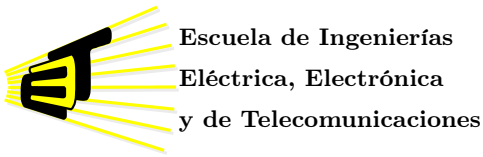
Extracción de parámetros de hardware para el modelado del tiempo de ejecución en GPU

Andersson Nicolás Sánchez Gil



Universidad Industrial de Santander
Facultad de Ingenierías Físico-Mecánicas
Escuela de Ingeniería Eléctrica, Electrónica y de Telecomunicaciones
Bucaramanga
2017

Extracción de parámetros de hardware para el modelado del tiempo de ejecución en GPU

Andersson Nicolás Sánchez Gil

Trabajo de investigación presentado como requerimiento parcial para optar por el título de Ingeniero Electrónico

Director:

William Alexander Salamanca Becerra

MSc. Ingeniería Electrónica

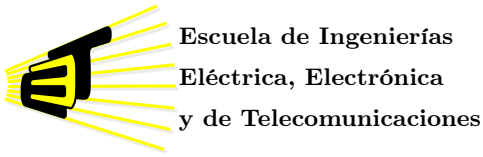
Co-Directores:

Ana Beatriz Ramírez Silva

PhD. Ingeniería Eléctrica

Dorfell Leonardo Parra Prada

MSc. Ingeniería Electrónica



Universidad Industrial de Santander

Facultad de Ingenierías Físico-Mecánicas

Escuela de Ingeniería Eléctrica, Electrónica y de Telecomunicaciones

Bucaramanga

2017

Agradecimientos

A todas las personas que de una u otra forma contribuyeron en el desarrollo de este trabajo de investigación, en especial al director y los codirectores por sus aportes en momentos en los que el trabajo llegó a puntos muertos y por el tiempo dedicado en el proceso de revisión del libro.
De la misma manera a todo esos héroes anónimos que no se pueden referenciar de manera formal y que hacen parte de la comunidad de internet: Wikipedia, Stackoverflow, Nvidia Developer Forums, GitHub, Stackexchange, Sci-Hub, Udacity, y muchos más que funcionan cuando la academia falla.

« *"A little more persistence, a little more effort, and what seemed hopeless failure may turn to glorious success."* »

Elbert Hubbard

« *"But man is not made for defeat, he said. A man can be destroyed but not defeated"* »

Ernest Hemingway
"The old man and the sea."

« Creedy: [Starts shooting the approaching V] *Die! Die! WHY WON'T YOU DIE?!* [His gun clicks empty] *Why won't you die?*
V: *Beneath this mask, there is more than flesh. Beneath this mask, there is an idea, Mr Creedy. And ideas are bullet-proof!* »

"V for Vendetta"
2005

ÍNDICE GENERAL

Pag.

ÍNDICE DE FIGURAS

Pag.

ÍNDICE DE TABLAS

Pag.

RESUMEN

TÍTULO:

Extracción de parámetros para el modelado del desempeño de una implementación GPU.*

AUTOR:
ANDERSSON NICOLÁS SÁNCHEZ GIL**

PALABRAS CLAVES:
GPU, Parámetros de hardware, Metodología, Extracción de parámetros, Herramientas, Procesamiento de datos

En este trabajo de investigación se propone y aplica una metodología para la extracción de los parámetros de hardware del modelo MWP-CWP. Inicialmente se brinda un marco teórico en el que se presentan los aspectos más relevantes de la GPU desde las perspectivas de hardware, software y arquitectura. También se presenta el modelo al que se desea extraer los parámetros y se define de manera formal los principios del análisis estadístico de los datos.
Luego se presenta la definición de parámetros de hardware y se especifican los parámetros del modelo que corresponden a este tipo. Una vez se tienen los parámetros se presenta un análisis de las herramientas disponibles para la extracción de los mismos y se asocia cada parámetro a la(s) herramienta(s) con las que se extraerá su valor. Acto seguido se define la metodología con la que se extraerán los parámetros y se precisa la manera en la que la metodología se utiliza para extraer cada parámetro.
En seguida se presenta la manera en la que se aplica la metodología para la extracción de los parámetros, esto se hace en dos capítulos: el primer capítulo está enfocado en los parámetros que se pueden extraer de manera directa y en la obtención de los datos que se requieren para los parámetros cuyo valor se extrae de manera indirecta. En el siguiente capítulo se expone la manera en la que la que se procesan estos datos, se presentan los resultados de este procesamiento y se extraen el valor de los parámetros restantes.
Finalmente se presenta el valor de los parámetros, se realiza un análisis de los mismos, se presentan las conclusiones del trabajo, los aportes adicionales del mismo y el trabajo futuro que se puede realizar a partir de la investigación.

*Trabajo de grado.

**Facultad de Ingenierías Físico-Mecánicas. Escuela de Ingenierías Eléctrica, Electrónica y de Telecomunicaciones. Directores Ph.D(c). William Alexander Salamanca Becerra, MS. Dorfell Leonardo Parra Prada, y Ph.D. Ana Beatriz Ramírez Silva.

ABSTRACT

TITLE:

Extracting the parameters for the modeling of the performance of a GPU implementation.*

AUTHOR:

ANDERSSON NICOLÁS SÁNCHEZ GIL**

KEYWORDS:

GPU, Hardware parameters, Methodology, Extracción de parámetros, Tools, Data processing

In this research work a methodology is proposed and applied in order to extract the hardware parameters of the MWP-CWP model. At first stance a theoretical framework is presented with the GPU's most relevant aspects from different perspectives: hardware, software and computational architecture. The model whose parameters are due to be extracted is presented and the statistical principles of the data processing are defined.

Afterwards, the definition of hardware parameters is done and the hardware parameters of the MWP-CWP model are specified. With those parameters specified an analysis of the tools available for the extraction is done, and each parameter is associated whit the tool(s) with which its value is extracted. Following that, the methodology is defined and the way this is used to extract each parameter is specified.

The way that the methodology is used to extract the parameters is presented in two chapters. The first one is focused on the parameters whose extraction is done directly and in the way to obtain the data that is required to extract the value of the remaining parameters. In the second one is exposed the way that the the data processing is done, the results of this processing are presented and the value of the remaining parameters are extracted.

Finally the values of the parameters are reported, their values are analyzed. The conclusions of the work are presented as well as some additional contributions and some future related work is proposed.

*Bachelor thesis.

**Facultad de Ingenierías Físico-Mecánicas. Escuela de Ingenierías Eléctrica, Electrónica y de Telecomunicaciones. MS. Directed by Dorfell Leonardo Parra Prada, Ph.D(c). William Alexander Salamanca Becerra and Ph.D. Ana Beatriz Ramírez Silva.

Introducción

En el campo de las aplicaciones de cómputo intensivo, las plataformas de cómputo heterogéneo (CPU+GPU) están desplazando a las plataformas de cómputo tradicional (solo CPU) en diversas aplicaciones que involucran cálculo científico como bioinformática, química computacional, dinámica de fluidos, ciencia de datos entre otras que aprovechan las ventajas que ofrecen este tipo de plataformas al permitir la ejecución de tareas en paralelo.

En ocasiones es necesario estimar el tiempo de ejecución de una aplicación en una GPU, bien sea porque se quiere comparar este tiempo con el tiempo de ejecución de la misma aplicación en CPU, porque se desea realizar una estimación del tiempo de ejecución de la aplicación antes de realizar su implementación o como parte de procesos de optimización de la misma aplicación. El modelo que se propone en [1,2], desarrollado en la universidad de Georgia en Estados Unidos, es un modelo general de ejecución que permite estimar el tiempo de ejecución de una aplicación sin entrar en detalle en la estructura la misma.

El objetivo de este trabajo de investigación es proponer y desarrollar una metodología que permita la extracción de los parámetros de hardware de ese modelo, de tal manera que se tenga una guía y/o manual para extraer los mismos. A este modelo no se le da una denominación en específico en la literatura [1] [2], pero ya que se desarrolla en torno a dos conceptos cuyas abreviaturas son MWP y CWP, para efecto de este trabajo se denominará en lo sucesivo el modelo MWP-CWP. La necesidad de realizar este trabajo de investigación surgió debido a que en el grupo de investigación CPS se estaba trabajando con este modelo en una tesis de maestría [3] y al revisar la literatura en la que se

[1] HONG Sunpyo; KIM Hyesoon. «An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness». En: *ACM SIGARCH Computer Architecture News*. Vol. 37. 3. ACM. 2009, págs. 152-163

[2] SIM, Jaewoong, *et al.* «A performance analysis framework for identifying potential benefits in GPGPU applications». En: *ACM SIGPLAN Notices*. Vol. 47. 8. ACM. 2012, págs. 11-22

[3] PARRA, Dorfell; SALAMANCA, William; RAMÍREZ, Ana. «Analytical Model to Estimate the Execution Time of a 3D Acoustic Wave Equation Implementation Using FDTD in a GPU». Magíster de Ingeniería Electrónica. Universidad Industrial de Santander, 2016

propone el mismo se encontró que no se definía la manera en la que se deberían extraer sus parámetros.

En el desarrollo de las etapas iniciales del trabajo se tomaron en cuenta los documentos en los que se plantea el modelo [4,5] y documentación de soporte a cerca de la programación y la arquitectura de la GPU[6,7,8,9,10,11,12].[13] En las etapas del planteamiento y el desarrollo de la metodología fue de vital importancia la documentación en la que se presentan ejemplos de uso de *microbenchmarking*[14,15,16],[17] parte de la información que se presenta en esta documentación se utilizó en el análisis de los resultados.

Para realizar este trabajo inicialmente se realizó una investigación a cerca de la programación y la arquitectura de la GPU, luego se tomó la documentación de soporte del *microbenchmarking* y se escribieron los primeros códigos de prueba. Ya con estos códigos funcionando se planteó una versión inicial de las pruebas y se empezó a perfeccionar los códigos utilizando ensayos a prueba y error. Al utilizar este tipo de ensayos el gasto de energía fue mayor que si se hubiera planteado la metodología completamente al comienzo y se hubiera desarrollado sobre los supuestos iniciales. Ya con los códigos funcionando y con el análisis de los datos obtenidos se planteó la metodología teniendo en cuenta todo el trabajo realizado, garantizando que todos sus procesos son funcionales. En síntesis la metodología se planteó a partir del trabajo realizado en el proceso de programación y depuración de los códigos con los que se obtuvieron los resultados, de tal manera que se garantiza que existe coherencia entre la metodología que se plantea y el trabajo técnico que se requiere para su desarrollo.

Los objetivos del trabajo se cumplen de acuerdo a como se plantearon en el plan: se tiene un listado de los parámetros de hardware del modelo, se tiene una relación de los parámetros y de las herramientas que se utilizan para extraer cada uno, se tiene la documentación del planteamiento y el desarrollo de la metodología y se documentan los resultados. Debido a que el fabricante no provee los valores de latencia de todos los parámetros que se hallaron no es posible evaluar los mismos de manera di-

[4]HONG Sunpyo; KIM Hyesoon. «An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness». En: *ACM SIGARCH Computer Architecture News*. Vol. 37. 3. ACM. 2009, págs. 152-163

[5]SIM, Jaewoong, *et al.* «A performance analysis framework for identifying potential benefits in GPGPU applications». En: *ACM SIGPLAN Notices*. Vol. 47. 8. ACM. 2012, págs. 11-22

[6]*NVIDIA, CUDA C Programming Guide v7.0*. 2015. URL: http://www.nvidia.com.

[7]*NVIDIA, CUDA Compiler Driver v7.0*. 2015. URL: http://www.nvidia.com.

[8]*NVIDIA, CUDA Binary Utilities driver v7.0*. 2015. URL: http://www.nvidia.com.

[9]*NVIDIA, Inline PTX Assembly in CUDA v8.0*. 2017. URL: http://www.nvidia.com.

[10]*NVIDIA, CUDA Samples v8.0*. 2017. URL: http://www.nvidia.com.

[11]*NVIDIA, Profiler User's Guide v8.0*. 2017. URL: http://www.nvidia.com.

[12]*NVIDIA, CUDA Parallel Thread Execution v7.0*. 2015. URL: http://www.nvidia.com.

[13]CHENG, John; GROSSMAN, Max; MCKERCHER, Ty. *Professional CUDA C Programming*. Jhon Wiley & Sons, 2014.

[14]MEI, Xinxin, *et al.* «Benchmarking the memory hierarchy of modern GPUs». En: *IFIP International Conference on Network and Parallel Computing*. Springer. 2014, págs. 144-156.

[15]COLLANGE, Sylvain. «Analyse de l'architecture GPU Tesla». En: *Rapport technique*. Université de Perpignan. 2010, págs. 42-54.

[16]ANDERSH, Michael, *et al.* «On latency in GPU throughput microarchitectures». En: *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*. IEEE. 2015, págs. 169-170.

[17]Ana, «Analytical Model to Estimate the Execution Time of a 3D Acoustic Wave Equation Implementation Using FDTD in a GPU».

recta, solo se pueden comparar con resultados provenientes de otros trabajos teniendo en cuenta que la metodología para la extracción de los parámetros es diferente en cada uno de ellos. Sin embargo, si se evalúa este trabajo de investigación desde una perspectiva de resultados "un árbol estaría impidiendo ver el bosque completo", ya que los aportes del mismo están mucho más allá de sus resultados.

Dentro de los aportes de este trabajo tiene la definición de los parámetros de hardware del modelo además de las herramientas disponibles para la extracción de los mismos. El mayor aporte de este trabajo consiste en el planteamiento de la metodología y la aplicación de la misma para la extracción de los parámetros. Adicionalmente, a partir de los resultados obtenidos al aplicar esta metodología se encuentra que esta es una poderosa herramienta que permite exponer la jerarquía de memoria de la GPU, se trabaja con memoria local (no se encuentra ninguna aproximación al trabajo con este tipo de memoria desde la perspectiva de hardware en la literatura utilizada), se expone el *pipelining* de las operaciones ariméticas de datos sin dependencia secuencial de resultados y se presenta un marco completo para el análisis de los datos de latencia,métricas y eventos, lo que permite aislar las métricas y eventos correspondientes a un grupo en particular de instrucciones. Por último se abre la puerta a trabajo futuro de optimización y modelado basado tanto en la metodología, como en los resultados aquí obtenidos.

El cuerpo del documento presenta en el primer capítulo un marco teórico que contiene aspectos de la arquitectura y la programación de la GPU, presenta el modelo y da luces a cerca de la estadística con la que se realiza el procesamiento de los datos, se recomienda encarecidamente realizar la lectura de este capítulo ya que en el mismo se presentan conceptos claves en torno a los que posteriormente se desarrolla el trabajo. En el segundo capítulo se define que es un parámetro de hardware en el contexto del modelo y se determinan los parámetros del modelo que son de este tipo. En el tercer capítulo se presentan y relacionan las herramientas de análisis con las que se realiza la extracción de cada parámetro. En el cuarto capítulo se plantea la metodología y se relaciona la misma con los parámetros. En el quinto capítulo se presentan los parámetros que requieren de menor complejidad para su extracción y se presentan los procesos de la metodología que se requieren para el diseño de los *microbenchmarks*. En el sexto capítulo se presenta el procesamiento de los datos resultantes de los *microbenchmarks*. Finalmente en el séptimo capítulo se presentan los resultados y las conclusiones del trabajo.

Capítulo 1

Marco Teórico

1.1. GPU

Una GPU *(Graphics Processing Unit)* es una unidad de hardware que inicialmente fue diseñada para realizar las operaciones 2D y 3D involucradas en la generación y manipulación de imágenes, de tal manera que se puedan generar imágenes, animaciones y vídeo para ser vistas en la pantalla de un computador. Desde un punto de vista técnico una GPU se define como: "un procesador en un único circuito integrado capaz de realizar transformaciones, iluminado, configuración y recorte de triángulos; que tiene motores de generación y es capaz de procesar por lo menos diez millones de polígonos por segundo" [1].

Desde una perspectiva de arquitectura representa una arquitectura *many-core*, con *multithreading*, MIMD *(Multiple Instruction Single Data)*, SIMD *(Single Instruction Multiple Data)* y paralelismo de datos [2]. Esta arquitectura en específico se denomina *Single Instruction, Multiple Thread* (SIMT).

Una interfaz conecta la GPU con la CPU a través del bus *PCI Express*. La memoria global de la GPU es de tipo DRAM, y es un recurso clave en muchas aplicaciones. El motor *GigaThread* permite asignar los bloques de *threads* a los *Warp Schedulers* de los *streaming multiprocessors*. La GPU cuenta también con una memoria L2 *cache* que se encuentra compartida para todos los *streaming multiprocessors*. En la figura 1 se puede observar el esquema de una GPU con arquitectura Kepler.

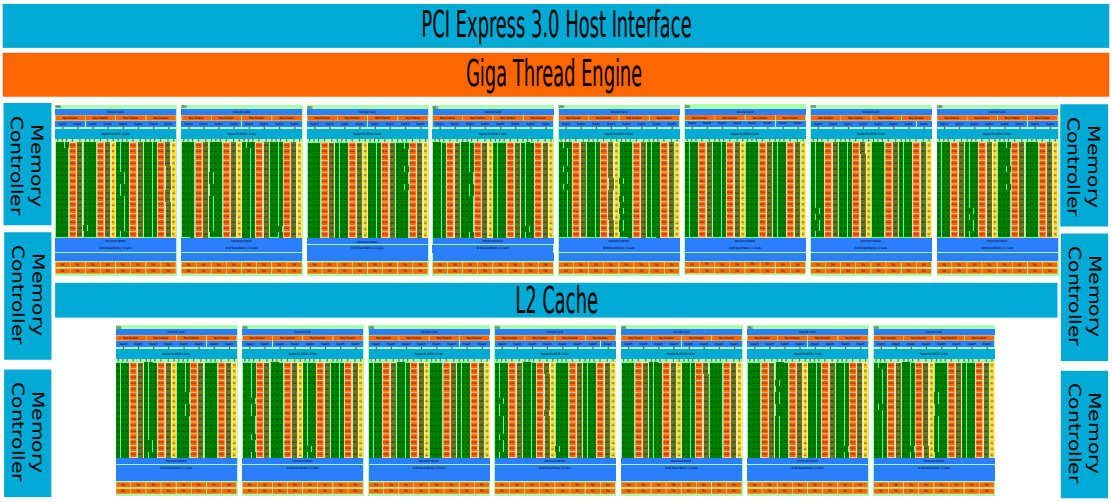
Es preciso definir dos conceptos:

- *Thread*: Representa la unidad básica de ejecución en un contexto de software. En el contexto GPUs se define un thread como el conjunto de instrucciones de uso de memoria y operaciones que se realiza sobre un conjunto de datos. Cada thread tiene asociado un índice mediante el cual accede a los datos y se tiene referencia del mismo en el contexto de ejecución en la GPU. Según el modelo de programación del fabricante estos threads se ejecutan en paralelo en la GPU.

[1] *NVIDIA, Graphics Processing Unit (GPU)*. 2015, disponible en: URL: http://www.nvidia.com

[2] CHENG, John; GROSSMAN, Max; MCKERCHER, Ty. *Professional CUDA C Programming*. Jhon Wiley & Sons, 2014

Figura 1: Diagrama de una GPU de la arquitectura Kepler.



Fuente: Adaptado de CHENG, John; GROSSMAN, Max; MCKERCHER, Ty. *Professional CUDA C Programming*. Jhon Wiley & Sons, 2014

El thread se asocia con la instrucción en el contexto de la arquitectura de *Single Instruction Multiple Thread* (SIMT).

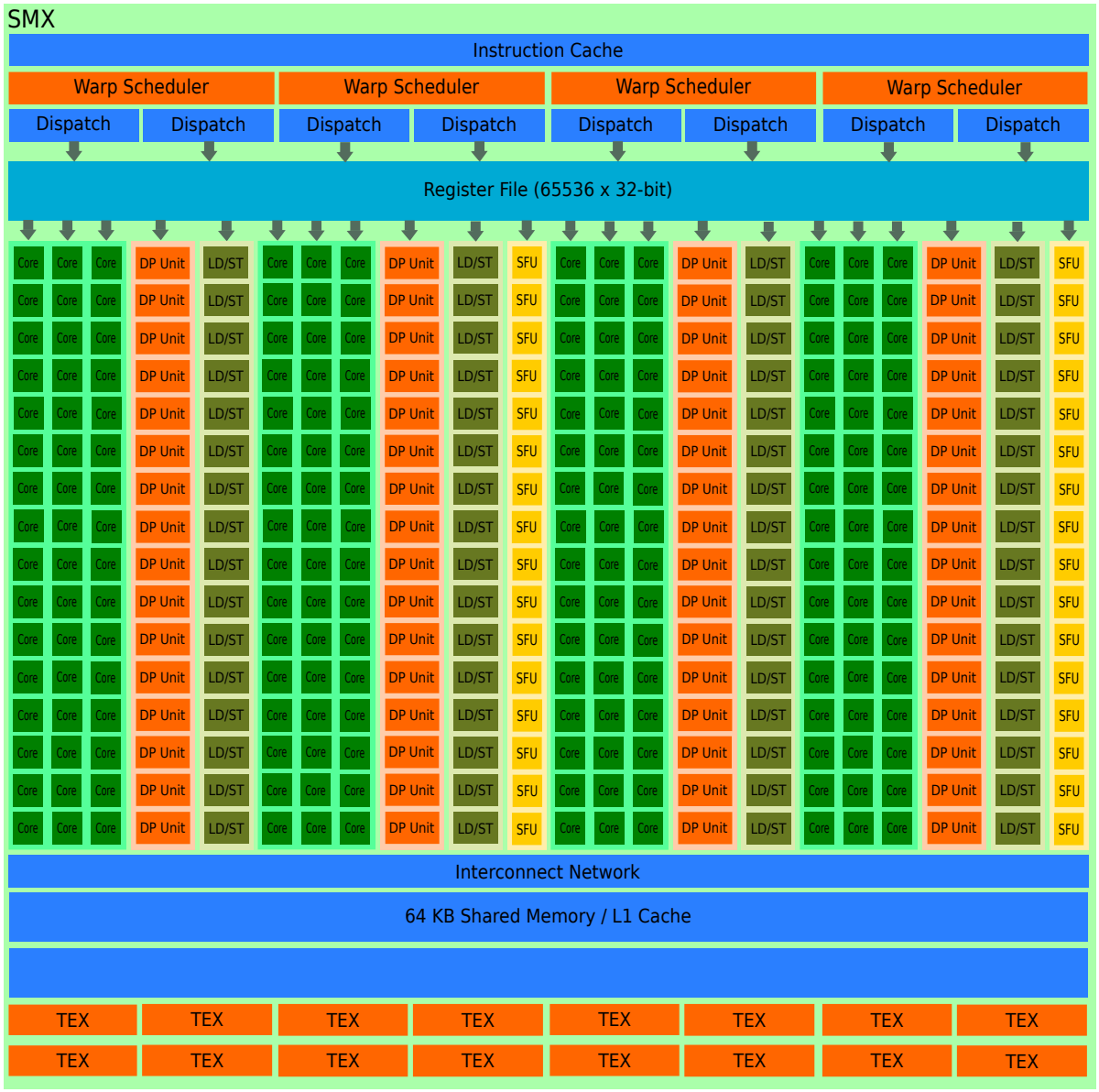
- *Warp*: representa la unidad de ejecución básica a nivel de hardware. El conjunto de instrucciones de uso de memoria y operaciones se ejecutan en hardware en grupos de threads. El tamaño del *warp* para la arquitectura Kepler es de 32 *threads*. Esto significa que cada instrucción que se ejecuta en el thread se ejecutará en hardware en grupos de a 32 de manera simultánea.

1.1.1 Streaming Multiprocessor. La GPU está construida en torno de un arreglo escalable de *Streaming Multiprocesor* (SM). Un bloque de *threads* (subprocesos) se asigna a un SM y permanece en este hasta que la ejecución se ha completado. Un SM puede tener asigando más de un bloque de *threads* a la vez. Aunque pueden existir variaciones dependiendo la arquitectura de la GPU con la que se trabaje, tienen algunos componentes comunes a todas las arquitecturas como se puede observar en la figura 2.

- *CUDA cores*: cada *CUDA core* posee una unidad lógica arimética (ALU) y una unidad de punto flotante (FPU), que ejecuta instrucciones de entero o punto flotante por ciclo de reloj.

- Memoria compartida/L1 *cache*: esta es una memoria en la que se dispone de 64[KB] configurables entre memoria compartida y L1 *cache*. La memoria compartida permite realizar comunicación y transferecia de datos entre sub-procesos (*threads*) ejecutados de manera concurrente. La memoria L1 *cache* permite acceso rápido a datos utilizando el principio de localidad.

Figura 2: Streaming Multiprocessor de la Arquitectura Kepler.

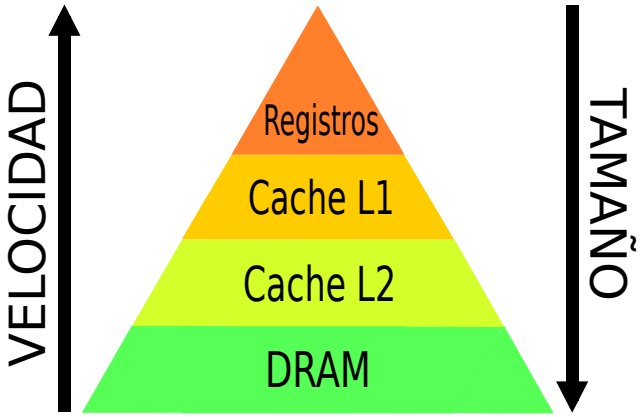


Fuente: Adaptado de CHENG, John; GROSSMAN, Max; MCKERCHER, Ty. *Professional CUDA C Programming*. Jhon Wiley & Sons, 2014.

- Registros: es el tipo de memoria más rápido de la GPU, y también la más pequeña. Se utiliza para gestionar datos propios de cada *thread*.
- Unidades de carga/almacenamiento: permiten calcular las direcciones de origen y de destino ode las operaciones.

- Unidades de funciones especiales (SFU): realizan operaciones como seno, coseno, raíz cuadrada, e interpolación; realizan estas operaciones en un ciclo de reloj por subproceso, ahorrando tiempo de ejecución.
- *Warp Scheduler* y unidad de despacho de instrucciones: se encargan de manipular la ejecución de los *threads* en los recursos de hardware disponibles (*CUDA cores*).

Figura 3: Herarquía de Memoria de una GPU.

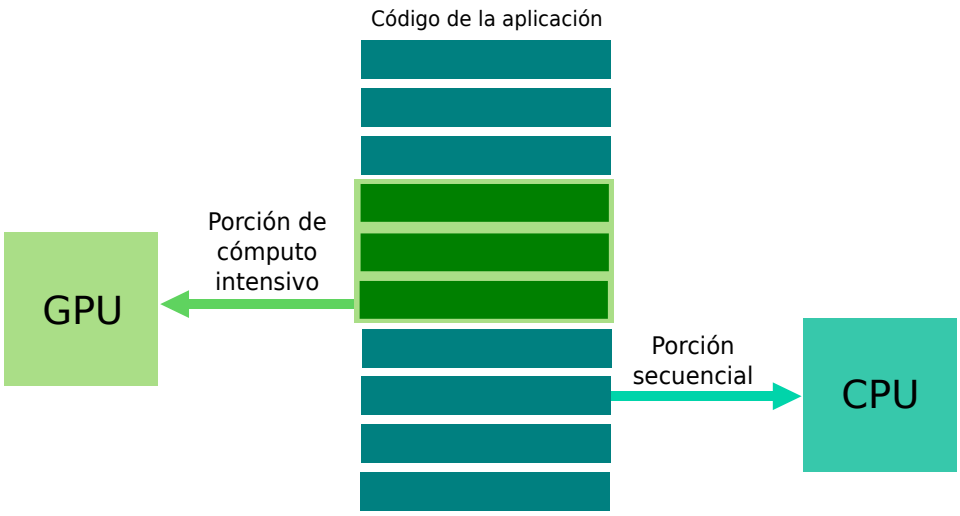


Fuente: Adaptado de CHENG, John; GROSSMAN, Max; MCKERCHER, Ty. *Professional CUDA C Programming*. Jhon Wiley & Sons, 2014.

