UNIVERSIDAD DE SANTIAGO DE CHILE

FACULTAD DE CIENCIA

Departamento de Matemática y Ciencia de la Computación

Implementación y análisis del comportamiento de dos nuevas técnicas para mapear hilos en la resolución de problemas con dominio triangular en GPU

AMARO ANDRÉS ESCOBAR GONZÁLEZ

SEBASTIÁN JESÚS LASTRA MENA

PROFESORES GUÍA

DR. CRISTÓBAL ALEJANDRO NAVARRO GUERRERO

DRA. LORNA FIGUEROA MORALES

Trabajo de Titulación presentado a la Facultad de Ciencia, en cumplimiento a los requisitos para optar al título de Analista en Computación Científica

Santiago - Chile

2017

Universidad de Santiago de Chile
Facultad de Ciencia
Departamento de Matemática y Ciencia de la Computación

UNIVERSIDAD DE SANTIAGO DE CHILE

INFORME DE SEMINARIO DE TESIS

En relación al trabajo de titulación de los Señores Amaro Andrés Escobar González y Sebastián Jesús Lastra Mena, de la Carrera de Licenciatura en Ciencia de la Computación, titulado: “Implementación y análisis del comportamiento de dos nuevas técnicas para mapear hilos en la resolución de problemas con dominio triangular en GPU", informo que el trabajo desarrollado es de un nivel destacable y ayuda a contestar preguntas de investigación que son de gran relevancia en el área de la computación paralela con GPUs. Ademas, los resultados obtenidos tienen el potencial de contribuir, con un aumento de rendimiento, a distintos problemas del mundo profesional.

En términos específicos, informo respecto a:

1. Relevancia del tema.....
   Muy Bueno.
2. Fundamento Teórico.....
   Muy Bueno.
3. Relación entre Objetivos y Logros.....
   Muy Bueno.
4. Estructura del Trabajo.....
   Muy Bueno.
5. Relación con los Objetivos de la Carrera.....
   Muy Bueno.
6. Resultados e Implementación.....
   Muy Bueno.
7. Conclusiones.....
   Muy Bueno.
8. Bibliografía.....
   ..... Muy Bueno.

Por lo anteriormente expuesto, califico el trabajo con nota 7.0 (siete).

Cristóbal Navarro Guerrero
Profesor Patrocinante

Santiago, Octubre 2017

UNIVERSIDAD DE SANTIAGO DE CHILE

Facultad de Ciencia
Departamento de Matemática y Ciencia de la Computación

INFORME DE TESIS

En relación al trabajo de titulación del Sr. **Amaro Andrés Escobar González** y del Sr. **Sebastián Jesús Lastra Mena** de la Carrera Licenciatura en Ciencia de la Computación, titulado: "**Implementación y análisis del comportamiento de dos nuevas técnicas para mapear hilos en la resolución de problemas con dominio triangular en GPU**", puedo indicar que se realizó un adecuado enfoque del tema con una aplicación metódica de los puntos planteados en ella; los conceptos están bien presentados, con una fundamentación teórica detallada y completa.

En términos específicos, informo respecto a:

1. Relevancia del tema..... 7
1. Fundamento Teórico..... 6
2. Relación entre Objetivos y Logros..... 7
3. Estructura del Trabajo..... 7
4. Relación con los Objetivos de la Carrera..... 7
5. Resultados e Implementación..... 6
6. Conclusiones..... 6

Por lo anteriormente expuesto, califico el trabajo con nota 6,6 (seis coma seis).

Dra. Lorna Figueroa Morales

Universidad de Santiago de Chile
Facultad de Ciencia
Departamento de Matemática y Ciencia de la Computación

UNIVERSIDAD DE SANTIAGO DE CHILE

INFORME DE SEMINARIO DE TESIS

En relación al trabajo de titulación de los Srs. Amaro Andrés Escobar González y Sebastián Jesús Lastra Mena, de la Carrera de Licenciatura en Ciencia de la Computación, titulado: “Implementación y análisis del comportamiento de dos nuevos técnicas para mapear hilos en la resolución de problemas con dominio triangular en GPU", informo que el trabajo desarrollado tiene un marco teórico acorde a la problemática y bien fundamentada la problemática. La solución cumplió los objetivos propuestos.

En términos específicos, informo respecto a:

1. Relevancia del tema..... Muy Bueno.
2. Fundamento Teórico..... Bueno.
3. Relación entre Objetivos y Logros..... Muy Bueno.
4. Estructura del Trabajo..... Muy Bueno.
5. Relación con los Objetivos de la Carrera..... Muy Bueno.
6. Resultados e Implementación..... Bueno.
7. Conclusiones..... Bueno.
8. Bibliografía..... Muy Bueno

Por lo anteriormente expuesto, califico el trabajo con nota 6,7 (seis coma siete).

ROSA BARRERA CAPOT
Profesora Informante

Santiago, Octubre 2017

Universidad de Santiago de Chile
Facultad de Ciencia
Departamento de Matemática y Ciencia de la Computación

UNIVERSIDAD DE SANTIAGO DE CHILE

INFORME DE SEMINARIO DE TESIS

En relación al trabajo de titulación de los Srs. Amaro Andrés Escobar González y Sebastián Jesús Lastra Mena, de la Carrera de Licenciatura en Ciencia de la Computación, titulado: “Implementación y análisis del comportamiento de dos nuevas técnicas para mapear hilos en la resolución de problemas con dominio triangular en GPU", informo que el objetivo de esta tesis fue conseguido dado que se realizó un adecuado enfoque del tema donde los conceptos estuvieron bien presentados.

En términos específicos, informo respecto a:

1. Relevancia del tema..... .7
2. Fundamento Teórico..... .7
3. Relación entre Objetivos y Logros..... .7
4. Estructura del Trabajo..... .7
5. Relación con los Objetivos de la Carrera..... .7
6. Resultados e Implementación..... .7
7. Formato, redacción y ortografía..... .6
8. Conclusiones..... .6

Por lo anteriormente expuesto, califico el trabajo con nota 6,8 (seis coma ocho).

Alfonso Lobos Basualto
Profesora Informante

RESUMEN

En este trabajo se presentan dos métodos de mapeo del espacio de cómputo en GPU para problemas triangulares. El primero fue desarrollado por los autores de esta memoria utilizando una funcionalidad de NVIDIA CUDA llamada Dynamic Parallelism. Con dicha funcionalidad hilos padres pueden engendrar hilos hijos, careciendo de la necesidad de la intervención de procesos de comunicación o sincronización CPU-GPU. Todo esto provoca una alta espectativa respecto *a lo que se podría desarrollar*, gracias a la potente mejora en rendimiento que se puede alcanzar utilizando esta nueva tecnología(Tang et al., 2017).

