que se esta procesando, que suele superar la cantidad de procesadores en el sistema, siendo esto algo importante para una correcta utilización de todos los procesadores de una GPU.

Nótese que en la nueva arquitectura Hopper de NVIDIA existe una nueva jerarquía *thread block cluster*. Este nuevo grupo lógico de hilos se encuentra encima de los *thread blocks* y por debajo del *grid*. El significado de esta nueva división se debe a que la memoria compartida de cada *thread block*, que se ejecuta dentro de un SM es accesible por otros SM que se encuentran dentro del mismo *thread block cluster* [12].

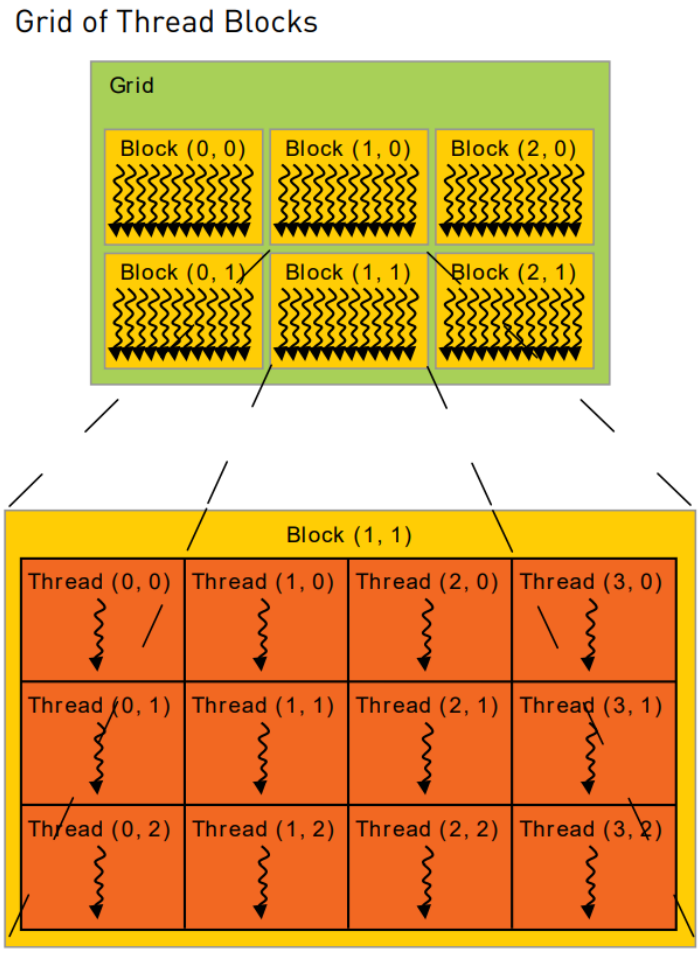

Figura 2.2: Diagrama de un *grid* en CUDA [4]

Jerarquía de Memoria

Los hilos de CUDA pueden acceder a datos desde múltiples espacios de memoria durante su ejecución, como se ilustra en la Figura 2.3. Cada hilo tiene su propia memoria local privada. Cada *thread block* tiene una memoria compartida visible para todos los hilos del *thread block* y con la misma duración que el *thread block*. Todos los hilos que se encuentran en el *grid* tienen acceso a la misma memoria global.

Además, hay dos espacios de memoria adicionales de solo lectura accesibles por todos los hilos: los espacios de memoria constante y textura. Los espacios de memoria global, constante y textura están optimizados para diferentes usos de memoria. La memoria de textura también ofrece diferentes modos de direccionamiento, así como filtrado de datos para algunos formatos de datos específicos.

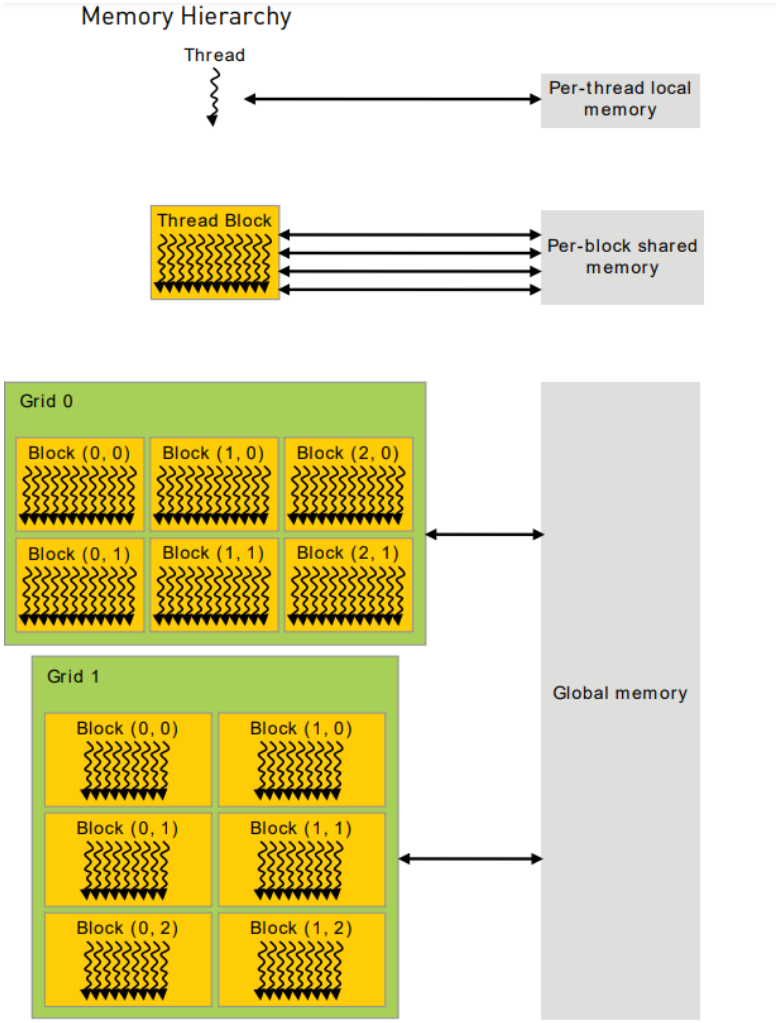

Figura 2.3: Diagrama de la jerarquía de memoria en CUDA [4])

2.1.2 Conjunto de instrucciones de GPUs de NVIDIA

Cuando NVIDIA introdujo CUDA al mercado decidieron seguir la tendencia de los fabricantes de GPU anteriores de crear un conjunto de instrucciones de la arquitectura de alto nivel (en ingles *ISA, Instruction Set Architecture*) para sus GPU llamado "Parallel Thread Execution ISA", denominado comúnmente como *PTX*. NVIDIA documenta este *ISA* con cada salida de una nueva arquitectura. Gracias a esto GPGPU-sim soporta simulaciones que aceptan de entrada ficheros *PTX*.

PTX es similar a un conjunto de instrucciones reducido *RISC*. Antes de poder ejecutar código *PTX* es necesario compilarlo al actual *ISA* que soporta el hardware. NVIDIA llama a este ISA *SASS* que proviene de "Streaming ASSembler". NVIDIA no documenta de forma completa *SASS* lo que complica a los investigadores académicos crear simuladores que sean precisos [13].

2.1.3 Arquitectura Interna

En esta sección se va a hablar sobre la arquitectura de una GPU desde la vista de cómo se modela en GPGPU-sim [5].

La GPU modelada por GPGPU-sim está compuesta por Single Instruction Multiple Threads (SIMT). Un núcleo SIMT modela un SIMD pipeline multihilo, que en terminología de NVIDIA se llama Streaming Multiprocesor (SM). la organización de un SM se puede ver en la figura 2.4.

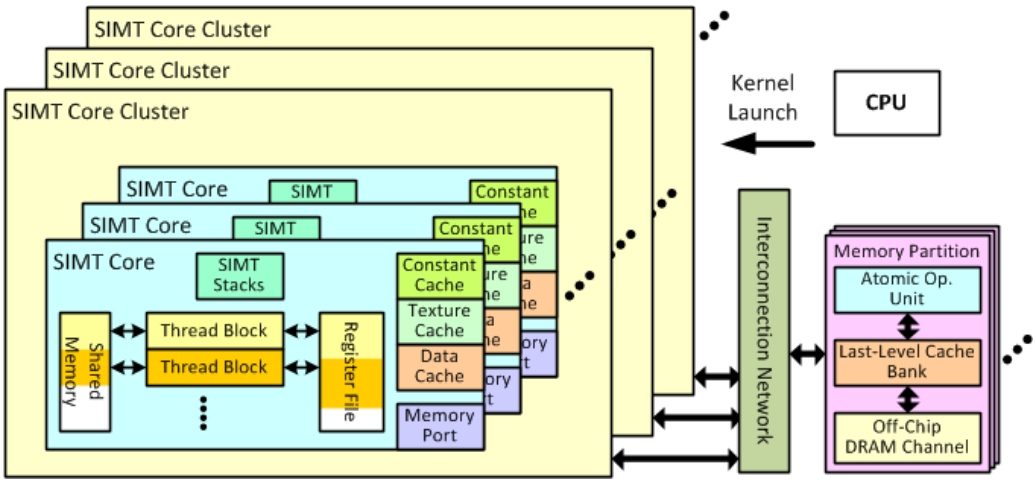

Figura 2.4: Arquitectura de una GPU dentro de GPGPU-sim [5]

Los núcleos SIMT, ilustrados en la figura 2.6. Están organizados en SIMT cluster que comparten un puerto de interconexión común. Cada SIMT cluster contiene una *response FIFO* que contiene los paquetes de la red de interconexión, estos paquetes son redirigidos a la caché de instrucciones del núcleo SIMT o a la pipeline de memoria (LDST unit).

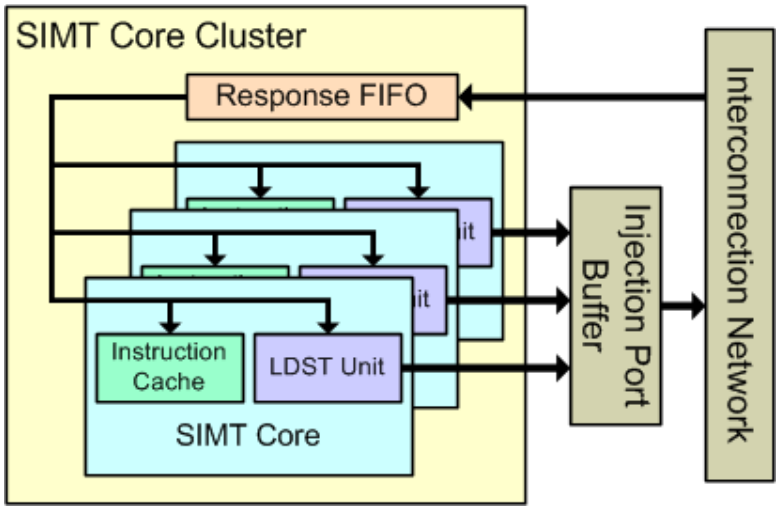

Figura 2.5: Modelado de SIMT core dentro de GPGPU-sim [5]

Podemos ver en detalle un núcleo SIMT en la figura 2.6. El núcleo SIMT se compone de dos partes, un front end pintado en verde y un back end pintado en amarillo. El front end se encarga: de buscar instrucciones, decodificarlas, meterlas dentro del buffer de instrucciones y buscar sus operandos en el score board. El back end recibe las instrucciones listas para ejecutar y las envía a ejecutar a las estructuras correspondientes dependiendo si son instrucciones de memoria o aritmético-lógicas.