1.1.2 Jerarquía de Memoria. La memoria principal de la GPU es de tipo *Dynamic Random Access Memory* (DRAM), la cual tiene latencias de acceso altas y su tamaño es el más grande dentro de la jerarquía de memoria. Las memorias *cache* son de tipo SRAM (*Static Random Access Memory*) y presentan latencias de acceso más bajas que la DRAM, pero su tamaño es más pequeño. Existe una memoria *cache* L2 que trabaja en un contexto global (todos los threads pueden acceder a esta) y una memoria *cache* L1 que se encuentra en cada *Streaming Multiprocessor* y a la que solo pueden acceder los threads que se ejecutan en cada SM. La memoria *cache* L2 es más grande que la memoria *cache* L1. El esquema de esta jerarquía de memoria se puede observar en al figura 3. Finalmente, los registros son el tipo de memoria más rápido y también el recurso más escaso dentro de la jerarquía de memoria. No es posible trabajar de manera directa con los espacios de memoria de esta jerarquía ya el fabricante define un modelo de programación en el que estos son invisible al usuario.
Esta jerarquía de memoria se utiliza para manipular los datos cuando el procesador se encuentra activo, de tal manera que se crea una ilusión de una memoria de gran tamaño pero de baja latencia; además esta jerarquía de memoria utiliza el principio de localidad (si una posición de memoria se encuentra referenciada, las posiciones adyacentes también podrían estarlo).

1.2. Computación Heterogénea

Figura 4: Modelo de arquitectura heterogénea.



Fuente: Adaptado de CHENG, John; GROSSMAN, Max; MCKERCHER, Ty. *Professional CUDA C Programming*. Jhon Wiley & Sons, 2014.

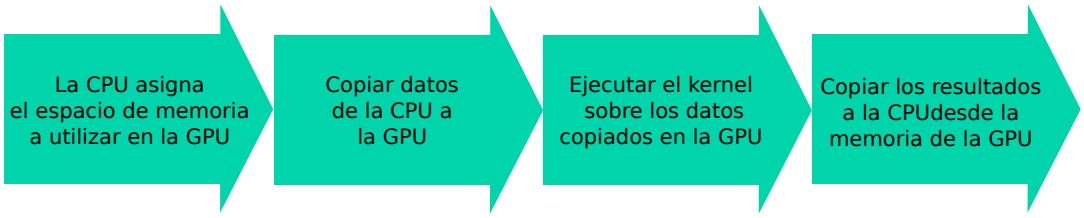
Un nodo de cómputo heterogéneo al más bajo nivel consiste en una CPU y una o más GPUs, como se puede observar en la figura 4. La GPU debe operar en conjunto con un *host* basado en una CPU a través de un bus *PCI-Express*. Es por esto que a la CPU se le denomida *"Host"* y a la GPU *"Device"*.Un programa ejecutado en una plataforma heterogénea es inicializado por la CPU que se encarga de preparar y compilar el código, entorno y datos antes de ser cargados a la GPU para realizar las tareas de cómputo intensivo que se requieran.

La arquitectura heterogénea de cómputo en paralelo de CPU+GPU se desarrolló debido a que individualmente tienen características complementarias que permiten que las aplicaciones utilicen lo mejor de cada arquitectura. Para un óptimo desempeño de esta arquitectura, las partes secuenciales o las tareas en paralelo se deben ejecutar en la CPU, y las tareas que impliquen uso paralelo de datos de manera intensiva se deben ejecutar sobre la GPU. La manera en la que una aplicación se ejecuta en una plataforma de cómputo de este tipo de presenta en la figura 5. Con el fin de poder ejecutar aplicaciones, utilizando esta arquitectura de cómputo, NVIDIA desarrollo una plataforma de programación llamada CUDA.

1.3. CUDA

CUDA (*Compute Unified Device Architecture*) es una plataforma de cómputo en paralelo de propósito general y un modelo de programación, que hace uso de los recursos de hardware en las GPUs de

Figura 5: Ejecución de una aplicación en una plataforma de cómputo heterogénea.



Fuente: Adaptado de CHENG, John; GROSSMAN, Max; MCKERCHER, Ty. *Professional CUDA C Programming*. Jhon Wiley & Sons, 2014.

NVIDIA, con el fin de resolver problemas de cómputo complejos que impliquen paralelismo.
El modelo de programación sobre la GPU está pensado para ejecutar programas que utilizan paralelismo de datos, sobre los cuales se ejecutan operaciones a manera de *threads* concurrentes. Una GPU tiene una arquitectura optimizada para esto, con una lógica de control simple y enfocada en optimizar el uso de los recursos de hardware.
La plataforma CUDA se puede trabajar a través de una extensión del lenguaje de programación C, llamada CUDA C. CUDA provee dos *Aplication Program Interface* (API) para gestionar los recursos de hardware de la GPU y organizar los datos:

- *CUDA Driver API*: una API de bajo nivel de programación, díficil de programar, pero que provee mayor control sobre la GPU.
- *CUDA Runtime API*: una API de alto nivel de programación que se encuentra sobre el CUDA *Driver API*.

La API *CUDA Runtime* descompone cada función en operaciones más sencillas que son ejecutadas sobre la plataforma *CUDA Driver API*. Se trabaja sobre la API CUDA Runtime, ya que facilita su programación utilizar CUDA C.

1.3.1 Estructura de programación de CUDA.

Es necesario definir dos términos para poder explicar la estructura de programación:

- *Host*: compuesto por la CPU y su memoria.
- *Device*: compuesto por la GPU y su memoria.

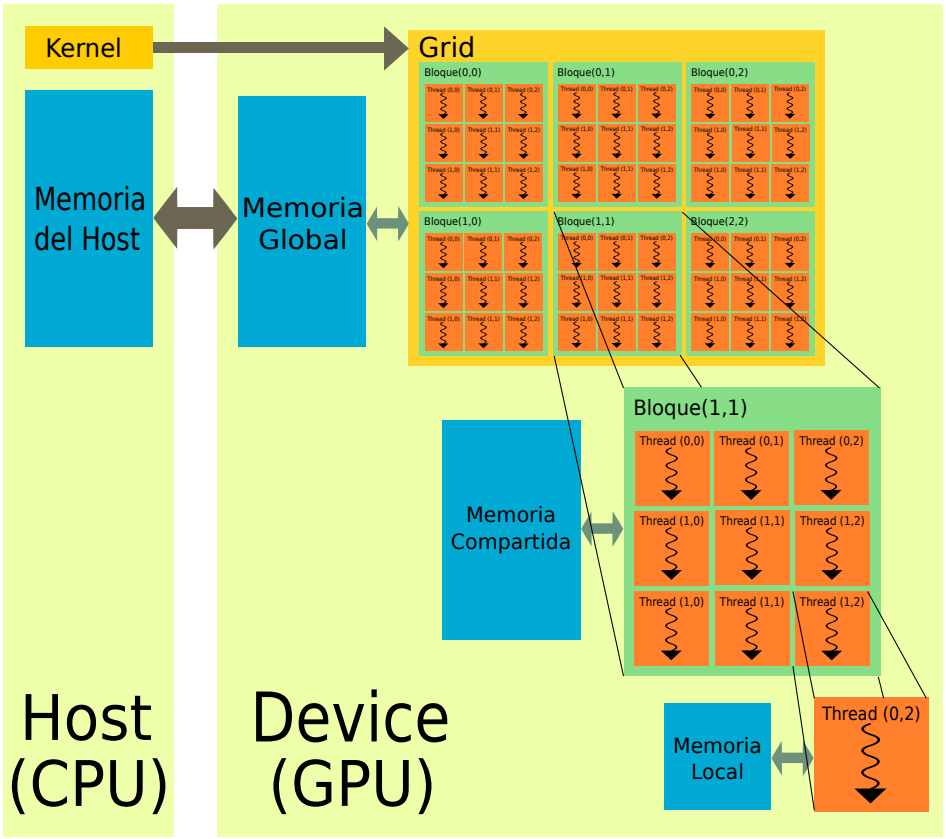
Un programa escrito en CUDA C está compuesto por el código del *host* (que es ejecutado por la CPU), codificado en lenguaje C, y el código del *device* (ejecutado por la GPU) codificado en CUDA C. El compilador de NVIDIA para la plataforma CUDA `nvcc` separa el código de *device* y del *host* durante la compilación. El compilador genera códigos ejecutables tanto para la GPU como para la CPU.

El código que es ejecutado sobre la GPU recibe el nombre de *kernel*. El *host* puede trabajar de manera independiente del *device*; cuando el *kernel* es ejecutado sobre la GPU, el control inmediatamente retorna a la CPU (a menos que se especifique o contrario), permitiendo que la CPU realice tareas adicionales mientras la GPU ejecuta su código.

1.3.2 Organización y modelo de memoria de los threads en CUDA. El modelo de memoria de CUDA y la manera en la que se organizan los *threads* se pueden observar en la figura 6. El orden del modelo de programación de CUDA se define mediante:

- *Threads*: Es la unidad básica del modelo de programación. Cada *thread* ejecuta el *kernel* de manera independiente para un conjunto de datos a los cuales accede a través de un índice, índice que también sirve para identificar el *thread* en el contexto de ejecución.

Figura 6: Organización de los threads y modelo de memoria en CUDA.



Fuente: Adaptado de CHENG, John; GROSSMAN, Max; MCKERCHER, Ty. *Professional CUDA C Programming*. Jhon Wiley & Sons, 2014.

- Bloques: consiste de un grupo de *threads* que pueden cooperar entre ellos utilizando una sincronización local de *threads* y una espacio de memoria denominado memoria compartida. Los *threads* que son de distintos bloques no pueden cooperar entre ellos.
- Grid: Todos los *threads* generados y ejecutados por un *kernel* se dice que están organizados en una *grid*. Todos los *threads* en una *grid* comparten el mismo espacio en misma memoria gobal del *device*. Una *grid* está formada por bloques de *threads*.

Usualmente una *grid* se organiza como un arreglo 2D de bloques, y los bloques se organizan como arreglos 3D de *threads*. El tamaño de *grid* y de bloques afecta el desempeño de un *kernel*; además sus dimensiones se encuentran limitadas por los recursos de hardware disponibles en el *device*.
Este modelo de memoria, que es visto desde la perspectiva de la programación, no debe confundirse con la jerarquía de memoria física del *device*; si bien la arquitectura del mismo y este modelo se encuentran relacionados, son modelos vistos desde perspectivas diferentes (software y hardware).

- Memoria local: cada *thread* tiene acceso a este tipo de memoria, y esta memoria es privada para cada *thread* (variables locales). Los *threads* pueden leer y escribir en la memoria local.
- Memoria compartida: cada *thread* que se ejecuta en el *Streaming Multiprocessor* tiene acceso a este tipo de memoria, este tipo de memoria se encuentra definida por bloque de *threads*.
- Memoria global: cada *thread* en el sistema completo (*grid*), en cualquier momento, puede leer y escribir desde y hasta este tipo de memoria.

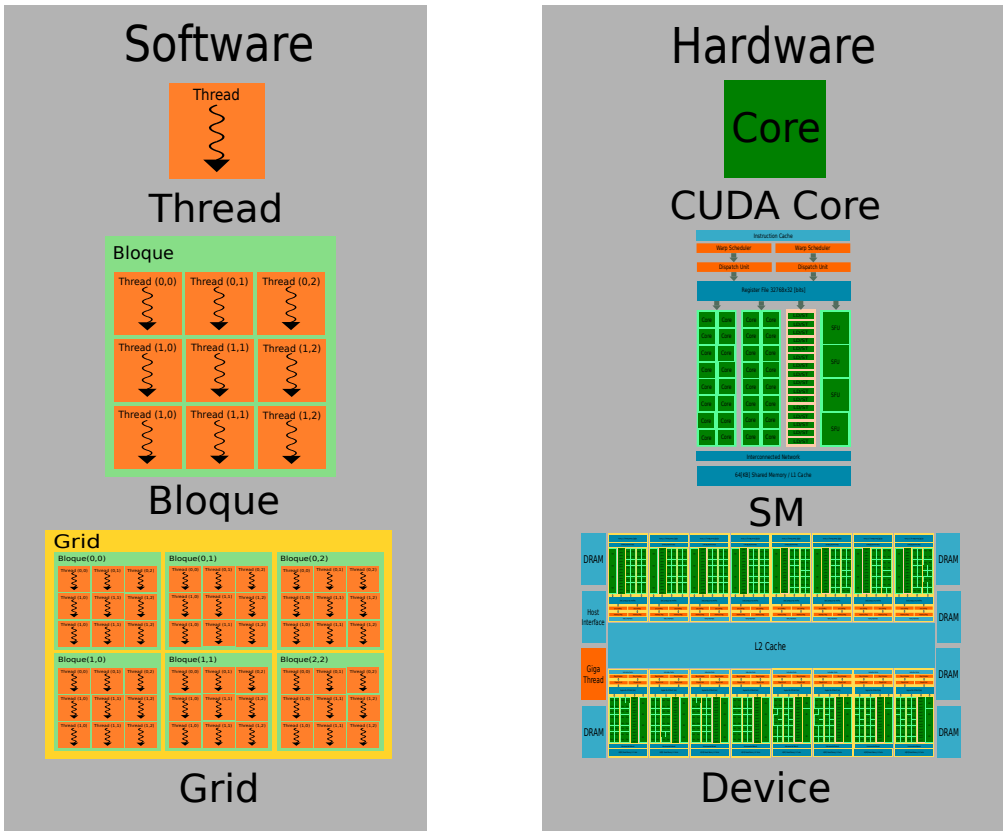
1.3.3 Modelo de ejecución de CUDA. Al ejecutar un *kernel* en la GPU, el *kernel* se ejecuta de manera global para todos los *threads* de la *grid*. Los bloques de *threads* definidos en la *grid* se asignan para ser ejecutados en los SM; esto implica que los recursos presentes en cada SM son repartidos entre los *threads* de cada bloque. Un bloque se denomina activo cuando los recursos de memoria compartida y registros han sido asignados a este. Las operaciones de cómputo que se ejecutan en cada thread son ejecutadas en el *CUDA-core*.
La manera en la que se organizan los *threads* en CUDA se relaciona con la estructura de la GPU de tal forma que la *grid* se asocia al *device*, los bloques a los *Streaming Multiprocessors* y los *threads* a los *CUDA-cores*, como se puede observar en la figura 7.

1.3.4 Estructura de Compilación en CUDA. Directamente de CUDA es poca la información que se puede obtener acerca de la ejecución en hardware de un *kernel*, pero en el proceso de compilación se generan algunos archivos que contienen información útil [3]. Algunos de estos archivos generados durante el proceso de compilación se muestran en la figura 8.

[3] *NVIDIA, CUDA Compiler Driver v7.0.* 2015. URL: http://www.nvidia.com

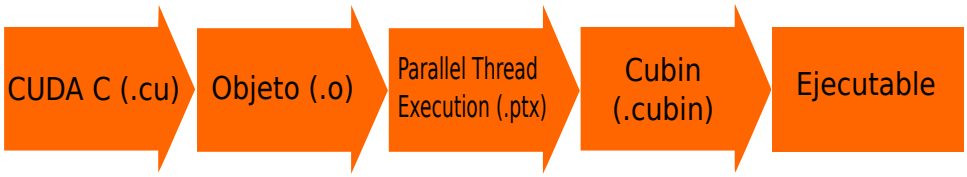
- Archivo CUDA C (.cu): es el archivo que se encuentra escrito en CUDA C, y en el que está escrito el *kernel*.
- Archivo objeto (.o): es un archivo que contiene contiene lo que se denomina código objeto, un código reubicable en formato máquina que Usualmente no se puede ejecutar directamente. Adicionalmente contiene información acerca del enlace y la depuración del código.
- Archivo PTX (.ptx): es un archivo intermedio de ensamblaje que contiene información acerca de la ejecución en paralelo de los threads.
- Archivo cubin (.cubin): es un archivo binario de CUDA de tipo ELF *(Executable and Linkable Format)*, contiene segmentos de código ejecutable, información acerca de símbolos,localizadores, información de la depuración y otros adicionales.
- Archivo ejecutable: es el archivo que como tal contiene la información binaria con la que se ejecuta en hardware el código.

Figura 7: Modelo de ejecución de CUDA.



Fuente: Adaptado de CHENG, John; GROSSMAN, Max; MCKERCHER, Ty. *Professional CUDA C Programming*. Jhon Wiley & Sons, 2014.

Figura 8: Archivos con información de la ejecución en hardware generados en la compilación.



Fuente: Adaptado de *NVIDIA, CUDA Compiler Driver v7.0.* 2015. URL: `http://www.nvidia.com`.

Es posible extraer y analizar la información contenida en estos archivos utilizando herramientas de análisis ofrecidas por CUDA.

1.3.5 Herramientas de análisis en CUDA. CUDA ofrece algunas herramientas para el análisis de la ejecución de un *kernel*, a partir del análisis del archivo binario con el que se ejecuta. Un archivo binario de CUDA (cubin) es un archivo que consta de segmentos de código ejecutable y de otras secciones que proveen información acerca de símbolos, información de la compilación y otros. Este tipo de archivo se carga para la ejecución dentro del CUDA *driver* API [4]. Las herramientas que provee CUDA para examinar y desensamblar este tipo de archivos son:

- cuobjdump: acepta archivos cubin y archivos ejecutables del *host* en los que se encuentran embebidos, y los presenta en un formato que hace que la información sea comprensible para un ser humano [4].
- nvdisasm: solo acepta archivos cubin, pero tiene maś opciones para la manipulación de la infomación que cuobjdump, y de la misma manera presenta la información en un formato humanamente comprensible [4].

Adicionalmente NVIDIA ofrece una serie de herramientas denominadas CUPTI *(NVIDIA CUDA Profiling Tools)*, las cuales son herramientas de análisis de desempeño que arrojan información detallada acerca de como las aplicaciones utilizan la GPU. Dentro de estas herramientas está nvprof, una herramienta que permite el análisis por línea de comandos.

1.4. GPU Tesla K40

Esta GPU tiene arquitectura Kepler y capacidad de cómputo 3.5. La capacidad de cómputo es un atributo que ofrece el fabricante y que hace referencia a las características de la arquitectura se pueden implementar en el *device*. Las características más relevantes de esta GPU se presentan en la tabla 1. Los *Streaming Multiprocessors* de la arquitectura Kepler tienen cuatro *warp schedulers* y ocho *instruction dispatch units*, lo que permite que cuatro *warps* sean seleccionados y ejecutados de manera

[4] *NVIDIA, CUDA Binary Utilities driver v7.0.* 2015. URL: `http://www.nvidia.com`

Tabla 1: Características de la GPU Tesla K40. (Adaptado del *deviceQuery* de la GPU).

Total amount of global memory:	11520 MBytes (12079136768 bytes)
(15) Multiprocessors, (192) CUDA Cores/MP:	2880 CUDA Cores
GPU Clock rate:	745 MHz (0.75 GHz)
Memory Clock rate:	3004 Mhz
Memory Bus Width:	384-bit
L2 Cache Size:	1572864 bytes
Maximum Texture Dimension Size (x,y,z)	1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
Maximum Layered 1D Texture Size, (num) layers	1D=(16384), 2048 layers
Maximum Layered 2D Texture Size, (num) layers	2D=(16384, 16384), 2048 layers
Total amount of constant memory:	65536 bytes
Total amount of shared memory per block:	49152 bytes
Total number of registers available per block:	65536
Warp size:	32
Maximum number of threads per multiprocessor:	2048
Maximum number of threads per block:	1024
Max dimension size of a thread block (x,y,z):	(1024, 1024, 64)
Max dimension size of a grid size (x,y,z):	(2147483647, 65535, 65535)
Maximum memory pitch:	2147483647 bytes

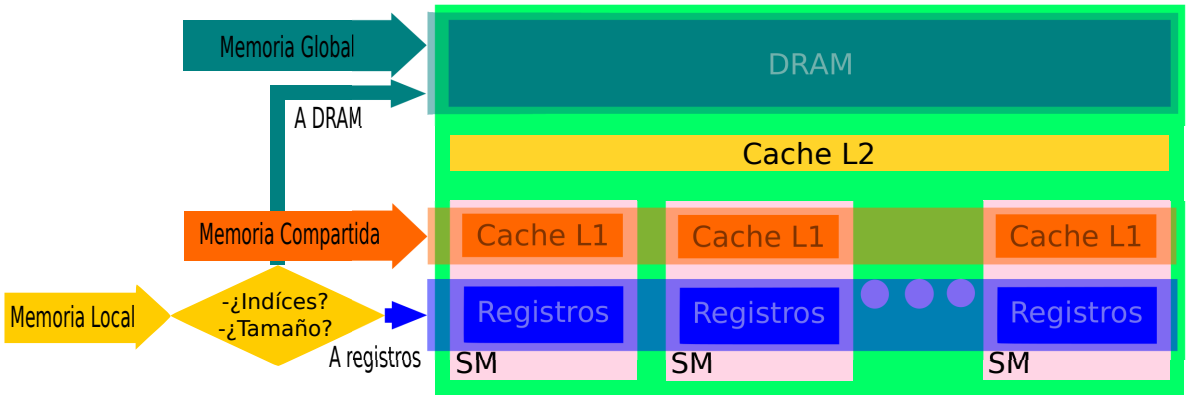
concurrente. Los *warp schedulers* seleccionan cuatro *warps* y pueden despachar dos instrucciones independientes por *warp* cada ciclo de reloj.[5]

1.4.1 Jerarquía de memoria y modelo de memoria de CUDA. Según el modelo de memoria de CUDA existen tres posibles espacios de memoria: local, compartida y global. De acuerdo a la jerarquía de memoria de la GPU los espacios físicos de memoria son: registros, memoria *cache* L1, memoria *cache* L2 y DRAM. La jerarquía de memoria permanece oculta la usuario a través del modelo de memoria de CUDA, aún así es posible determinar el espacio de memoria dentro de la jerarquía de en el que se almacena la información como se puede observar en la figura 9:

- Memoria local: los datos se guardan en registros, si los datos tienen índices que no se conocen durante la compilación o si el tamaño del dato es muy grande, de tal forma que no se puede almacenar utilizando los registros disponibles por thread, los datos se almacenan en DRAM.
- Memoria compartida: los datos de memoria compartida se almacenan físicamente en la memoria *cache* L1. Esta memoria se encuentra en cada SM, y aunque compartan el mismo espacio de memoria que la *cache* L1, el método de acceso es diferente.
- Memoria global: los datos de memoria global se guardan en la DRAM del *device*.

[5] *NVIDIA, NVIDIA Kepler GK110 Architecture Whitepaper*. 2015. URL: http://www.nvidia.com.

Figura 9: Jerarquía de memoria y modelo de memoria de CUDA. En esta imagen se pueden observar los espacios de memoria dentro de la jerarquía en los cuales se almacena la información de los espacios de memoria definidos en el modelo de memoria de CUDA.



La manera en la que la jerarquía y el modelo de memoria de CUDA interactúan depende de la arquitectura de la GPU, en este caso la arquitectura es Kepler. Ya que las operaciones numéricas que se realicen sobre los datos se hacen a nivel de registros, las operaciones de carga y almacenamiento más relevantes que se ejecutan en al GPU se hacen desde los espacios de memoria al espacio de registros de la GPU.

Si el espacio de memoria corresponde a memoria local y el dato se encuentra almacenado en registros, la operación que se realiza es un movimiento entre registros o se podría pensar en trabajar directamente con el registro en el cual se encuentra almacenado el dato. Si el dato se encuentra en memoria DRAM las operaciones de carga y almacenamiento de hacen a través de la memoria *cache* L1 y la memoria *cache* L2. De tal manera que al realizar una operación de carga o almacenamiento de un dato en memoria local en DRAM existe la posibilidad de encontrar el dato en memoria *cache* L1, memoria *cache* L2 o DRAM, comoo se puede observar en la figura 10.

Para la memoria global las operaciones de carga y almacenamiento de datos se realizan a través de la memoria *cache* L2, si bien la memoria cache L1 se encuentra físicamente conectada entre la memoria *cache* L1 y el espacio de registros para este tipo de memoria la GPU realiza una operación de *bypass* sobre la memoria *cache* L1 [6]. De tal manera que al realizar una operación de carga o almacenamiento de datos es memoria global existe la posibilidad de encontrar el dato en memoria *cache* L2 o en DRAM como se puede observar en la figura 11.

[6] *NVIDIA, CUDA Parallel Thread Execution v7.0.* 2015. URL: http://www.nvidia.com

Figura 10: Memoria local y jerarquía de memoria.

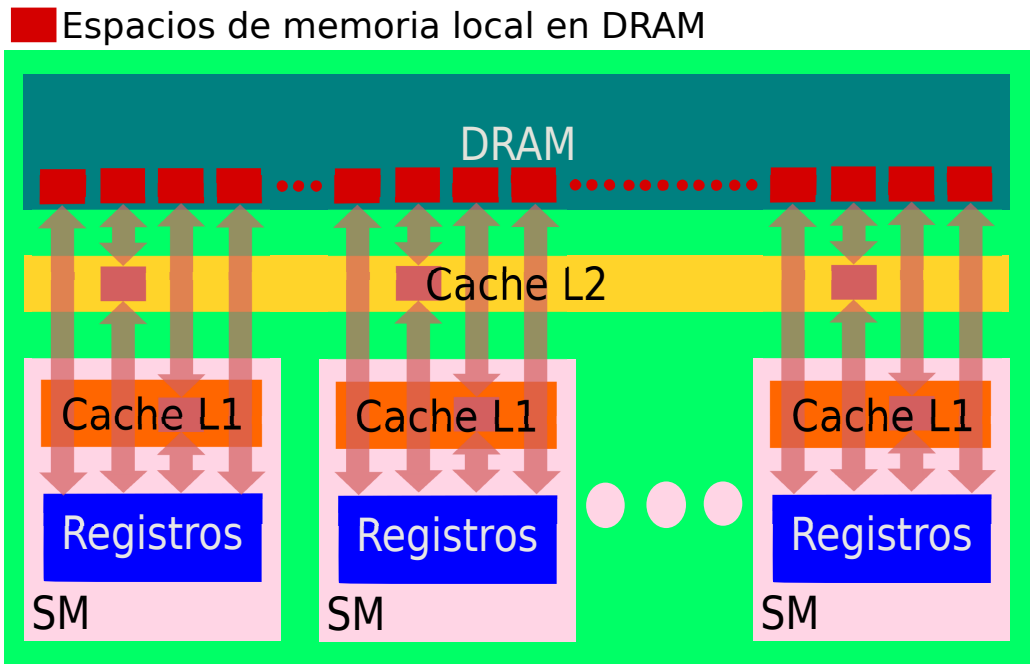
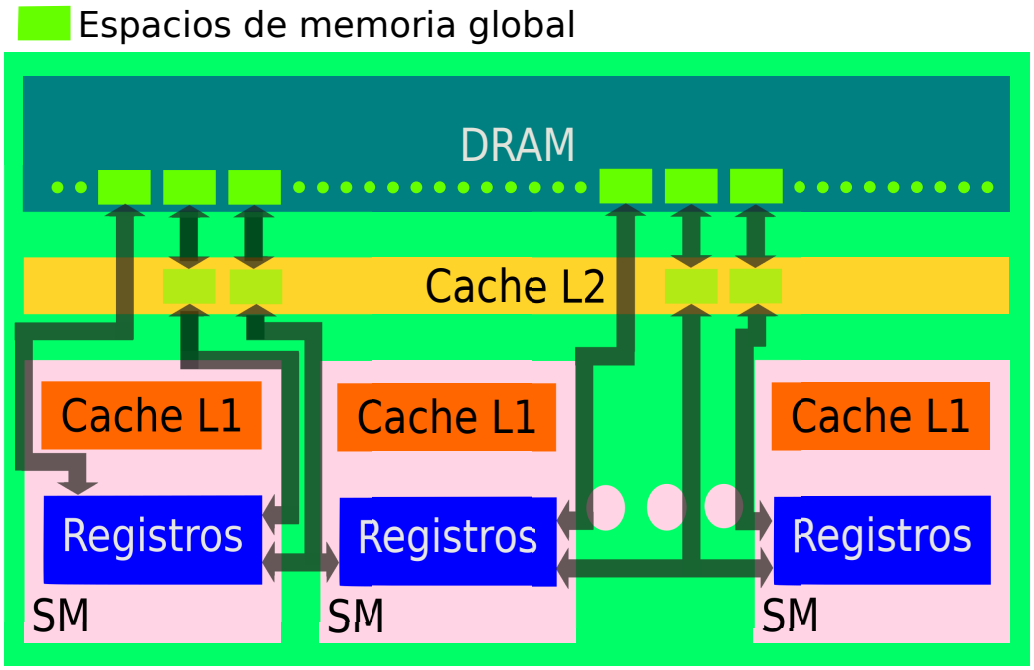


Figura 11: Memoria global y jerarquía de memoria.