El segundo método presentado, fue desarrollado de forma teórica por el doctor Cristóbal Navarro e implementado computacionalmente por primera vez por los autores de esta memoria. Este método utiliza otro paradigma, ya que resuelve ecuaciones matemáticas óptimamente para realizar el mapeo en un sólo paso computacional.

Adicionalmente, se implementan y adaptan dos métodos conocidos y utilizados por otros autores para resolver problemas con dominios triangulares en GPUs, los que corresponden al método Bounding Box y método Lower Triangular Matrix. Junto con la implementación, se llevan a cabo experimentos y se realiza la comparación de estos dos últimos métodos con el método Dynamic Parallelism y el método Flat Recursive.

Todos los métodos mencionados cumplen una función de mapeo del espacio de cómputo de la GPU al dominio del problema, que en éste caso tiene forma triangular. Al observar problemas en el ámbito del High Performance Computing (HPC), resulta esencial obtener una técnica de mapeo que sea eficiente. Es por esto que se presenta un análisis de resultados de eficacia y rendimiento de los métodos evaluados en distintas plataformas computacionales.

ABSTRACT

Two methods of mapping the space of computation from a GPU to a triangular domain problem are presented in this report, one of them was developed by the authors of this report and the second one was developed theoretically by PhD in Computer Science Cristóbal Navarro, but developed programmatically for the first time by the authors of this report, second of all the authors implemented two well knowns methods, one is the Bounding Box method and the second one is the Lower Triangular Matrix method and a comparison is made between all of them. The first algorithm is called Dynamic Parallelism Method, and use a tool from from NVIDIA CUDA that is called Dynamic Parallelism, that has the capability to use recursive calls in GPU without the need of CPU, the second algorithm is called Flat Recursive Method, and use another paradigm, because use mathematics formulas to complete the mapped space in one computing step.

AGRADECIMIENTOS

Mirando el camino recorrido, no queda nada más que agradecer a mis padres,que sin ellos no hubiese podido llegar hasta donde he llegado, en especial a mi madre que tanta paciencia ha tenido en los momentos dificiles, a mis amigos, que han soportado mi mal genio, a mi compañero de tesis, con quien hemos transitado este camino tormentoso que es a veces el trabajar en grupo más aún en este tipo de desafios que marcan un antes y un después en nuestras vidas, agradecer a los profesores de la Universidad, pero en especial a nuestro profesor guía Cristóbal Navarro, quien creyó en nosotros desde el primer momento, incluso cuando no teníamos definido exactamente lo que íbamos a hacer. Sé que quedan personas por nombrar, pero deben saber que siempre estaré agradecido con todos, por ayudarme a ser la persona que soy el día de hoy y aportar su granito de arena para poder completar ésta tarea.

Amaro Andrés Escobar González

AGRADECIMIENTOS

Resulta difícil a esta altura agradecerles a todos por su esfuerzo y paciencia: polola, familiares, amigos, colegas, entre otros. El camino ha sido muy largo y a veces demasiado complicado, muchas veces hasta dejamos de creer en nosotros y en nuestro trabajo, pero principalmente no puedo hacer más que darte las gracias, Cristóbal, gracias a tu paciencia, pedagogía y constancia, es que hoy estamos entregando este trabajo. El mérito también es nuestro, por supuesto, pero sin un buen lider, las cosas no hubiesen salido de la misma forma.

Gracias a mi compañera de vida, por haber tenido tanta paciencia todos estos meses, por haberme ayudado a pararme cuando no tenía fuerzas ni ganas de terminar esto. Ahora ambos alcanzamos esta meta, espero que alcancemos muchas más juntos. **Te amo.**

Al mi tribu, por darme la oportunidad de estar aquí. Por haber luchado siempre para entregarnos lo mejor, pero siempre pensando en el ***todo para todos***. Para mi han sido siempre un ejemplo a seguir y es así como intento llevar mi vida. Los amo infinitamente.

A mis amigos, por siempre dar alegrias y compañía. Se siente bien tener gente que no sea tu propia familia y que te quiera.

Al guata, por bancarme tanto tiempo. Te quiero caleta.

Sebastián Jesús Lastra Mena

Índice general

Índice de figuras

Índice de tablas

INTRODUCCIÓN

En el transcurso de la última década la evolución mundial de las GPUs ha resultado en el posicionamiento de éstas como un factor de suma importancia en el campo del High Performance Computing (HPC) (Navarro et al., 2014), debido a su bajo costo monetario y a su alto poder de procesamiento en paralelo, obteniendo resultados en un periodo de tiempo mucho más acotado comparado con hardware multi-core (CPU). Si bien hay algoritmos que son relativamente simples, hay otros ejemplos que dejan de ser triviales y la implementación de los mismo requiere de una mejora en su performance y esto se alcanza mediante el paralelismo.

Por otro lado, se ha desarrollado una importante masificación de estas tecnologías pero en otro nivel: para el uso doméstico y cotidiano a través de la industria de los videojuegos. Esto se realizó incorporando tecnologías con mayor capacidad de procesamiento para alcanzar resultados que permitan simular con mayor perfección y fluidez nuestra realidad. Junto con esto, en algunas tarjetas se agregan herramientas que permiten escribir programas que se ejecutan en paralelo, generando un impacto importante debido a que se abre una oportunidad para iniciarse en el campo del HPC sin necesitar, por ejemplo, una gran inversión de capital para la compra de un supercomputador. Una de estas herramientas es CUDA, que la podemos describir como una arquitectura e interfaz de bajo nivel para implementar algoritmos en las tarjetas de video.

CUDA se basa en llamadas a funciones que realizan trabajo en la GPU, las cuales se denominan kernels. Dentro de cada kernel es donde se realiza el cómputo del problema que se intenta resolver. Cada kernel cuenta con tres estructuras: threads, blocks y grids. Un grid está compuesta por blocks, los que a su vez están compuestos por threads. Estas estructuras representan el espacio de cómputo de la GPU, el cual debemos mapear y amoldar al dominio del problema que se debe resolver. A esto se le denomina mapeo del espacio de cómputo al dominio del problema.