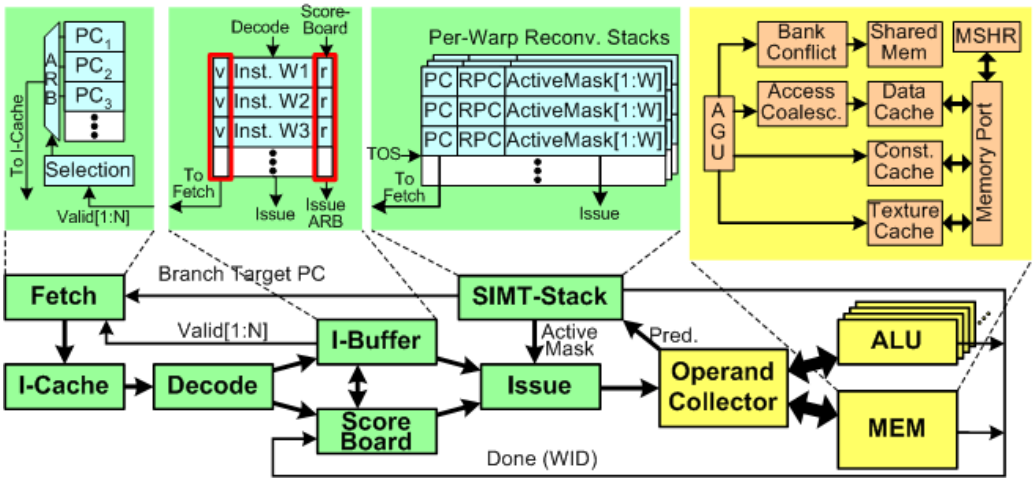

Figura 2.6: Esquema del interior de un núcleo SIMT dentro de GPGPU-sim [5]

2.1.4 Políticas de remplazo de caché

Las cachés son estructura de memoria limitadas. Se encuentran cerca de los núcleos de los procesadores y su trabajo es mantener los datos más usados dentro del procesador para que los accesos sean rápidos. Debido a esta limitación, las cachés no son de un gran tamaño y se dividen en diferentes niveles en lo que se llama la jerarquía de cachés.

La caché que se encuentra más cerca del núcleo se le suele llamar L1 y son las más rápidas y también las más pequeñas. En el caso de las GPUs la L1 también es utilizada para guardar registros, permitiendo así el rápido cambio de contexto entre un warp y otro.

El reducido tamaño de las cachés dan lugar a un problema, cuando necesitamos más datos en la caché y la caché esta completa ¿Qué dato tenemos que expulsar? Esta pregunta es la que intenta solucionar la política de remplazo de cachés.

La política de remplazo de cachés es un aspecto crítico en cuanto el rendimiento de cualquier procesador, ya que las instrucciones que mas tardan en ejecutarse son las de memoria. Esto no es excepción en las GPUs, sino que se agrava al no tener una jerarquía de caché tan grande como suelen tener las CPU.

La política de remplazo de caché más utilizada es LRU (Least Recently Used). Esta política utiliza un grupo de bit (age bits) para llevar la cuenta de cuál es el bloque de caché mas antiguo. Cuando se realiza un acceso de caché se actualizan los *age bits* del resto de líneas de caché y se escriben a 0 las líneas de caché utilizadas. Cuando hay que expulsar una línea de caché se elige la línea con mayor número en los *age bits*.

El problema que presenta esta alternativa es que no funciona bien cuando hay referencias lejanas en el tiempo de los datos [3]. En el caso de las GPUs una de las posibles razones por las que existan referencias alejadas en el tiempo puede ser, por ejemplo, en el caso donde un warp se bloquee por una petición de memoria. Por tanto, el planificador de warp elige otro warp. Hasta que vuelva a aparecer ese warp habrán pasado diferentes warp por la pipeline con diferentes peticiones de memoria. Nuestro primer warp al coger otra vez la pipeline necesitará el mismo dato con el que se ha bloqueado, pero al haber estado mucho tiempo sin utilizarse con una política LRU este puede haber sido expulsado y necesitará realizar la petición otra vez habiendo utilizado una costosa petición de memoria para nada.

Una solución para la problemática de los accesos lejanos en el tiempo es la política de remplazo de cachés NRU (Non Rencently Used). Esta política de caché utiliza un bit por bloque de caché llamado *nru-bit*, este bit tiene dos estados *Recently Used* y *Non Recently Used*. Cuando el bit está a 1 significa que ha sido usado recientemente y cuando el bit está a 0 significa que no ha sido usado recientemente.

Cuando hay un fallo de caché, el bit *nru-bit* del bloque que ha provocado ese fallo de caché se pone a 1 ya que se da por hecho que va a ser referenciado en un futuro cercano y el resto de bloques del conjunto se ponen a 0. En el caso de haber un acierto de caché el *nru-bit* se actualiza a 1. Finalmente, cuando se tiene que expulsar un dato en caché se busca un bloque que tenga el *nru-bit* a 0, si todos los bloques de caché tienen el *nru-bit* a 1 entonces se elegirá uno aleatoriamente.

2.2 Roofline Model

El Roofline Model es una forma gráfica de representar el rendimiento de una GPU o de cualquier chip de cómputo como una CPU o una TPU y poder compararlo con otros chips de características similares (similares en ancho de banda y potencia de cómputo) [14]. El Roofline Model esta orientado para ser capaz de visualizar de una manera sencilla la localidad de datos, el ancho de banda de memoria y la potencia de cómputo. Para esto se utiliza una medida intrínseca de las aplicaciones: La intensidad aritmética.

La intensidad aritmética se define como el ratio de todas las instrucciones de punto flotante con la cantidad de datos que se mueven en total. Esta se representa por $\frac{GFLOP}{Byte}$. Cuanto mas intensidad aritmética tenga una aplicación habrá un mayor aprovechamiento de las unidades de cómputo del procesador en comparación con la presión de memoria. Por lo tanto cuanto mayor intensidad aritmética, se necesitará un menor ancho de banda para aprovechar la potencia de cómputo total de la unidad de procesamiento.

Dependiendo de la aplicación, el ratio de intensidad aritmética puede aumentar según aumentamos el tamaño de entrada. En la figura 2.7 podemos ver diferentes aplicaciones y el orden de crecimiento de las mismas.

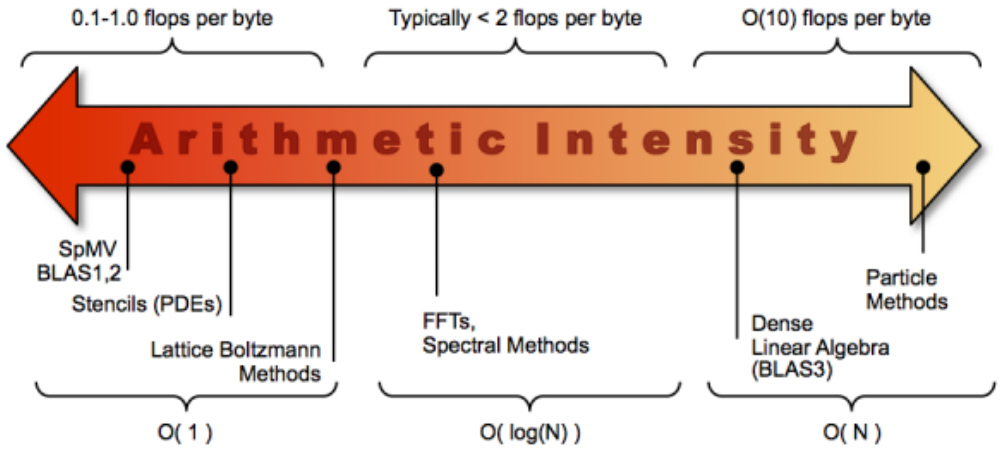

Figura 2.7: Diferentes Intensidades Aritméticas según aplicaciones [6]

Para la construcción de un Roofline Model se necesita dos valores específicos de la arquitec-

tura que queremos caracterizar y estos son: el pico de ancho de banda y el pico de FLOPS. Con estos dos valores se calcula el punto donde $PEAKbw \cdot AI = PeakGFLOPS$. Dibujamos así los FLOPS en función de la intensidad aritmética obteniendo un gráfico como el de la figura 2.9

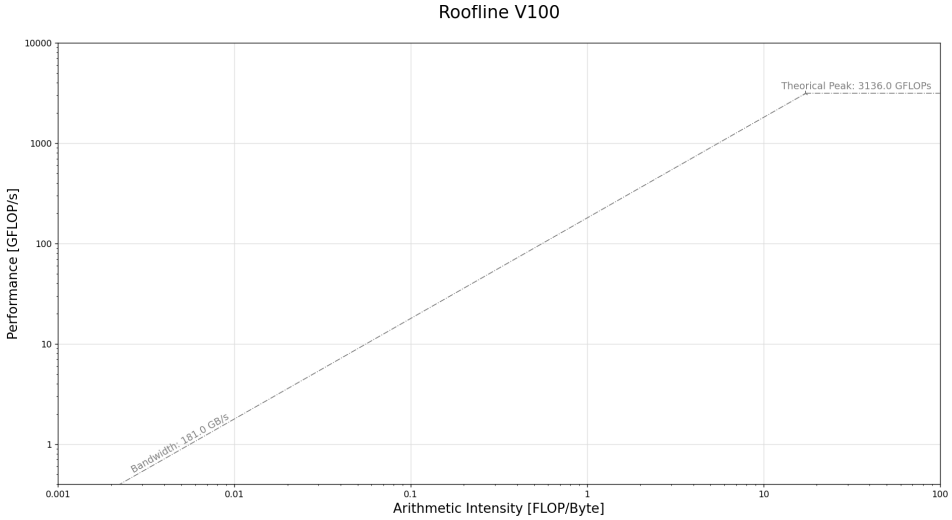

Figura 2.8: Roofline de la GPU Nvidia QV100 generado con el programa explicado en el capítulo 3

Ahora teniendo el Roofline podemos dibujar las aplicaciones que ejecutamos en la gráfica para poder comparar su rendimiento según su Intensidad Aritmética, teniendo en cuenta que el máximo teórico que una aplicación puede alcanzar es la línea superior.

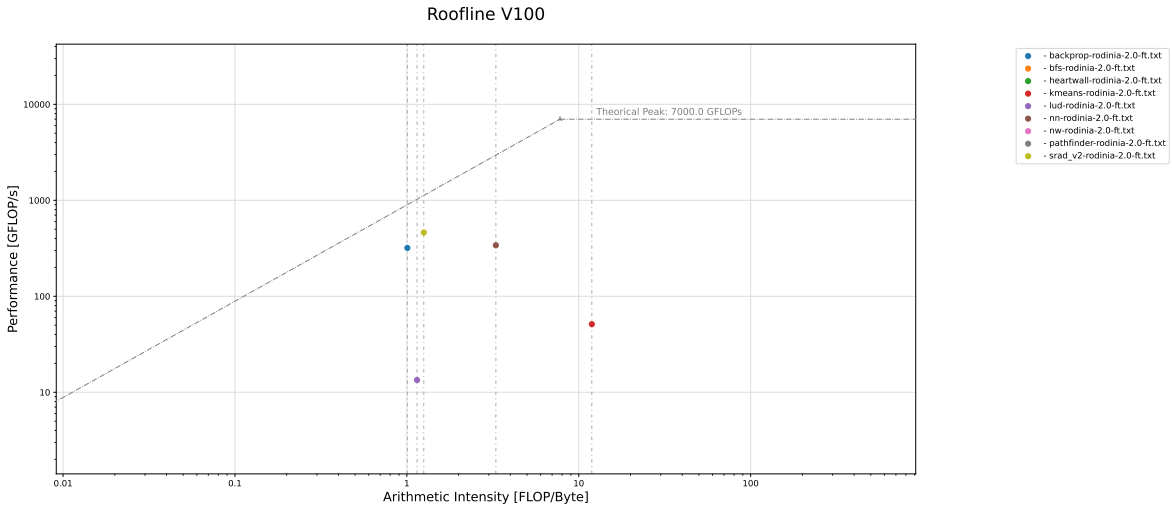

Figura 2.9: Roofline de la GPU Nvidia QV100 con benchmarks generado con el programa explicado en el capítulo 3

2.3 Accel-sim

Accel-sim es una herramienta para lanzar simulaciones de GPGPUs, podemos ver su estructura general en la figura 2.10. Para este propósito el simulador que utiliza es GPGPU-sim 4.0. Este es un simulador de GPGPUs orientado a las GPUs de NVIDIA, dentro de su funcionalidad esta la de simular la ejecución de una gráfica pasándole como parámetro una configuración determinada y el archivo ptx con el kernel a simular. En esta simulación devuelve estadísticas relevantes, como el número de ciclos que ha tardado el programa en ejecutarse, el tipo y cantidad de instrucciones que se han ejecutado en la gpu, el número de ciclos que el pipeline ha tenido que ser parado y mucha más información sobre la caché y la dram.