1.5. PTX e ISA

PTX es la abreviatura de *Parallel Thread Execution*, y hace referencia a una máquina virtual de bajo nivel que permite desarrollar un modelo de ejecución en paralelo de *threads*. Este modelo va acompañado de una arquitectura de un *Instruction Set Architecture* (ISA), las cuales son instrucciones similares a las utilizadas en assembler. Al hacer la conversión de PTX a los archivos binarios de la GPU se permite que las GPUs sean vistas como una computadora paralela programable [7].
Los compiladores de alto nivel, como el nvcc utilizado para compilar los programas escritos en CUDA C, generan instrucciones PTX que luego se optimizan y se traducen a las instrucciones de la arquitectura del hardware sobre las cuales deben ser ejecutadas [7]. El set de instrucciones cambia de una arquitectura de GPU a otra, y virtualmente permite las mismas instrucciones de ejecución de un *kernel* que CUDA C, pagando el precio de una programación más compleja.

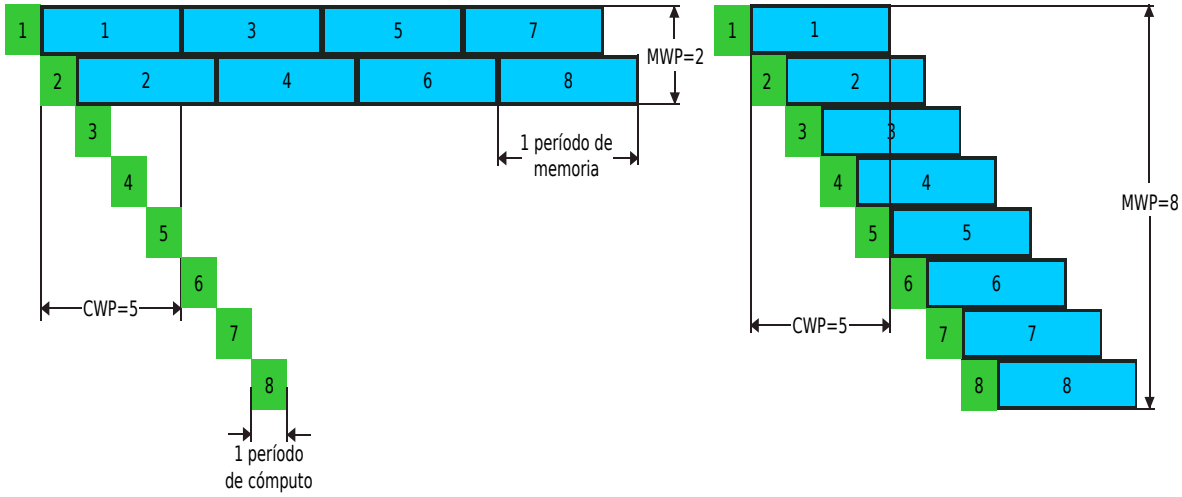
La ventaja de utilizar PTX es que permite una programación más cercana al hardware al utilizar instrucciones de tipo assembler, lo cual permite una manipulación de partes específicas de hardware y la escritura de *kernels* más eficientes al momento de su ejecución.

1.6. Modelo MWP-CWP

Este modelo se encuentra documentado en [8] [9] y se utiliza para estimar el tiempo de ejecución de una aplicación en una GPU.
Este modelo analítico se basa en dos conceptos: el paralelismo de memoria de los *warps*, *memory warp parallelism* (MWP), y el paralelismo de cómputo de los *warps computation warp parallelism*(CWP). Según este modelo, hallar el tiempo de ejecución de las operaciones de memoria es la clave para determinar el desempeño de las aplicaciones en una GPU, debido a que el tiempo de ejecución de estas depende principalmente del tiempo de latencia de las operaciones de memoria[8].
El paralelismo de memoria de los *warps* se refiere específicamente a cuantas operaciones de memoria pueden ser ejecutadas de manera concurrente. El paralelismo de cómputo de los *warps* se encuentra relacionado con la cantidad de cómputo que pueden realizar otros *warps*, mientras un *warp* se encuentra realizando una operación de memoria. Este modelo se encuentra diseñado como un modelo estático (que no requiere de una implementación para estimar el tiempo de ejecución de la aplicación).
Este modelo se construye en base a ecuaciones que utilizan valores constantes (parámetros). Estos parámetros se deben estimar de antemano y sus valores pueden depender o no de otros parámetros de la aplicación para la que se desea realizar la estimación.

[7] *NVIDIA, CUDA Parallel Thread Execution v7.0*. 2015. URL: http://www.nvidia.com
[8] HONG Sunpyo; KIM Hyesoon. «An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness». En: *ACM SIGARCH Computer Architecture News*. Vol. 37. 3. ACM. 2009, págs. 152-163
[9] SIM, Jaewoong, *et al.* «A performance analysis framework for identifying potential benefits in GPGPU applications». En: *ACM SIGPLAN Notices*. Vol. 47. 8. ACM. 2012, págs. 11-22

Figura 12: Representación gráfica del modelo MWP-CWP.



Fuente: Adaptado de HONG Sunpyo; KIM Hyesoon. «An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness». En: *ACM SIGARCH Computer Architecture News*. Vol. 37. 3. ACM. 2009, págs. 152-163.

Este modelo está basado en un análisis de las operaciones de cómputo, de acceso a memoria y la manera en la que estos accesos se hacen. Este modelo fue desarrollado para determinar los posible cuellos de botella que se puedan llegar a generar y la manera en la que se pueden realizar optimizaciones para mejorar el desempeño de las aplicaciones [10].

Este modelo requiere de parámetros que involucran características propias del hardware (respecto a la disponibilidad de recursos), la manera en la que están organizados los *threads*, los tiempos en los que se realizan operaciones de cómputo, y finalmente los tiempos y la manera en la que se realizan los accesos a memoria.

El modelo se desarrollado en torno a dos conceptos:

- MWP: Hace referencia al número máximo de de *warps* por *streaming multiprocessor* que pueden acceder a memoria de manera simultánea.
- CWP: Se refiere al número máximo de *warps* que pueden finalizar un periodo de cómputo de una operación, durante un periodo de espera de un acceso a memoria, más uno.

En el caso en el que MWP es menor que CWP el costo en tiempo de las operaciones de cómputo se encontrará oculto por las operaciones de acceso a memoria, y el costo total de ejecución se encuentra determinado por las operaciones de memoria. El otro caso ocurre cuando MWP es mayor que CWP, en

[10] SIM, Jaewoong, *et al.* «A performance analysis framework for identifying potential benefits in GPGPU applications». En: *ACM SIGPLAN Notices*. Vol. 47. 8. ACM. 2012, págs. 11-22

este el costo de las operaciones de memoria se encuentra oculto por el de las operaciones de cómputo, de tal manera que el costo total de ejecución estará dado por las operaciones de cómputo [11]. En la figura 12 se puede observar una representación gráfica de estos conceptos.
A partir del desarrollo de estos dos conceptos, se derivan una serie de parámetros de este modelo que buscan determinar el cálculo del tiempo de ejecución. Estos parámetros se encuentran documentados en [11,12].

1.7. *Microbenchmarks* y extracción de parámetros

En literatura consultada [11][13,14,15] la principal herramienta que se encuentra para la extracción de parámetros son los *microbenchmarks*, un *microbenchmark* es un *kernel* en el cual se toman medidas de tiempo en torno un segmento de código el cual ha sido diseñado para determinar el comportamiento de alguna parte específica del hardware. Usualmente estos *microbenchmarks* se ejecutan dentro de un bucle de tal manera que se cuente con múltiples resultados que garanticen la fiabilidad de la medida. Esta herramienta resulta útil para determinar características de acceso a memoria, optimizaciones y de ejecución, que el fabricante no provee; si bien, se encuentra documentada la arquitectura del dispositivo, existen algunos vacíos respecto a la manera en que operan algunos de sus componentes (como las memorias y la forma en que se accede a las mismas). Estas herramienta tiene la flexibilidad de ser usadas cuando se requieren tomar métricas de alguna característica en específico dentro de la ejecución de un algoritmo. También existen algunas herramientas asociadas a CUDA que permiten conocer parámetros en particular del hardware sobre el cual se está trabajando.

1.8. Suma y resta de variables aleatorias

Se tienen dos variables aleatorias continuas independientes X y Y cada una con funciones de densidad de probabilidad *f(x)* y *g(y)* respectivamente, la función de densidad de probabilidad de la suma de las dos variables aleatorias $Z=X+Y$ es la convolución de *f(x)* y *g(y)*.

$$(f * g)z = \int_{-\infty}^{\infty} f(z-y)g(y)dy = \int_{-\infty}^{\infty} g(z-x)f(x)dx \tag{1.1}$$

Para el caso discreto se parte de dos variables aleatorias discretas independientes X y Y con funciones de densidad de probabilidad *m1(x)* y *m2(x)* respectivamente. La función de densidad de probabilidad

[11] HONG Sunpyo; KIM Hyesoon. «An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness». En: *ACM SIGARCH Computer Architecture News*. Vol. 37. 3. ACM. 2009, págs. 152-163

[12] SIM, Jaewoong, *et al.* «A performance analysis framework for identifying potential benefits in GPGPU applications». En: *ACM SIGPLAN Notices*. Vol. 47. 8. ACM. 2012, págs. 11-22

[13] MEI, Xinxin, *et al.* «Benchmarking the memory hierarchy of modern GPUs». En: *IFIP International Conference on Network and Parallel Computing*. Springer. 2014, págs. 144-156

[14] WONG, Henry, *et al.* «Demystifying GPU microarchitecture through microbenchmarking». En: *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*. IEEE. 2010, págs. 235-246

[15] COLLANGE, Sylvain. «Analyse de l'architecture GPU Tesla». En: *Rapport technique*. Université de Perpignan. 2010, págs. 42-54

correspondiente a la suma de las dos variables aleatorias discretas *Z=X+Y* será la convolución de las funciones de densidad de probabilidad *m1(x)* y *m2(x)*.

$$(m1 * m2)z = \sum_k m_1(k)m_2(j-k) \tag{1.2}$$

Se define la función de distribución acumulativa a partir de la integral:

$$F_X(x) = \int_{-\infty}^{x} f(u)du \tag{1.3}$$

La función de densidad de probabilidad se define a partir de la función de distribución acumulativa como:

$$f(x) = \frac{d}{dx}F_X(x) \tag{1.4}$$

La resta de dos variables aleatorias se define a partir de la suma como *Z=X+(-Y)*, de manera tal que se halla la función de densidad de probabilidad de $-Y$ y se aplica convolución de la misma manera que para la suma. Para hallar la función de densidad de probabilidad de $-Y$ se parte de la función de distribución acumulativa:

$$F_Y(y) = P(Y \leq y) \tag{1.5}$$

$$F_{-Y}(y) = P(-Y \leq y) = P(Y \geq -y) = 1 - P(Y < y) = 1 - F_Y(-y) \tag{1.6}$$

Derivando estas funciones de distribución acumulativa se obtiene la función de densidad de probabilidad (ecuación 1.4).

$$\frac{d}{dy}F_{-Y}(y) = \frac{d}{dx}(1 - F_Y(-y)) \tag{1.7}$$

$$f_{-Y}(y) = f_Y(-y) \tag{1.8}$$

De tal forma que si se tienen dos variables aleatorias X y Y con funciones de densidad de probabilidad *f(x)* y *g(y)* respectivamente, la función de densidad de probabilidad de la resta de las dos variables aleatorias *Z=X-Y=X+(-Y)* será igual a la convolución de *f(x)* y *g(-y)*.

Capítulo 2

Parámetros de Hardware del modelo MWP-PCWP

Respecto al modelo con el que se trabaja y que se presenta en [1,2] es necesario hacer algunas aclaraciones de la manera en la que se plantea el mismo

- En la documentación en la que se presenta el modelo este no recibe un nombre en específico, el nombre de "modelo MWP-CWP" se utiliza en este trabajo para referirse al mismo ya que el modelo se basa en los conceptos de MWP y CWP (ver marco teórico).
- Este modelo sirve para estimar el tiempo de ejecución de una aplicación en GPU, este modelo no contempla el tiempo total de ejecución de la aplicación en la plataforma de cómputo heterogénea, luego los tiempos de ejecución de la aplicación en CPU y de transferencia de datos entre CPU y GPU no se toman en cuenta en el mismo.
- El modelo se basa en los conceptos de MWP y CWP que se presentan en el marco teórico, los cuales están directamente relacionados con las latencias de acceso a memoria y de operaciones ariméticas, sin embargo, la complejidad del modelo está más allá de estos dos conceptos en la medida que se toman en cuenta características de hardware de la GPU y de la aplicación de la que se desea estimar el tiempo de ejecución.
- El modelo no específica los parámetros de hardware del mismo, debido a esto, es necesario definir qué es un parámetro de hardware en el contexto del modelo y hallar los parámetros del mismo que son de este tipo.

2.1. *Parámetros de hardware*

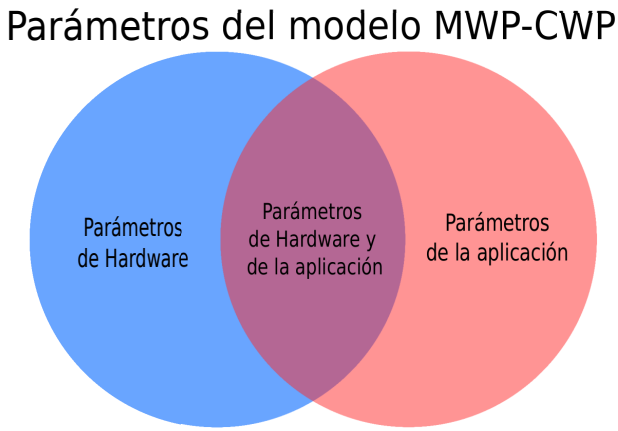
Este modelo, al realizar una estimación del tiempo de ejecución de una aplicación, tiene algunos parámetros que dependen del problema que se desea solucionar y de las condiciones de ejecución de

[1] SIM, Jaewoong, *et al.* «A performance analysis framework for identifying potential benefits in GPGPU applications». En: *ACM SIGPLAN Notices*. Vol. 47. 8. ACM. 2012, págs. 11-22

[2] HONG Sunpyo; KIM Hyesoon. «An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness». En: *ACM SIGARCH Computer Architecture News*. Vol. 37. 3. ACM. 2009, págs. 152-163

la aplicación, y otros que dependen directamente de la GPU sobre la que se desee implementar la aplicación como se puede observar en la figura 13; a este último tipo de parámetros se le denominará en lo sucesivo parámetros de *hardware*. Estos parámetros de hardware se encuentran ligados a las especificaciones y la arquitectura de la GPU, y no requieren de diseñar o implementar una aplicación para ser hallados.

Figura 13: Tipos de parámetros del modelo MWP-CWP de acuerdo a su dependencia del hardware y de la aplicación. Se presentan los tipos de parámetros del modelo MWP-CWP según su dependencia del hardware y de la aplicación



Al realizar un análisis del modelo, se llega a la conclusión que los parámetros de hardware son los siguientes;

2.1.1 ***Average instruction latency*** Este parámetro hace referencia al promedio del costo de operaciones de punto flotante en la GPU [3]. Si bien, la naturaleza de las operaciones de cómputo se encuentran definidas por la aplicación, las latencias dependen la GPU sobre la GPU sobre la cual esta se desee implementar; y por esta razón se le considera un parámetro de hardware. Dentro del modelo se encuentra definida como avg_instr_lat[4,3].

2.1.2 ***Core frequency*** La unidad lógico-arimética y la unidad de operaciones de punto flotante se encuentran en los *CUDA-cores*. La frecuencia a la que operan estos *CUDA-cores* determina la latencia de las operaciones, en términos de tiempo.

[3]SIM, Jaewoong, *et al.* «A performance analysis framework for identifying potential benefits in GPGPU applications». En: *ACM SIGPLAN Notices*. Vol. 47. 8. ACM. 2012, págs. 11-22

[4]HONG Sunpyo; KIM Hyesoon. «An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness». En: *ACM SIGARCH Computer Architecture News*. Vol. 37. 3. ACM. 2009, págs. 152-163.

2.1.3 *Warp size* Hace referencia al número de *threads* que ejecutan una instrucción al mismo tiempo. Este representa el bloque de ejecución básico a nivel de hardware. Dentro del modelo se encuentra definido como *warp_ size*.

2.1.4 *SIMD width* Un *Streaming Multiprocessor* posee un número limitado de *CUDA-cores*, lo que limita el número máximo de warps que pueden ejecutar operaciones de manera simultánea. Este parámetro se define como el número de *CUDA-cores* por *Streaming Multiprocessor*.

2.1.5 *Hit latency* Este modelo contempla los efectos del uso de las memorias *cache*, por lo que es necesario calcular las latencias de acceso a las mismas. La estimación se realizará para memoria *cache* L1 y memoria *cache* L2.

2.1.6 *DRAM latency* La memoria DRAM es la memoria principal de la GPU. En algún punto todos los datos de entrada y de salida deben realizar accesos a este tipo de memoria. Ya que este modelo se basa en las operaciones de memoria, es de vital importancia calcular este parámetro.

2.1.7 *Memory peak bandwidth* Este parámetro hace referencia al máximo ancho de banda con el que se puede realizar una transferencia de datos entre la memoria DRAM del *device* y los *Streaming Multiprocessors*. Según el modelo cada *Streaming Multiprocessor* consume el mismo ancho de banda. Para hallar este parámetro se asumirá que el máximo ancho de banda con el que se puede realizar la transferencia de datos será igual al ancho de banda de la memoria DRAM.

Capítulo 3

Herramientas de Análisis

En este capítulo se presentan las herramientas de análisis que se utilizarán en el proceso de extracción de los parámetros y se relacionan con cada uno de estos.

3.1. *NVIDIA profiling tools*

NVIDIA provee una serie de herramientas que se pueden utilizar para entender y optimizar el desempeño de aplicaciones. *Visual Profiler* es una herramienta gráfica de *profiling* (perfilamiento) que muestra mediante una línea de tiempo la actividad de una aplicación, tanto en GPU como en CPU, y que también incluye un motor de análisis para identificar posibles oportunidades de optimización[1].
La herramienta *Visual Profiler* trabaja con base a dos conceptos:

- Eventos: se define como una actividad, acción, u ocurrencia en el *device*. Corresponde a un contador individual en hardware, cuyo valor se toma durante la ejecución del *kernel*[1].
- Métricas: se define como una característica de una aplicación que se calcula a partir de uno a más eventos[1].

En este trabajo de investigación no se utiliza el visual profiler de manera directa, se utilizan eventos y métricas que son extraídos a un archivo de texto para su posterior manipulación y análisis.

3.2. *Paralell Thread Execution ISA (PTX)*

Debido a que CUDA es un lenguaje de programación de alto nivel y para en este trabajo de consideran parámetros de hardware, es necesario trabajar con un lenguaje de programación que esté a un nivel más próximo al hardware. Es en este punto en el que resulta útil trabajar con una aproximación al set de instrucciones de la arquitectura, como lo es el *Paralell Thread Execution ISA (PTX)*. PTX define una máquina virtual de bajo nivel para la ejecución de threads en paralelo *Paralell Thread Execution*

[1] *NVIDIA, Profiler User's Guide v8.0.* 2017. URL: http://www.nvidia.com

y un set de instrucciones de la arquitectura ISA (*Instruction Set Architecture*).[2] Cabe anotar que este no es directamente el set de la arquitectura, sino más bien una abstracción del mismo; sin embargo esto es lo más cercano que se puede estar al mismo, NVIDIA no provee el set de la arquitectura como tal.
Es posible escribir segmentos de código utilizando esta herramienta dentro de *kernel* escritos en CUDA,[3] lo cual facilita el trabajo y el análisis de los *microbenchmarks* que se deban implementar.

3.3. *CUDA binary utilities*

Esta herramienta se utiliza para analizar archivos cubin. Un CUDA *binary* (cubin) es un archivo de formato ELF (*Executable and Linkable Format*) que consiste de segementos ejecutables de código, símbolos, información de depuración, etc.[4] Este tipo de archivos usualmente se encuantran embebidos dentro del código que ejecuta el host, pero pueden ser generados de manera separada para su análisis. Existen dos herramientas:

- cuobjdump: acepta archivos cubin y archivos ejecutables del host en los que se encuentran embebidos, y los presenta en un formato que hace que la información sea comprensible para un ser humano.[5]
- nvdisasm: solo acepta archivos cubin, pero tiene maś opciones para la manipulación de la infomación que cuobjdump, y de la misma manera presenta la información en un formato humanamente comprensible.[6]

Estas herramientas permiten un análisis de código desde una perspectiva del ISA de la arquitectura a partir de un código en CUDA; lo que resulta muy útil al hacer análisis de la ejecución de una aplicación visto desde la arquitectura de la GPU.

3.4. *CUDA Samples*

CUDA provee una serie de ejemplos que se incluyen con el NVIDIA CUDA *Toolkit*[7]. Los ejemplos que se uilizarán se encuentran en la sección de *Utilities Reference*.

- *Device Query*: este ejemplo enumera las propiedades de los CUDA *devices* del sistema[7].
- *Bandwidth Test*: es un programa que mide el ancho de banda de la operación memcpy *memory copy* a través del puerto *PCI-express*. Este ejemplo permite medir en ancho de banda de operaciones de copia de memoria *host to device* y *device to host*[7].

[2] *NVIDIA, CUDA Parallel Thread Execution v7.0.* 2015. URL: http://www.nvidia.com.
[3] *NVIDIA, Inline PTX Assembly in CUDA v8.0.* 2017. URL: http://www.nvidia.com.
[4] *NVIDIA, CUDA Binary Utilities driver v7.0.* 2015. URL: http://www.nvidia.com.
[5] Ibíd.
[6] Ibíd.
[7] *NVIDIA, CUDA Samples v8.0.* 2017. URL: http://www.nvidia.com

El uso de estos ejemplos permite obtener los valores de algunos de los parámetros de manera directa.

3.5. Parámetros y herramientas de análisis

Tabla 2: Se listan los párametros y las herramientas se planean utilizar para su extracción.

Parámetro	**Herramientas**
Core frequency	*CUDA Samples: Device Query*
Warp size	*CUDA Samples: Device Query*
SIMD width	*CUDA Samples: Device Query*
Memory peak andwidth	*CUDA Samples: Band Test*
Average instruction latency	PTX, *CUDA Binary Utilities*
Hit latency	PTX, *CUDA Binary Utilities*, *NVIDIA Profiling Tools*
DRAM latency	PTX, *CUDA Binary Utilities*, *NVIDIA Profiling Tools*

Capítulo 4

Metodología para la extracción de parámetros

Esta metodología se plantea para extraer los parámetros de hardware del modelo MWP-CWP GPUs de NVIDIA programando en CUDA. El esquema de la metodología se presenta el la figura 14.

4.1. Procesos de la metodología

Esta metodología se define en torno al uso de procesos como se puede observar en la figura 14. El conjunto de procesos que se deben utilizar para hallar cada parámetro depende de la complejidad que requiera su extracción. Para determinar qué procesos se deben utilizar se deben contestar las siguientes preguntas:

4.1.1 ¿Se puede extraer directamente el parámetro? Es posible que el valor del parámetro se pueda extraer directamente de documentación ofrecida por el fabricante, o que el fabricante ofrezca alguna herramienta para su extracción. En caso tal que el parámetro se pueda extraer de manera directa de documentación la extracción del mismo se realiza de manera directa. Si el fabricante ofrece alguna herramienta que permita extraer el parámetro se debe responder la siguiente pregunta:

4.1.2 ¿Al trabajar con la herramienta los datos obtenidos requieren algún tratamiento estadístico? Si la herramienta trabaja ejecutando algún código sobre el *hardware* es necesario utilizar la herramienta el número de veces que sea necesario para comprobar que los resultados que se obtienen son consistentes, esto implica realizar un análisis sobre los datos que se obtengan, lo que lleva al siguiente proceso:

4.1.3 Realizar análisis estadístico y extraer parámetros. Es necesario determinar el número de veces que se debe ejecutar la prueba, en la medida que los resultados de la misma se encuentren agrupados se considerará que la validez de los mismos. Esto se puede verificar analizando algunas de las medidas de dispersión de los datos. En el caso que los datos sean válidos se procede a extraer el parámetro. Si los datos no se consideran válidos se debe determinar si existe alguna fuente

de errores, y si existe la posibilidad de realizar algún sesgo sobe los datos. Es caso de no ser posible se debe descartar la herramienta para extraer el parámetro y buscar otra manera de hallar el parámetro. En caso de no poder extraer el parámetro de manera directa se debe implementar un *microbenchmark* siguiendo esta línea de procesos en la metodología:

4.1.4 Definir el contexto. En este proceso se definen las condiciones de tamaño de *grid*, tamaño de bloque y tipo de datos bajo las cuales se ejecutará el *microbenchmark*. Estas condiciones se definen a partir del parámetro que se quiere extraer y su relación con la arquitectura de la GPU. Al definir este contexto se define el número de pruebas que tendrá el *microbenchmark*.

4.1.5 Codificar los programas de host y device. De acuerdo con la estructura de programación para una plataforma de cómputo heterogénea se requiere un código para la CPU y otro que contenga el kernel que se ejecuta en la GPU. En el código de la CPU se definen los valores de entrada del kernel, se definen los espacios de memoria que se requieren en la GPU, se lanza el *kernel*; en caso que se requiera se define algún mecanismo de verificación de los resultados de la GPU y se imprimen los datos de la prueba. El código del *kernel* se implementa utilizando CUDA y PTX *inline*, en este punto la ventaja de usar PTX *inline* es que permite trabajar de cerca con el *hardware*. Se utiliza CUDA para operaciones de declaración de variables auxiliares y copia de resultados dentro del mismo *device*. Utilizando PTX *inline* se realiza el código principal del *kernel* en el que se realizan las mediciones, en esta parte del código se define qué operaciones se van a medir y la manera en al que se realizará la medición.

4.1.6 Definir condiciones de compilación. Debido al uso de PTX *inline*, el uso de librerías, el modelo de memoria y la jerarquía de memoria de la GPU es posible que se requiera ejecutar la prueba utilizando condiciones especiales de compilación que garanticen que el *hardware* se comporta de acuerdo a las condiciones que se asumieron al programar el código del *device*. Estas condiciones se encuentran ligadas a la parte de *hardware* que se esté analizando y hacen especial referencia al número de registros que se utiliza por *thread*, las optimizaciones del compilador y el uso de las memorias *cache*. En este punto el código para cada prueba ya se debe poder compilar y ejecutar.

4.1.7 Verificar el kernel. Una vez compilados y ejecutados los códigos, se analiza el ejecutable generado utilizando las herramientas de manipulación de archivos binarios de CUDA. Se busca verificar que el *kernel* efectivamente está realizando la medición al realizar el análisis del código del ISA de la arquitectura que se obtiene de las herramientas de manipulación de archivos binarios de CUDA.

4.1.8 Determinar si existe costo del registro especial clock. Debido a que las pruebas involucran mediciones de valores de latencia se utiliza el registro especial *clock* para medir tiempo. El registro especial *clock* es un contador de ciclos de reloj, cuyo valor se puede consultar utilizando

la función *clock()* en CUDA o pasando su valor a un registro al utilizar PTX *inline*. Al utilizar este registro como instrumento de medida es posible que exista un costo adicional de tiempo en la medida. Para determinar si este costo existe se realiza una prueba en la que se mida el uso de este registro especial. Si se comprueba que existe este costo se deben crear códigos para cada prueba que permitan medir el mismo. Al realizar la medición de este costo para cada prueba existirán entonces dos códigos independientes: el de medición de latencia y el de costo de uso del registro especial *clock*. Para cada uno de estos códigos se debe crear el mismo entorno de ejecución, de tal forma que para la extracción de cada parámetro existirán dos *microbenchmarks*: uno de medición de latencias y otro de uso del registro especial *clock*. El costo de uso de este registro especial se mide para cada *microbenchmark* por separado.