La mayoría de los problemas comunes pueden ser encapsulados en espacios de cómputo cuadrados o rectangulares, son los llamados *problemas de tipo caja*. Muchas veces estos espacios se ven representados por las figuras antes mencionadas, pero también existen problemas que no pueden ser modelados utilizando formas cuadradas. Un ejemplo de aquello, son los problemas de dominio triangular o *td-problems* como es el problema de los NCuerpos o problemas basados en interacciones sobre espacios estructurados. Para solucionar los *td-problems* existen variadas técnicas de mapeos conocidas, tales como bounding box strategy (BB), recursive partition (REC) y lower triangular mapping (LTM)(Navarro & Hitschfeld, 2013), entre otros. Todas estas técnicas apuntan a resolver el problema principal que se da con éste tipo de espacios triangulares, el cuál es el gasto excesivo de recursos en la GPU que no computan información y se

descartan en ejecución. En la misma línea, esta memoria aporta con una nueva técnica de mapeo basada en Dynamic Parallelism, la cual es una funcionalidad especial en el modelo de programación de CUDA. Además de la implementación de un nuevo método propuesto por (Navarro et al., 2016) llamado flat-recursive.

Esta memoria realiza la comparación de los métodos anteriormente mencionados, todos implementados en CUDA, utilizando sólo la GPU para determinar el valor más óptimo en términos de tiempo de ejecución, buscando la mejor versión de cada algoritmo. Además de realizar una evaluación de la técnica Dynamic Parallelism en el área de mapeos de espacios de cómputo para problemas triangulares, la cual, hasta la fecha de defensa de esta memoria no ha sido utilizada en este campo. Para ésto se explican cada una de las características del hardware involucrado, así como las herramientas utilizadas para llevar a cabo los métodos antes mencionados. Finalmente, se explica cómo funciona cada método y también se analiza la optimización requerida en algunos casos.

ANTECEDENTES Y MOTIVACIÓN

Al buscar el tema de tesis a desarrollar, siempre fue un objetivo centrarse en un área de la ciencia de la computación que, además de llamar poderosamente la atención, fuese un área no explorada por los autores hasta entonces, pero que tuviera matices de herramientas o conceptos conocidos y manejados por ambos, buscando que la curva de aprendizaje no fuera tan pronunciada. Así fue como el dr. Cristóbal Navarro nos propuso explorar un campo reciente de la investigación computacional, como lo es el cómputo paralelo utilizando GPU, la que gracias a su alta capacidad de procesamiento paralelo y su arquitectura, permiten estudiar y analizar conceptos aprendidos a lo largo del proceso de formación de pregrado, como lo es puntualmente en este caso la programación en *C++* y la programación Paralela. Todo esto enfocado a involucrar teoría respecto a CPU y GPU, el aprendizaje de una API potente como lo es CUDA y utilizarla para aprovechar la tecnología desarrollada recientemente por NVIDIA, la funcionalidad Dynamic Paralellism.

Por otro lado, fue un objetivo encontrar un problema real y relevante para los autores de esta memoria, para su profesor guía y por sobretodo, para la comunidad científica en general. Cristóbal tomó el conocimiento y el área en la cual se desarrolla como docente y propuso enfocar el trabajo en problemas computacionales que poseen una forma particular en el dominio de datos. Dichos problemas resultan costosos para la GPU resolverlos.

Resulta importante para el desarrollo del ser humano resolver este tipo de problemas en menor tiempo, por tanto el presente trabajo posee un valor agregado en sí mismo.

Llama la atención que pese a que la programación en GPU no es algo nuevo, no existe tanta investigación al respecto. Creemos que se debe a la complejidad del tema, pero no se considera el gran aporte que ello podría significar. Por lo mismo, existe una serie de métodos, en su mayoría matemáticos, que realizan optimizaciones al mapeo de los problemas triangulares. Un ejemplo de ello es (Navarro et al., 2016), quien además posee otras publicaciones en el área en los que propone más de una solución (dos de los cuales están presentes en esta memoria). Todo esto resulta interesante de analizar y sobre todo se presenta un objetivo desafiante, ya que pone a prueba nuestra capacidad de aprendizaje y análisis en un problema real, aplicando un paradigma computacional aprendido en el transcurso de la carrera, pero ahora aplicado en otro tipo de dispositivo, los cuales poseen una tecnología en constante desarrollo y expansión. Se exploraron los límites de la programación paralela particularmente el desarrollado por NVIDIA CUDA Dynamic Parallelism, comparándolo con algunos métodos conocidos y poderosos, obteniendo sus rendimientos para compararlos y determinar el óptimo bajo distintas configuraciones, todo ésto evaluado en un servidor de HPC orientado a procesamiento computacional en GPU del Departamento de Matemática y Computación (DMC) de la Universidad de Chile, acceso proporcionado y supervisado por Cristóbal Navarro.

DESCRIPCIÓN DEL PROBLEMA

El área del HPC (High Performance Computing) está siempre en búsqueda de nuevos métodos para la solución de problemas tan complejos como (Stone, 2010)(NVIDIA, 2017e) que requieren tanto poder de cómputo que las soluciones basadas en supercomputadores, e incluso granjas de supercomputadores no pueden cumplir el desafío en un tiempo viable para los científicos, ya que incluso esta capacidad de cómputo no es suficiente.

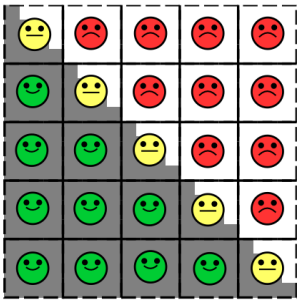
Ésta gama de problemas es amplio, pero algunos tienen la cualidad de ser paralelizables, e incluso cabe la posibilidad de ser altamente paralelizables (embarrassing parallelling), lo que quiere decir que no precisan o requieren muy poco de algún resultado anterior, lo que los hace ideales para sistemas de hardware paralelos, o en nuestro caso, para la GPU.

Además de ser paralelizables, estos problemas, como todos los demás, tiene un dominio, que abarca todo el problema y este mismo tiene una forma determinada. En algunos casos son formas comunes, como por ejemplo cuadradas, las cuales la GPU se puede adaptar

fácilmente, aumentando inmediata y significativamente el resultado de ellos (Schatz et al., 2007). Por otro lado, existen problemas con un dominio más complejo, con formas que no son fácilmente adoptadas por la GPU. Un ejemplo de ello son los problemas de dominio triangular.