Accel-sim añade funcionalidades a GPGPU-sim como la posibilidad de ejecutar simulaciones con trazas de SASS de NVIDIA, sacar trazas de kernels con una GPU física, tunear ficheros de configuración para que simulen lo más fiel posible nuestra gpu y correlación de los resultados de la simulación con la ejecución real, para comprobar la precisión del simulador.

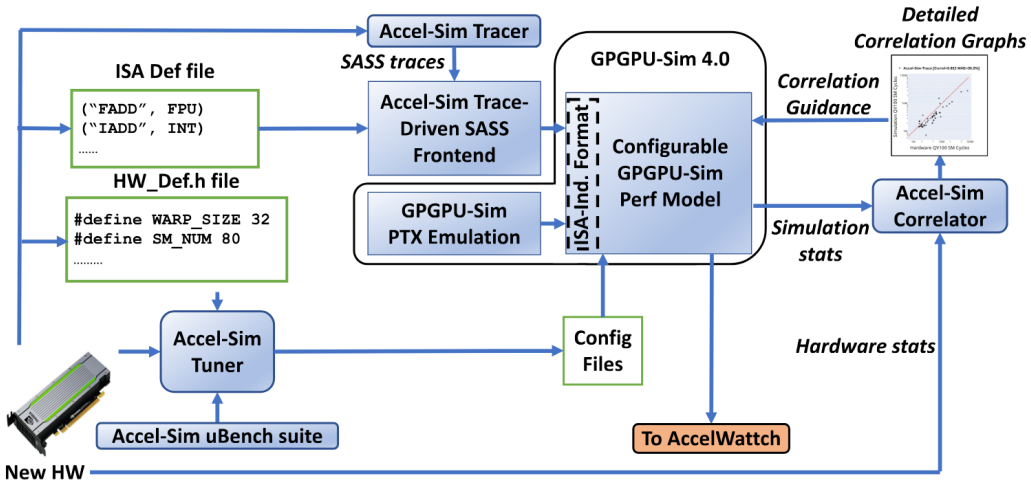

Figura 2.10: Esquema de las funcionalidades de Accel Sim [1]

2.3.1 Planificadores de Warp dentro de GPGPU-sim

Dentro de cada Streaming Multiprocesor, la unidad mínima de ejecución es lo que se define como warp. Cada SM cuenta como mínimo con un Warp Scheduler con una política común de planificación. Como su nombre indica el Warp Scheduler se encarga de seleccionar el siguiente warp que va a entrar a la pipeline, en la figura 2.11 podemos ver el funcionamiento básico de un planificador de warps. La tarea de un Warp Scheduler o planificador de warps en español, es la de conseguir ocultar la latencia de memoria y ejecución de las instrucciones.

Para que sea posible la figura del Warp Scheduler y la forma en la que funciona la ejecución en una GPU, cada SM contiene miles de registros para poder cambiar de contexto rápidamente entre un warp y otro. Por ejemplo, la suma de todos los registros de la arquitectura de NVIDIA Fermi daba un tamaño de 2 MB capaz de mantener 20,000 hilos en vuelo.

Dentro de este TFG se trabaja con 4 schedulers incluidos en el simulador GPGPU-sim, estos son:

- **lrr**: (Loose Round Robin) Este planificador como su nombre indica se basa en una selec-

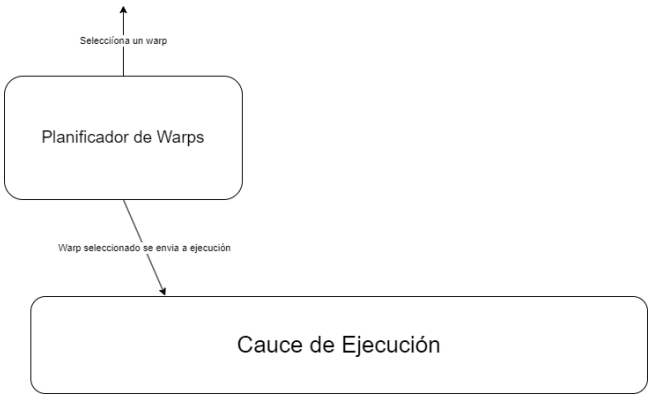

Figura 2.11: Esquema de un planificador de warps

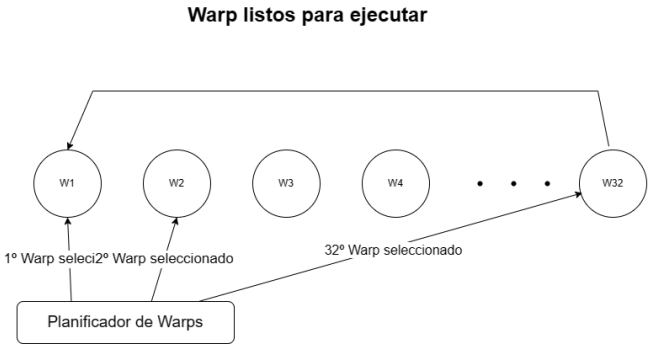

Figura 2.12: Política de planificación de warps Loose Round Robin

ción circular en orden de cada warp. Cada vez que es llamado el planificador, el siguiente warp es seleccionado. En la figura 2.12 se puede ver la selección de un warp siguiendo una lista Round Robin

- **rrr**: (Restricted Round Robin) Al igual que el round robin, la lista de warps que se van a ir ejecutando a continuación del Warp que se ejecuta actualmente, es de forma circular. Este es muy parecido a **lrr**, la diferencia se haya en que cuando se llama al planificador **rrr**, elige siempre el Warp que se este ejecutando actualmente. Elegirá el siguiente warp cuando este este bloqueado por una petición de memoria o una barrera.

- **OF**: (Oldest First) Cada vez que se llama al planificador se elije el warp que ha estado más tiempo sin ejecutarse. En la figura 2.13 se puede ver la selección de un warp siguiendo una lista Oldest First.

- **GTO**: (Gready the Oldest) Sigue una dinámica similar a **rrr**. Hasta que un hilo no esté bloqueado no se elige otro hilo, y en el caso de elegirse otro hilo se elige según el hilo mas tiempo esperando.

2.3.2 Ciclos de Parada

En las estadísticas de GP-GPUsim podemos ver bastante información sobre la ejecución del kernel. Una de estas estadísticas es la cantidad y tipo de ciclos de parada, estos se dividen en tres tipos:

Figura 2.13: Política de planificación de warps Oldest First

- **Stall** : Este tipo de Ciclos de parada significa que ninguna instrucción no se ha podido emitir y por lo tanto tenemos la pipeline parada.

- **W0_Scoreboard** : Este tipo de ciclo parada se da cuando un Warp esta parado esperando un dato por un riesgo RAW, posiblemente causado por una instrucción a memoria.

- **W0_Idle** : Este tipo de ciclo de parada ocurre cuando hay un riesgo de control o el procesador no esta ejecutando ninguna instrucción porque no hay instrucciones que ejecutar.

2.3.3 Estructura de ficheros principales GPGPU-sim

Se han realizado varias modificaciones a GPGPU-sim, se pueden ver la jerarquía de ficheros modificados en la figura 2.14. Para una mayor comprensión de la utilidad de los ficheros que se han modificado vamos a listarlos brevemente.

- **gpu-sim.h** Se definen variables globales para todo el simulador como las variables que se utilizan para imprimir estadísticas globales.

- **gpu-sim.cc** Fichero que junto **gpu-sim.h** agrega varios ficheros que se encargan de funcionalidades específicas del simulador, de inicializar variables globales, actualizarlas e imprimirlas en los ficheros de configuración.

- **gpu-cache.h** Este fichero se encarga de definir las estructuras de las cachés y las funciones que van a tener. Aquí es donde se definen las líneas de caché y sus atributos.

- **gpu-cache.cc** Aquí es donde se implementa el funcionamiento de las cachés y las funciones que van a ser llamadas después por el núcleo.

- **shader.h** y **shader.cc** Estos ficheros modelan el SIMT core y son los ficheros mas importantes a la hora de hacer modificaciones que involucren ver la pipeline de cada SM.

También existen los ficheros que se encargan de parsear los diferentes códigos *PTX* de diferentes arquitecturas. Ha sido necesario investigar para poder entender como se traducen las instrucciones dentro del simulador. Sin embargo, no se ha realizado ninguna modificación en ellos.

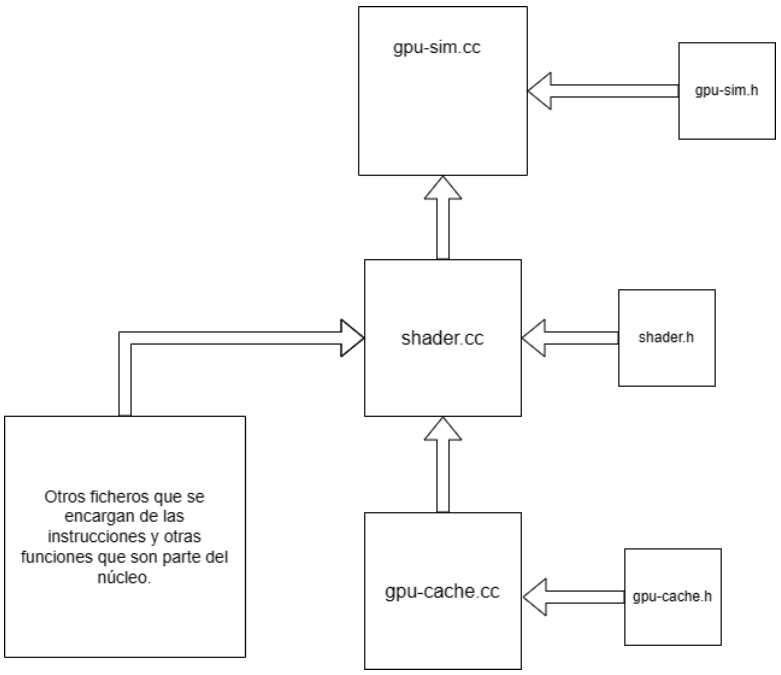

Figura 2.14: Jerarquía de los ficheros modificados dentro de GPGPU-sim

3 Generación de Roofline Models con Accel-Sim

En este apartado se va a hablar de la realización de un programa que permite extraer la información necesaria dentro de los ficheros de estadísticas de GPGPU-sim y dibuja el Roofline Model [15]. Para que el programa pueda extraer las configuraciones necesarias, primero ha sido necesario modificar el simulador para que calcule y muestre las estadísticas necesarias para dibujar cada becnhmark dentro del Roofline Model. Por tanto, este apartado se divide en dos secciones: la modificación del simulador GPGPU-sim para conseguir más información de los ficheros de estadísticas de salida y el manual de uso.

3.1 Modificación de GPGPU-sim para conseguir para posicionar los Benchmarks en el Roofline Model

Como se comentó en la sección 2.2 para conseguir dibujar el resultado de un Benchmark dentro de un Roofline Model necesitamos dos medidas:

Intensidad Aritmética: La Intensidad Aritmética se define como el número de instrucciones de punto flotante por byte leído, por lo tanto necesitamos conseguir el número de instrucciones de tipo punto flotante y el número de bytes que se han leído de memoria.

FLOPS: Para calcular el número de instrucciones de punto flotante por segundo, vamos a necesitar la misma información que para la intensidad aritmética, además de saber el tiempo que tarda en ejecutarse el kernel.

Ninguna de las estadísticas que he mencionado se muestran explícitamente en los ficheros de estadísticas que genera el simulador, debido a esto ha sido necesario modificar el código del simulador para definir nuevos contadores que nos dieran la información que necesitamos.