En este punto se debe determinar si los datos de latencia que se obtienen son suficientes para determinar el valor del parámetro o si es necesario utilizar métricas y eventos de CUDA que permitan la extracción del parámetro. Si no se requiere de métricas y eventos se puede pasar al proceso de codificar y ejecutar los *microbenchmarks*, si se requiere utilizar métricas primero se debe definir cuáles son necesarias.

4.1.9 Definir métricas y eventos necesarios. Las métricas y eventos se definen de acuerdo a las partes de *hardware* que se encuentren involucradas en la extracción de los parámetros. Ya que es posible que se tenga acceso a contadores de eventos para los que no se consolide su valor en una métrica, se toma el valor de estos eventos para crear una nueva métrica.

4.1.10 Codificar y ejecutar los microbenchmarks. Al codificar los *microbenchmarks* se define el número de iteraciones que tendrá cada prueba y la manera en la que se guardarán los datos. En caso de necesitar de eventos y métricas se incluyen las condiciones de ejecución necesarias para su extracción y el almacenamiento de sus valores. Esto se debe hacer tanto para el *microbenchmark* de medición de latencias como para el *microbenchmark* de costo de uso del registro especial *clock*.

4.1.11 Procesar datos de medición de latencia. Se debe estimar el valor de latencia de las operaciones que se miden en cada *microbenchmark*, para esto se debe encontrar la manera de sustraer el costo de uso del registro especial *clock* de las mediciones de latencia, esto se logra restando el costo de uso del registro especial *clock* de las mediciones de latencia obtenidas del *microbenchmark*. Teniendo en cuenta que son variables aleatorias independientes se llevan los datos de medición de latencia y de costo de uso del registro especial *clock* a distribuciones de probabilidad normalizadas y se estima la distribución de probabilidad del resultado utilizando convolución.

En ese punto si de los datos de la estimación de los valores de latencia se pueden extraer los parámetros se puede pasar al proceso de presentación de resultados y extracción de parámetros de lo contrario se debe procesar la información de las métricas y eventos.

4.1.12 Procesar datos de métricas y eventos.

En este proceso se busca obtener los valores de las métricas y eventos que corresponden puntualmente a las operaciones a las que mide latencia. Las métricas y eventos que se obtienen para cada prueba tienen validez en un contexto de ejecución de *kernel*, es decir corresponden a todas las operaciones que se ejecutan dentro del *kernel*. Para estimar el valor de las métricas y eventos de las operaciones a las cuales se mide latencia se utiliza una estrategia similar a la utilizada en el procesamiento de los datos de latencia. Se toman los datos de las métricas y eventos de los *microbenchmarks* de medición de latencia y de costo de uso del registro especial *clock*, se llevan los datos a distribuciones de probabilidad normalizadas y se realiza convolución.

4.1.13 Analizar relación entre datos de latencia y métricas y eventos.

A partir de los datos de las estimaciones de latencia y de las métricas y eventos se realiza un análisis de acuerdo al contexto que se definió para cada *microbenchmark*, de tal manera que se puedan separar los valores de latencia correspondientes a la parte del hardware que se requiera para poder extraer el parámetro.

4.1.14 Presentar resultados y extraer parámetros.

Los resultados se presentan de acuerdo al contexto que se definió para cada *microbenchmark* utilizando gráficas y a partir de los datos de estas gráficas se extrae el valor de cada parámetro.

4.2. Parámetros y metodología.

Al aplicar la metodología a cada parámetro en particular se encuentran diferentes caminos dentro de los procesos de la metodología dependiendo de las herramientas que sean necesarias para realizar la extracción. Los parámetros de *Core frequency*, *Warp size*, *SIMD width* y *Memory peak bandwidth* se extraen de manera directa, solo el parámetro de *Memory peak bandwidth* requiere un análisis estadístico adicional debido a que la herramienta con la que se determina su valor se basa en una aplicación que se ejecuta sobre la GPU.

Para los parámetros de *Average instruction latency*, *Hit latency* y *DRAM latency* es necesario implementar *microbenchmarks*. Los parámetros de *Hit latency* y *DRAM latency* requieren de los procesos de análisis de métricas y eventos. Los parámetros, la manera en la cual se extraen y los análisis que se requieren para su extracción se listan en la tabla 3.

A partir de la metodología y de la manera en la que se plantea su desarrollo para cada parámetro se desarrollan los siguientes capítulos. En el capítulo 5 se aborda el desarrollo de la metodología para los parámetros de *Core frequency*, *Warp size*, *SIMD width* y *Memory peak bandwidth*. También se abordan los procesos iniciales de la metodología para los parámetros que requieren la implementación de *microbenchmarks*. Este capítulo se desarrolla enfocando el desarrollo de la metodología a cada parámetro, con excepción del proceso de estimación de costo del uso del registro especial *clock*, proceso que es común a los parámetros para los que se implementan *microbenchmarks*. Los procesos de la metodología que se desarrollan en el capítulo 5 se pueden observar en la figura 15.

Tabla 3: Se listan los parámetros, la manera en la que se extraen y el tipo de análisis que se requieren para su extracción.

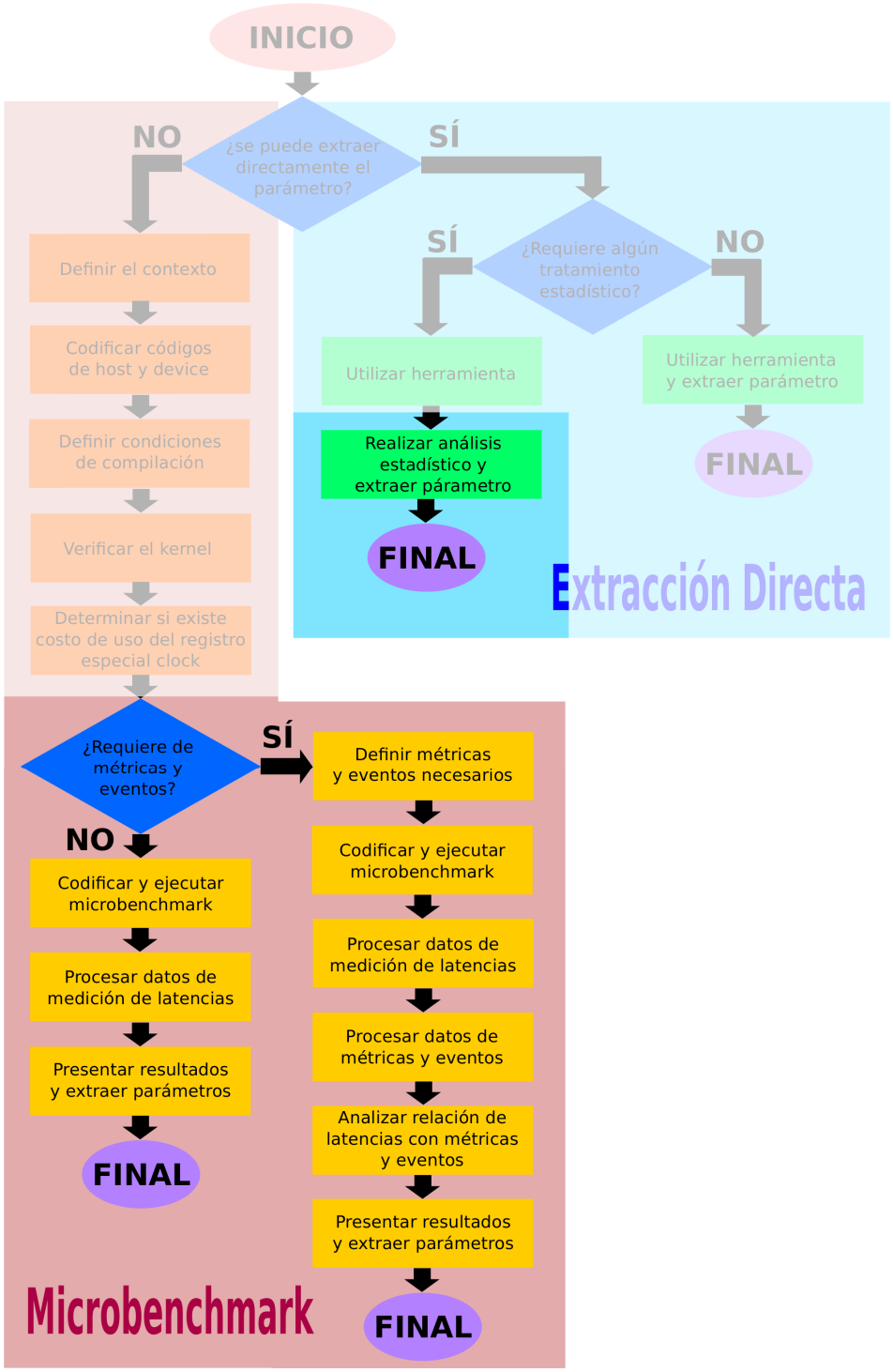
Parámetro	**Extracción**	**Análisis Estadístico**	**Análisis de métricas y eventos**
Core frequency	Directa	NO	N/A
Warp size	Directa	NO	N/A
SIMD width	Directa	NO	N/A
Memory peak andwidth	Directa	SI	N/A
Average instruction latency	*Microbenchmark*	SI	NO
Hit latency	*Microbenchmark*	SI	SI
DRAM latency	*Microbenchmark*	SI	SI

En el capítulo 6 se desarrollan los procesos de análisis para los parámetros en los que se implementaron *microbenchmarks*. Este capítulo se desarrolla por procesos, no por parámetros como se hizo en el capítulo 5. Los procesos de la metodología que se desarrollan en este capítulo se presentan en la figura 16.

Figura 14: Metodología planteada para la extracción de los parámetros. Los procesos de la metodología se pueden agrupar según el parámetro se pueda extraer de manera directa o si se requiere implementar un *microbenchmark* para su extracción.

Figura 15: Procesos de la metodología que se abordan en el capítulo 5.

Figura 16: Procesos de la metodología que se abordan en el capítulo 6.



Capítulo 5

Aplicación de la Metodología I: Programación y Ejecución

La metodología se desarrollará para extraer los parámetros de *hardware* del modelo MWP-CWP para la GPU Tesla K40 de Nvidia con arquitectura Kepler y capacidad de cómputo SM 3.5.
En este capítulo se aborda el desarrollo de la metodología para los parámetros de *Core frequency*, *Warp size*, *SIMD width*. También se presenta el uso de la herramienta con la que se extrae el parámetro de *Memory peak bandwidth*. Finalmente se abordan los procesos iniciales de la metodología para los parámetros que requieren la implementación de *microbenchmarks*. Estos procesos se presentan en este capítulo por parámetro, con excepción del proceso de estimación de costo del uso del registro especial *clock*. El desarrollo del proceso de estimación de costo del uso del registro especial *clock* se presenta de manera común para los parámetros para los que se requiere implementar *microbenchmarks*. Los procesos que se desarrollan en este capítulo se muestran en la figura 15 en el capítulo 4.

5.1. *Core frequency, Warp size, SIMD width*

Estos parámetros se determinan utilizando el ejemplo *deviceQuery* que viene dentro de la herramienta *CUDA Samples*, el cual soporta capacidades de cómputo SM 2.0, SM 3.0, SM 3.2, SM 3.5, SM 3.7, SM 5.0, SM 5.2, SM 5.3, SM 6.0, SM 6.1 y se puede ejecutar sobre sistemas operativos Linux, Windows y OS X.[1] Al utilizar esta herramienta se obtiene un listado de propiedades de la GPU (ver tabla 4).
Al utilizar esta herramienta el parámetro de *core frequency* se toma del valor de *GPU Clock rate*, en este caso 745[MHz]. El parámetro de *warp size* se obtiene directamente de esta herramienta y corresponde a 32[*threads*]. Finalmente el parámetro de *SIMD width* se toma del valor de (192) CUDA Cores/MP que indica el número de *CUDA-cores* por *streaming multiprocessor*.

[1] *NVIDIA, CUDA Samples v8.0*. 2017. URL: http://www.nvidia.com.

Tabla 4: Resultados obtenido al utilizar la herramienta deviceQuery, para una GPU Tesla K40

Device 0: Tesla k40c	
CUDA Driver Version / Runtime Version	6.5 / 6.5
CUDA Capability Major/Minor version number:	3.5
Total amount of global memory:	11520 MBytes (12079136768 bytes)
(15) Multiprocessors, (192) CUDA Cores/MP:	2880 CUDA Cores
GPU Clock rate:	745 MHz (0.75 GHz)
Memory Clock rate:	3004 Mhz
Memory Bus Width:	384-bit
L2 Cache Size:	1572864 bytes
Maximum Texture Dimension Size (x,y,z)	1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
Maximum Layered 1D Texture Size, (num) layers	1D=(16384), 2048 layers
Maximum Layered 2D Texture Size, (num) layers	2D=(16384, 16384), 2048 layers
Total amount of constant memory:	65536 bytes
Total amount of shared memory per block:	49152 bytes
Total number of registers available per block:	65536
Warp size:	32
Maximum number of threads per multiprocessor:	2048
Maximum number of threads per block:	1024
Max dimension size of a thread block (x,y,z):	(1024, 1024, 64)
Max dimension size of a grid size (x,y,z):	(2147483647, 65535, 65535)
Maximum memory pitch:	2147483647 bytes
Texture alignment:	512 bytes
Concurrent copy and kernel execution:	Yes with 2 copy engine(s)
Run time limit on kernels:	No
Integrated GPU sharing Host Memory:	No
Support host page-locked memory mapping:	Yes
Alignment requirement for Surfaces:	Yes
Device has ECC support:	Enabled
Device supports Unified Addressing (UVA):	Yes
Device PCI Bus ID / PCI location ID:	3 / 0
Compute Mode:	

5.2. *Memory peak bandwidth*

Este parámetro se halla utilizando el ejemplo *bandTest* que viene dentro de la herramienta *CUDA Samples*, el cual soporta capacidades de cómputo SM 2.0, SM 3.0, SM 3.2, SM 3.5, SM 3.7, SM 5.0, SM 5.2, SM 5.3, SM 6.0, SM 6.1 y se puede ejecutar sobre sistemas operativos Linux, Windows y OS X.[2] Al utilizar esta herramienta se obtienen las medidas de ancho de banda para movimientos de datos de *host* a *device*, *device* a *host* y *device* a *device* (ver tabla 5). La medida que se considera para hallar el parámetro es la que corresponde a la del movimiento de datos de *device* a *device*. Ya que al utilizar esta herramienta es posible que los valores que se obtengan presenten cierta desviación entre

[2] *NVIDIA, CUDA Samples v8.0.* 2017. URL: http://www.nvidia.com.

prueba y prueba, se plantea realizar la misma prueba un determinado número de veces y analizar los resultados para presentar un valor consolidado del parámetro. El número de iteraciones de la prueba se establecen en 1000 basados en el tiempo de ejecución de la misma y en el tamaño de los datos que se deben procesar. El análisis de estos datos se aborda en el capítulo 6.

Tabla 5: Resultados obtenido al utilizar la herramienta *bandTest*, para una GPU Tesla K40

```
Device 0: Tesla K40c
Quick Mode

Host to Device Bandwidth,       1 Device(s)
PINNED Memory Transfers
Transfer Size (Bytes)           Bandwidth(MB/s)
33554432                        10282.9

Device to Host Bandwidth,       1 Device(s)
PINNED Memory Transfers
Transfer Size (Bytes)           Bandwidth(MB/s)
33554432                        10291.7

Device to Device Bandwidth,     1 Device(s)
PINNED Memory Transfers
Transfer Size (Bytes)           Bandwidth(MB/s)
33554432                        182816.3
```

Hasta el momento se ha presentado la aplicación de la metodología para los parámetros que se pueden extraer de manera directa. En las siguientes secciones se presenta la aplicación de la metodología para los parámetros que requieren ser hallados mediante el uso de *microbenchmarks*.

5.3. *Hit latency: L1*

Basándose en la jerarquía de memoria de la GPU y en el modelo de memoria de CUDA, para estimar los valores de latencia de *hit* en memoria *cache* L1 se toman medidas de latencia de acceso a memoria local. Mediante un análisis de métricas y eventos se hallan los valores de acceso a memoria local que corresponden a *hits* en memoria *cache* L1. En esta sección se presenta la aplicación de los procesos iniciales de la metodología que se utilizan para hallar este parámetro hasta el proceso previo a hallar el costo de uso del registro especial *clock*.

5.3.1 Definir el Contexto. Según la arquitectura de la GPU la memoria *cache* L1 se encuentra definida por *streaming multiprocessor*. Teniendo en cuenta lo anterior, las pruebas se realizan en contexto de bloques ya que los bloques del modelo de memoria de CUDA se ejecutan por *streaming multiprocessor*.

De acuerdo al *deviceQuery* de la GPU se cuenta con 15 *Streaming Multiprocessors*, el máximo número de threads por bloque es de 1024 *threads* y el máximo número de threads por *Streaming Multiprocessors* es de 2048 *threads*. El tamaño del *warp* es de 32 *threads*. Según esta información el máximo número de *threads* que pueden ser ejecutados ocurrirá cuando exista una configuración de 30 bloques por *grid* con 1024 *threads* por bloque. Con esta información, las parámetros de la prueba se plantean de la siguiente manera:

- Se hará un barrido desde 1 hasta 15 bloques, con tamaños de bloque desde 128 *threads* hasta 1024 *threads*(el máximo número de *threads* permitidos por bloque); el barrido del tamaño del bloque se hará en incrementos de 128 *threads* (128 es múltiplo del tamaño del warp, 32 *threads*). De esta manera se busca la ejecución de un bloque por *Streaming Multiprocessor*.
- Se correrá una prueba con 30 bloques, haciendo un barrido en el tamaño del bloque desde 128 hasta 1024 *threads*. Esto garantiza que se ejecutarán 2 bloques por *Streaming Multiprocessor*. Bajo estas condiciones, se alcanza el máximo número de *threads* que se pueden ejecutar por *Streaming Multiprocessor*.

Las operaciones de carga se realizarán con un dato de tipo flotante de 32[bits].

5.3.2 Codificar el código del *host* y del *device*. El *kernel* del código del device debe implementar una parte en *inline* PTX en el que se medirá la latencia de las operaciones de carga. En esta parte del código se debe declarar un espacio de memoria local que cumpla las condiciones para quedar alojado en la memoria DRAM del *device* (que sea mayor que el número de registros disponibles que tiene el *thread* o que al utilizarlo no se conozcan los índices del mismo, si se trata de un arreglo), con el fin de garantizar que al realizar la operación de carga el dato pasa a través toda la jerarquía de memoria de la GPU . Para lograr esto se declaró un arreglo de tamaño 256, cuando se definan las condiciones de compilación se debe limitar el número máximo de registros que puede usar el thread de tal manera que el dato tenga que ser alojado en la memoria DRAM del device.
Se debe generar un valor aleatorio en el código del *host* que se debe pasar como argumento al *kernel* que se ejecutará en el *device*, el cual a su vez será guardado en el primer espacio de memoria local que anteriormente se ha declarado. Se realizan tres operaciones de carga del dato de tal manera que se aumente la probabilidad de encontrar el dato en memoria *cache* L1. Se debe utilizar el modificador `ca`en la operación de carga para que utilice memoria *cache* a través de toda la jerarquía de memoria de la GPU.[3]
Se debe definir registros que guarden el valor del registro especial *clock* antes y después de realizar cada operación de carga y así medir la latencia al hacer la resta de los dos valores que se obtenga. Una vez hechas las cargas de los datos se realiza la suma entre los tres valores, este valor se pasa al *host*

[3] *NVIDIA, CUDA Parallel Thread Execution v7.0.* 2015. URL: `http://www.nvidia.com`.

para verificar que la operación de carga se realizó correctamente, así se valida de manera indirecta las cargas del dato. A continuación se presenta un extracto del código en PTX *inline* que se utilizó en el *kernel*.

```
1  ".local.f32 cosa[256];            \n\t" // Declaring array cosa (local.memory)
2  ".reg.u32 %index;                 \n\t" // index register declaration
3  ".reg.f32 %aux;                   \n\t" // Aux. register to save the value
4  "mov.f32 %aux, %15;               \n\t" // aux = a, initial value
5  "mov.u32 %index, cosa[0];         \n\t" // index = address(cosa[0])
6  "st.local.cs.f32 [%index], %aux;  \n\t" // cosa[0]=aux
7  ".reg.f32 %1st;                   \n\t" // Aux. register to save the first load
8  ".reg.f32 %2nd;                   \n\t" // Aux. register to save the second load
9  ".reg.f32 %3rd;                   \n\t" // Aux. register to save the third load
10 ".reg.f32 %sum;                   \n\t" // Aux. register to store de first oper.
11 ".reg.f32 %sum2;                  \n\t" // Aux. register to store the second oper.
12 "bar.sync 0;                      \n\t" // Synchronization barrier
13 "mov.u32 %2, %%clock;             \n\t" // Start timer for the first load
14 "ld.local.ca.f32 %1st, [%index];  \n\t" // First load
15 "mov.u32 %3, %%clock;             \n\t" // Ending timer for the first load
16 "bar.sync 0;                      \n\t" // Synchronization barrier
17 "mov.u32 %4, %%clock;             \n\t" // Start timer for the second load
18 "ld.local.ca.f32 %2nd, [%index];  \n\t" // Second load
19 "mov.u32 %5, %%clock;             \n\t" // Ending timerfor the second load
20 "bar.sync 0;                      \n\t" // Synchronization barrier
21 "mov.u32 %6, %%clock;             \n\t" // Start timer for the third load
22 "ld.local.ca.f32 %3rd, [%index];  \n\t" // Third load
23 "mov.u32 %7, %%clock;             \n\t" // Ending timer for the third load
24 "bar.sync 0;                      \n\t" // Synchronization barrier
25 "add.f32 %sum, %1st, %2nd;        \n\t" // First oper. sum=a+b
26 "add.f32 %sum2, %3rd, %sum;       \n\t" // Second oper. sum2=(a+b)+c
```

Extracto del código PTX inline con el que se realiza la medición de la latencia de las operaciones de carga de un dato en memoria local.

5.3.3 Definir las condiciones de compilación Se debe limitar el número de registros que puede utilizar un *thread*, para esto se puede utilizar la bandera -maxreregcount. En este caso se dejará esta bandera como -maxreregcount=32. Debido a que el compilador aplica optimizaciones que no se pueden controlar estas se deben desactivar utilizando la bandera -O. El nivel de optimizaciones debe estar en cero para garantizar que el uso de registros especiales se realice en la parte del código en la

que se utilizan, para esto la bandera de optimizaciones se define como -O0. Una vez definidas estas condiciones, se procede a compilar y ejecutar el código.

5.3.4 Verificar el *kernel* de los *microbenchmarks* A partir de los archivos cubin y con ayuda de las herramientas de manipulación de archivos binarios de CUDA se debe llevar el archivo binario del kernel a su versión en PTX, esta operación se denomina "desensamblar"(*disassemble*). Se debe analizar la parte en la cual se realizan las operaciones de carga de memoria con el fin de verificar que las condiciones en las que se está midiendo la latencia.

Figura 17: Segmento de código en PTX *inline* del *kernel*. En este segmento de código se puede observar las operaciones de carga de memoria local y el uso del registro especial *clock* para realizar la medida de latencia.

```
BAR.SYNC 0x0;
S2R R19, SR_CLOCKLO;
MOV R26, R19;
MOV R19, R21;
LDL R19, [R19];
MOV R22, R19;
S2R R19, SR_CLOCKLO;
MOV R25, R19;
BAR.SYNC 0x0;
S2R R19, SR_CLOCKLO;
MOV R24, R19;
MOV R19, R21;
LDL R19, [R19];
MOV R27, R19;
S2R R19, SR_CLOCKLO;
MOV R19, R19;
BAR.SYNC 0x0;
S2R R20, SR_CLOCKLO;
MOV R20, R20;
MOV R21, R21;
LDL R21, [R21];
MOV R28, R21;
S2R R21, SR_CLOCKLO;
MOV R21, R21;
BAR.SYNC 0x0;
```

Al analizar el código de la figura 17 y realizar una comparación con el PTX *inline* del kernel, se puede observar que existen tres operaciones adicionales de movimiento de registros (mov) al realizar la consulta del registro especial *clock*. Esto se debe a que el ISA que provee PTX es una abstracción del verdadero set de instrucciones de la arquitectura, sobre el que no se tiene control; además el compilador realiza rutinas de optimización sobre las cuales tampoco se posee el control. Estos movimientos entre registros bien pueden ser inherentes a la operación de carga del dato o al uso del registro especial *clock*.

5.3.5 Determinar las métricas y eventos de nvprof que se requieran Debido a que se está trabajando con memoria local, y está se trabaja a través de toda la jerarquía de memoria de

la GPU, se deben incluir métricas y eventos de nvprof que permitan estimar el uso de las memorias *cache* L1 y L2, DRAM, así como métricas propias del uso de la memoria local (Ver tabla 6).

5.3.6 Codificar y ejecutar los *microbenchmarks* El número de iteraciones para cada prueba se define en cien basándose en el tiempo que se demora en correr el *microbenchmark* (alrededor de 72 horas) y la cantidad de datos resultantes que se deben procesar (4[GB] aprox.). El *microbenchmark* debe ser capaz de guardar la información de la medida de la latencia y de los resultados obtenidos de las métricas y eventos por separado para su posterior procesamiento.

5.4. *Hit latency*: L2 y DRAM *latency*

De acuerdo al modelo de memoria de CUDA y a la jerarquía de memoria de la GPU al trabajar con memoria global los datos se guardan físicamente en la memoria DRAM de la GPU. Con esta información, y teniendo en cuenta que para los *streaming multiprocessors* de la arquitectura Kepler la memoria *cache* L1 no se utiliza para operaciones que impliquen el uso de memoria global, se mide la latencia de acceso a memoria global. Una vez se tienen estos valores de latencia se realiza un análisis de métricas y eventos sobre los mismos de tal forma que se puedan separar los valores correspondientes a *hit* en memoria *cache* L2 y accesos a DRAM. En esta sección se presenta la aplicación de los procesos iniciales de la metodología que se utilizan para hallar estos parámetro hasta el proceso previo a determinar el costo de uso del registro especial *clock*.

5.4.1 Definir el contexto La memoria *cache* L2 y la memoria DRAM se encuentran definidas en un contexto de *device*, luego los parámetros del *microbenchmark* se definen para que los resultados del mismo permitan realizar un análisis en un contexto de *device*. Con esto en mente los parámetros del *microbenchmark* se definen de la siguiente manera:

- Se hará un barrido desde 1 hasta 30 bloques, con tamaños de bloque desde 128 *threads* hasta 1024 *threads*(el máximo número de *threads*permitido por bloque); el barrido del tamaño del bloque se hará en incrementos de 128 *threads* (128 es múltiplo del tamaño del warp, 32 *threads*). De esta manera se garantiza que se realiza un barrido de tamaño de bloque y de *grid* un contexto de *device*.

Las operaciones de carga se realizarán con un dato de tipo flotante de 32[bits]. Si bien los parámetros de esta prueba están en función del tamaño de *grid* y bloque, el análisis se realiza en un contexto global por número de threads.

5.4.2 Codificar el código del *host* y del *device* El *kernel* del código del device debe implementar una parte en *inline* PTX que con el que se medirá la latencia de las operaciones de carga. En esta parte del código se debe declarar un espacio de memoria global, de tal manera que el dato

Tabla 6: Métricas y Eventos para el análisis de latencia de memoria local.