Los problemas triangulares, como su nombre lo indica, son problemas en los que su dominio tiene forma triangular. Debido a esto, la GPU no logra adaptarse completamente a su dominio y desperdicia muchos recursos al momento de computar este tipo de problemas, si se utiliza la forma normal de mapeo (la cual tienen forma de array, matrices, vectores, entre otros). Es más, si por ejemplo se considera cualquier problema triangular de dos dimensiones, al utilizar el mapeo de la GPU, es decir, el mapeo más trivial, como se observa en la figura 1.1 automáticamente se pierde un $50\,\%$ de los recursos de la GPU.

Figura 1.1: Problema del dominio triangular.



Si se tiene en cuenta lo anterior, si lo que se quiere lograr es un mejor rendimiento en GPU de este tipo de problemas, es necesario crear una técnica de mapeo especial, la que se amolde al espacio de cómputo de la GPU, obteniendo la menor cantidad de pérdida de recursos. Esto ayuda inmediatamente al rendimiento de la solución de los problemas, permitiendo así determinar la mejor versión del mismo.

SOLUCIÓN PROPUESTA

En el presente trabajo de título, se propone un nuevo método para el mapeo de problemas triangulares. Dicho método se denomina método Dynamic Parallelism. El algoritmo se desarrolló utilizando una nueva funcionalidad incorporada en CUDA 5[1] (NVIDIA, 2013), llamada Dynamic Parallelism, la cual tiene la característica de invocar kernels sin la ayuda de la CPU,

[1] https://devblogs.nvidia.com/parallelforall/introduction-cuda-dynamic-parallelism/

disminuyendo el overhead presente en la comunicación entre los dispositivos, pero también entregando completa independencia a la GPU en el control de su concurrencia[2]. Las aplicaciones no estructuradas son aplicaciones que poseen flujos irregulares y en memoria trabajan siempre sobre los mismos miles de millones de datos, los cuales no están estructurados. Ejemplos de esto es la simulación de combustión utilizando refinamiento de mallas adaptativas, la generación de árboles en la indexación de información en motores de búsqueda y, en general, todos los problemas que poseen dependencia de información(DiMarco & Taufer, 2013).

Adicionalmente, se realizó la implementación y adaptación de dos métodos desarrollados por el doctor Cristóbal Navarro, los cuales tienen por nombre inverse-square-root y flac-recursive, éste último siendo implementado por primera vez en ésta memoria, ya que está en proceso de publicación. Incluso, se realizó el mismo proceso con el método bounding-box, el cual es el método de fuerza bruta. Cada método utiliza funciones implementadas en base a soluciones matemáticas y que optimizan el desempeño del algoritmo resolviendo problemas costosos en cómputo, como lo suele ser por ejemplo resolver una raíz cuadrada, en este caso muy utilizada en el cálculo de la distancia. Ejemplo de ésto, es lo mencionado por (Trapnell & Schatz, 2009a), respecto al análisis de secuencias de ADN y la importancia en encontrar la distancia entre ellas. Muchas implementaciones en CUDA han ayudado a optimizar problemas que anteriormente no tenían una solución tan accesible y con tanto soporte a nivel mundial. Es indiscutible el éxito de esta tecnología, ya que incluso existen hasta ofertas laborales que buscan a programadores con experiencia en la librería de NVIDIA.

Algo a tener en cuenta es que hasta donde alcanza el estado del arte de ésta memoria, no existe una comparación entre tantos métodos orientados al mapeo de problemas triangulares, más aún, se desconoce el comportamiento de Dynamic Parallelism en éste tipo de problemas y su rendimiento versus los demás métodos, que además de no utilizar recursividad, utilizan fórmulas matemáticas para alcanzar el mapeo completo del espacio de cómputo al dominio del problema.

OBJETIVO GENERAL

Investigar, implementar, optimizar y comparar resultados de algoritmos de mapeo que resuelven problemas de dominio triangular utilizando la plataforma de procesamiento paralelo de NVIDIA, CUDA.

[2]YANG2015

Objetivos Específicos

1. Investigar sobre los métodos existentes que resuelven problemas de dominio triangular.
2. Investigar tecnologías y conceptos necesarios para plantear una nueva solución.
3. Construir algoritmo método Dynamic Parallelism.
4. Analizar desempeño del método Dynamic Parallelism.
5. Implementar algoritmo método Flat Recursive.
6. Analizar desempeño del método Flat Recursive.
7. Optimizar los métodos para así lograr el máximo rendimiento de cada uno de ellos.

ALCANCES Y LIMITACIONES

Alcances

Uno de los alcances pricipales es el estudio del comportamiento de las diferenctes técnicas de mapeo, dos de las cuales no han sido implementadas anteriormente. Además, las técnicas de mapeo presentes en esta memoria pueden ser utilizadas en cualquier dispositivo que sea compatible con CUDA/Dynamic Parallelism. Por otro lado, las técnicas son independientes del problema a resolver, lo que quiere decir que mientras sea de dominio triangular, las técnicas de mapeo son válidas.

Limitaciones

Una de las limitaciones principales es el límite de memoria física de las GPU, lo que genera que no se pueda experimentar con un N mayor a 32768 e incluso en algunos casos con un N mayor a 16384. En términos de hardware, las técnicas de mapeo obtenidas sólo puede ser trabajadas en una tarjeta de video NVIDIA, que soporte CUDA y Dynamic Parallelism.

CONTRIBUCIÓN DEL TRABAJO

Algunas de las contribuciones principales, son el estudio del comportamiento de las diferentes técnicas de mapeo bajo distintas configuraciones en su ejecución. Además se realizó la adaptación de los métodos para ser lanzados considerando la diagonal y para ser ejecutados en función de los *blocks*. Si bien el método Flat Recursive fue publicado por (Navarro et al., 2016), hasta la fecha no existe implementación en CUDA.

Cabe destacar la relevancia del método Dynamic Parallelism, ya que no ha sido implementado hasta la fecha y, dado sus resultados, presenta una importante evidencia para tomar decisiones en torno a las investigaciones que se llevan o pudiesen llevar a cabo en el futuro.

Además, las técnicas de mapeo presentes en esta memoria pueden ser utilizadas en cualquier dispositivo que sea compatible con CUDA/Dynamic Parallelism, incluso se podría pensar en modificaciones menores para lograr ejecuciones en cluster de GPUs de HPC.

Finalmente, los métodos fueron desarrollados y programados para ser ejecutados independiente del problema a resolver, lo que quiere decir que mientras sea de dominio triangular, los métodos funcionarán.

ORGANIZACIÓN DEL DOCUMENTO

El primer capítulo del presente trabajo corresponde a la introducción al tema que se desarrolla posteriormente.