La primera modificación necesaria para poder conseguir nuestro objetivo es incluir un contador de instrucciones de punto flotante, que se encargue de contar cada vez que una instrucción de punto flotante es ejecutada por cada núcleo. Para ello primero nos tenemos que fijar en la definición de las instrucciones dentro del simulador. Estas se encuentran en el fichero **abstract_hardware_model.h** Dentro de este fichero lo que nos importa es la definición de una serie de funciones booleanas que nos indican el tipo de la instrucción

```
bool is_fp() const { return ((sp_op == FP__OP));}
bool is_fpdiv() const { return ((sp_op == FP_DIV_OP));}
bool is_fpmul() const { return ((sp_op == FP_MUL_OP));}
bool is_dp() const { return ((sp_op == DP___OP));}
bool is_dpdiv() const { return ((sp_op == DP_DIV_OP));}
bool is_dpmul() const { return ((sp_op == DP_MUL_OP));}
bool is_imul() const { return ((sp_op == INT_MUL_OP));}
```

```
bool is_imul24() const { return ((sp_op == INT_MUL24_OP));}
bool is_imul32() const { return ((sp_op == INT_MUL32_OP));}
bool is_idiv() const { return ((sp_op == INT_DIV_OP));}
bool is_sfu() const {return ((sp_op == FP_SQRT_OP) || (sp_op == FP_LG_OP) || (
    ↪ sp_op == FP_SIN_OP) || (sp_op == FP_EXP_OP) || (sp_op == TENSOR__OP));}
bool is_alu() const {return (sp_op == INT__OP);}
```

Teniendo esta información ya podríamos saber si una instrucción es de punto flotante o no creando una función que recopile todas las instrucciones que podrían ser de punto flotante.

```
bool is_fp_all() const {return ((this->is_fp() || this->is_fpdiv() || this->
    ↪ is_fpmul() || this->is_dpdiv() || this->is_dpmul() || this->is_dp() || 
    ↪ this->is_sfu()));}
```

Una vez sabemos si un objeto instrucción es o no de punto flotante, necesitamos contar las veces que un Stream Multiprocessor esta ejecutando una instrucción y si esta es una instrucción de punto flotante añadirla al contador de instrucciones de punto flotante que ya se creó previamente en el fichero **gpu-sim.h** y se inicializó a 0 en el fichero **gpu-sim.cc**.

Para saber si la instrucción se debe contar o no dentro del fichero **shader.cc** dentro de la función **warp_inst_complete** hacemos la comprobación.

```
if(inst.is_fp_all())
  m_gpu->gpu_sim_insn_fp += inst.active_count();
```

Algo que mencionar en esta función es la función **active_count()**, esta función nos devuelve 1 si la instrucción que está ejecutando dentro de un warp está activa o no, ya que esta puede estar desactivada para manejar el control de instrucciones condicionales.

Una vez solucionado el número de instrucciones de punto flotante necesitamos conseguir el número de bytes leídos de memoria por el kernel. Una de las maneras de conseguir esta información es contar los bytes que se cargan desde memoria global, ya que esta memoria se encuentra en la VRAM y toda la información que necesite el kernel tiene que pasar por la memoria global.

Para conseguir esto nos tenemos que fijar en las instrucciones de SASS de Nvidia que nos traen datos de la memoria global. Esto lo podemos encontrar en la documentación de Nvidia [16].

En el simulador podemos ver que dentro de la función **icnt_inject_request_packet** se encuentra el lugar donde la estadística del número de instrucciones de lectura global se suma. Aprovechamos que aquí se realiza la suma de operaciones de lectura en memoria global para actualizar nuestros bytes leídos. Cogemos la petición de memoria y cogemos el tamaño de los datos concretos de esa petición de memoria que es de tipo lectura a memoria global y lo sumamos a la estadística nueva que hemos creado.

```
void simt_core_cluster::icnt_inject_request_packet(class mem_fetch *mf) {
 // stats
 if (mf->get_is_write())
   m_stats->made_write_mfs++;
 else
   m_stats->made_read_mfs++;
 switch (mf->get_access_type()) {
   case CONST_ACC_R:
```

```
9       m_stats->gpgpu_n_mem_const++;
10      break;
11    case TEXTURE_ACC_R:
12      m_stats->gpgpu_n_mem_texture++;
13      break;
14    case GLOBAL_ACC_R:
15      m_gpu->bytesMemReaded += mf->get_data_size();
16      m_stats->gpgpu_n_mem_read_global++;
17      break;
```

El tiempo de ciclo nos lo da ya de forma nativa el simulador, así que ya tenemos todas las estadísticas que necesitamos y modificaciones dentro del simulador.

Con todos los datos disponibles ya podemos calcular los GFLOPs utilizando la frecuencia de la GPU, los ciclos que ha tardado el kernel en ejecutar y el número de instrucciones de punto flotante.

```
1  printf("gpu_gflops = %f\n", (m_config.core_freq * gpu_sim_insn_fp/↩
       ↪ gpu_sim_cycle)/1000000000);
```

También calculamos la intensidad aritmética dividiendo las instrucciones de punto flotante por los bytes leídos.

```
1printf("gpu_arithmetic_intensity = %2.14f\n", (double)gpu_sim_insn_fp/↩
     ↪ bytesMemReaded2);
```

3.2 Programa de dibujo del Roofline Model en Python

Teniendo ya la información dentro de los ficheros de estadísticas que nos da como salida el simulador, necesitamos obtener esta información y tratarla para poder dibujar el Roofline Model. Tanto la fase de obtener la información de los ficheros de estadísticas como la fase de dibujar el Roofline Model se realizan en el lenguaje Python.

La parte que analiza las estadísticas abre tantos ficheros como los ficheros que haya dentro del directorio que se le haya pasado como parámetro. Para cada fichero se leen la intensidad aritmética, los GFLOPs y los ciclos que tardan en ejecutarse en cada kernel. Dentro de cada fichero de estadísticas que pertenece a un benchmark en concreto puede haber varios kernels con estadísticas diferentes. En el caso de nuestro programa se hace una media ponderada con los ciclos de ejecución de cada kernel con las diferentes medidas (IA, GFLOPs), dando así el valor medio de IA y GFLOPs que se va a utilizar para imprimir el punto del benchmark dentro del Roofline Model.

3.3 Manual de uso

Para poder utilizar el programa de generador de Roofline Model se tienen que cumplir los siguientes requisitos

- Tener instalado Python Versión 3 o superior.
- Tener instalado la biblioteca de Python Matplotlib.

- Un fichero de entrada que indique el nombre de la GPU, el número de GFLOPs que tiene como máximo la gráfica y el número máximo de GB/s que tiene de bandwidth con el siguiente formato.

```
1  Nombre de la gráfica
2  GFLOPs
3  GB/s
```

- Tener todas las estadísticas de cada benchmark que se quiera pintar en la gráfica dentro de un mismo directorio. Además, estos ficheros deben contener las medidas añadidas en la sección 3.1.

La línea de comandos que se debe ejecutar es la siguiente: `python .\rooflineCreator.py <Fichero de salida.pdf> <Fichero de arquitectura> <Directorio donde estan las trazas>`

Un ejemplo sería `python .\rooflineCreator.py .\roofline.pdf .\nvidia_arquitectura .\estadisticas\rodinia\` que nos daría como salida esta figura 3.1.

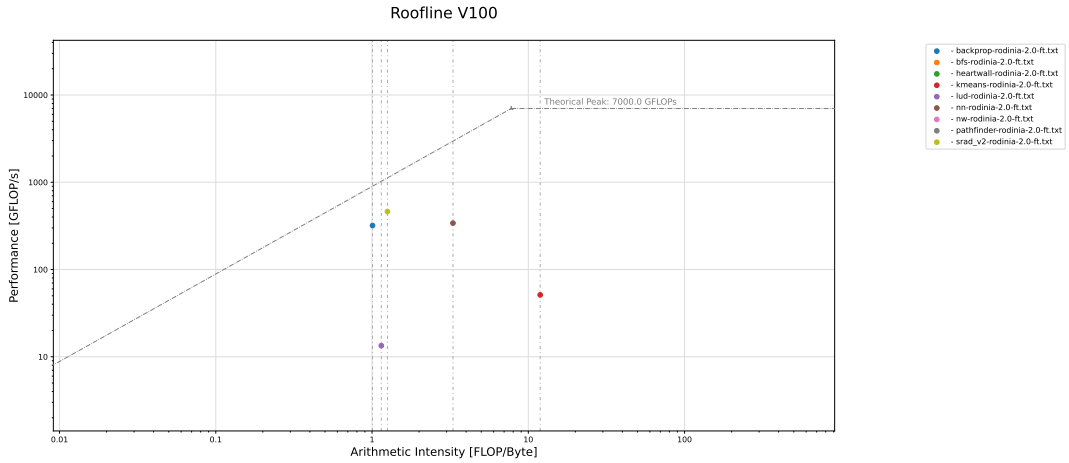

Figura 3.1: Roofline Model generado por nuestro programa

El programa, al ejecutarse, abre una ventana donde se puede ver el Roofline Model. Este tiene un panel de herramientas.

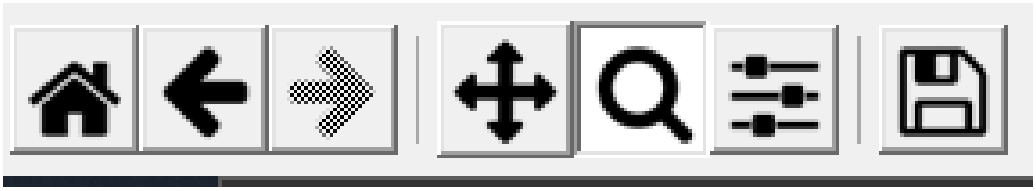

Figura 3.2: Herramientas dentro de la vista del Roofline Model

Tenemos diferentes herramientas que podemos ver en la figura 3.2, donde cada una de estas

contiene una pequeña auto explicación cuando pasas el ratón por encima. Una de las cosas que podemos hacer es hacer zoom al gráfico y luego guardarlo en formato pdf. Un ejemplo de un pdf guardado con zoom en la figura 3.3. En el caso de guardar el Roofline Model modificado nos pedirá el nombre y lugar donde queremos que guardemos el documento.

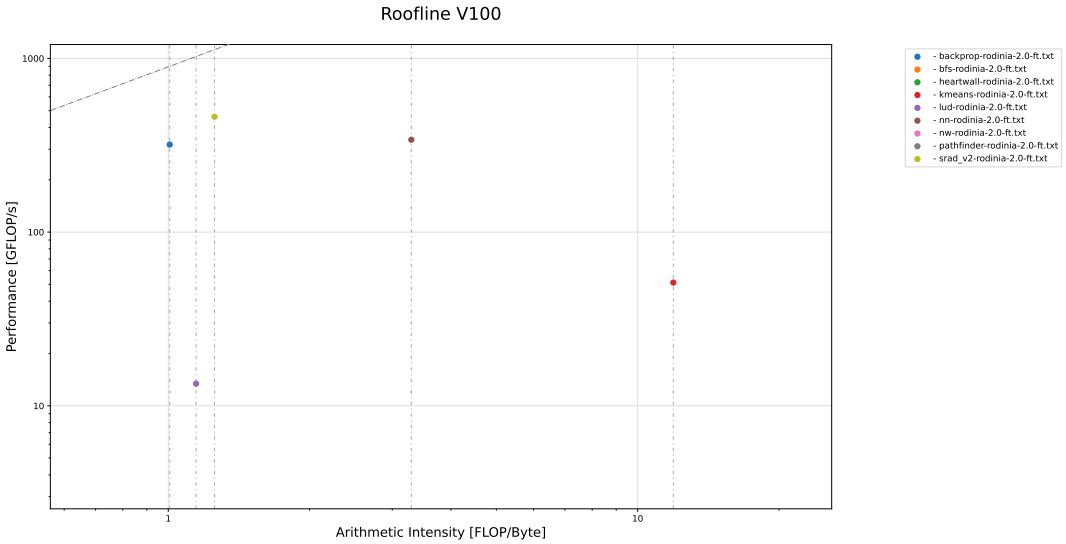

Figura 3.3: Roofline Model con aumento

Al finalizar el programa se nos guardará en el fichero de salida un documento con formato pdf en el mismo directorio donde se ha ejecutado el programa. La generación de este fichero es independiente de otros ficheros que se puedan guardar al usar la herramienta visual del programa.