Evento o Métrica	Descripción
fb_subp0_read_sectors:	Number of DRAM read requests to sub partition 0, increments by 1 for 32 byte access.
fb_subp1_read_sectors:	Number of DRAM read requests to sub partition 1, increments by 1 for 32 byte access.
l2_subp0_read_sector_misses:	Number of read misses in slice 0 of L2 cache. This increments by 1 for each 32-byte access.
l2_subp1_read_sector_misses:	Number of read misses in slice 1 of L2 cache. This increments by 1 for each 32-byte access.
l2_subp2_read_sector_misses:	Number of read misses in slice 2 of L2 cache. This increments by 1 for each 32-byte access.
l2_subp3_read_sector_misses:	Number of read misses in slice 3 of L2 cache. This increments by 1 for each 32-byte access.
l2_subp0_read_l1_sector_queries:	Number of read requests from L1 to slice 0 of L2 cache. This increments by 1 for each 32-byte access.
l2_subp1_read_l1_sector_queries:	Number of read requests from L1 to slice 1 of L2 cache. This increments by 1 for each 32-byte access.
l2_subp2_read_l1_sector_queries:	Number of read requests from L1 to slice 2 of L2 cache. This increments by 1 for each 32-byte access.
l2_subp3_read_l1_sector_queries:	Number of read requests from L1 to slice 3 of L2 cache. This increments by 1 for each 32-byte access.
l2_subp0_read_l1_hit_sectors:	Number of read requests from L1 that hit in slice 0 of L2 cache. This increments by 1 for each 32-byte access.
l2_subp1_read_l1_hit_sectors:	Number of read requests from L1 that hit in slice 1 of L2 cache. This increments by 1 for each 32-byte access.
l2_subp2_read_l1_hit_sectors:	Number of read requests from L1 that hit in slice 2 of L2 cache. This increments by 1 for each 32-byte access.
l2_subp3_read_l1_hit_sectors:	Number of read requests from L1 that hit in slice 3 of L2 cache. This increments by 1 for each 32-byte access.
local_load:	Number of executed load instructions where state space is specified as local, increments per warp on a multiprocessor.
l1_local_load_hit:	Number of cache lines that hit in L1 cache for local memory load accesses. In case of perfect coalescing this increments by 1,2, and 4 for 32, 64 and 128 bit accesses by a warp respectively.
l1_local_load_miss:	Number of cache lines that miss in L1 cache for local memory load accesses. In case of perfect coalescing this increments by 1,2, and 4 for 32, 64 and 128 bit accesses by a warp respectively.

Fuente: Adaptado de *NVIDIA, Profiler User's Guide v8.0*. 2017. URL: `http://www.nvidia.com`.

quede almacenado en la memoria DRAM del *device* y al realizar la operación de carga el dato quede en la memoria *cache* L2.

Se debe generar un valor aleatorio en el código del *host* que se debe pasar como argumento al *kernel* que se ejecutara en el *device*, el cual a su vez será guardado en el espacio de memoria global que anteriormente se ha declarado. Se definen tres operaciones de carga del dato desde memoria global con el fin de aumentar la probabilidad de encontrar el dato en la memoria *cache* L2. Al definir las operaciones de carga se debe utilizar el modificador `cg`, para que utilice memoria *cache* a nivel global.[4] Se debe definir registros que guarden el valor del registro especial *clock* antes y después de realizar las operaciones de carga y así medir la latencia, al hacer la resta de los dos valores que se obtengan. Una vez hechas las cargas de los datos se realiza la suma entre los tres valores, este valor se debe pasar al *host* para ser comparado y verificar que las cargas se realizaron correctamente. A continuación se presenta un segmento del código en PTX inline en el que se realizan las operaciones de carga del dato y la medición de la latencia.

```
1  ".global.f32 cosa;                \n\t" // Declaring variable cosa (global.memory)
2  ".global.f32 suma;                \n\t" // Declaring variable suma (global.memory)
3  "st.global.wt.f32 [cosa], %15;    \n\t" // Storing the input value (global.memory)
4  ".reg.f32 %1st;                   \n\t" // Temporary register
5  ".reg.f32 %2nd;                   \n\t" // Temporary register
6  ".reg.f32 %3rd;                   \n\t" // Temporary register
7  ".reg.f32 %sum;                   \n\t" // Temporary register
8  ".reg.f32 %sum2;                  \n\t" // Temporary register
9  "bar.sync 0;                      \n\t" // Synchronization barrier
10 "mov.u32 %2, %%clock;             \n\t" // Starting timing for the first load
11 "ld.global.cg.f32 %1st, [cosa];   \n\t" // First load
12 "mov.u32 %3, %%clock;             \n\t" // Ending timing for the first load
13 "bar.sync 0;                      \n\t" // Synchronization barrier
14 "mov.u32 %4, %%clock;             \n\t" // Starting timing for the second load
15 "ld.global.cg.f32 %2nd, [cosa];   \n\t" // Second load
16 "mov.u32 %5, %%clock;             \n\t" // Ending timing for the second load
17 "bar.sync 0;                      \n\t" // Synchronization barrier
18 "mov.u32 %6, %%clock;             \n\t" // Starting timing for the third load
19 "ld.global.cg.f32 %3rd, [cosa];   \n\t" // Third load
20 "mov.u32 %7, %%clock;             \n\t" // Ending timing for the third load
21 "bar.sync 0;                      \n\t" // Synchronization barrier
22 "add.f32 %sum, %1st, %2nd;        \n\t" // sum1=(a+b)
23 "add.f32 %sum2, %3rd, %sum;       \n\t" // sum2=(a+b)+c
```

[4] *NVIDIA, CUDA Parallel Thread Execution v7.0.* 2015. URL: http://www.nvidia.com.

Extracto del código PTX inline con el que se realiza la medición de la latencia de las operaciones de carga

5.4.3 Definir las condiciones de compilación Debido a que el compilador aplica optimizaciones que no se pueden controlar, estas se deben desactivar utilizando la bandera `-O`.El nivel de optimizaciones debe estar en cero, en este caso la bandera de optimizaciones quedaría definida como `-O0`.Las optimizaciones se desactivan para garantizar que los registros especiales se utilizan en las partes del código en las que se definen.

5.4.4 Verificar el *kernel* de los *microbenchmarks* A partir de los archivos cubin y con ayuda de las herramientas binarias de CUDA se debe llevar el archivo binario del *kernel* a su versión en PTX. Se debe analizar la parte en la cual se realizan las operaciones de carga de memoria con el fin de verificar que se está realizando la medición de la latencia.

Figura 18: Segmento de código en PTX *inline* del *kernel*. En este segmento de código se puede observar las operaciones de carga de memoria local y el uso del registro especial *clock* para realizar la medida de latencia.

```
BAR.SYNC 0x0;
S2R R14, SR_CLOCKLO;
MOV R22, R14;
MOV32I R14, 32@lo($__cosa__576);
MOV32I R15, 32@hi($__cosa__576);
MOV R14, R14;
MOV R15, R15;
LD.E.CG R14, [R14];
MOV R26, R14;
S2R R14, SR_CLOCKLO;
MOV R23, R14;
BAR.SYNC 0x0;
S2R R14, SR_CLOCKLO;
MOV R21, R14;
MOV32I R14, 32@lo($__cosa__576);
MOV32I R15, 32@hi($__cosa__576);
MOV R14, R14;
MOV R15, R15;
LD.E.CG R14, [R14];
MOV R27, R14;
S2R R14, SR_CLOCKLO;
MOV R24, R14;
BAR.SYNC 0x0;
S2R R14, SR_CLOCKLO;
MOV R25, R14;
MOV32I R14, 32@lo($__cosa__576);
MOV32I R15, 32@hi($__cosa__576);
MOV R14, R14;
MOV R15, R15;
LD.E.CG R14, [R14];
MOV R15, R14;
S2R R14, SR_CLOCKLO;
MOV R14, R14;
BAR.SYNC 0x0;
```

Al analizar el código de la figura 18 y realizar una comparación con el PTX *inline* del *kernel*, se puede observar que existen operaciones adicionales a las del código PTX desensamblado. Estas operaciones adicionales a la operación de carga como tal, tal vez se deban a la naturaleza del espacio de memoria en la que se encuentra almacenado el dato y al uso del registro especial *clock*. Para eliminar el efecto del uso del registro especial *clock* se utiliza la estrategia de las dos pruebas planteadas en el costo del uso del registro especial *clock*.

Tabla 7: Métricas y eventos para el análisis de la latencia de memoria global.

Evento o Métrica	Descripción
fb_subp0_read_sectors:	Number of DRAM read requests to sub partition 0, increments by 1 for 32 byte access.
fb_subp1_read_sectors:	Number of DRAM read requests to sub partition 1, increments by 1 for 32 byte access.
l2_subp0_read_sector_misses:	Number of read misses in slice 0 of L2 cache. This increments by 1 for each 32-byte access.
l2_subp1_read_sector_misses	Number of read misses in slice 1 of L2 cache. This increments by 1 for each 32-byte access.
l2_subp2_read_sector_misses	Number of read misses in slice 2 of L2 cache. This increments by 1 for each 32-byte access.
l2_subp3_read_sector_misses	Number of read misses in slice 3 of L2 cache. This increments by 1 for each 32-byte access.
l2_subp0_read_l1_hit_sectors:	Number of read requests from L1 that hit in slice 0 of L2 cache. This increments by 1 for each 32-byte access.
l2_subp1_read_l1_hit_sectors:	Number of read requests from L1 that hit in slice 1 of L2 cache. This increments by 1 for each 32-byte access.
l2_subp2_read_l1_hit_sectors:	Number of read requests from L1 that hit in slice 2 of L2 cache. This increments by 1 for each 32-byte access.
l2_subp3_read_l1_hit_sectors:	Number of read requests from L1 that hit in slice 3 of L2 cache. This increments by 1 for each 32-byte access.
gld_inst_32bit:	Total number of 32-bit global load instructions that are executed by all the threads across all thread blocks.
gld_inst_64bit:	Total number of 64-bit global load instructions that are executed by all the threads across all thread blocks.
gld_inst_128bit:	Total number of 128-bit global load instructions that are executed by all the threads across all thread blocks.
l2_read_transactions:	Memory read transactions seen at L2 cache for all read requests.
dram_read_transactions:	Device memory read transactions.
global_replay_overhead:	Average number of replays due to global memory cache misses for each instruction executed.
global_cache_replay_overhead:	Average number of replays due to global memory cache misses for each instruction executed.
gld_transactions_per_request:	Average number of global memory load transactions performed for each global memory load

Fuente: Adaptado de *NVIDIA, Profiler User's Guide v8.0.* 2017. URL: `http://www.nvidia.com`.

5.4.5 Determinar las métricas y eventos de nvprof que se requieran Debido a que se está trabajando con memoria global y en la jerarquía de memoria de la GPU esta trabaja a través de memoria *cache* L2 y DRAM, se deben incluir métricas y eventos de nvprof que permitan estimar el uso de las memorias *cache*L2 y DRAM, así como métricas propias del uso de la memoria global. Los eventos y métricas que se utilizarán en esta prueba se listan en la tabla 7.

5.4.6 Codificar y ejecutar los *microbenchmarks* De manera similar que para el *microbenchmark* de medición de latencia de memoria local, el número de iteraciones para cada prueba se define en cien basándose en el tiempo que se demora en correr el *microbenchmark* (alrededor de 72 horas) y la cantidad de datos resultantes que se deben procesar (12[GB] aprox.). El *microbenchmark* debe ser capaz de guardar la información de la medida de la latencia y de los resultados obtenidos de las métricas y eventos por separado para su posterior procesamiento.

5.5. *Average instruction latency*

Para hallar este parámetro se realizarán mediciones de latencia de operaciones de cómputo para datos de tipo punto flotante de 32[bits]. Las operaciones que a las cuales se medirá la latencia son adición, sustracción, multiplicación, división y multiplicación-adición fusionadas. En esta sección se abordan los procesos de la metodología que se utilizan para hallar estos parámetros hasta el proceso previo a determinar el costo de uso del registro especial *clock*.

5.5.1 Definir el contexto Se asume que la latencia de las operaciones de cómputo dependen de los *CUDA-cores*. Los valores de las mediciones de latencia deberían ser independientes del número de bloques y de *threads* con los que se realice la prueba. Basándose en lo anterior los parámetros de las pruebas del *microbenchmark* se definen de la siguiente manera:

- Se hará un prueba con un tamaño de *grid* de un bloque, con tamaños de bloque desde 32 *threads* hasta 1024 *threads* (el máximo número de *threads*permitido por bloque); el barrido del tamaño del bloque se hará en incrementos de 32 *threads* (32 es múltiplo del tamaño del warp, 32 *threads*).De esta manera se garantiza que se ejecutará un bloque en un *Streaming Multiprocessor*.
- Se hará un prueba con un tamaño de *grid* de quince bloques, con tamaños de bloque desde 32 *threads* hasta 1024 *threads* (el máximo número de *threads*permitido por bloque); el barrido del tamaño del bloque se hará en incrementos de 32 *threads* (32 es múltiplo del tamaño del warp, 32 *threads*).De esta manera se garantiza que se ejecutará un bloque en cada *Streaming Multiprocessor*.
- Se hará un prueba con un tamaño de *grid* de treinta bloques, con tamaños de bloque desde 32 *threads* hasta 1024 *threads* (el máximo número de *threads*permitido por bloque); el barrido

del tamaño del bloque se hará en incrementos de 32 *threads* (32 es múltiplo del tamaño del warp, 32 *threads*).De esta manera se garantiza que se ejecutarán dos bloques en cada *Streaming Multiprocessor*.

Las operaciones se realizarán con un dato de tipo flotante de 32[bits]. Las operaciones que se tomarán en cuenta para la prueba son adición, resta, multiplicación, división y multiplicación-adición fusionadas, Se efectuarán operaciones sucesivas sin dependencia serial de datos, desde una hasta ocho operaciones.

5.5.2 Codificar el código del *host* y del *device* El *kernel* del código del *device* debe implementar una parte en *inline* PTX que con el que se medirá la latencia de las operaciones. Al definir la operación de división se debe utilizar el modificador de redondeo `rn`en la operación. En el caso de la operación de multiplicación- adición fusionadas se debe utilizar el mismo modificador de redondeo.

Se deben generar valores aleatorios en el código del *host* que se debe pasar como argumentos al *kernel* que se ejecutara en el *device*. Se debe definir registros que guarden el valor del registro especial *clock* antes y después de realizar las operaciones y así medir la latencia, al hacer la resta de los dos valores que se obtengan. Una vez hechas las cargas realizadas las operaciones el datos resultante se debe pasar al *host* para ser comparado con el valor de las mismas operaciones en CPU y así puede validar la prueba. Debido a las diferencias de hardware entre GPU y CPU se debe definir un porcentaje de aceptación de error en las operaciones que realizan redondeo, división y multiplicación-adición fusionadas, en este caso se definirá un porcentaje de menos del 0.001 %. Hecho lo anterior se puede hacer un *kernel* que permita realizar las operaciones de cálculo de latencia por*thread*; estos datos se deben pasar al *host*, el cual en su código los debe imprimir. Se debe tener especial precaución al copiar los datos de vuelta de GPU a CPU, ya que se deben copiar la cantidad de operaciones que se estén realizando por cada *thread*. A continuación se presenta un segmento de código en PTX *inline* en el que se mide la latencia de la operación de multiplicación-adición fusionadas.

```
1 "bar.sync 0;                    \n\t" // Synchronization barrier
2 "mov.u32 %0, %%clock;           \n\t" // Start cycle
3 "mad.rn.f32 %2, %10, %11, %12;  \n\t" // d0 = (a0 * b0) + c0
4 "mad.rn.f32 %3, %13, %14, %15;  \n\t" // d1 = (a1 * b1) + c1
5 "mad.rn.f32 %4, %16, %17, %18;  \n\t" // d2 = (a2 * b2) + c2
6 "mov.u32 %1, %%clock;           \n\t" // Stop cycle
7 "bar.sync 0;                    \n\t" // Synchronization barrier
```

Extracto del código PTX inline con el que se realiza la medición de la latencia de las operaciones, en este caso multiplicación-adición fusionadas

Figura 19: Segmento del código PTX de un *kernel* sobre el que se toman las medidas de latencia en este caso para la división con 8 operaciones

```
MOV R30, R4;
BAR.SYNC 0x0;
S2R R4, SR_CLOCKLO;
MOV R31, R4;
MOV R4, R3;
MOV R5, R0;
CAL `($_Z20div_f32_lat_ubmk_knlPfS_S_PiS0_S0_S0_S0_S0_S0_S0_S0_$__cuda_sm20_div_rn_f32);
MOV R0, R4;
MOV R3, R0;
STL [R1+0x54], R3;
LDL R0, [R1+0x34];
MOV R4, R0;
LDL R0, [R1+0x38];
MOV R5, R0;
CAL `($_Z20div_f32_lat_ubmk_knlPfS_S_PiS0_S0_S0_S0_S0_S0_S0_S0_$__cuda_sm20_div_rn_f32);
MOV R0, R4;
MOV R3, R0;
STL [R1+0x58], R3;
LDL R0, [R1+0x3c];
MOV R4, R0;
LDL R0, [R1+0x40];
MOV R5, R0;
CAL `($_Z20div_f32_lat_ubmk_knlPfS_S_PiS0_S0_S0_S0_S0_S0_S0_S0_$__cuda_sm20_div_rn_f32);
MOV R0, R4;
MOV R3, R0;
STL [R1+0x5c], R3;
LDL R0, [R1+0x44];
MOV R4, R0;
LDL R0, [R1+0x48];
MOV R5, R0;
CAL `($_Z20div_f32_lat_ubmk_knlPfS_S_PiS0_S0_S0_S0_S0_S0_S0_S0_$__cuda_sm20_div_rn_f32);
MOV R0, R4;
MOV R3, R0;
STL [R1+0x60], R3;
LDL R0, [R1+0x4c];
MOV R4, R0;
LDL R0, [R1+0x50];
MOV R5, R0;
CAL `($_Z20div_f32_lat_ubmk_knlPfS_S_PiS0_S0_S0_S0_S0_S0_S0_S0_$__cuda_sm20_div_rn_f32);
MOV R0, R4;
MOV R0, R0;
STL [R1+0x64], R0;
MOV R4, R20;
MOV R5, R21;
CAL `($_Z20div_f32_lat_ubmk_knlPfS_S_PiS0_S0_S0_S0_S0_S0_S0_S0_$__cuda_sm20_div_rn_f32);
MOV R0, R4;
MOV R20, R0;
MOV R4, R22;
MOV R5, R23;
CAL `($_Z20div_f32_lat_ubmk_knlPfS_S_PiS0_S0_S0_S0_S0_S0_S0_S0_$__cuda_sm20_div_rn_f32);
MOV R0, R4;
MOV R21, R0;
MOV R4, R24;
MOV R5, R25;
CAL `($_Z20div_f32_lat_ubmk_knlPfS_S_PiS0_S0_S0_S0_S0_S0_S0_S0_$__cuda_sm20_div_rn_f32);
MOV R0, R4;
MOV R0, R0;
S2R R3, SR_CLOCKLO;
MOV R3, R3;
BAR.SYNC 0x0;
```

5.5.3 Verificar el *kernel* de los *microbenchmarks*

A partir de los archivos cubin y con ayuda de las herramientas binarias de CUDA se debe llevar el archivo binario del *kernel* a su versión en PTX. Se debe analizar la parte en la cual se realizan las operaciones con el fin de verificar que las condiciones en las que se está midiendo la latencia.

Al analizar el código de la figura 19 y realizar una comparación con el PTX *inline* del *kernel*, se puede observar que existen operaciones adicionales a las del código PTX desensamblado. Estas operaciones adicionales a la operaciones con las que se plantea la prueba en el PTX *inline*, tal vez se deban a la naturaleza de la operación; lo que sí se debe verificar en este caso es que se están realizando el número de operaciones con los que se debe ejecutar la prueba.

5.5.4 Codificar y ejecutar los *microbenchmarks*.

El número de iteraciones se define en cincuenta basados en la cantidad de datos que se obtienen por prueba, esta cantidad de datos varía de acuerdo al número de *threads* que tenga la prueba y al número de operaciones que se deban verificar en la misma. Del *kernel* de medida de latencias se reciben los resultados de las operaciones. Estos valores se debe comparar con el resultado de las mismas operaciones en CPU para de esta manera validar la prueba. Las condiciones de ejecución para el tamaño de bloque y *grid* se encuentran definidas en el contexto. El *microbenchmark* debe ser capaz de guardar la información de la medida de la latencia para su posterior procesamiento.

5.6. Costo del uso del registro especial *clock*

Debido a que se deben medir latencias entre operaciones se utiliza el registro especial *clock*. Este registro es un contador de ciclos de reloj de 32 bits sin signo.[5] Para comprobar su uso se puede crear un *kernel* que utilice PTX *inline* y que haga dos consultas sucesivas de este registro especial como se puede observar en el siguiente extracto de código en PTX *inline*.

```
1 "bar.sync 0;                    \n\t" // Synchronization barrier
2 "mov.u32 %2, %%clock;           \n\t" // Starting timer
3 "mov.u32 %3, %%clock;           \n\t" // Ending timer
4 "bar.sync 0;                    \n\t" // Synchronization barrier
```

Extracto del código PTX inline con el que se desea comprobar la existencia de costo en el uso del registro especial clock

Al compilar el código del *kernel* y desensamblar los archivos binarios del mismo utilizando las herramientas de CUDA *Binary Utilities* se observa que existe un movimiento entre registros (`mov`) adicional a la consulta del registro especial como se observa en la figura 20). De manera similar al consultar el valor de latencia obtenido al ejecutar este *kernel* se observa que los valores son mayores a cero, lo que implica que al utilizar este registro especial para medir la latencia se tiene una distorsión en la medida.

[5] *NVIDIA, CUDA Parallel Thread Execution v7.0.* 2015. URL: http://www.nvidia.com.

Figura 20: Segmento de código PTX del *kernel* de medida de costo del registro especial *clock*.

```
BAR.SYNC 0x0;
S2R R20, SR_CLOCKLO;
MOV R26, R20;
S2R R20, SR_CLOCKLO;
MOV R25, R20;
BAR.SYNC 0x0;
```

Ya que es posible medir esta distorsión y se tienen los *microbenchmarks* de medición de latencia para cada parámetro es posible estimar los valores de esta distorsión removiendo las operaciones sobre las cuales se mide la latencia en cada *kernel* y volviendo a ejecutar los *microbenchmarks* con estas modificaciones. Los resultados de esta prueba se deben guardar por separado de los de medición de latencia de las operaciones de uso de memoria y operaciones de cómputo. Esto se debe realizar para los *microbenchmarks* de medición de latencia de operaciones ariméticas, memoria local y memoria global. El objetivo de guardar los valores de costo del registro especial *clock* es poder restar la distorsión que este produce en las mediciones de latencia utilizando convolución basados en el hecho que el uso del registro especial *clock* y las operaciones sobre las que se mide la latencia son variables aleatorias independientes.

Capítulo 6

Aplicación de la Metodología II: Análisis de datos y Presentación de resultados

En este capítulo se presenta la manera en la que se estimó estadísticamente el valor de latencia para las operaciones ariméticas, los accesos a memoria local y los accesos a memoria global. Adicionalmente se presenta el análisis de métricas que se realizó para las pruebas en las que se utilizó memoria local y memoria global con el fin de extraer los parámetros de latencia de *hit* en memoria *cache* L1, *hit* en memoria *cache* L2 y accesos a DRAM. Finalmente se presentan los resultados obtenidos y se extraen los parámetros. Este capítulo tiene un enfoque por procesos, los procesos que se tratan en este capítulo se pueden observar en la figura 16 del capítulo 4.

6.1. Procesar datos de medición de latencias

El procesamiento de los datos obtenidos para la medición de latencias se realiza de la misma manera para los resultados de las pruebas de operaciones ariméticas, accesos a memoria local y accesos a memoria global.

Este procesamiento de datos se realiza partiendo de que se tienen dos variables aleatorias independientes como lo son: los datos obtenidos de la medición hecha en los *microbenchmarks* (operaciones ariméticas, accesos a memoria local y accesos a memoria global) y la medición de costo de uso del registro especial *clock*. Se debe asumir que las mediciones hechas en los *microbenchmarks* son la suma de la medición de latencia y el costo de uso del registro especial *clock*. En términos de variables aleatorias:

$$Z = X + Y \tag{6.1}$$

En dónde Z representa los datos obtenidos de medición hecha en los *microbenchmarks*, X representa la medida de latencia de las operaciones y Y el costo de uso del registro especial *clock*. La variable aleatoria que se debe hallar es X, entonces:

$$X = Z - Y \tag{6.2}$$

Si se tienen las distribuciones de densidad de probabilidad de Z y Y, se puede estimar la distribución de densidad de probabilidad de X utilizando convolución:

$$f_X(x) = f_Z(z) * f_{-Y}(y) = f_Z(z) * f_Y(-y) \tag{6.3}$$

Los datos de las mediciones hechas en los *microbenchmarks* y de costo de uso del registro especial *clock* deben ser procesados como se observa en la figura 21.
Al ejecutar los *microbenchmarks* la información que se obtuvo de estos se guardó en archivos de texto para cada prueba. Los datos de cada prueba son las mediciones hechas en cada *microbenchmarks* y las mediciones de costo de uso del registro especial *clock*. El tratamiento se realizó utilizando los siguientes procesos:

- Vectorizar los datos: Es necesario llevar la información de los archivos de texto a vectores de tal forma que el procesamiento de los datos se pueda realizar utilizando alguna rutina escrita en algún lenguaje de programación. Los vectores tendrán un tamaño que depende de la prueba, del número de iteraciones y del número de mediciones que se realizó en cada prueba.

- Diezmar los datos: Se observó que al ejecutar los *microbenchmarks* que los datos obtenidos de los mismos se guardo en grupos de a 32, esto debido a que la medición de las latencias se realizó en el contexto de ejecución de la GPU en grupos de a 32 *threads* (warp). Al tener los datos organizados de esta manera es posible realizar un diezmado de tal forma que se tome en consideración solo un dato de cada grupo de 32 datos, eliminado de esta manera información redundante. El tamaño de los vectores al realizar el diezmado se escala por $1/32$.

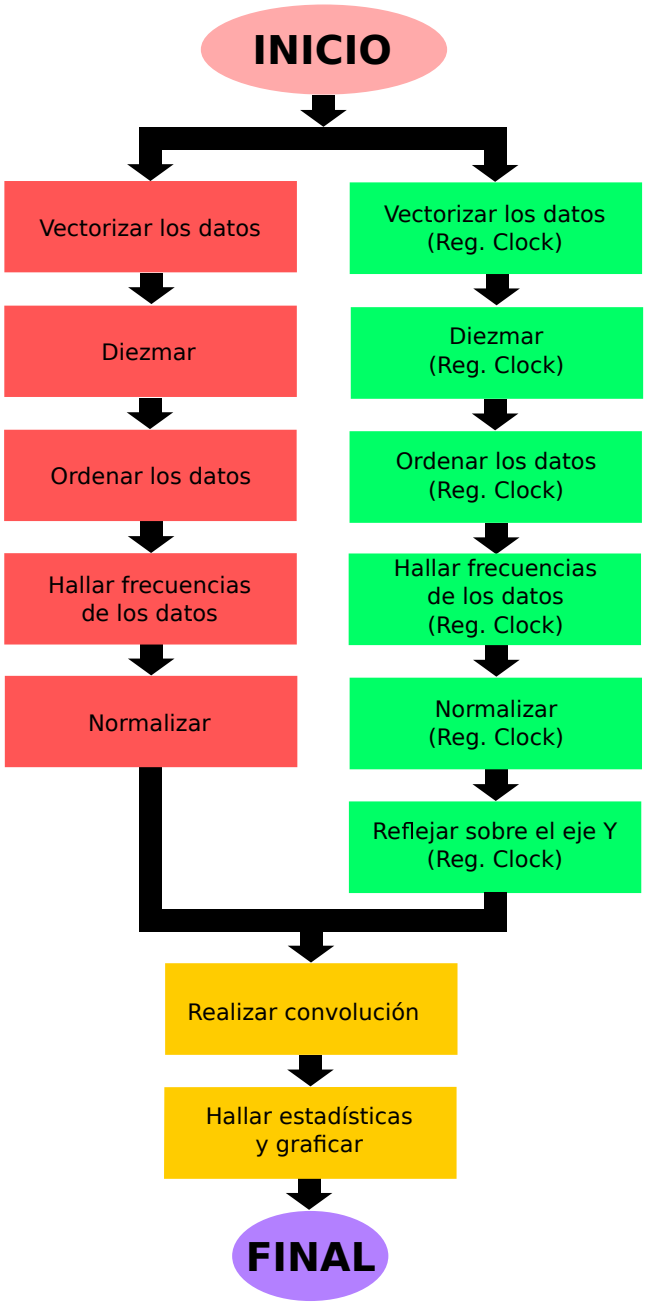
- Ordenar los datos: Los datos de los vectores se deben ordenar de menor a mayor con el fin de facilitar el proceso de hallar la frecuencia de los datos.