El segundo capítulo corresponde a la contextualización teórica que se lleva a cabo mediante el marco teórico. En dicho capítulo se presenta información necesaria para el lector y el entendimiento en líneas generales del trabajo. Esto se desarrolla siempre en dos niveles, buscando entregar líneas generales sobre un tema, pero además aportando antecedentes vitales y específicos sobre algunos tópicos y tecnologías.

En el tercer capítulo se presenta una introducción a los problemas de tipo triangular con el objetivo de interiorizar un poco más en el tema central.

El capítulo cuarto describe los métodos existentes. Notar que se incluye el método Flat Recursive, pero en su implementación computacional, sin embargo, se aportan antecedentes de este método también.

Por otro lado, el quinto capítulo presenta la solución utilizando Dynamic Paralellism, para luego en el sexto capítulo analizar los resultados obtenidos de los experimentos, junto con la metodología explicada en el análisis.

Finalmente el séptimo capítulo muestra al desarrollo de las conclusiones, y el octavo capítulo los posibles trabajos futuros que se desprenden de esta memoria.

MARCO TEÓRICO

CONCURRENCIA Y PARALELISMO

Pese a que los términos concurrencia y paralelismo están relacionados, existen diferencias fundamentales que permiten entender mejor estos conceptos. Desde la aparición de los procesadores multicore para uso general y/o masivo, estos dos conceptos se han utilizado con mayor frecuencia por la comunidad dedicada al estudio de la Ciencia de la Computación, sobre todo por el área del High Performance Computing (desde ahora HPC)(Navarro et al., 2014). Pese a esto, hay otras áreas de la computación en las que existe cierta confusión entre sus diferencias, o en si efectivamente existe alguna (Breshears, 2009).

Dada la importancia de ambos conceptos para el presente trabajo, resulta prioritario definirlos y explicarlos.

Concurrencia

La concurrencia no es un concepto que está ligado sólo a la computación, es un fenómeno natural presente en nuestro día a día. Muchas cosas suceden al mismo tiempo y muchas de ellas ni siquiera las percibimos. Cuando arranca el sistema operativo, se deben preparar y disponibilizar los dispositivos y servicios para que puedan ser utilizados por el usuario. Si estos procesos no tuvieran concurrencia en algún nivel, el tiempo de carga sería mayor. Es por esto que el procesamiento concurrente cobra especial importancia en el diseño de los sistemas operativos.

Un buen caso a analizar para comprender la principal diferencia entre concurrencia y paralelismo, es el comportamiento que se genera al ejecutar una aplicación concurrente en procesadores que poseen solamente un core. En todo sistema operativo, el orden de ejecución de cada tarea lo determinan los algoritmos y las políticas de prioridades establecidas en el scheduler del kernel del mismo(Stallings, 2009). De esta forma, si se ejecuta una aplicación concurrente en alguno de los procesadores con las características recién mencionadas, en algún momento existirán dos o más threads en ejecución y el sistema operativo priorizará, en base a algún criterio, y ejecutará secuencialmente los procesos, generando la ilusión de que se están ejecutando simultáneamente(P. & H., 2012). Por otro lado, en el caso de los procesadores con más de un núcleo, dicha ejecución no tendría solapamiento en sus procesos, dado que cuenta con infraestructura para ejecutarlos simultáneamente.

Una definición consistente y formalmente aceptada de concurrencia, es lo mencionado por (Navarro et al., 2014): Concurrencia es la propiedad de un programa (etapa de diseño), en donde dos o más tareas pueden estar en progreso simultáneamente.

Paralelismo

Los programas paralelos son diseñados para tomar ventaja de las múltiples unidades de procesamiento disponibles, con el objetivo de resolver problemas computacionales complejos en menor tiempo. Es gracias a esto que la GPU y su arquitectura se abren terreno, mostrándose como una alternativa atractiva para los científicos ya que ofrece una enorme capacidad de procesamiento al costo de un computador de escritorio (Navarro et al., 2014).

La clave a la hora de diseñar algoritmos paralelos es reducir la dependencia entre los datos, para que sea posible realizar operaciones en unidades de cómputo independientes, intentando reducir el costo en tiempo que provoca la comunicación entre procesos y sobre todo lograr utilizar todos los recursos computacionales disponibles (Cao, 2011).

El paralelismo es un tipo de concurrencia, es decir, el paralelismo es un subconjunto dentro del conjunto de la concurrencia(Breshears, 2009). En otras palabras, los algoritmos paralelos generan concurrencia, pero no todos los procesos concurrentes son ejecutados paralelamente.

Se puede convenir que una definición consistente y formalmente aceptada de paralelismo es la propiedad de un programa (tiempo de ejecución), en donde dos o más tareas se ejecutaron simultáneamente (Navarro et al., 2014).

PROGRAMACIÓN PARALELA

La programación paralela es la apuesta para lograr tiempos de ejecución menores en aplicaciones desarrolladas para encontrar soluciones a problemas presentes en distintas áreas, sobre todo en el área de la medicina. Algunas de las tecnologías de cómputo paralelo más recientes, fueron diseñadas específicamente para ser utilizada en aplicaciones de bioinformática. Algunos ejemplos son: analizar secuencias biológicas apuntando a crear nuevas drogas para curar enfermedades y condiciones médicas; o la caracterización funcional y estructural de genes y proteínas para entender y fundamentalmente influenciar procesos biológicos(Grama, 2003); y también para optimizar y obtener mejores resultados en el cálculo de secuencias de ADN(Trapnell

& Schatz, 2009b). Éstos son algunos de los problemas que pueden y necesitan ser resueltos diseñando algoritmos paralelos eficientes, precisamente para obtenerlos en un tiempo menor y poder simular procesos cada vez más complejos.

En términos simples, la programación paralela se define como un conjunto de técnicas utilizadas para resolver un gran problema, dividiéndolo en sub-problemas mucho más pequeños, resolviendo cada uno simultáneamente en un procesador físico diferente(Cao, 2011).

Existen dos formas de dividir el problema en cuestión e identificar cuál técnica es mejor, es un paso clave al momento de diseñar algoritmos paralelos. Podemos decir que PD es un **problema de paralelización de información** si D está compuesta de di subconjuntos de elementos iguales en tamaño, y si para resolver el problema se requiere aplicar una función f(di) a todo el dominio de datos. Esto puede quedar representado utilizando la siguiente ecuación:

$$f(D) = f(d_1) + f(d_2) + ... + f(d_k) = \sum_{i=1}^{k} f(d_i) \tag{2.1}$$

Este tipo de problemas son ideales para resolverlos con la ayuda de una GPU debido a que con dicha arquitectura se obtienen mejores resultados cuando todos los hilos ejecutan las mismas instrucciones, pero aplicadas en subdominios de información más pequeños.