4 Modelado de una GPU con Accel-sim

Una de las funcionalidades de Accel-Sim es el "tuneo" y generación de trazas de una GPU real. Para poder generar un fichero de configuración que contenga las características de nuestra GPU se necesitan 4 recursos:

- Un fichero que define información pública de la GPU que es difícil de obtener mediante micro-benchmarking. Este fichero lo debe rellenar el usuario.
- Se necesita ejecutar una serie de microbenchmarks para descubrir la configuración de cache, memoria y configuración de las unidades de ejecución.
- Se realiza una CUDA device query. Esta petición nos proporciona configuraciones del hardware como: el número de Streaming Multiprocesors, el ancho del bus de memoria, etc.
- Para los parámetros que no se pueden obtener directamente mediante la ejecución de microbenchmarks, el Tuner de Accel-Sim hace una búsqueda extensiva de las posibles combinaciones de parámetros en un conjunto de microbenchmarks de ancho de memoria.

El funcionamiento de la herramienta Accel-sim Tuner con el uso de los recursos listados anteriormente se puede ver en la figura 4.1

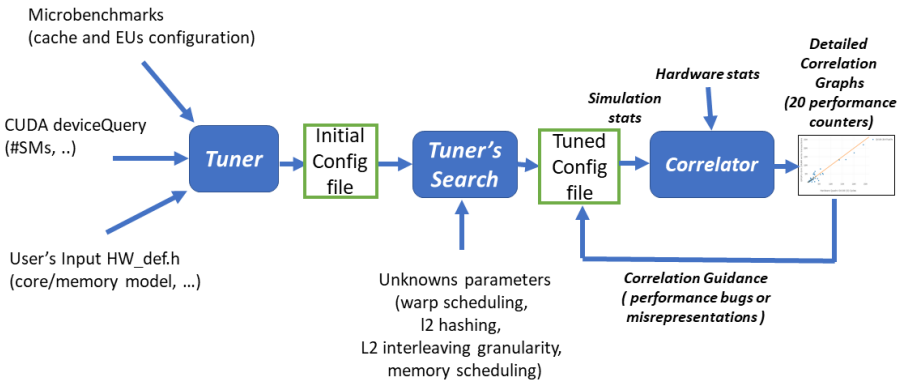

Figura 4.1: Diagrama de Accel-sim Tuner [1]

4.1 Utilización de Accel-sim Tuner en una GPU real

Con el objetivo de probar esta herramienta de Accel-sim se ha realizado una extracción de características de la GPU **MSI RTX 3060 TI VENTUS 2X OC V1 LHR**. Para esto se han seguido los pasos que se indican en el GitHub oficial de Accel-sim dentro de la sección Accel-Sim Tuner.

Al seguir estos pasos nos encontramos con un primer problema, los microbenchmark que nos proporcionan dan errores de compilación. Para solucionar este primer problema lo primero que se ha hecho es descargar un fichero CUDA que es necesario y se puede encontrar en helper_string.h. Después de descargar el fichero se tendrá que incluir **helper_string.h** al fichero **deviceQuery.cpp**.

Una vez se han ejecutado los microbenchmarks se pasan las estadísticas que se han generado y se ejecuta la herramienta Tuner. Al ejecutar esta herramienta nos genera un fichero de configuración llamado **gpgpusim.config** donde se encontrarán todos los parámetros que acepta el simulador GPGPU-sim para simular una versión aproximada de nuestra GPU. En nuestro caso de ejemplo el fichero de configuración se generó con los siguientes fallos: no se asemejan a la arquitectura real y además no permitían que fuera simulada.

Uno de estos fallos es un error en el número de conjuntos de la cache L2. El programa nos indica que nuestra GPU tiene 128 conjuntos lo cual no es posible ya que la versión superior, la GPU Nvidia GTX 3070 tiene 64 conjuntos. Para solucionar este problema indicamos el número de conjuntos de la GPU Nvidia GTX 3070 ya que ambas son de la misma arquitectura, Ampere.

Otro de los problemas del fichero de configuración que nos proporciona se encuentra en el número de canales de memoria principal. Este número es únicamente de 2 controladores de memoria. Este número aparte de ser cierto o no, no es compatible con el estado actual del simulador. El simulador solo acepta como mínimo 32 canales (teniendo en cuenta que están subdivididos). Esta casuística puede ser por dos motivos, que el simulador esté preparado para GPUs de alta gama y no contemple estos casos o que el Tuner haya errado en el número de canales de la memoria principal, siendo esto último lo más probable. La raíz del problema de compatibilidad con un número de canales menor a 32 es la falta de casos dentro de la función de hashing. Para solucionarlo por tanto se tienen que añadir nuevos casos cuando el conjunto de bancos es menor que 32. Para añadir nuevos casos dentro de la función de hashing tenemos que modificar el fichero **hashing.cc**. Uno de los casos añadidos en este fichero es:

```
if (bank_set_num == 2) {
   std::bitset<64> a(higher_bits);
   std::bitset<1> b(index);
   std::bitset<1> new_index(index);

   new_index[0] =
       a[0] ^ b[0];

   return new_index.to_ulong();

 }
```

La herramienta de Tuner también genera un fichero de trazas (**trace.conf**) para poder analizar trazas que se generen desde la GPU real. El fichero de trazas que fue generado por la GPU de ejemplo era defectuoso y no funcionaba. Para solucionar este problema se ha copiado el fichero **trace.conf** de la GPU Nvidia GTX 3070 ya que son de la misma arquitectura.

4.2 Generar trazas con una GPU real

Una de las funciones de Accel-Sim es poder utilizar trazas SASS para realizar las simulaciones. Se va a utilizar la misma GPU que en la sección anterior 4.1.

Para conseguir las trazas de una cierta GPU se necesita tener una versión de CUDA que permita un SM compatible con la arquitectura específica de Nvidia. En el caso de la GPU usada en este ejemplo, la arquitectura Ampere, es compatible con SM86 siendo necesaria la versión 11.1 de CUDA o superior, vemos estas versiones en la figura 4.2.

Fermi†,	Kepler†	Maxwell‡	Pascal	Volta	Turing	Ampere	Ada (Lovelace)	Hopper
sm_20	sm_30	sm_50	sm_60	sm_70	sm_75	sm_80	sm_89	sm_90
	sm_35	sm_52	sm_61	sm_72 (Xavier)		sm_86		sm_90a (Thor)
	sm_37	sm_53	sm_62			sm_87 (Orin)		

† Fermi and Kepler are deprecated from CUDA 9 and 11 onwards
‡ Maxwell is deprecated from CUDA 11.6 onwards

Figura 4.2: SM según arquitectura de NVIDIA

Accel-sim te permite generar trazas de kernels específicos utilizando tracer_nvbit o realizarlo mediante un script de python que te proporcionan donde puedes generar trazas de suites de benchmarks incluidas en Accel-sim. En nuestro caso en concreto hemos elegido esta opción generando trazas de la suite de benchmarks Rodinia 2.0, dándonos como resultado un directorio con todas las trazas SASS.

Las trazas generadas de Rodinia 2.0 son las trazas utilizadas en los siguientes apartados de este trabajo.

5 Planificación de Warps en GPGPU-sim

Como se ha explicado en el apartado 2.3, uno de los componentes esenciales de la arquitectura de una GPU es el planificador de Warps. El planificador de Warps es clave para que una GPU funcione como está planeada. La visión de un programador dentro de una GPU es de "infinitos" hilos ejecutándose en paralelo y totalmente independientes. Para poder conseguir que esto se haga realidad a bajo nivel, es necesario la estructura de un planificador de warps que seleccione el grupo de hilos que va a entrar en la pipeline de ejecución y ocultar la latencia de las operaciones haciéndolas parecer ejecutarse en paralelo.

GPGPU-sim implementa varios planificadores de warps. El planificador que Accel-sim Tuner derivó de la GPU real en la que hicimos las pruebas es un planificador GTO. Aparte de este planificador existen otros ya explicados en el apartado2.3.1, todos estos planificadores tienen una lógica detrás que intenta aprovechar diferentes propiedades de una arquitectura de una GPU. Como podría ser la localidad espacial de los datos o confiar en que un warp diferente al que se está ejecutando consiga resolver riesgos RAW, dejando pasar otro warp que va a escribir el dato que necesita. En contraposición a todo esto, hemos decidido implementar en GPGPU-sim un planificador de warps que no intente aprovecharse de ninguna característica de la GPU y así tener una medida básica de comparación. Este planificador de warps es el random scheduler, el funcionamiento del mismo es muy sencillo, cada vez que la GPU le pida al planificador de Warps un hilo para la ejecución este será elegido aleatoriamente. De esta forma es imposible aprovechar ninguna característica del programa o de la arquitectura.

5.1 Implementación de Random Scheduler en GPGPUsim

Para poder implementar este planificador vamos a aprovechar el código de los planificadores que ya estan implementados y añadirlo a la lista de opciones que se pueden elegir dentro del fichero de configuración de una GPU en GPGPU-sim. Para esto lo primero que tenemos que hacer es declarar un nuevo planificador dentro de la parte del código que se encarga de parsear los planificadores. Dentro del fichero **shader.h** añadimos `CONCRETE_SCHEDULER_RANDOM` al enumerado que contiene todos los planificadores declarados (aunque no todos estan implementados).

```
// Each of these corresponds to a string value in the gpgpsim.config file
// For example - to specify the LRR scheudler the config must contain lrr
enum concrete_scheduler {
  CONCRETE_SCHEDULER_LRR = 0,
  CONCRETE_SCHEDULER_GTO,
  CONCRETE_SCHEDULER_TWO_LEVEL_ACTIVE,
  CONCRETE_SCHEDULER_RRR,
  CONCRETE_SCHEDULER_WARP_LIMITING,
  CONCRETE_SCHEDULER_OLDEST_FIRST,
```

```
  CONCRETE_SCHEDULER_RANDOM,
  NUM_CONCRETE_SCHEDULERS
};
```

También es necesario crear una clase que defina este nuevo planificador que hereda de una clase genérica de planificadores. Esta clase también se define en **shader.h** ya que en este fichero se calcula la mayoría de la lógica que ocurre dentro de un Streaming Multiprocesor.

```
class random_scheduler : public scheduler_unit {
 public:
  random_scheduler(shader_core_stats *stats, shader_core_ctx *shader,
              Scoreboard *scoreboard, simt_stack **simt,
              std::vector<shd_warp_t *> *warp, register_set *sp_out,
              register_set *dp_out, register_set *sfu_out,
              register_set *int_out, register_set *tensor_core_out,
              std::vector<register_set *> &spec_cores_out,
              register_set *mem_out, int id)
      : scheduler_unit(stats, shader, scoreboard, simt, warp, sp_out, dp_out,
                       sfu_out, int_out, tensor_core_out, spec_cores_out,
                       mem_out, id) {}
  virtual ~random_scheduler() {}
  virtual void order_warps();
  virtual void done_adding_supervised_warps() {
    m_last_supervised_issued = m_supervised_warps.end();
  }
};
```

Necesitamos declarar en **shader.h** la función que se va a utilizar para decidir el orden de cada planificador, incluyendo el nuestro.

```
template <typename T>
  void order_random(
      typename std::vector<T> &result_list,
      const typename std::vector<T> &input_list,
      const typename std::vector<T>::const_iterator &last_issued_from_input,
      unsigned num_warps_to_add);
```