- Hallar frecuencia de los datos: Se halla la frecuencia de cada dato en los vectores y esta información se guarda en nuevos vectores de frecuencia de datos.

- Normalizar: Se normalizan los vectores de frecuencia de datos dividiendo cada elemento de los vectores de frecuencia en el número de datos que contienen los vectores de los datos diezmados. Al realizar la normalización se obtienen vectores que contienen la información correspondiente a distribuciones de probabilidad de dos variables aleatorias independientes: la medición de latencia y el costo de uso del registro especial *clock*.

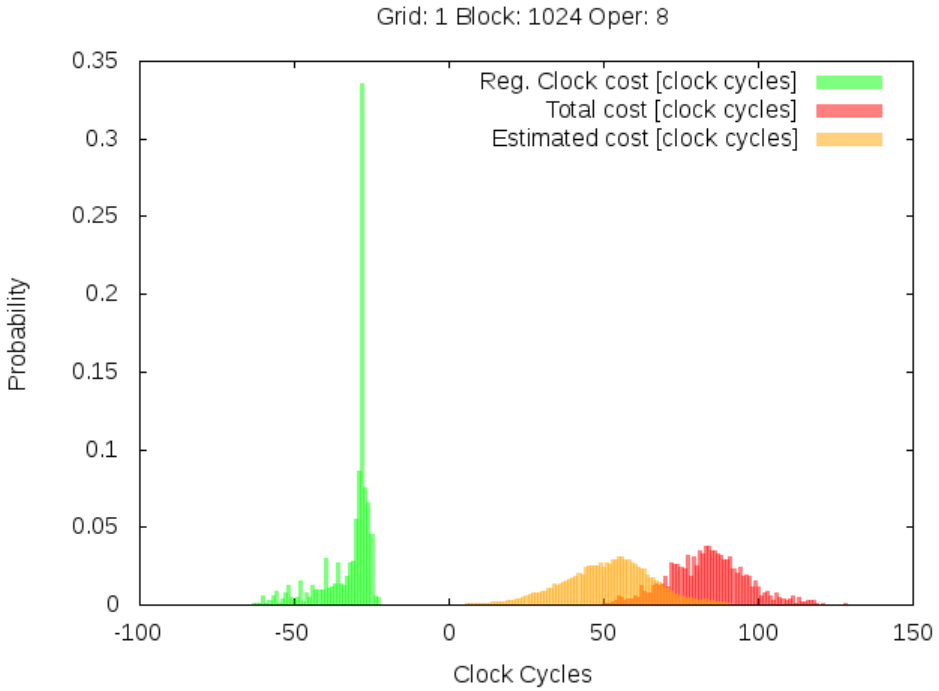
- Reflejar sobre el eje Y: Se realiza sobre los datos correspondientes a la distribución de probabilidad de costo de uso del registro especial *clock*. Esta operación se hace para realizar la convolución y poder garantizar que se está eliminando la distorsión en la medición debida al uso del registro especial *clock*.

Figura 21: Diagrama de flujo del procesamiento de los datos. Los procesos en color rojo se hacen sobre los datos de medición de latencia, los procesos en color verde se realizan sobre los datos de costo de uso del registro especial *clock*. Los procesos en color naranja se hacen con los dos grupos de datos.



- Realizar convolución: Se realiza la convolución entre los datos correspondientes a las distribuciones de probabilidad de medición de latencia y costo de uso del registro especial *clock*. El resultado de esta convolución se guarda en un nuevo vector.

Figura 22: Convolución para estimar el valor de latencia. En este se estima el valor de latencia en la prueba para la instrucción de sustracción con 8 operaciones, 1 bloque y 1024 *threads*.



- Hallar estadísticas y graficar: Con el vector en el que se guardaron los datos de la convolución se calcula el valor de la media y de la desviación estándar de esta distribución de probabilidad. Es conveniente graficar los datos ya que esto permite detectar posibles errores en el proceso y permiten visualizar los resultados del mismo.

En la gráfica número 22 se puede observar la operación de convolución utilizada para estimar la latencia. La gráfica en color rojo representa la distribución de probabilidad de las mediciones realizadas sobre las operaciones, la gráfica en color verde representa la distribución de probabilidad de costo de uso del registro especial *clock* y la gráfica en color naranja representa la distribución de probabilidad de la estimación.

En este punto ya se podría presentar los resultados y extraer los parámetros de latencia de operaciones ariméticas, pero esto se deja para el final del capítulo para presentar los resultados de manera conjunta en una sola sección. Los datos de latencia de acceso a memoria global y a memoria local requieren de un análisis adicional de métricas para poder extraer los parámetros de de *hit* en memoria *cache* L1, *hit* en memoria *cache* L2 y accesos a DRAM.

6.2. Procesar datos de métricas y eventos

Las distribuciones de probabilidad de latencia de acceso a memoria global puede contener latencias de acceso a memoria *cache* L2 y a DRAM. De manera similar para el caso en el que se utiliza memoria local se pueden encontrar accesos a memoria *cache* L1, memoria *cache* L2 y DRAM. El análisis de métricas se realiza para poder analizar estas latencias y poder separar los valores de estos accesos a memoria.

6.2.1 Supuestos iniciales.

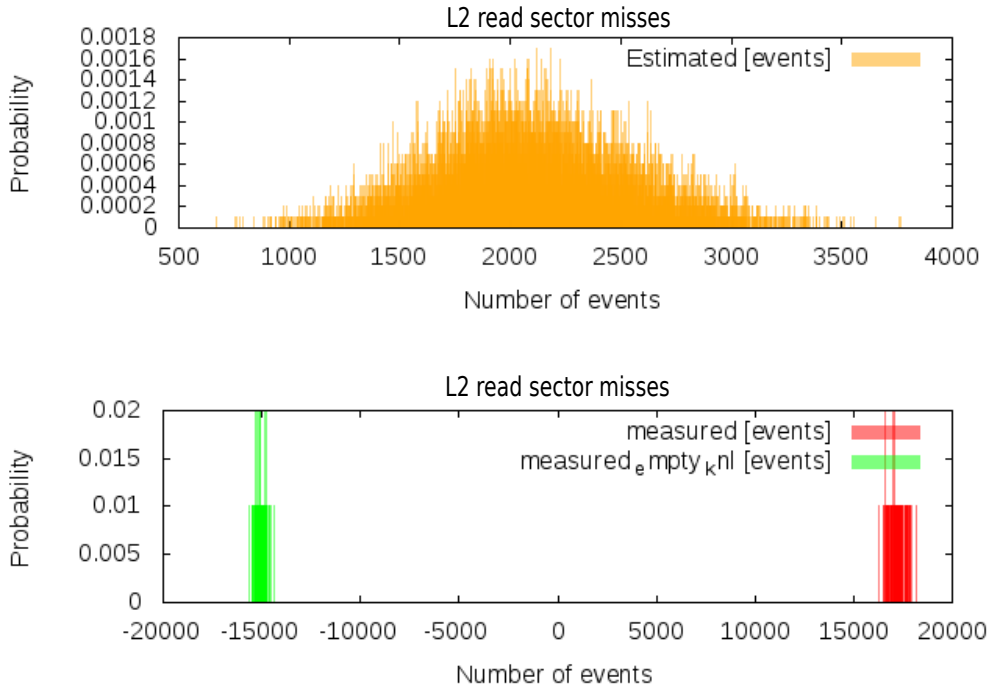
Para realizar este análisis se debe partir de :

- Las operaciones de acceso a memoria cargan datos desde un determinado espacio en la jerarquía de memoria la GPU a registros.
- De acuerdo con la jerarquía de memoria de la GPU, los accesos a memoria más rápidos serán los de memoria *cache* L1, seguidos por los accesos a memoria *cache* L2; los accesos de memoria más lentos serán los de DRAM.
- Para las pruebas en las que se tiene latencias de acceso a memoria global existen casos en los que solo existen accesos a memoria *cache* L2, el otro escenario posible es que existan accesos a memoria *cache* L2 y a DRAM.
- Para las pruebas en las que se tiene las latencias de acceso a memoria local existen tres posibles casos: pruebas con latencia de acceso a memoria *cache* L1, pruebas con latencia de accesos a memoria *cache* L1 y a memoria *cache L2* y finalmente pruebas con accesos a memoria *cache* L1, memoria *cache* L2 y DRAM.
- Si existe acceso a un solo tipo de memoria para una prueba, los datos estarán agrupados en torno a un valor y su dispersión no debería ser muy grande.
- No existen traslapes en los valores de latencia obtenidos para cada prueba en caso de que existan accesos a más de un tipo de memoria en la misma.
- Con ayuda de las métricas debería poder determinarse las pruebas para las que existen cambios en los accesos a los diferentes tipos de memoria dentro de la jerarquía de memoria de la GPU.
- Los valores de las métricas y eventos tienen validez en un contexto de *kernel*, estos datos se toman para todas las operaciones que se realizan en el *kernel*.
- Al evaluar la métrica *gld_transactions_per_request* se debe tener para todas las pruebas su valor es de 1.0, esto garantiza que los accesos a memoria global fueron *coalesced*, es decir los *threads* accedieron a espacios de memoria aledaños.

- Es posible determinar las métricas correspondientes únicamente a los accesos a memoria global y a memoria local utilizando una estrategia similar a la utilizada en el procesamiento de los datos de latencia.

El análisis de métricas se realizó para las pruebas de acceso a memoria global, ya que se asume que no existe traslape en los valores de acceso a diferentes espacios dentro de la jerarquía de memoria de la GPU y que para los accesos a memoria global solo existen accesos a memoria *cache* L2 y a DRAM, es posible utilizar el valor mínimo de latencia de cada prueba de memoria global como el mínimo valor de acceso a memoria *cache* L2 para las pruebas de latencia de acceso a memoria local y de esta manera separar los valores de latencia de memoria *cache* L1 para las pruebas de latencia de acceso a memoria local.

Figura 23: Convolución para estimar el valor de la métrica de *l2_ read_ sector_ misses*. Se realiza la estimación para la prueba con 20 bloques y 1024 *threads*. La gráfica en color naranja representa la estimación del valor de esta métrica para los accesos a memoria global para esta prueba. Las gráficas en color rojo y verde representan los datos de esta métrica para la prueba de medición de latencia de acceso a memoria global y los datos de la prueba de costo de uso del registro especial *clock* respectivamente.



6.2.2 Procesamiento de métricas de accesos a memoria global. De los *microbenchmarks* tienen los datos de las métricas y eventos de la prueba en las que se realizó la medición de

latencia de los accesos a memoria, también se dispone de los datos de las métricas y eventos de la prueba en la que se midió el costo de uso del registro especial *clock*. Debido a que estas métricas se encuentran definidas en un contexto de *kernel* es preciso encontrar alguna manera de separar los valores de las métricas correspondientes a los accesos a memoria global. Para realizar esto se procesan los datos de la misma manera que se procesaron los datos de latencia (ver figura 21). Las métricas que se extrajeron se definen en la tabla 8 y se extrajeron para todas las pruebas de accesos a memoria global. En la figura 23 se observa el resultado del procesamiento para una de las métricas.

Tabla 8: Métricas para el análisis de memoria global. Fue necesario definir una métrica de *misses* al realizar lecturas en memoria *cache* L2 a partir de eventos.

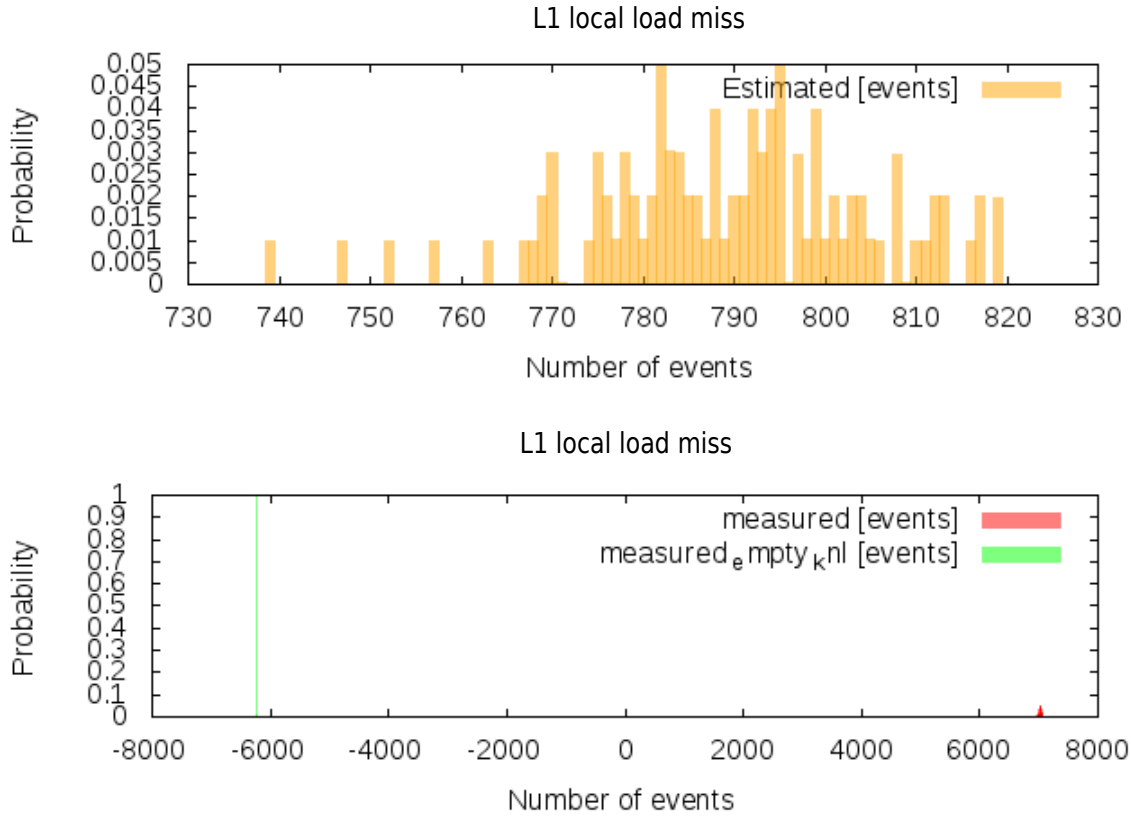
Métrica	Eventos y/o Métrica
L2 read sector misses	l2_subp0_read_sector_misses l2_subp1_read_sector_misses l2_subp2_read_sector_misses l2_subp3_read_sector_misses
L2 read transactions	l2_read_transactions
dram read transactions	dram_read_transactions

Tabla 9: Métricas para el análisis de latencia de memoria local. Algunas de las métricas (columna derecha) se definen a partir de la suma de eventos (columna izquierda).

Métrica	Evento y/o Métrica
dram read transactions	dram_read_transactions
L2 read sector misses	l2_subp0_read_sector_misses l2_subp1_read_sector_misses l2_subp2_read_sector_misses l2_subp3_read_sector_misses
L2 L1 read L1 hit sectors	l2_subp0_read_l1_sector_queries l2_subp1_read_l1_sector_queries l2_subp2_read_l1_sector_queries l2_subp3_read_l1_sector_queries
L2 L1 read transactions	l2_subp0_read_l1_hit_sectors l2_subp1_read_l1_hit_sectors l2_subp2_read_l1_hit_sectors l2_subp3_read_l1_hit_sectors
local load transactions	local load
L1 local load hit	l1 local load hit
L1 local load miss	l1 local load miss

6.2.3 Procesamiento de métricas de accesos a memoria local. Las métricas que se deben considerar para el análisis de este tipo de memoria se presentan en la tabla 9. Se incluyen métricas que permiten el análisis de la memoria *cache* L1.

Figura 24: Estimación del valor de la métrica de *l1_local_load_miss*. La prueba con 15 bloques y 1024 *threads*. La gráfica en color naranja representa la estimación del valor de esta métrica para los accesos a memoria local, la gráfica en color rojo los datos de la métrica para de medición de latencia de acceso a memoria local y la gráfica en color verde representan y los datos de la métrica para la medición del costo de uso del registro especial *clock*.



Los datos de las métricas para los accesos de memoria local se procesan de la misma manera que para memoria global. En la figura 24 se presenta el resultado del procesamiento para una de estas métricas. Aunque estas métricas no se utilizan en el desarrollo de esta metodología, ya que resulta más sencillo separar los accesos a memoria *cache* L1 utilizando los resultados que se obtienen del análisis de memoria global, si pueden ser útiles para el análisis de memoria local en un contexto de análisis desde la perspectiva de la jerarquía de memoria de la GPU.

6.3. Análisis de métricas y latencias

Figura 25: *L2 read sector misses* para los accesos a memoria global. Para esta gráfica se observa que hasta aproximadamente 10000 *threads* los *miss* en *cache* L2 tienen un valor cercano a cero.

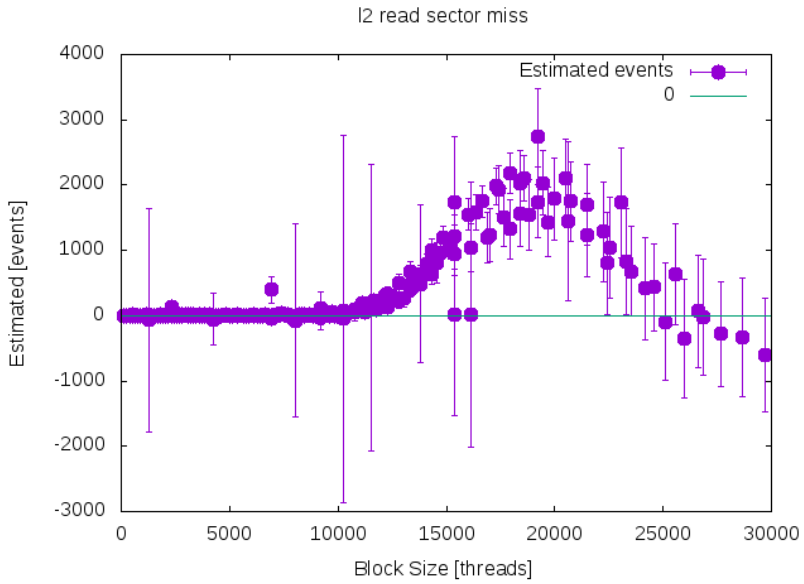
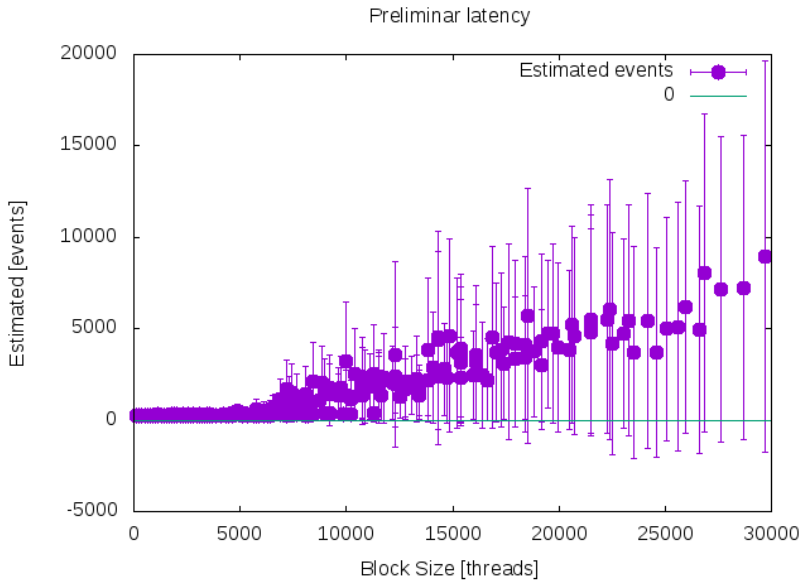


Figura 26: Valores de media arimética para latencia de acceso a memoria global. Se puede observar que estos valores son cercanos a cero para menos de 10000 *threads*, de laMedia arimética de latencia de accesos a memoria global para pruebas con menos de 10000 *threads*. misma forma que para la gráfica de *L2 sector misses*.



6.3.1 *Hit latency: L2* y DRAM *latency.*

Figura 27: Valores de la métrica *L2 read sector misses* para pruebas de accesos a memoria global de menos de 10000 *threads*.Se puede observar que dentro de este intervalo los valores de la métrica son menores a 20 eventos.

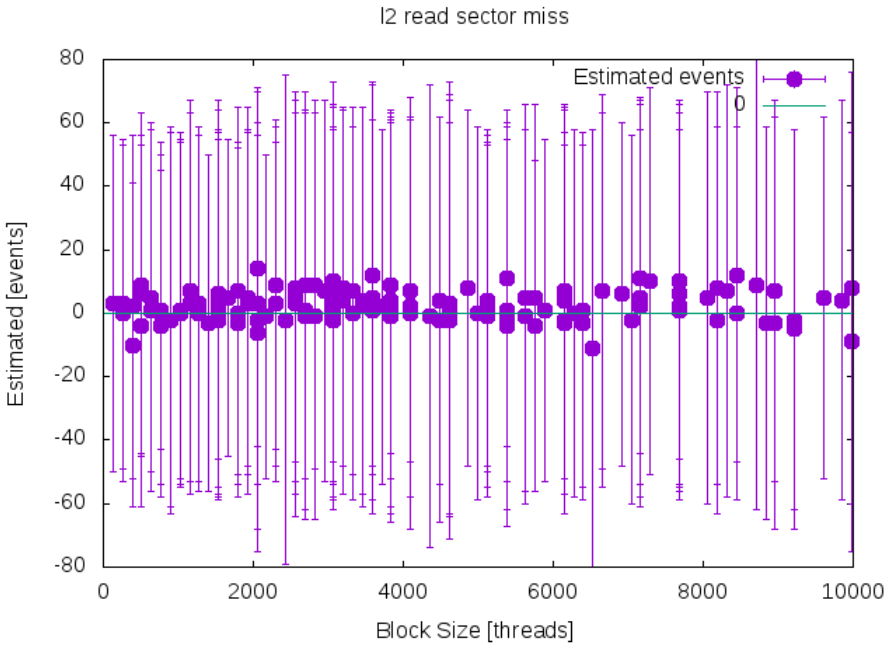
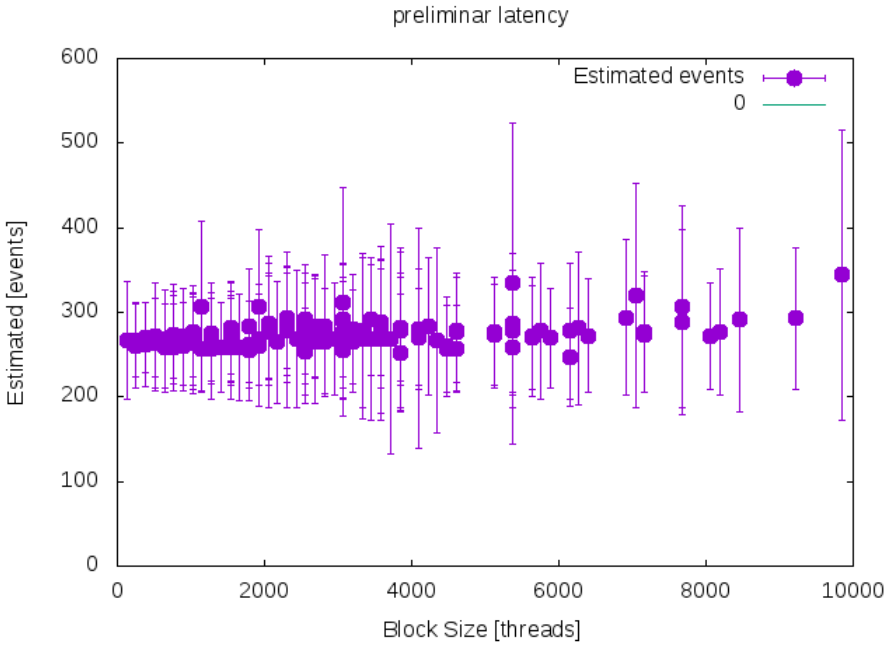


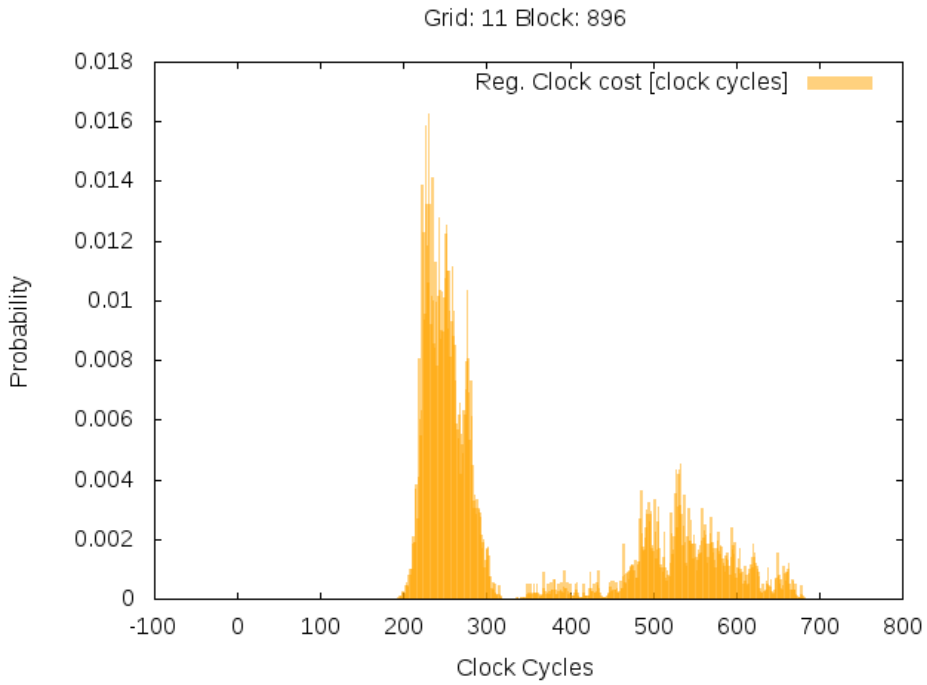
Figura 28: Media arimética de latencia de accesos a memoria global para pruebas con menos de 10000 *threads*.Se puede observar que el valor de la media arimética para las pruebas en este intervalo tienen un valor menor a 350 ciclos de reloj.



Para extraer estos parámetros se realiza un análisis de las distribuciones de latencia de acceso a memoria global y de sus métricas, con el fin de determinar las latencias de *hit* en memoria *cache* L2 y accesos a DRAM dentro de cada distribución.

El punto de partida para el análisis debe hacerse con las métricas de *DRAM transactions* o *L2 read sector misses* (estas gráficas deben ser similares). A partir de esta gráfica debe establecerse un intervalo de análisis en el cual se pueda suponer que solamente existieron accesos a memoria *cache* L2, en las gráficas debe haber un punto de inflexión en el cual empiezan a existir accesos a memoria DRAM o lo que es lo mismo, *miss* en memoria *cache* L2 como se puede observar en la figura 25. A partir de estas gráficas se establece un intervalo de análisis de acceso a memoria cache L2 desde 128 *threads* hasta 9984 *threads*.

Figura 29: Medidas de latencia de accesos a memoria global para la prueba con 11 bloques y 896 *threads*.