Por otro lado, podemos decir que PD es un **problema de paralelización de tareas** si D está compuesta por di funciones, y si para resolver el problema es necesario aplicar cada función a un dominio común de información S. Esto puede quedar representado con la siguiente ecuación:

$$D(S) = d_1(S) + d_2(S) + ... + d_k(S) = \sum_{i=1}^{k} d_k(S) \tag{2.2}$$

Este tipo de problemas son ideales para resolverlos utilizando la arquitectura CPU porque, dentro de sus características, permite que diferentes tareas sean ejecutadas en cada hilo.

MEDIDA DE DESEMPEÑO: SPEEDUP

Cuando se modelan problemas utilizando computación paralela, siempre es necesario responder preguntas similares a ¿Cuánto efectivamente mejoró el desempeño del algoritmo utilizando programación paralela Existe un consenso general al respecto(Sahni & Thanvantri, 2002):

$$\frac{tiempo_s}{tiempo_p} \tag{2.3}$$

Donde tiempo S es el tiempo de ejecución secuencial y tiempo P es el tiempo de ejecución paralelo.

Las medidas de desempeño permiten encontrar las mejores configuraciones en la ejecución de los algoritmos paralelos, posibilitando obtener, por ejemplo, la cantidad óptima de threads como también el valor asociado al peor desempeño.

El speedup es la proporción existente entre el tiempo de ejecución del mejor algoritmo secuencial, versus su mejor versión paralela. Muchas veces se utiliza como un valor expresado en porcentaje, pero generalmente se utiliza como un factor de mejora. Esto se explica ya que es más intuitivo leer que el algoritmo tuvo un speedup de 3x, fácilmente se puede inferir que tomó tres veces menos tiempo respecto a su versión secuencial.

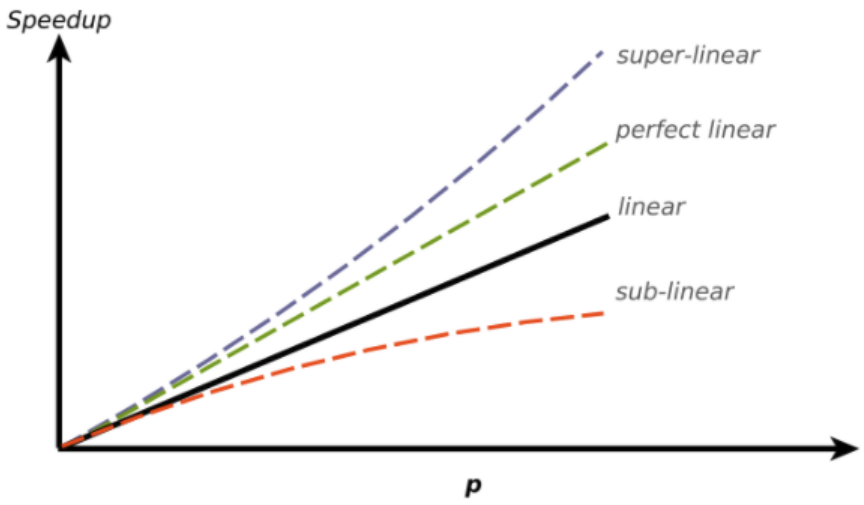
Sea n el tamaño del problema, $T_s(n, 1)$ el tiempo de la mejor versión secuencial, p el número de procesadores, y $T(n, p)$ el tiempo de la mejor versión paralela, tenemos que el speedup queda expresado como:

$$S_p = \frac{t_i(n, 1)}{t(n, p)} \tag{2.4}$$

Existen cuatro posibles escenarios al medir el speedup de un algoritmo:

1. Speedup lineal: si el valor del speedup aumenta linealmente en función de p.
2. Speedup ideal: ocurre cuando se obtiene el máximo desempeño teórico, cuando n es fijo.
3. Speedup sublineal: es lo que comúnmente encontramos en los algoritmos, desempeños que no son malos pero que distan un poco a los óptimos.
4. Speedup superlineal: pese a que no existe un consenso respecto a la real existencia de este escenario, algunos autores apoyan la idea de que es posible alcanzar resultados de speedup mejores que los teóricamente esperados si se considera una definición más general de Sp, en vez de considerar la proporción del tiempo se considera la velocidad(Gustafson, 1990).

Figura 2.1: Diferentes tipos de speedups.



Existen tres formas de medir el speedup:

1. **Tamaño fijo:** corresponde a la forma recién explicada y es la utilizada en este trabajo.
2. **Escalado:** consiste en hacer variar n y p, tal que el tamaño del problema se mantenga constante por procesador.
3. **Tiempo fijo:** consiste en hacer variar n y p, tal que la cantidad de trabajo por procesador se mantenga constante.

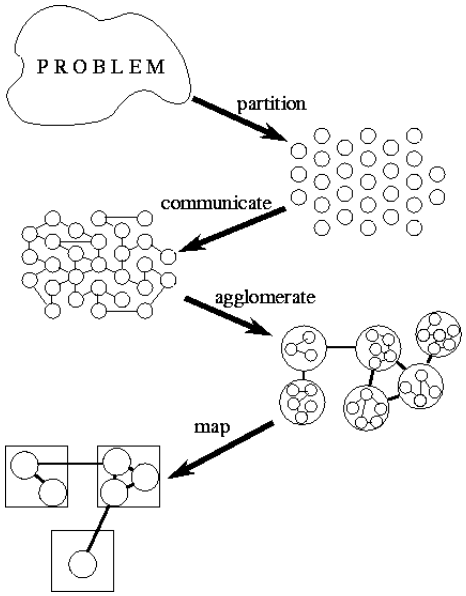
ESTRATEGIA PARA DISEÑAR ALGORITMOS PARALELOS, PCAM IAN FOSTER

Los primeros intentos utilizando GPUs para usos no gráficos, fueron estudios relacionados a ecuaciones diferenciales. Trazado de partículas, simulación de fluidos, efectos ópticos, entre otros[20], son ejemplo de los distintos usos que tienen los algoritmos paralelos. Es por esto que resulta importante diseñarlos bien desde el inicio.