Una vez declarados los diferentes componentes del Random Scheduler, lo primero que tenemos que modificar es la zona donde se lee del fichero de configuraciones. Esta zona del código tiene como objetivo, saber cual va a ser el planificador de warps que va a usar la simulación. Esta funcionalidad se encuentra dentro del fichero **shader.cc**, dentro de la función que se encarga de crear el planificador específico para esa simulación `create_schedulers()`. Nuestro Random Scheduler va a ser definido dentro de las opciones de configuración como "random", por lo tanto tenemos que añadir el caso de esta cadena dentro de la función `create_schedulers()`.

```
  // scedulers
  // must currently occur after all inputs have been initialized.
  std::string sched_config = m_config->gpgpu_scheduler_string;
  const concrete_scheduler scheduler =
      sched_config.find("lrr") != std::string::npos
          ? CONCRETE_SCHEDULER_LRR
```

```
        : sched_config.find("two_level_active") != std::string::npos
             ? CONCRETE_SCHEDULER_TWO_LEVEL_ACTIVE
             : sched_config.find("gto") != std::string::npos
                  ? CONCRETE_SCHEDULER_GTO
                  : sched_config.find("rrr") != std::string::npos
                       ? CONCRETE_SCHEDULER_RRR
                  : sched_config.find("old") != std::string::npos
                       ? CONCRETE_SCHEDULER_OLDEST_FIRST
                       : sched_config.find("warp_limiting") !=
                                std::string::npos
                            ? CONCRETE_SCHEDULER_WARP_LIMITING
                            : sched_config.find("random") != std::string::↩
                               ↪ npos
                                 ? CONCRETE_SCHEDULER_RANDOM
                            : NUM_CONCRETE_SCHEDULERS;
 assert(scheduler != NUM_CONCRETE_SCHEDULERS);
```

Dentro del simulador cada "core" (Streaming Multiprocesor) puede tener varios planificadores y el número de planificadores es un parámetro que se puede pasar por la configuración. Por lo tanto, dentro de la función de creación de planificadores se va rellenando una lista con los planificadores que se van a utilizar. Tenemos que añadir el caso donde haya sido seleccionado Random Scheduler y añadirlo a la lista de schedulers.

```
case CONCRETE_SCHEDULER_RANDOM:
      schedulers.push_back(new random_scheduler(
         m_stats, this, m_scoreboard, m_simt_stack, &m_warp,
         &m_pipeline_reg[ID_OC_SP], &m_pipeline_reg[ID_OC_DP],
         &m_pipeline_reg[ID_OC_SFU], &m_pipeline_reg[ID_OC_INT],
         &m_pipeline_reg[ID_OC_TENSOR_CORE], m_specilized_dispatch_reg,
         &m_pipeline_reg[ID_OC_MEM], i));
      break;
```

Dentro del simulador está desacoplado la función de ordenar los warps que se van a seleccionar y la función que se utiliza para ordenarlos, por tanto tenemos que conectarlos en la función `order_warps()`

```
void random_scheduler::order_warps(){
 order_random(m_next_cycle_prioritized_warps, m_supervised_warps,
          m_last_supervised_issued, m_supervised_warps.size());
}
```

Finalmente, dentro de `order_random()` se encuentra la funcionalidad del planificador. En esta función lo que realizamos es generar un número aleatorio con la función `rand()` de c++ y hacerle el módulo del tamaño de la lista de warps que pueden ser electos para ser planificados. Este va a ser el warp que va a ser seleccionado. Después de esto tenemos que rellenar la lista que vamos a devolver con los warps restantes.

```
template <class T>
void scheduler_unit::order_random(
   std::vector<T> &result_list, const typename std::vector<T> &input_list,
   const typename std::vector<T>::const_iterator &last_issued_from_input,
```

```
    unsigned num_warps_to_add) {
  assert(num_warps_to_add <= input_list.size());
  result_list.clear();
  typename std::vector<T>::const_iterator iter =
      (last_issued_from_input == input_list.end()) ? input_list.begin()
                                                   : last_issued_from_input + 1;

  unsigned int randN = rand() % input_list.size();
  result_list.push_back(&warp(randN));

  for (unsigned count = 0; count < num_warps_to_add; ++iter, ++count) {
    if (iter == input_list.end()) {
      iter = input_list.begin() +1;
    }
    result_list.push_back(*iter);
  }
}
```

Estos son todos los pasos que han sido necesarios para implementar un nuevo planificador de warps dentro de GPGPU-sim.

5.2 Comparación de diferentes planificadores de Warps con Random Scheduler

Como se ha mencionado en la sección anterior, uno de los objetivos de implementar Random Scheduler además de para explicar como se puede añadir un planificador en gpgpu-sim, sirve para poder compararlo con otros planificadores [17].

Para poder realizar esta comparación se han realizado diferentes experimentos, estos experimentos han sido realizados con las trazas del benchmark Rodinia 2.0 [2] y el fichero de configuración de GPU obtenidos en el apartado 4.1.

Los planificadores utilizados para la realización de experimentos han sido:

- **gto** : Gready the oldest
- **lrr** : Loose Round Robin
- **rrr** : Restricted Round Robin
- **old** : Oldest First
- **random** : Random

Se han realizado simulaciones con las trazas de los siguientes benchmarks:

- **backprop**
- **hearwall**
- **bfs**

- **pathfinder**
- **streamcluster**
- **nw**

Se han utilizado dos scripts de Python para manejar los datos de las simulaciones, tanto estos scripts como todos los datos utilizados se encuentran en este repositorio de GitHub [17].

Dentro de este repositorio podemos ver dos ficheros Python. Uno de ellos es **ejecuciones.py**, este fichero es utilizado para realizar todas las ejecuciones de las trazas del benchmark rodinia 2.0. También tenemos el fichero **parseador.py**, este programa se encarga de buscar dentro de todas las trazas que genera **ejecuciones.py**. Además se encarga de hacer una media ponderada de diferentes estadísticas de cada benchmark. Esta media esta ponderada por el porcentaje de tiempo en el que se ejecuta cada kernel dentro del conjunto del benchmark. El programa da como salida un fichero con el nombre del benchmark y la media de estadísticas formateadas para poder copiar y pegarlas en una hoja de cálculo para poder trabajar con ellas.

```
resultados de : streamcluster-rodinia-2.0-ftGTO.txt
0,61531727184
59224847
40529430
120184587
resultados de : bfs-rodinia-2.0-ftGTO.txt
0,456431090207
301580
4639555
9039458
resultados de : nw-rodinia-2.0-ftGTO.txt
0,8661
2040
6932764
1805220
resultados de : heartwall-rodinia-2.0-ftGTO.txt
0,6901
108682
80680
568769
resultados de : pathfinder-rodinia-2.0-ftGTO.txt
0,823423084482
16199
153432
664475
resultados de : backprop-rodinia-2.0-ftGTO.txt
0,34998042251
1113486
131132
695315
```

5.2.1 Resultados

El objetivo de las simulaciones realizadas son dos. Uno de ellos es asegurarse de que la modificación del simulador funciona. Otro es ver experimentalmente la diferencia entre diferentes planificadores de Warps dentro de diferentes cargas.

Como se ha comentado en el inicio de este apartado uno de los principales objetivos de Random Scheduler es comparar este planificador con otros planificadores que si busquen obtener una mejora en el tiempo de ejecución.

En la tabla 5.1 se encuentran los speedup respecto al planificador GTO calculados a partir de la media aritmética del número de ciclos de los diferentes benchmarks simulados con el mismo planificador.
Comparando el tiempo de ejecución de los diferentes planificadores podemos ver que el mejor tiempo de ejecución es del planificador Oldest First, aunque esta diferencia es del 0.1%. Además de esto si tenemos en cuenta que la cantidad de benchmark y el tamaño de los mismos no es muy grande debido a limitaciones de recursos de computo. No podemos derivar que Oldest First sea el mejor planificador dentro del conjunto simulado. Un resultado interesante dentro de estos resultados es la ineficacia de Restricted Round Robin, si comparamos este planificador con el planificador que acabamos de implementar, Random. Podemos ver que RRR es casi un 4% mas lento que Random. Esto nos indica que si elegimos al azar el warp que va a entrar en ejecución, vamos a conseguir mejor rendimiento que si lo elegimos mediante RRR. El planificador RRR busca sacar partido de la localidad espacial del mismo hilo hasta bloquearse. Sin embargo, RRR no logra aprovecharse de esta localidad.

Planificadores	GTO	LRR	RRR	OLD	RANDOM
Speed Up vs GTO	1	0.987002	0.938676	1.001286	0.976722

Tabla 5.1: Speed Up de los diferentes planificadores vs GTO

En la visión de cada benchmark ejecutado con un planificador diferente que podemos ver en la figura 5.1 GTO y Oldest First están igualados en los diferentes benchmarks teniendo ligeras mejoras y desmejoras. Además el planificador RRR es el peor en todos los benchmark estando siempre por debajo del planificador random confirmando que es un planificador subóptimo.

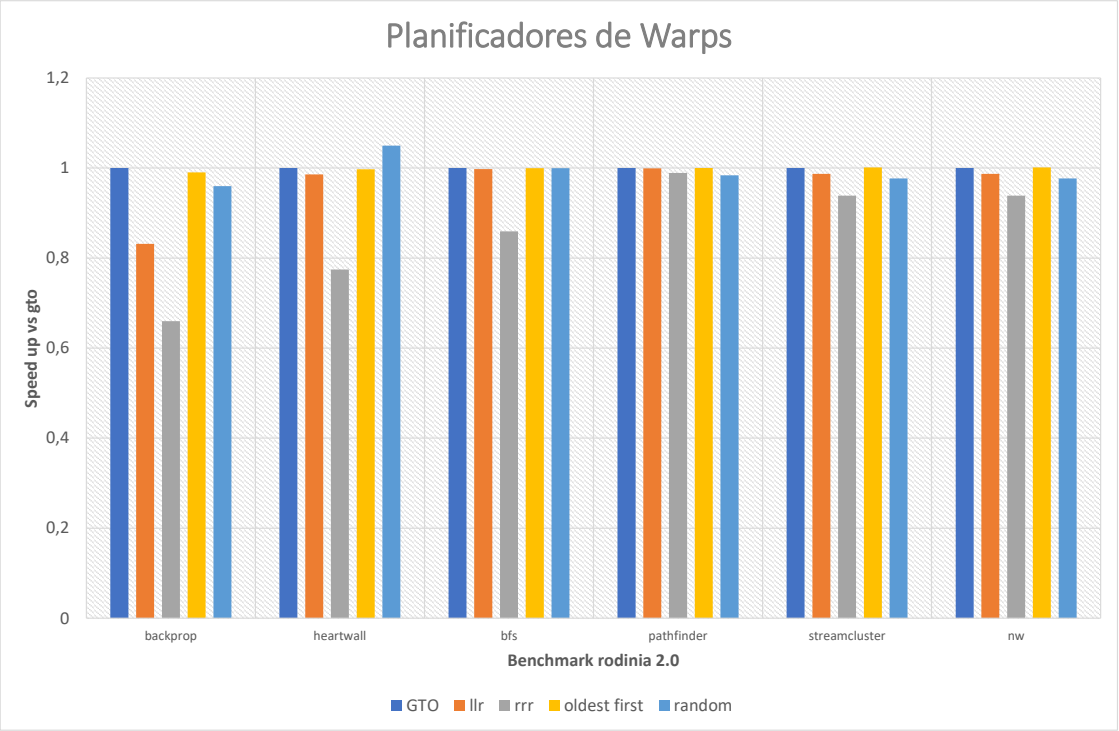

Figura 5.1: Diferencias de rendimiento entre planificadores de Warps en cada becnhmark

Es interesante ver la degradación del planificador RRR respecto a LRR ya que ambos se basan en una lista circular pero tienen una diferencia mayor que GTO y Oldest First que ambos se basan en una lista del mas viejo primero. Fijándonos en los ciclos de parada, el planificador RRR tiene mayor número de paradas de warps que LRR, esto se acentúa en los *warp Stall*. Cuando miramos la diferencia entre Oldest First y GTO, figura 5.3, vemos que no hay mucha diferencia entre sus ciclos de parada. Esta mayor incidencia de ciclos de parada explica el decremento de eficacia del planificador RRR. Podemos verlo en la figura 5.2.