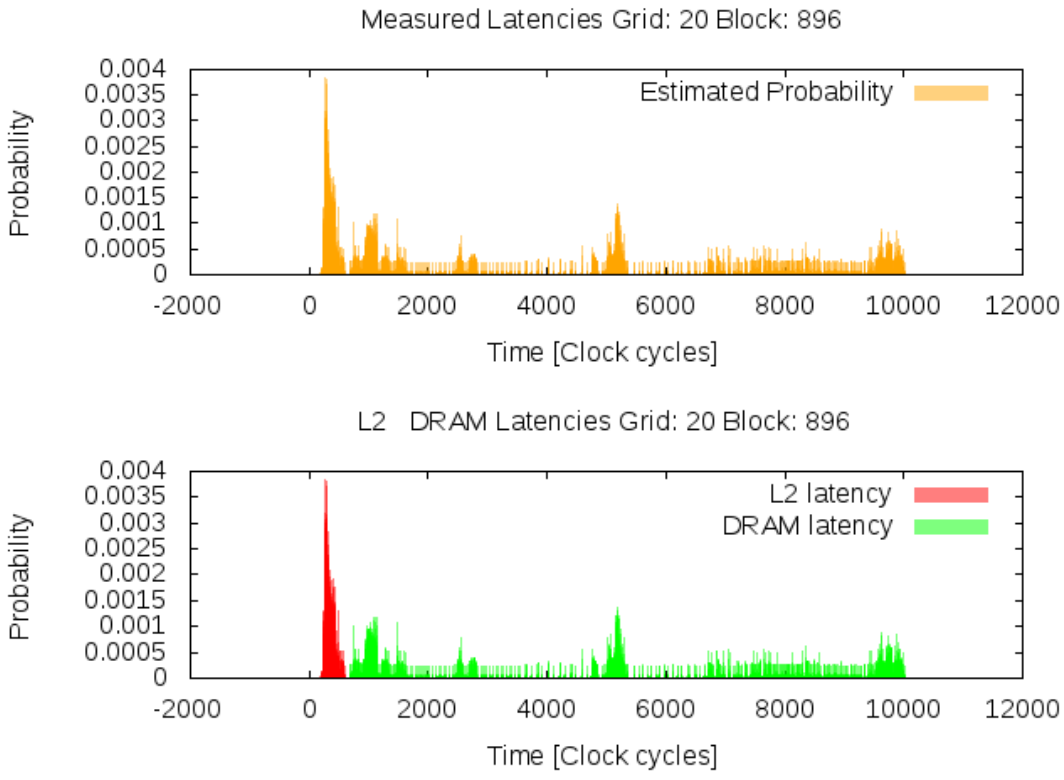


Ahora se debe tomar en cuenta los valores de latencia. Se supone que para latencia si únicamente existen accesos a memoria *cache* L2 en una prueba, sus valores deben estar agrupados y su dispersión debe ser pequeña; este comportamiento se puede observa en la figura 26. Se debe realizar entonces un análisis de los valores tanto de latencia, como de la métrica de *L2 sector miss* en el intervalo definido a partir de la gráfica de la métrica *L2 read sector misses*, como se observa en las gráficas 27 y 28.

De las gráficas 27 y 28 se puede concluir que para las pruebas en las que se asume que solo existen accesos a memoria *cache* L2 el valor de su media arimética para latencia es no mayor a 350 ciclos de

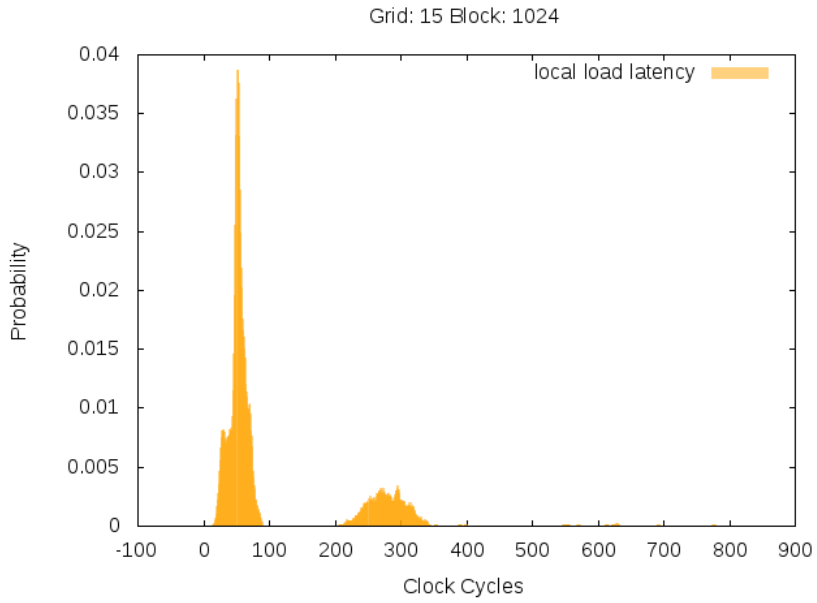
reloj, y el número de eventos de la métrica *L2 read sector misses* es no mayor a 20 eventos.
Con esto en mente se realiza un barrido dentro del intervalo de análisis para encontrar la prueba con el mayor número de threads que cumpla las dos condiciones: un valor de media arimética para latencia menor a 350 ciclos de reloj, y un valor de la métrica *L2 read sector misses* menor a 20 eventos. Al realizar este análisis se encuentra que este punto pertenece a la prueba de 11 bloques con 896 *threads*, la medidas de latencia para esta prueba se observa en la figura 29).
Asumiendo que esta será la prueba con el mayor número de *threads* que tendrá solamente accesos a memoria *cache* L2 es posible determinar el valor que se utilizará como frontera entre los accesos a memoria *cache* L2 y DRAM, tomando el valor más grande de la distribución para esta prueba: 683 ciclos de reloj. Con este valor se puede calcular la latencia de *hit* en memoria *cache* L1 y la latencia de acceso a DRAM, a partir de las mediciones de latencia que se tienen para cada prueba. Esto se hace separando los valores de las mediciones de latencia que se tienen utilizando como frontera este valor, como se observa en la figura 30).

Figura 30: Estimación de los valores de latencia de *hit* en memoria *cache* L2 y accesos a DRAM.Se presentan los valores estimadas para la prueba con 20 bloques y 896 *threads*. La gráfica en color naranja representa la estimación de las medidas de latencia para la prueba, la gráfica en color rojo representa la latencia de los *hit* en memoria *cache* L2, y la gráfica en color verde la latencia de accesos a DRAM



Es necesario hallar el límite inferior de la distribución de latencia para cada prueba con el fin de utilizarlo como frontera para hallar los valores de latencia de *hit* de memoria *cache* L1.

Figura 31: Latencia de accesos a memoria local para la prueba con 15 bloques y 1024 *threads*Esta medida de latencia incluye latencias de accesos a memoria *cache* L1, memoria *cache* L2 y DRAM. Se debe encontrar alguna manera de aislar los valores de acceso a memoria *cache* L1.



6.3.2 ***Hit latency: L1.*** Para esta prueba se utiliza memoria local(este tipo de memoria se encuentra definido por *software*), la cual en términos de hardware trabaja a través de memoria *cache* L1, memoria *cache* L2 y DRAM; por consiguiente los datos de medición de latencias tendrán medidas de latencia de todos los tipos de memoria a través de los cuales se trabaja como se observa en la figura 31.

Se debe encontrar alguna manera de aislar los datos de acceso a memoria *cache* L1, para esto se toma el límite inferior de las distribuciones obtenidas para el cálculo de latencia de memoria *cache* L2 para cada prueba con el mismo número de bloques y *threads*. Ya que se asume que no existe traslape en los valores de latencia para distintos espacios de memoria en hardware, se asume que el valor máximo de latencia que puede llegar a tener un acceso de memoria *cache* L1 debe ser menor que el valor mínimo de latencia que puede tener acceso de memoria *cache* L2, con esto se puede definir el intervalo de datos que se considerarán como accesos a memoria *cache* L1 como se observa en la figura 32.

Figura 32: Latencia de acceso a memoria *cache* L1 para la prueba con 15 bloques y 1024 *threads*. La gráfica en color rojo representa los valores acceso a memoria *cache* L1, valores definidos a partir del límite inferior de la distribución de memoria *cache* L2.

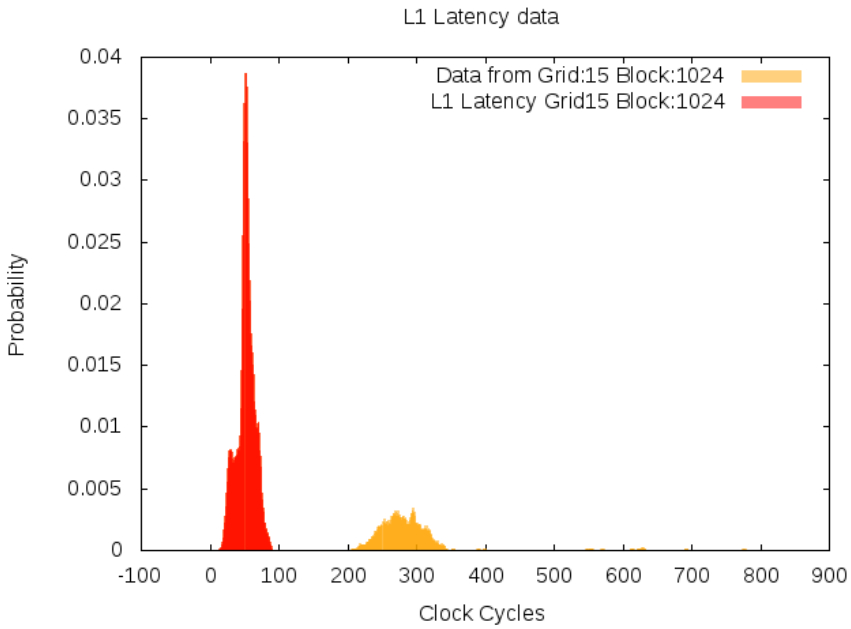
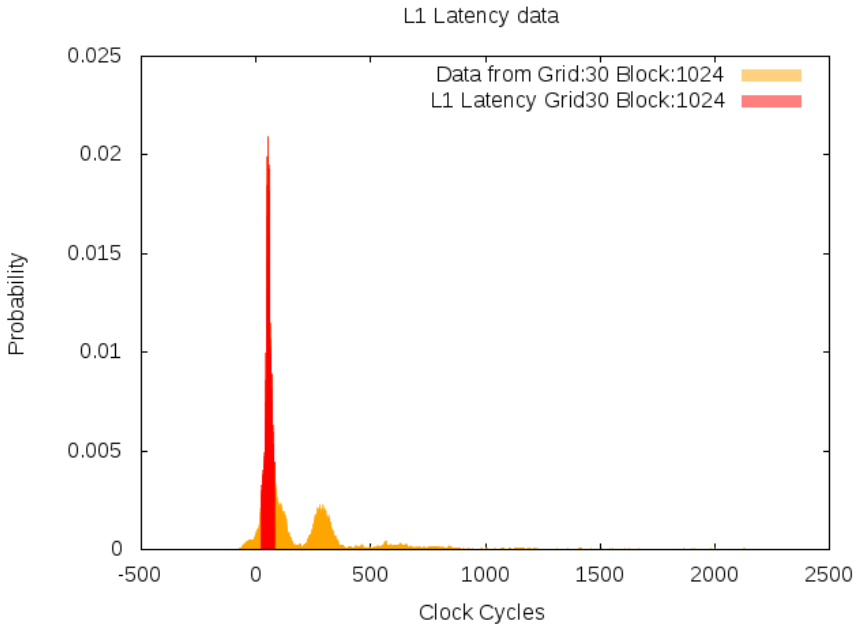


Figura 33: Latencia de acceso a memoria *cache* L1 para 15 bloques con 1024 *threads*. El intervalo que se define para los accesos a memoria *cache* L1 se presenta en color rojo, este intervalo se define en torno a la moda de los datos con el mismo ancho por derecha e izquierda. Se puede observar que a la izquierda del intervalo existen valores que no se consideran dentro del mismo.



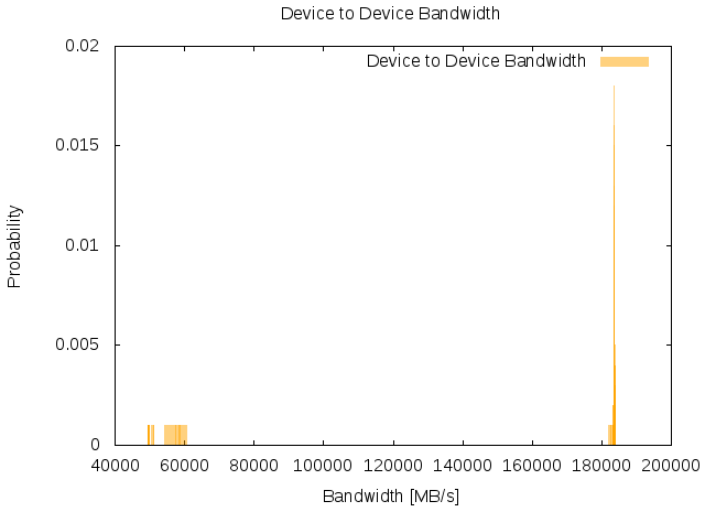
Este intervalo se define en torno a la moda (el valor con mayor probabilidad de ocurrencia), de tal manera que el intervalo en el que se tomen los datos tenga el mismo ancho por derecha e izquierda, garantizando que el valor de la media se aproximará a la moda y que minimizando la distorsión que pueda ocurrir al tomar un número mayor de datos por la izquierda del intervalo, esto se puede observar con mayor claridad en la figura 33.
En caso de no tener un límite de accesos a memoria *cache* L2, las métricas que se toman para la prueba de memoria *cache* L1 pueden ser utilizadas para realizar un análisis similar al de memoria *cache* L2 y DRAM para determinar el intervalo de datos correspondientes a accesos de memoria *cache* L1.

6.4. Presentación de resultados y extracción de Parámetros

En esta sección presentan los resultados obtenidos de los *microbenchmarks* de acuerdo al contexto definido para cada uno. En las gráficas se presenta cada punto como la media de los datos para la prueba, y se define el intervalo de confianza de los datos utilizando barras. El intervalo de confiabilidad de estos resultados se define tomando una vez el valor de la desviación estándar de los datos, de tal manera que se garantiza que existe una probabilidad del 68.27 % de encontrar el valor de la prueba en este intervalo.
La manera manera más apropiada de utilizar estos resultados es considerar el número de de *threads* para los que se desea conocer el valor del parámetro y con esto reportar el valor del mismo y el valor del intervalo de confianza del mismo. Si bien existen parámetros en los cuales no se evidencia dependencia del valor de los mismos del número de *threads*, el hecho que para cada prueba se tengan valores de desviación estándar diferentes hace que no se pueda consolidar un un intervalo de confianza único.
El valor del parámetro que se calcula en cada gráfica es la media de los datos obtenidos de las pruebas que se ejecutaron. Este valor sirve de guía en caso que no tenga un número de *threads* en específico para el que se quiera determinar el valor del parámetro. Para este valor no se puede definir un intervalo de confianza.

6.4.1 *Memory peak bandwidth.*

Figura 34: *Memory peak bandwidth*. Para hallar este parámetro se utilizó la herramienta *bandTest*. El dato que se toma con esta herramienta es el de ancho de banda de una transferencia de datos en la memoria DRAM del *device*. El tamaño de la transferencia de datos esta definido por la herramienta y es de 33.554432[kB]. La prueba se ejecutó 1000 veces. Los valores de la parte izquierda de la gráfica se consideran pruebas nulas y no se tomarán en cuenta para el cálculo del parámetro. Bajo estas condiciones el valor del parámetro es de 183.5[GB/s].



6.4.2 *Average instruction latency.*

Figura 35: Latencia para la operación de suma. Los puntos en color rojo representan los resultados para la prueba con un bloque, los puntos en color verde para 15 bloques y los puntos en color naranja para 30 bloques. Se observa que se obtienen resultados similares para cada prueba. A partir de estos datos se estima el valor de latencia para esta operación en 17[*clock cycles*].

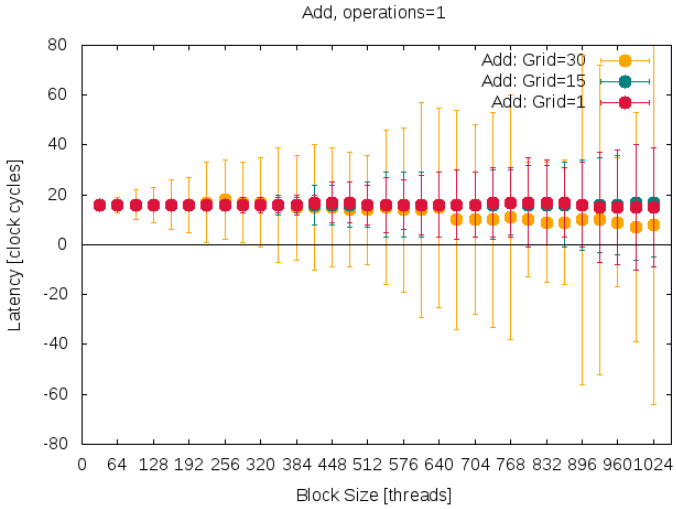


Figura 36: Latencia para la operación de resta. Los puntos en color rojo representan los resultados para la prueba con un bloque, los puntos en color verde para 15 bloques y los puntos en color naranja para 30 bloques. Se observa que se obtienen resultados similares para cada prueba. A partir de estos datos se estima el valor de latencia para esta operación en 17[*clock cycles*].

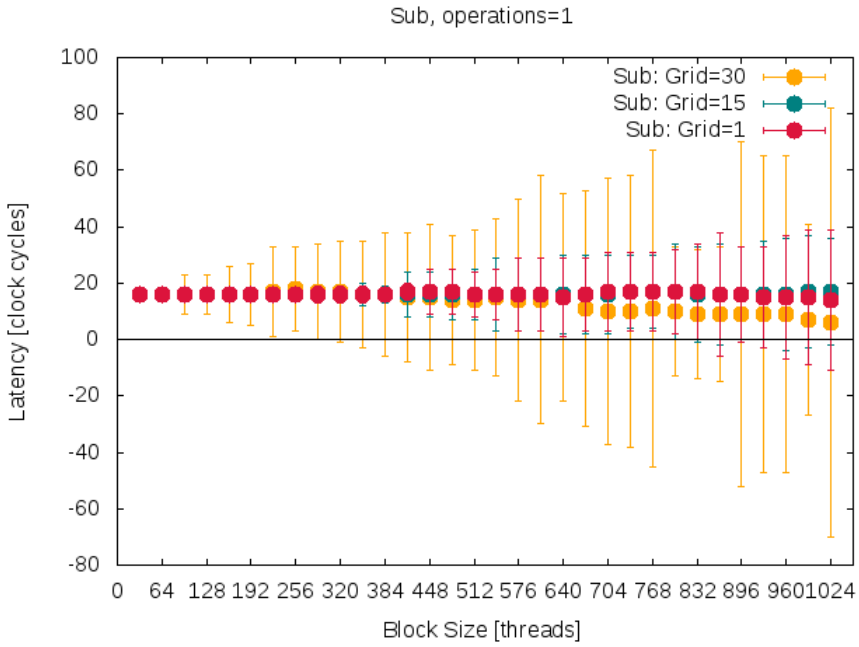


Figura 37: Latencia para la operación de multiplicación. Los puntos en color rojo representan los resultados para la prueba con un bloque, los puntos en color verde para 15 bloques y los puntos en color naranja para 30 bloques. Se observa que se obtienen resultados similares para cada prueba. A partir de estos datos se estima el valor de latencia para esta operación en 17[*clock cycles*].

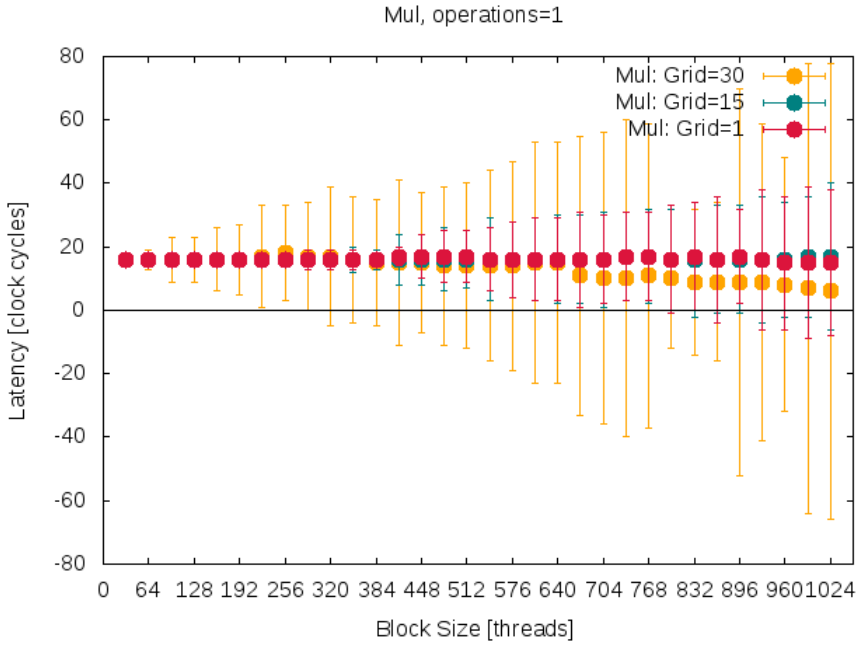


Figura 38: Latencia para la operación de división. Los puntos en color rojo representan los resultados para la prueba con un bloque, los puntos en color verde para 15 bloques y los puntos en color naranja para 30 bloques. Se observa que se obtienen resultados similares para cada prueba. A partir de estos datos se estima el valor de latencia para esta operación en 960[*clock cycles*].

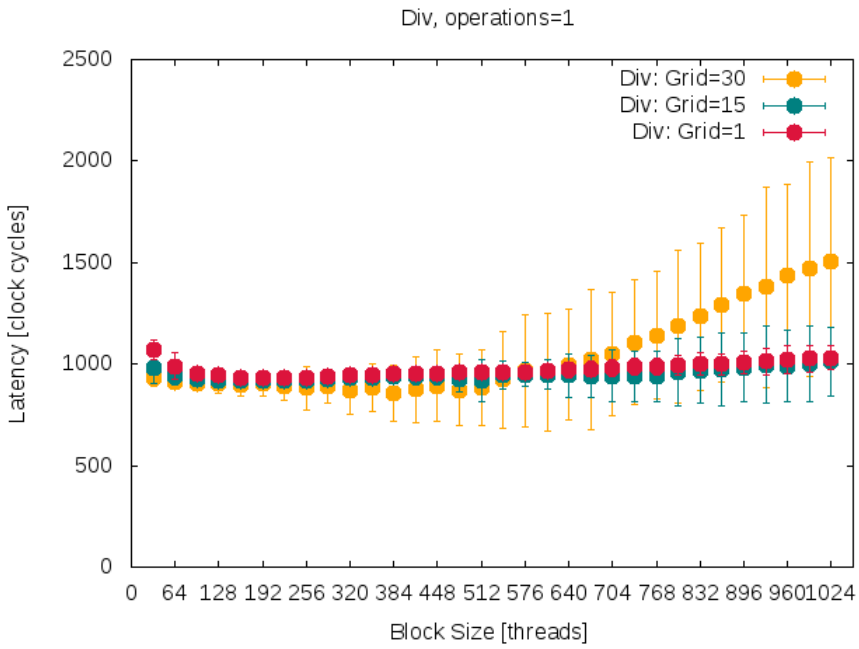
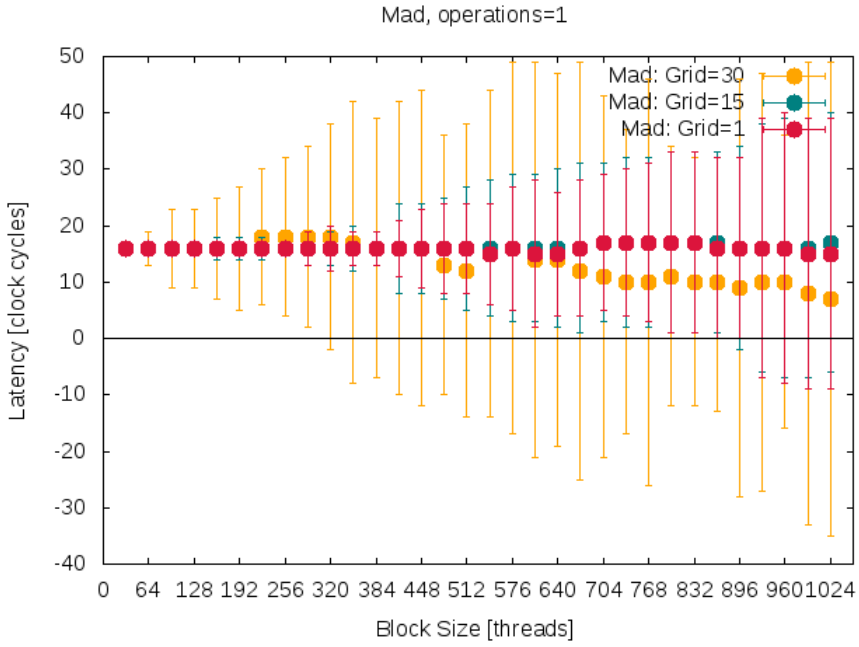


Figura 39: Latencia para la operación de multiplicación-adición fusionadas. Los puntos en color rojo representan los resultados para la prueba con un bloque, los puntos en color verde para 15 bloques y los puntos en color naranja para 30 bloques. Se observa que se obtienen resultados similares para cada prueba. A partir de estos datos se estima el valor de latencia para esta operación en 17[*clock cycles*].



6.4.3 *Hit latency: L2* y DRAM *latency.*

Figura 40: *DRAM Latency.* Los resultados de las pruebas para este parámetro se presentan por número de *threads*. De la gráfica se puede observar que los valores de este parámetro no son independientes de la configuración de la prueba (número de *threads*). Al realizar un análisis de los datos, se observa que se puede realizar una regresión lineal sobre estos datos para aproximar el valor de este parámetro en función del número de *threads*.

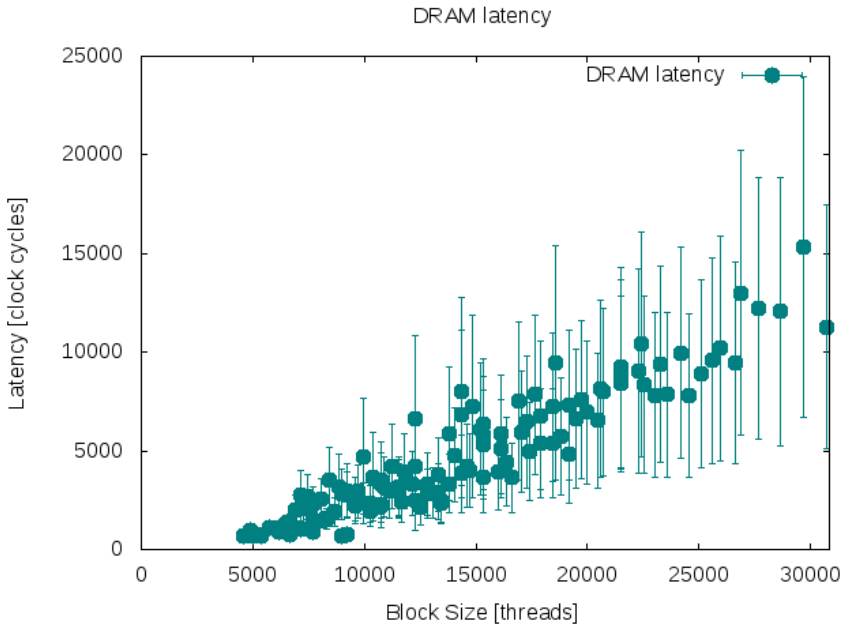
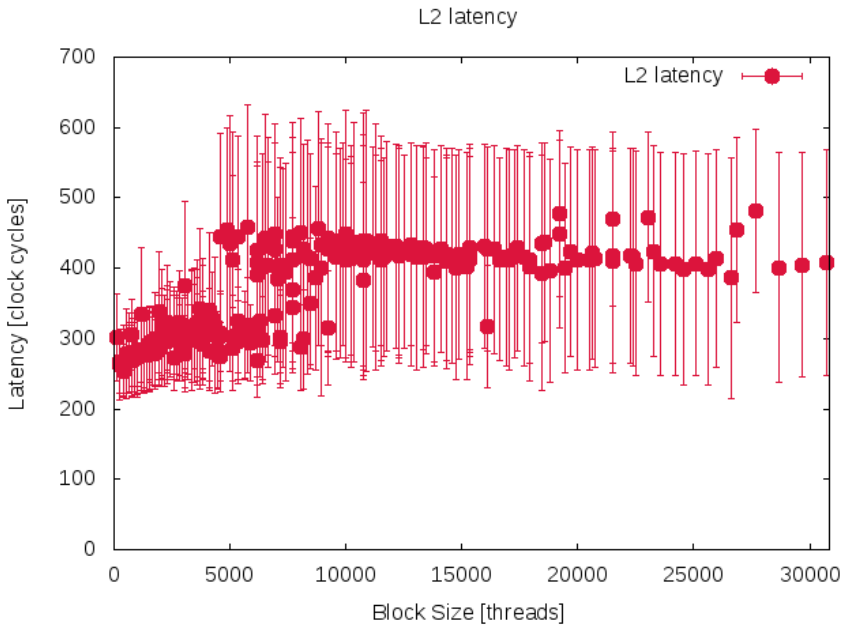


Figura 41: *Hit latency: cache L2.* Los resultados de esta pruebas se presentan de acuerdo al contexto por número de threads. Al realizar el promedio de los valores obtenidos se puede estimar el valor de este parámetro en 365[*clock cycles*].