Es así como (Foster, 1995) identifica una estrategia que consiste en prestar atención a cuatro pasos que aparecen con frecuencia en la etapa de diseño de algoritmos paralelos. Estas etapas son: particionado, comunicación, aglomeración y mapeo. Gracias a las iniciales de cada

etapa, es conocida bajo la sigla PCAM. Esta estrategia no es una receta, es más bien una metodología que permite identificar los elementos que deberían estar presentes en cada algoritmo paralelo. Pese a esto, cada paso cuenta con un listado de preguntas que deberían verificarse luego de realizar cada análisis. Este trabajo no tiene como objetivo describirlas en detalle, ni tampoco describirlas, se menciona sólo como referencia para el lector.

Figura 2.2: Diagrama algoritmo Ian Foster



Particionado

La ejecución concurrente de diferentes tareas, utilizando todos los recursos computacionales disponibles, es la clave de todo proceso paralelo. Si no se utilizan todos los procesadores, se verá una caída en la performance(Cao, 2011).

Teniendo en consideración lo anterior, se puede inferir que el principal objetivo de esta etapa es encontrar la mejor partición posible, la que genere la mayor cantidad de subproblemas, aprovechando así todo el hardware disponible.

Antes de particionar el problema, se debe identificar el dominio del problema a paralelizar, es decir, si se trata de un problema de paralelización de información se debe realizar una descomposición del dominio del problema, es decir, dividir la información en pequeñas porciones del mismo tamaño para luego asignar un conjunto de instrucciones que operarán en cada subdominio. Por otro lado, si se tratase de un problema de paralelización de procesos es necesaria una

descomposición funcional, en otras palabras, se deben identificar los procesos y/o instrucciones para distribuirlas entre los procesadores. Pese a que se identifican diferencias claras entre los dos tipos de problemas, un algoritmo podría contener ambos tipos de problemas implementados.

Comunicación

Luego de dividir la información, es necesario enviarla-recibirla a los sub-problemas. Pese a que existe un grado de independencia en la ejecución de los procesos, la ejecución general no es independiente completamente, ya que muchas veces un proceso requiere datos asociada a otro proceso, por lo que se necesita realizar una transferencia de información para lograr la comunicación entre los procesos o tareas.

Existen dos tipos de comunicación:

1. Comunicación local: cada proceso puede comunicarse con un listado pequeño de otros procesos vecinos, pudiendo utilizar algún patrón geométrico.
2. Comunicación global: cada proceso puede comunicarse con todo el resto de los procesos. Esto muchas veces obliga a identificar posibles casos de borde o procesos de sincronización que aseguren que todo funciona correctamente.

Aglomeración

Algo esperado al ejecutar el algoritmo paralelo es que existan tantas tareas como procesadores o hilos disponibles, asumiendo que no tendrá procesos desocupados a lo largo de su ejecución. El concepto de granularidad ayuda a mejorar la performance de un algoritmo paralelo, identificando la cantidad de cómputos que se le asigna a cada proceso. A mayor granularidad, mayor cantidad de cómputos. Por el contrario, si la cantidad de cómputos es menor, menor es la granularidad. Con este antecedente, la granularidad de un proceso debe ser lo suficientemente grande como para ejecutar el proceso completo, pero lo suficientemente pequeña para controlar situaciones que podría presentarse al tener muy pocos datos.

Mapeo

Es posible encontrarse con casos en donde una alta granularidad implique volver a distribuir y lanzar procesos. Para lograr esto será necesario reasignar ejecuciones, combinando

algunas tareas con otras más grandes, por ejemplo. Cada tarea debe ser asignada a un procesador de manera eficiente, es decir, intentar disminuir al máximo la latencia en tiempo que puede provocarse con la comunicación entre los procesos, maximizando la utilización de los recursos en paralelo. Al realizar el mapeo de los datos, se pueden utilizar dos estrategias:

- Aumentar concurrencia: se vuelven a dividir los subprocesos para que puedan ser ejecutados en distintos procesadores simultáneamente.
- Aumentar localidad: asociar procesos con datos en común y ejecutarlos en el mismo procesador simultáneamente.

El problema del mapeo es conocido como un problema np-completo, lo cual implica que no existe un algoritmo que entregue una solución general, sin embargo existen heurísticas que permiten optimizarlos y encontrar soluciones aceptables. El patrón más simple es el mapeo geométrico 1:1 entre aglomeraciones y procesadores.

LEY DE AMDAHL

Esta ley es utilizada para predecir el máximo speedup que puede obtener la ejecución de un algoritmo(del Rio et al., 2016). Sea f la fracción de un algoritmo que corre en paralelo, $(1-f)$ la parte que se ejecuta secuencialmente, y S_f la cantidad de procesadores disponibles(Amdahl, 1967), tenemos que:

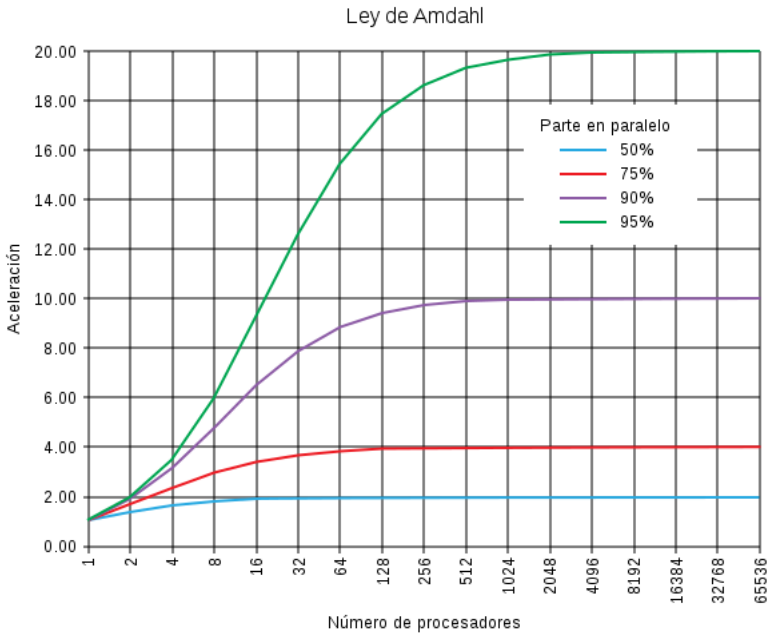
$$S(f) = \frac{1}{(1 - f + \frac{f}{S_f})} \tag{2.5}$$

Además, si $S_f = \infty$, la ecuación anterior se transforma en:

$$S(f) = \frac{1}{1 - f} \tag{2.6}$$

Esta ley considera que si en el HPC del centro de simulación ambiental de la NASA, existen clusters de computadores con una innumerable cantidad de procesadores disponibles, el mejor speedup siempre estará determinado por el tiempo de ejecución de la parte secuencial del algoritmo, entregando un panorama pesimista del cómputo paralelo, dado que asume que el tiempo de la parte secuencial es siempre igual, sin considerar el tamaño del problema como una variable. Para propósitos académicos, es común fijar el tamaño del problema y medir tiempo versus cantidad de procesadores.