Podemos entonces ver como la técnica de aguantar con el mismo warp hasta que este se bloquee solo funciona en el caso de que estemos eligiendo los warp por su tiempo sin ejecutarse y no de forma de lista. El objetivo al implementar un planificador en forma de lista rotativa es que cada warp vaya realizando avances de forma equitativa, al obligar a mantener los warps en ejecución mientras estos no se ven bloqueados, hacemos que la idea de reparto equitativo deje de funcionar y vemos los efectos de esto en una penalización de rendimiento. En contraste GTO está basado en una lista de acceso al warp más "viejo", por lo tanto, está dando la oportunidad a esos warp que están esperando durante más tiempo. Estos warps probablemente hayan solucionado los riesgos por los que dejaron de ejecuarse.

Los tamaños de las simulaciones con LRU y GTO han sido los siguientes para cada benchmark, tabla 5.2. Por lo tanto solo podemos tomar los resultados como un indicio.

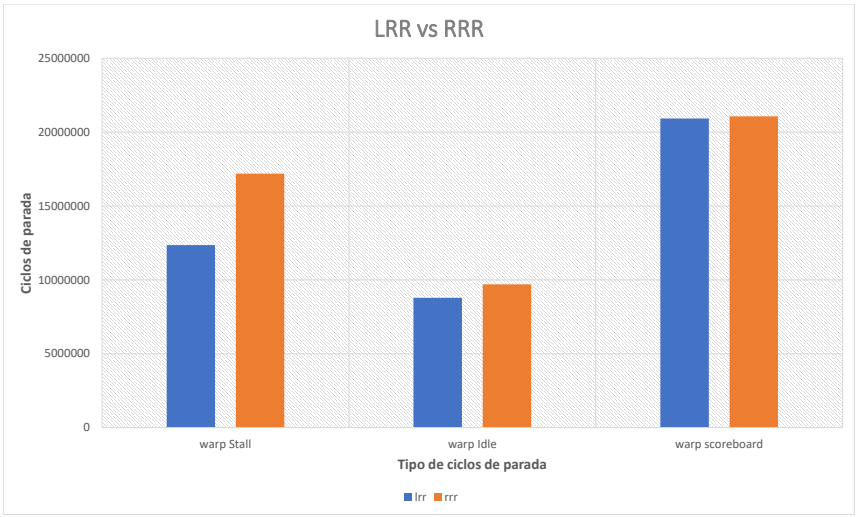

Figura 5.2: Ciclos medios de parada del planificador LRR y RRR

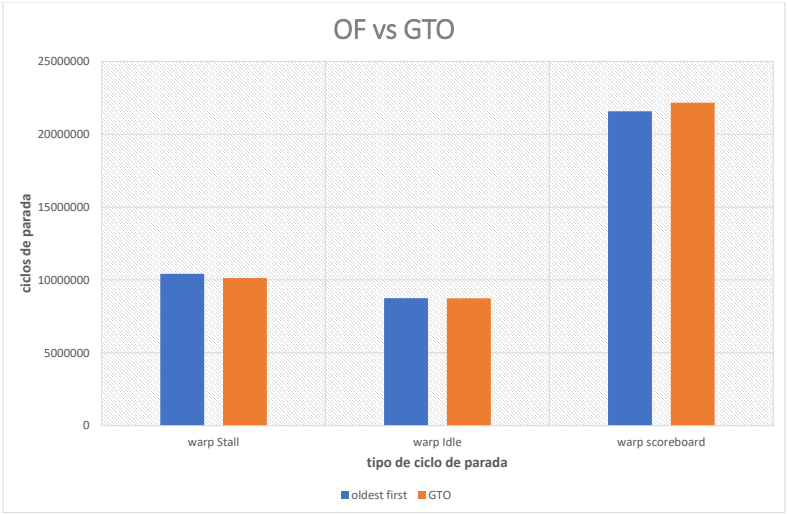

Figura 5.3: Ciclos medios de parada del planificador OF y GTO

Benchmark	Número de instrucciones	Número de ciclos total
backprop	8'827'230	22'828
bfs	1'288'820	138'214
heartwall	6'559'668	12'128
nw	473'136	136'538
pathfinder	870'814	35'617
streamcluster	20'147'595	1'094'080

Tabla 5.2: Número de instrucciones y tiempo de ejecución de los becnhmark con el planificador GTO y la política de caches LRU

6 Política de Remplazo de Cachés en GPGPU-sim

6.1 Implementación de política de remplazo de caché NRU

Para implementar NRU dentro de GPGPU-sim al igual que ocurre con los planificadores de warps hay que seguir las estructuras que hay ya definidas en el simulador. GPGPU-sim organiza toda la lógica de definición de la estructura de cachés, como se acceden a ellas y de que manera se manejan los datos dentro de las cachés. Todo esto se encuentra en dos ficheros: **gpu-cache.h** y **gpu-cache.cc**

En el fichero **gpu-cache.h** Podemos encontrar la definición de una estructura abstracta que indica los campos que debe tener una bloque de caché. Esta estructura se llama `cache_block_t` dentro de esta clase se encuentran datos como los bits que indican si una línea de caché es válida, está reservada o está modificada. Como nuestro interés es implementar la política de remplazo NRU, definimos 3 funciones para modificar el bit que nos indica si el bloque de caché ha sido accedido recientemente y otra función para saber el valor de este bit.

```
virtual void set_recently_used() = 0;
virtual void unset_recently_used() = 0;
virtual bool is_recently_used() = 0;
```

La instanciación de una estructura que implementa métodos e instancia atributos de las líneas de caché es la estructura `line_cache_block`. Dentro de esta estructura lo primero que hacemos es añadir el bit de uso reciente `m_recently_used` dentro de los atributos de la línea.

```
private:
 unsigned long long m_alloc_time;
 unsigned long long m_last_access_time;
 unsigned long long m_fill_time;
 cache_block_state m_status;
 bool m_ignore_on_fill_status;
 bool m_set_modified_on_fill;
 bool m_set_readable_on_fill;
 bool m_set_byte_mask_on_fill;
 bool m_readable;
 bool m_recently_used;
 mem_access_byte_mask_t m_dirty_byte_mask;
```

Una vez tenemos definido este atributo necesitamos inicializarlo junto con todos los demás en la función que se encarga de inicializarlos. El atributo se inicializa a 0 ya que ninguna línea ha sido usada al inicializarse la caché.

```
line_cache_block() {
   m_alloc_time = 0;
   m_fill_time = 0;
   m_last_access_time = 0;
   m_recently_used = 0;
   m_status = INVALID;
   m_ignore_on_fill_status = false;
   m_set_modified_on_fill = false;
   m_set_readable_on_fill = false;
   m_readable = true;
 }
```

Implementamos las funciones que nos permiten acceder y modificar.`m_recently_used`

```
virtual void set_recently_used(){
   m_recently_used = true;
 }
 virtual void unset_recently_used(){
   m_recently_used = false;
 }
 virtual bool is_recently_used(){
   return m_recently_used;
 }
```

Teniendo el objetivo de añadir esta nueva política de remplazo de caché a las configuraciones del simulador, tenemos que añadir el caso en el que se especifique NRU con el carácter 'N'. Esto se hace en el fichero **gpu-cache.h** en la clase `cache_config` dentro de la función `init()`.

```
switch (rp) {
    case 'L':
      m_replacement_policy = LRU;
      break;
    case 'F':
      m_replacement_policy = FIFO;
      break;
    case 'N':
      m_replacement_policy = NRU;
      break;
    default:
      exit_parse_error();
  }
```

En el fichero **gpu-cache.cc** encontramos las implementaciones de funciones relacionadas con peticiones a caché que va llamar el SM modelado en el fichero **shader.cc** dentro de este fichero nos interesa modificar la parte del código donde se comprueba que una petición es un: acierto, fallo, o versiones de estos. La función que se encarga de discernir estos escenarios se llama `probe()`. Necesitamos modificar la parte del código donde se identifica la línea que va a ser expulsada de caché. Una vez sabemos esto, elegimos la primera línea con el bit de "Usado recientemente" a 0.

```
// valid line : keep track of most appropriate replacement candidate
        if (m_config.m_replacement_policy == LRU) {
          if (line->get_last_access_time() < valid_timestamp) {
            valid_timestamp = line->get_last_access_time();
            valid_line = index;
          }
        } else if (m_config.m_replacement_policy == FIFO) {
          if (line->get_alloc_time() < valid_timestamp) {
            valid_timestamp = line->get_alloc_time();
            valid_line = index;
          } else if (m_config.m_replacement_policy == NRU){
            if(!line->is_recently_used()) {
              valid_line = index;
            }
          }
```

En la función `probe()` se recorren todas las líneas de la caché asociadas a una petición. Tenemos que tener en cuenta el caso donde ninguna de las líneas tiene el bit de recientemente usado a 0, es decir, todas las líneas se han usado recientemente. En este caso la posible línea a expulsar estará vacía y tendremos que seleccionar una línea cualquiera (en este caso nos apoyamos de la función `rand()`).

```
if (valid_line == (unsigned)-1) {
   valid_line = set_index * m_config.m_assoc + (rand()% m_config.m_assoc);
 }
```

Una vez seleccionamos una línea de caché para ser expulsada, tenemos que restablecer a 0 el bit de recientemente en todas las vías.

```
for(unsigned way = 0; way < m_config.m_assoc; way++){
   unsigned index = set_index * m_config.m_assoc + way;
   cache_block_t *line = m_lines[index];
   line->unset_recently_used();
 }
```

Habiendo terminado la lógica cuando la petición va a fallar y expulsar una línea de caché, lo siguiente es actualizar el bit de recientemente usado cuando se da un acierto de caché. Para esto vamos a modificar la función `access()` que gestiona los metadatos de la caché cuando se accede a ella. En el caso de que haya sido un acierto de caché vamos a actualizar el bit de usado recientemente.

```
case HIT:
     m_lines[idx]->set_last_access_time(time, mf->get_access_sector_mask());
     m_lines[idx]->set_recently_used();
     break;
```

Las únicas políticas de remplazo que admite el simulador ahora mismo con esta modificación son tres: LRU, que ya estaba implementada, FIFO y la que acabamos de implementar, NRU. En el fragmento de código donde actualizamos el bit de usado recientemente podríamos discernir entre la política de LRU y NRU. Sin embargo, no vemos esta necesidad al ser solo dos llamadas que actualizan un valor en memoria. En el caso donde se implementen más políticas de remplazo,

se puede discernir entre los diferentes casos.

6.2 Simulaciones y resultados

Para la realización de simulaciones con la política de remplazo de caché de NRU se han utilizado los mismos recursos que en la sección 5.2, siendo por lo tanto el mismo número de simulaciones pero con la política de remplazo NRU. Se ha realizado de esta forma con el objetivo de poder comparar las dos políticas de remplazo.

Como podemos ver en la figura 6.1. La media aritmética de las ejecuciones entre las dos políticas de remplazo es muy similar diferenciándose menos de un 0.1%. Esto puede ser debido a que no se han realizado la suficiente cantidad de simulaciones para no sesgar los datos con los comportamientos específicos de cada benchmark.