6.4.4 *Hit latency: L1.*

Figura 42: *Hit latency: cache L1.* Se presentan los resultados obtenidos para las pruebas de acuerdo al contexto. Según el contexto, en estas pruebas siempre se ejecutó un bloque por *streaming multiprocessor* activo. El valor de ese parámetro se estima en 51 [*clock cycles*].

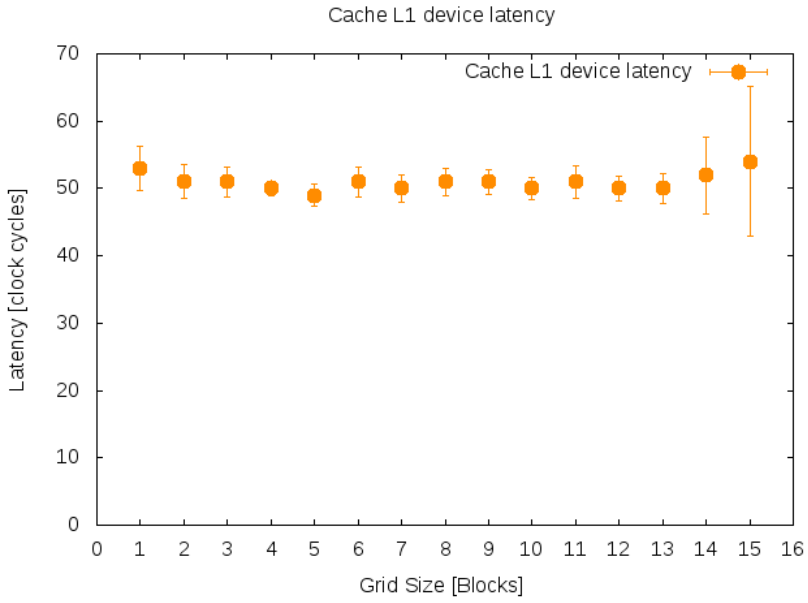
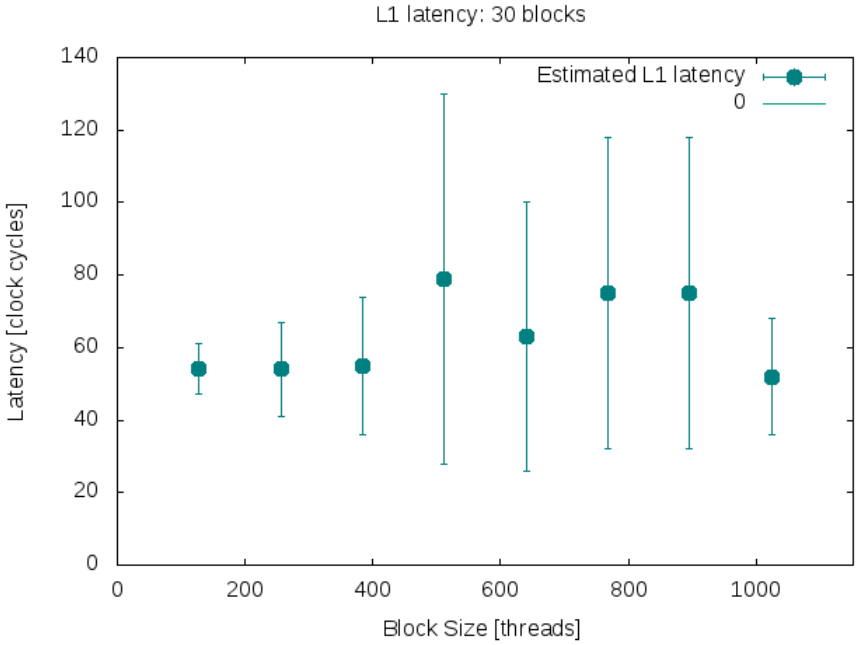


Figura 43: *Hit latency: cache L1, Grid= 30[bloques].* Según lo definido en el contexto para esta prueba siempre se ejecutan dos bloques por *streaming multiprocessor*. El valor del parámetro se estima en 64[*clock cycles*].



Capítulo 7

Conclusiones

En este capítulo se presentan los resultados obtenidos para los parámetros y se realiza un análisis de los mismos, se presentan las conclusiones del trabajo y finalmente se presentan algunas observaciones y posible trabajo futuro con base en este trabajo.

7.1. Resultados

Tabla 10: Se listan los parámetros y los valores obtenidos para cada uno. La variable X en la ecuación de la latencia de acceso a DRAM tiene unidades de número de threads. Esta ecuación tiene validez siempre y cuando se tenga certeza que se hacen accesos a este espacio de memoria.

Parámetro	**Valor**
Core frequency	745 [MHz]
Warp size	32 [*threads*]
SIMD width	192 [*CUDA-cores*]
Memory peak andwidth	183.5[GB/s]
Average instruction latency	Add: 17[*clock cycles*] Sub: 17[*clock cycles*] Mul: 17[*clock cycles*] Div: 960[*clock cycles*] Mad: 17[*clock cycles*]
Hit latency	*Cache L1*: 51[*clock cycles*] *Cache L2*: 365[*clock cycles*]
DRAM latency	$Y = 0{,}466X - 1911{,}2[clock\_cycles], (R^2 = 0{,}879918)$ (7.1)

7.1.1 Análisis de resultados. Los parámetros de *core frequency*, *warp size*, *SIMD width* vienen dados directamente por la arquitectura lo cual deja fuera de discusión los valores obtenidos para los mismos. En el caso del parámetro de *memory peak bandwidth* al consultar la página web del fabricante este da un valor de ancho de banda para la memoria DRAM de 288[GB/s] con el mecanismo de ECC inactivo. El valor de este parámetro de acuerdo a la metodología aplicada es de 183.5[GB/s]. La diferencia en este valor respecto al suministrado por el fabricante probablemente se debe a que en el *microbenchmark* utilizado en la metodología se contempla el uso del mecanismo de ECC. Al utilizar este mecanismo se obtiene un valor para este parámetro dentro de un ambiente de operación similar a las condiciones reales de trabajo de la GPU.

Los valores obtenidos para los parámetros de latencia se comparan con los estimados estimados en la tesis de maestría *Analytical Model to Estimate the Execution Time of a 3D Acoustic Wave Equation Implementation Using FDTD in a GPU* [1], en la cual se desarrolla el modelo MWP-CWP para una GPU K40. Como referencias adicionales se toman los valores estimados para GPUs con arquitectura Kepler documentados en *Analizing GPGPU Pipeline Latency* [2] y *Benchmarking the Memory Hierarchy of Modern GPUs* [3].

Tabla 11: Los valores de latencia tomados de [3] y [2] corresponden a GPUs con arquitectura Kepler. Los valores tomados de [1] corresponden a la GPU K40 con arquitectura Kepler, que fue la misma sobre la que se desarrolló la metodología.

GPU	GTX78 [3]	GK104 [2]	K40 [1]	K40 (Metodología)
Latencia	[clock cycles]	[clock cycles]	[clock cycles]	[clock cycles]
Add	–	9	12	17
Sub	–	9	12	17
Mul	–	9	12	17
Div	–	758	206	960
Mad	–	9	12	17
Hit Cache L1	110	30	29	51
Hit Cache L2	230	175	215.5	365
DRAM	>370	300	400	Ecuación

En [3] se utiliza un *benchmark* de tipo *P-chase* para analizar las estructuras de *cache/transalation lookaside buffer* (TLB), para medir la latencia se utiliza la función clock de CUDA, la cual es una consulta del registro especial *clock*. No se documenta que se haya tenido en cuenta el efecto del uso del registro especial *clock* y el contexto de la prueba se hizo utilizando accesos a memoria de tipo

[1] PARRA, Dorfell; SALAMANCA, William; RAMÍREZ, Ana. «Analytical Model to Estimate the Execution Time of a 3D Acoustic Wave Equation Implementation Using FDTD in a GPU». Magíster de Ingeniería Electrónica. Universidad Industrial de Santander, 2016

[2] ANDERSH, Michael, *et al.* «On latency in GPU throughput microarchitectures». En: *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on.* IEEE. 2015, págs. 169-170

[3] MEI, Xinxin, *et al.* «Benchmarking the memory hierarchy of modern GPUs». En: *IFIP International Conference on Network and Parallel Computing.* Springer. 2014, págs. 144-156

single-warp, lo que hace que los resultados de la prueba no se aproximen a los de las condiciones reales de operación.

Para medir la latencia de las operaciones en[4] se utiliza una serie de operaciones con resultados dependientes entre sí, se toma el tiempo de ejecución de las operaciones y se divide este tiempo entre el número de operaciones. No se considera que pueda llegar a existir segmentación en las operaciones ni se toma en cuenta el uso del registro especial *clock*. Respecto a la latencia en las operaciones de acceso a memoria no se documenta la manera en la que se tomaron las medidas.

Lo más aproximado a la metodología utilizada se encuentra en.[5] Para las operaciones se utiliza un *microbenchamrk* similar al que se utiliza en esta metodología, además se considera el uso del registro especial *clock* pero con un enfoque distinto al de esta metodología, en este caso se implementan *microbenchmarks* para estimar el valor de uso del registro especial y de un movimiento adicional entre registros, luego al valor obtenido de la latencia se le resta el estimado de los dos valores. En el caso de las latencias de acceso a memoria se plantean *microbenchamrks* de manera similar a los de la metodología, pero el contexto se define solo por bloques, no se toma en cuenta la jerarquía de memoria al momento de realizar el análisis de los datos, el uso del registro especial *clock* se considera de manera similar al de las operaciones y las latencias de acceso a memoria *cache* se estiman en base a ecuaciones y mediciones que no toman en cuenta el espacio de hardware sobre el que se toma la medición.

Ya que el fabricante no provee los valores de latencia de operaciones ni de acceso a memoria no existe una forma de determinar un error en las mediciones respecto a un valor teórico.

7.2. Conclusiones

- La metodología planteada en este trabajo cumple con su objetivo principal que es la extracción de los parámetros, desde el principio se buscaba crear un manual que sirviera de guía para la extracción de los parámetros.
- En general los parámetros que se asumieron como parámetros de hardware demostraron comportarse como tal (los valores no presentaban cambios significativos al variar los parámetros de la prueba, esto es el número de threads). El único parámetro que demostró tener dependencia con el número de threads fue el de *DRAM_ average_ latency*.
- Las herramientas que se utilizaron para hallar cada parámetro cumplieron con su propósito, en especial en lo que se refiere a las latencias de acceso a memoria. Ya que no se puede determinar de manera directa el espacio físico de memoria al cual se accede gracias al análisis de estas métricas se puede lograr estimar el espacio de memoria al cual se está accediendo de manera indirecta.

[4] ANDERSH, Michael, *et al.* «On latency in GPU throughput microarchitectures». En: *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*. IEEE. 2015, págs. 169-170.

[5] PARRA, Dorfell; SALAMANCA, William; RAMÍREZ, Ana. «Analytical Model to Estimate the Execution Time of a 3D Acoustic Wave Equation Implementation Using FDTD in a GPU». Magíster de Ingeniería Electrónica. Universidad Industrial de Santander, 2016.

- Es de vital importancia definir el contexto en el cual se plantean las pruebas. Debido a que no se puede trabajar de manera directa con el hardware para hallar algunos de los parámetros, definir el contexto de las pruebas facilita el análisis de los resultados obtenidos al trabajar con los mecanismos de software que se definen para la GPU.

- La metodología planteada presenta un marco completo de trabajo que permite realizar la extracción de los parámetros. Adicionalmente dentro del análisis de latencia de accesos a espacios de memoria definidos a partir de software se brinda indirectamente un marco de análisis de la jerarquía de memoria de la GPU. Este es un aporte muy significativo de este trabajo ya que brindar una herramienta que permite exponer la jerarquía de memoria de la GPU hace de este trabajo pionero en este aspecto.

- El procesamiento de datos representa una gran carga de trabajo dentro del desarrollo de la metodología. El correcto procesamiento estadístico de los datos y de las métricas obtenidas de los *microbenchamrks* hace que se obtengan estimaciones consistentes con la aplicación de la metodología.

- Realizar convolución en lugar de trabajar con promedios al realizar el análisis de los datos obtenidos de los *microbenchamrks* busca garantizar que se conserva la forma de las distribuciones de densidad de probabilidad de los datos. Este es un punto crítico al realizar el análisis de la jerarquía de memoria de la GPU a partir de las mediciones obtenidas al trabajar con el modelo de memoria de la GPU, permitiendo exponer la jerarquía de memoria al analizar estas distribucioness de densidades de probabilidad.

- Debido a que el fabricante no proporciona valores para los parámetros en los cuales se midió latencia no es posible determinar un porcentaje de error en los resultados. Aunque existen resultados de trabajos en los cuales se han estimado los valores de latencia para estos parámetros, la información insuficiente acerca de como estos fueron hallados, o las diferencias con la metodología que se plantea en este trabajo hace que no tenga sentido una comparación directa entre resultados.

7.3. Observaciones adicionales y trabajo futuro

7.3.1 *Pipelining* de operacciones ariméticas. La prueba de medición de operaciones aritméticas se planteó desde una hasta ocho operaciones sin dependencia de resultados. Estas pruebas se plantearon de esta manera con el fin de determinar que tanto se puede llegar a enmascarar la latencia de los accesos a memoria al realizar más de una operación arimética del mismo tipo por thread.
Según los resultados de la tabla 12 se puede ahorrar tiempo al ejecutar varias operaciones por thread en lugar de ejecutar una sola operación. Esto sería válido únicamente para casos en los que el kernel

Tabla 12: Valores de latencia para varias operaciones. El hecho que la latencia para varias operaciones no sea un múltiplo escalar de la latencia para una sola operación evidencia la existencia de mecanismos de *pipelining* para la ejecución de operaciones ariméticas en la GPU.

Número de operaciones	1	2	3	4	5	6	7	8
Operación	Latencia [clock cycles]							
Add	17	6	28	28	33	40	42	40
Sub	17	6	28	28	33	40	42	40
Mul	17	6	28	28	33	40	42	40
Div	960	1609	2262	2926	3666	4572	5803	6791
Mad	17	6	24	31	31	40	25	89

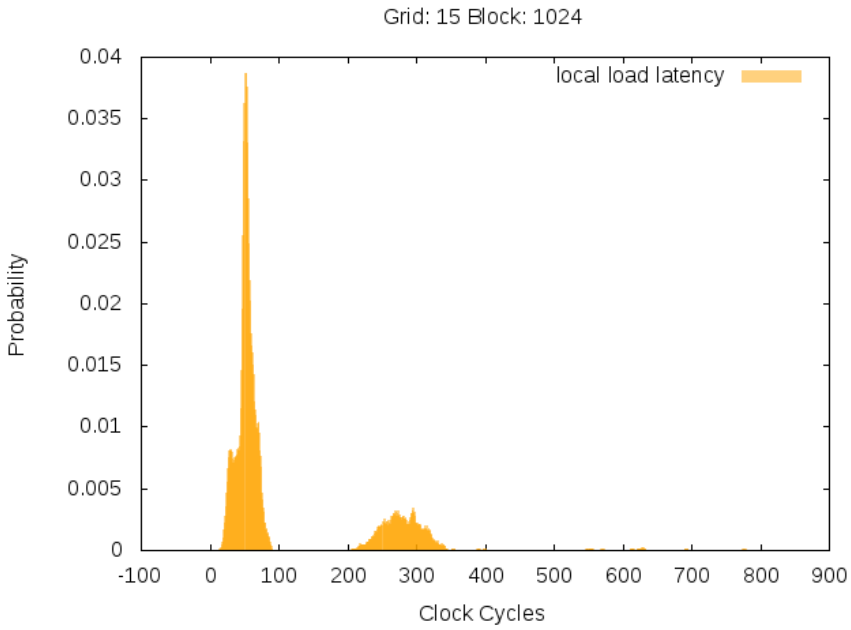
realice rutinas intensas de ejecución de operaciones ariméticas, y además se debe tener en cuenta el costo de los accesos a memoria adicionales que se deben hacer por cada thread.

7.3.2 Jerarquía de memoria en memoria local. En este trabajo se utilizó memoria local para determinar la latencia de *hit* en memoria *cache* L1. En la metodología como tal no se desarrolla un análisis completo de este tipo de memoria, sin embargo los datos de latencia obtenidos permiten analizar la relación entre este tipo de memoria y el hardware. Para tomar un punto de partida se analizan los resultados para la prueba con 15 bloques y 1024 *threads*, los datos de esta prueba se presentan en la figura 44.

De acuerdo a la manera en la que se planteó la prueba, los datos que se guardaron en memoria local se encuentran físicamente en memoria DRAM. Al realizar la operación de carga de estos datos se atraviesa toda la jerarquía de memoria de hardware de la GPU: DRAM. memoria *cache* L1 y memoria *cache* L2; luego se espera que al tomar la latencia de estos accesos se encuentren latencias de acceso a estos espacios de memoria. En la gráfica 44 se observan algunos datos agrupados en el intervalo de 0 a 100 ciclos de reloj, otros en el intervalos de 200 a aproximadamente 350 ciclos de reloj, y finalmente se observan algunos datos dispersos para más de 350 ciclos de reloj. Teniendo en cuenta la jerarquía de memoria de la GPU, se puede pensar que los datos correspondientes al primer intervalo son accesos a memoria *cache* L1, los del segundo intervalo accesos a memoria *cache* L2 y los datos que se encuentran dispersos accesos a DRAM. Más relevante aún, la distribución de probabilidad indicaría que tan probable es que al realizar la carga de un dato desde memoria local (con el dato almacenado físicamente en DRAM) la carga se realice desde memoria *cache* L1, de memoria *cache* L2 o desde la misma DRAM.

El desafío consiste en probar que los datos de cada intervalo corresponden a cada espacio de memoria de los que se asume se esta haciendo la carga del dato. Al realizar un análisis de la métrica de transacciones de memoria DRAM en un contexto de device para los datos de acceso a memoria local se obtiene la gráfica de la figura 45.

Figura 44: Datos de latencia de memoria local para la prueba con 15 bloques y 1024 *threads*.

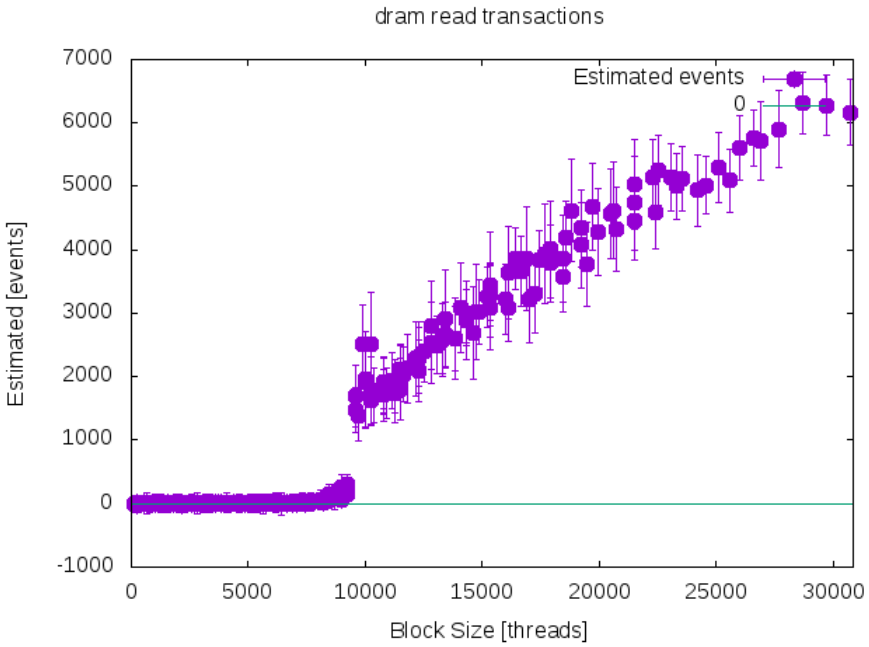


A partir de esta gráfica se puede evidenciar el número de threads a partir de los que empiezan a existir accesos a memoria DRAM. Lo cual permite separar los datos de latencia de memoria DRAM del resto. Ahora solo falta una manera de separa los datos correspondientes a los accesos de memoria *cache* L1 y L2. De los datos de memoria DRAM se puede mencionar que se evidencia una velocidad de acceso mayor que para los datos de acceso a memoria global, datos para los que los accesos a memoria DRAM se asumió estaban por encima de los 683 ciclos de reloj. Esto comprobaría en parte lo que menciona el fabricante respecto a la velocidad de acceso de memoria local, la cual se menciona es mucho más rápida que la memoria global. El análisis completo en un contexto de *device* se deja planteado de esta manera en este trabajo.

7.3.3 Jerarquía de memoria en memoria global. Para medir la latencia de *hit* en memoria *cache* L2 y en DRAM se utilizó memoria global. De la misma manera que con memoria local, los datos de latencia permiten realizar un análisis del modus operandi de este tipo de memoria en hardware. Como punto de partida se tomar[an los datos obtenidos para la prueba con 20 bloques y 640 *threads* como se observa en la figura 46.

En esta gráfica se encuentran agrupados los datos de latencia de memoria DRAM y *hit* en memoria *cache* L2. Es posible separar las latencias de cada acceso a memoria mediante el análisis de métricas. Esta distribución de probabilidad permite determinar que tan probable es que un dato que se carga desde memoria global se cargue físicamente desde memoria *cache* L2 o desde DRAM. Lo ideal sería

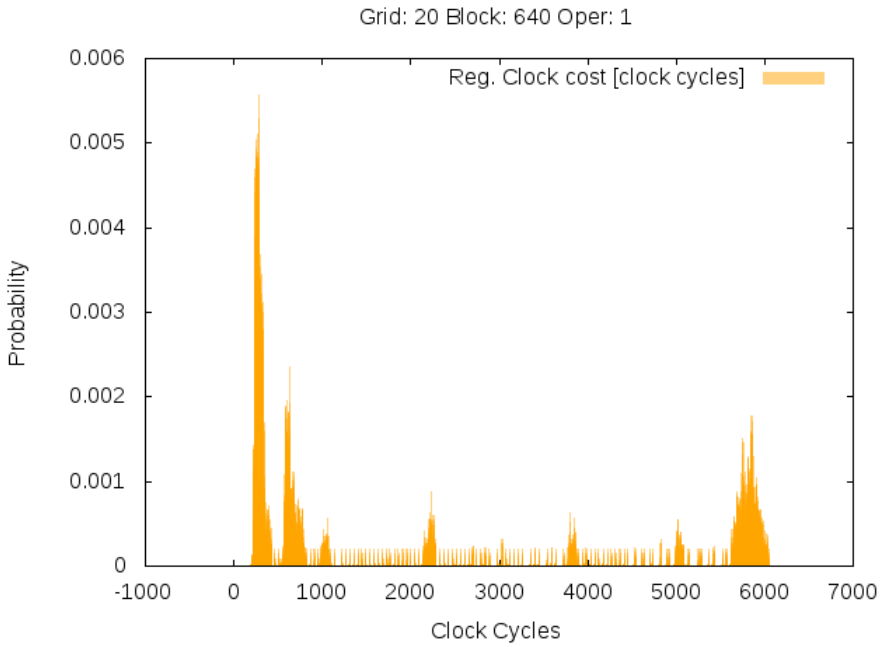
Figura 45: Datos obtenidos para la métrica de transacciones de memoria DRAM. Se observa que existe un salto en los datos cerca de los 10000*threads*. Este salto representa el punto en el que empiezan a existir accesos a memoria DRAM.



encontrar la manera de cargar el dato directamente como un parámetro de entrada del kernel, lo cual no se pudo hacer en esta prueba debido a la complejidad en el manejo de las direcciones de memoria al trabajar con PTX.

7.3.4 Modelo de ejecución basado en probabilidad de acceso a espacios de memoria. Al tener las distribuciones de las latencias de acceso a diferentes tipos de memoria (global, local) y la probabilidad de acceso desde los espacios físicos de memoria (*cache* L1, *cache* L2 y DRAM), se puede pensar en mejorar el modelo MWP-CWP teniendo en cuenta las distribuciones, de tal manera que se tenga en cuenta las probabilidades de acceso a diferentes espacios de memoria en hardware. El modelo debe tener en cuenta las condiciones en las que se realizan las cargas de memoria para plantear los *microbenchmarks* de latencia. La estructura de ejecución de las operaciones de cómputo y de acceso a memoria sería la misma del modelo MWP-CWP.

Figura 46: Latencia de accesos a memoria global para la prueba de 20 bloques y 540 *thrads*.



Bibliografía

ANDERSH, Michael, *et al.* Analyzing GPGPU Pipeline Latency. Tenth International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems, Fiuggi, Italy (ACACES' 14). July 2014.

CHENG, John; GROSSMAN, Max; MCKERCHER, Ty. Professional Cuda C Programming. John Wiley & Sons, 2014.

COLLANGE, Sylvain. Analyse de l'architecture GPU Tesla. Rapport technique, Université de Perpignan, 42-54, 2010.

HONG, Sunpyo; KIM, Hyesoon. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. En ACM SIGARCH Computer Architecture News. ACM, 2009. p. 152-163.

MEI, Xinxin, *et al.* Benchmarking the memory hierarchy of modern GPUs. En IFIP International Conference on Network and Parallel Computing. Springer, Berlin, Heidelberg, 2014. p. 144-156.

NVIDIA, "CUDA C Programming Guide v7.0"[en línea].2015 [fecha de consulta: Julio 2015]. Disponible en: http://www.nvidia.com.

NVIDIA, "CUDA Compiler Driver v7.0"[en línea].2015 [fecha de consulta: Julio 2015]. Disponible en: http://www.nvidia.com.

NVIDIA, "CUDA Binary Utilities driver v7.0"[en línea].2015 [fecha de consulta: Julio 2015]. Disponible en: http://www.nvidia.com.

NVIDIA, “Inline PTX Assembly in CUDA v8.0”[en línea].2017 [fecha de consulta: Julio 2015]. Disponible en: http://www.nvidia.com.

NVIDIA, “Profiler User’s Guide v8.0”[en línea].2017 [fecha de consulta: Enero 2017]. Disponible en: http://www.nvidia.com.

NVIDIA, “CUDA Parallel Thread Execution v7.0”[en línea].2015 [fecha de consulta: Julio 2015]. Disponible en: http://www.nvidia.com.

NVIDIA, “Graphics Processing Unit (GPU)”[en línea].2015 [fecha de consulta: Julio 2015]. Disponible en: http://www.nvidia.com.

NVIDIA, “NVIDIA Kepler GK110 Architecture Whitepaper” [en línea].2014 [fecha de consulta: Julio 2015]. Disponible en: http://www.nvidia.com.

PARRA, Dorfell; SALAMANCA, William; RAMÍREZ, Ana. Analytical Model to Estimate the Execution Time of a 3D Acoustic Wave Equation Implementation Using FDTD in a GPU. Tesis (Magíster en Ingeniería Electrónica), UIS. Escuela de Ingenierías Eléctrica, Electrónica y de Telecomunicaciones, 2016.

SIM, Jaewoong, *et al.* A performance analysis framework for identifying potential benefits in GPGPU applications. En ACM SIGPLAN Notices. ACM, 2012. p. 11-22.

WONG, Henry, *et al.* Demystifying GPU microarchitecture through microbenchmarking. En Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on. IEEE, 2010. p. 235-246.