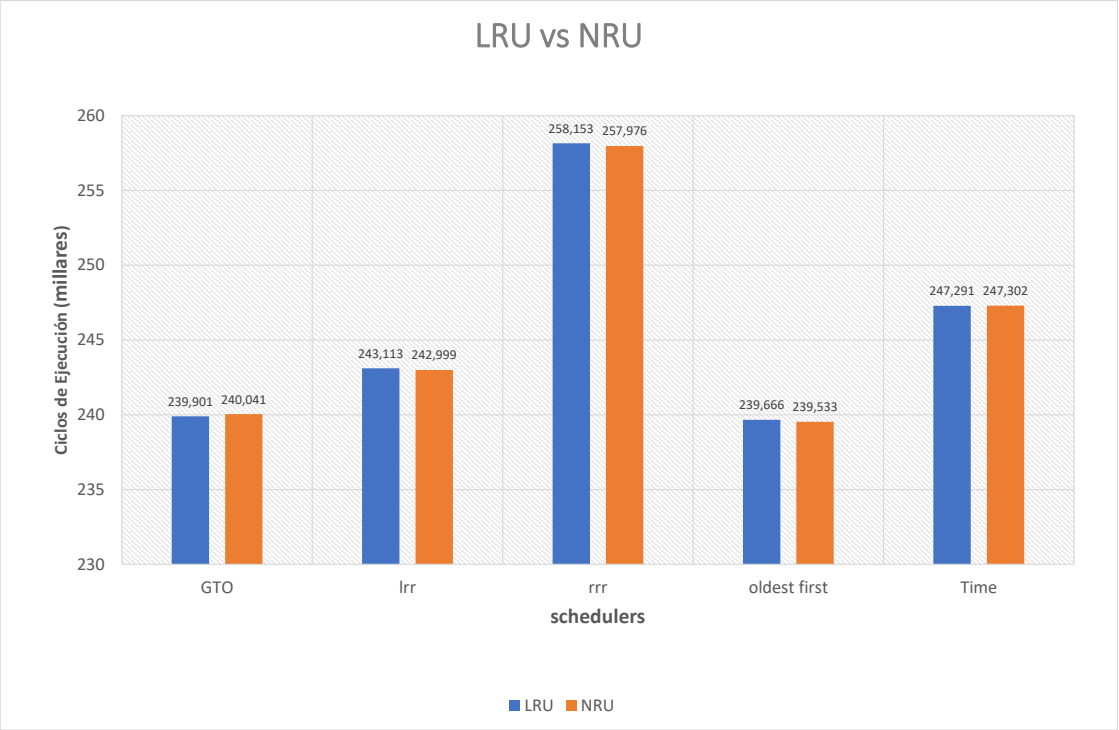

Figura 6.1: Ciclos medios de ejecución de LRU y NRU

Si nos fijamos en la diferencia de tiempo de ejecución de cada benchmark con LRU y NRU nos damos cuenta que de los benchmark que hemos utilizado solo hay uno donde esta diferencia es positiva, figura 6.2. El benchmark backprop obtiene un speedup del 2.2%. Este benchmark es relevante ya que se basa en el back propagation que se realiza en las redes neuronales (técnica de inteligencia artificial muy utilizada actualmente). Dentro de las mejoras de este becnhmark respecto a NRU el planificador que mejor funciona es LRR con una mejora respecto al planificador que peor funciona, GTO, de un 4.4%. La cantidad de ciclos que tardan en

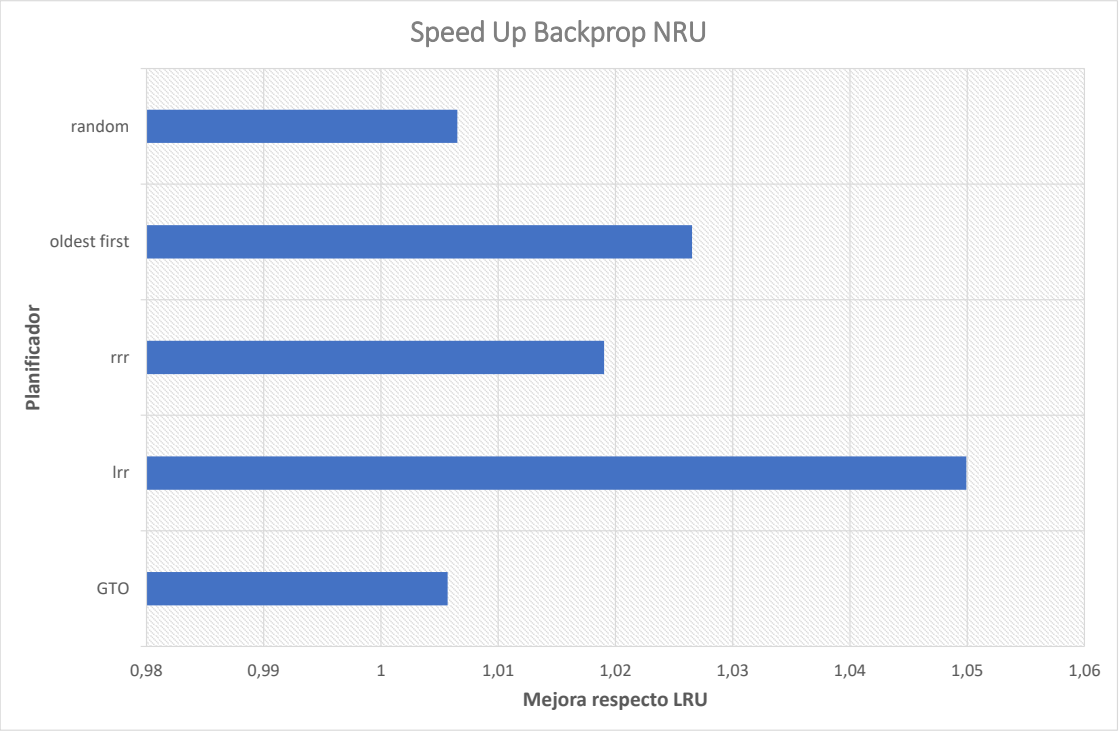

Figura 6.2: Mejora de Speed Up Backprop NRU respecto a LRU

ejecutarse estos benchmarks no es superior a 30 mil ciclos, por lo tanto, estos datos pueden no ser significativos. Sin embargo, sí que pueden marcar la posible existencia de una correlación entre el planificador de hilos utilizado y la política de remplazo de caché.

Dentro de los experimentos realizados esta correlación parece incrementarse cuando la utilización de NRU respecto a LRU es mejor utilizando cualquiera de los schedulers. En el resto de casos la correlación es de alrededor de un 1%.

7 Conclusiones y vías futuras

En este trabajo se han abordado diferentes acciones dentro del marco del simulador Accel-sim. Dentro de estas acciones el primer objetivo realizado fue un programa capaz de generar Roofline Models a partir de ficheros de estadísticas de GPGPU-sim. Para el funcionamiento de esta herramienta también ha sido necesario modificar el simulador para imprimir más estadísticas que la versión base no muestra. Para ello fue necesario modificar el código fuente de GPGPU-sim. Para la visualización de un Roofline Model es también necesario indicar al programa características básicas de la GPU para que pueda dibujar la línea del "roof". Una posible mejora de este programa sería la utilización de benchmarks que calcularan de forma automática estas características gracias a la utilización del simulador. Aunque esto supondría un problema ya que para poder generar una simulación dentro de GPGPU-sim necesitas de un fichero de configuración de GPU.

Es posible modelar una GPU real gracias a la herramienta Tuner de Accel-sim pero como se ha probado en este trabajo, esta herramienta no siempre da los resultados correctos. Además, sigue siendo necesario indicar ciertas especificaciones de la GPU para poder modelarla. También se ha realizado la generación de trazas con Accel-sim. Debido a limitaciones técnicas estas trazas han sido de tamaño reducido. Al utilizar estas trazas en el resto de simulaciones este problema se repite durante el desarrollo del TFG. A sí mismo, hay otra herramienta dentro de Accel-sim que no ha sido posible probar dentro de este TFG. Esta herramienta consiste en un correlacionador que compara las simulaciones realizadas con una configuración de GPU con la ejecución de las mismas trazas en una GPU real. Se intentó utilizar la herramienta correlator de Accel-sim para comparar las simulaciones con el fichero de configuración que modela la RTX 3060Ti utilizada. Sin embargo, no ha sido posible utilizar correctamente esta herramienta. Una de las posibles razones de esto es debido al cambio de los contadores entre diferentes arquitecturas de NVIDIA. Solucionar los problemas de esta herramienta no ha sido trivial y se ha marcado como fuera del alcance de este proyecto.

Otro objetivo de este trabajo ha sido la implementación de un nuevo planificador de warp dentro de GPGPU-sim. La implementación de este planificador Random ayudó a discernir entre planificadores de warp que están aprovechando ventajas de la arquitectura de otros que simplemente lastran el rendimiento y son peores que no implementar ningún planificador. En adición a esto se implementó también una política de caché dentro de GPGPU-sim proporcionando la posibilidad de realizar simulaciones con diferentes políticas de remplazo de cachés y diferentes planificadores para buscar algún tipo de correlación entre ellas. Los resultados no indicaron muchas diferencias entre los planificadores y las políticas de caché, pero sí que hubo un resultado destacable donde el planificador LRR obtuvo un 4.4% de mejora sobre el planificador GTO en comparación con ejecuciones de estos mismos planificadores dentro de la política de remplazo de caché LRU. Desgraciadamente, estos resultados no son determinantes pues el tamaño de los

experimentos ha sido reducido debido a limitaciones de hardware.

Una de las mayores limitaciones del trabajo realizado en este TFG ha sido el espacio de las trazas. Accel-sim incluye un programa para poder descargar trazas que los creadores de Accel-sim utilizaron para corroborar su estudio. Los tamaños de las trazas son los siguientes:

- Rodinia 3.1: 302 GB
- CUDA SDK: 18 GB
- Parboil: 250 GB
- Polybench: 743 GB
- CUTLASS 2.5 TB
- Deepbench: 2.6 TB

El menor tamaño de estas trazas es de 18 GB, el siguiente es Parboil, con 250 GB. En la realización de este trabajo se ha utilizado un ordenador personal que no tiene la intención específica de ejecutar simulaciones y alojar trazas. Para trabajos futuros sería necesario utilizar un servidor con las características suficientes para poder alojar y ejecutar al menos Parboil. Esto nos permitiría tener una mejor visión del comportamiento de los diferentes planificadores de warps y su interacción con la política de remplazo de caché. Teniendo una vista más amplia se procedería al diseño de un planificador de warps que aproveche la utilización de la política de remplazo de caché NRU. Otra posibilidad sería seguir realizando experimentos con diferentes políticas de remplazo de caché y con más schedulers de la literatura más reciente, para poder demostrar una correlación entre ellos.

Bibliografía

[1] Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. Accel-sim: An extensible simulation framework for validated gpu modeling. *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 473–486, 2018.

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

[3] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, and Joel Emer. High performance cache replacement using re-reference interval prediction (rrip). *Association for Computing Machinery*, 38(3):60–71, 2010.

[4] *CUDA C++ Programming Guide*. NVIDIA, 2021.

[5] Tayler H. Hetherington or M. Aamodt, Wilson W.L. Fung. Gpgpu-sim manual. URL `http://gpgpu-sim.org/manual/index.php/Main_Page`.

[6] Ashish Baghudana and Vartan Kesiz Abnousi. A parallel algorithm for lda using stochastic collapsed variational bayesian inference. 2018.

[7] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(10):115–133, 1943.

[8] OpenAI and : Achiam et all. Gpt-4 technical report, 2023.

[9] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Advances in neural information processing systems.

[10] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, 1972.

[11] Inderpreet Singh, Arrvindh Shriraman, Wilson W. L. Fung, Mike O'Connor, and Tor M. Aamodt. Cache coherence for gpu architectures. *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 578–590, 2013.

[12] Ronny Krashinsky Nick Stam Vishal Mehta Gonzalo Brito Michael Andersch, Greg Palmer and Sridhar Ramaswamy. Nvidia hopper architecture in-depth. URL `https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/`.

[13] Timothy G. Rogers Tor M. Aamodt, Wilson Wai Lun Fung. *General-Purpose Graphics Processor Architecture*. Synthesis Lectures on Computer Architecture, 2018.

[14] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. 52(4):65–76, 2009.

[15] Juan José Castillo Otón. Roofline maker. `https://github.com/JunjoFor/rooflineMaker`, 2024.

[16] Nvidia cuda documentation. URL `https://docs.nvidia.com/cuda/cuda-binary-utilities/`.

[17] Juan José Castillo Otón. Estadisticas tfg. `https://github.com/JunjoFor/estadisticas`, 2024.

[18] Juan José Castillo Otón. gpgpu-sim. `https://github.com/JunjoFor/gpgpu-sim`, 2024.