Figura 2.3: Ley de Amdahl.



LEY DE GUSTAFSON

Por otro lado, si analizamos la ley de Gustafson, nos podemos dar cuenta que sí toma en consideración el tamaño del problema.

Consideremos un programa paralelo corriendo con n procesadores, y sea A y B la porción de tiempo destinado a la parte secuencial y a la paralela respectivamente (Gustafson, 1988), se tiene que:

$$T(p) = A + B \tag{2.7}$$

Y si también consideramos que el tiempo de ejecución secuencial, va a estar determinado por:

$$T_s = A + pB \tag{2.8}$$

Tenemos el siguiente speedup:

$$S(f) = \frac{A + pB}{A + B} = \frac{A}{A + B} + \frac{pB}{A + B} \tag{2.9}$$

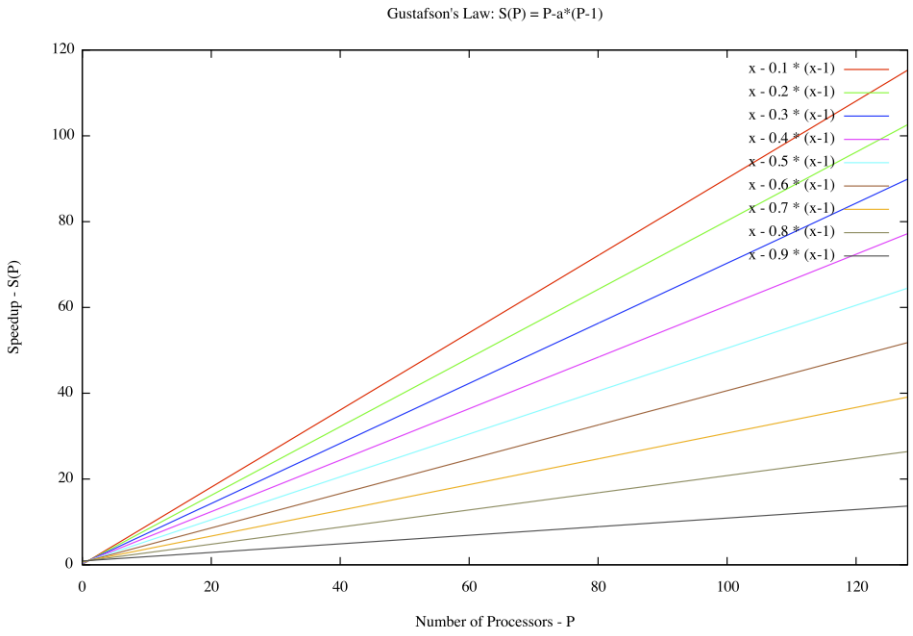
Sea $\alpha = \frac{A}{(A+b)}$ la porción de tiempo consumido ejecutando las operaciones secuenciales y $(1 - f) = \frac{B}{(A+b)}$ la parte paralela, finalmente tenemos que:

$$S(f) = \alpha + p(1 - f) = p - \alpha(p - 1) \tag{2.10}$$

Entre más pequeño o cercano a cero sea , entonces el speedup va a estar muy cerca al valor de p, alcanzando un speedup linealmente perfecto.

Esta ley ayuda a favorecer al cómputo paralelo y a promover su utilización. Algoritmos Monte Carlo para la simulación de eventos aleatorios(L Harrison, 2010) y para la optimización de algoritmos de encriptación(Flores Carapia et al., 2012), algoritmos para realizar predicciones del clima (Michalakes & Vachharajani, 2008), entre otros, son ejemplos de aplicaciones en donde el tamaño del problema podría incrementar si existe mayor capacidad de cómputo paralelo.

Figura 2.4: Ley de Gustafson.



TAXONOMÍA DE FLYNN

Existen muchas formas de clasificar la arquitectura de los computadores. Una muy utilizada es la Taxonomía de Flynn (Flynn's Taxonomy). La taxonomía de Flynn es una clasificación de arquitecturas de computadoras propuesta por (Flynn, 1972). Este esquema de clasificación divide las arquitecturas en cuatro tipos y éstas se basan en la cantidad de instrucciones concurrentes y en los flujos de datos disponibles en la arquitectura:

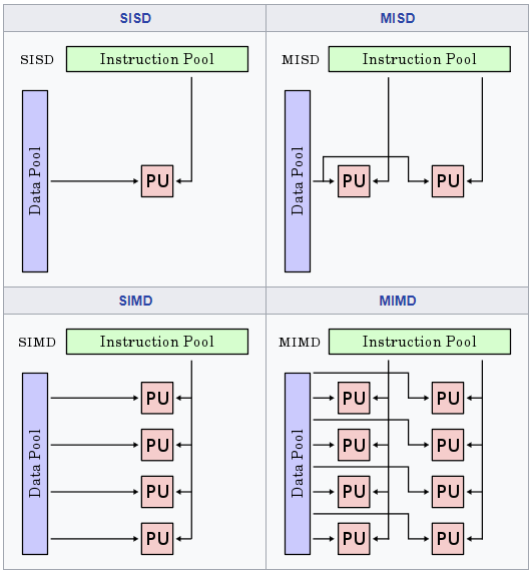
- Una Instrucción, Un Dato(Single Instruction, Single Data - SISD): corresponden a las arquitectura de un sólo núcleo, computadores equipados con procesadores de un core, es decir, cada instrucción se ejecuta secuencialmente y cada operación con los datos se realiza a través de un único flujo de datos.

- Una Instrucción, Múltiples Datos(Single Instruction, Multiple Data - SIMD): corresponde a un tipo de arquitectura paralela en la que un computador puede tener múltiples núcleos. Todos los cores ejecutan instrucciones en cualquier momento, y cada operación en múltiples flujos de datos. Los computadores vectoriales son reconocidos como SIMD, debido a la naturalidad del paralelismo de sus operaciones.

- Múltiples Instrucciones, Un Dato(Multiple Instruction, Single Data - MISD): es una arquitectura poco común. Un computador tiene muchos cores operando bajo distintos flujos de instrucción en un único flujo de datos. Este tipo de arquitectura es común encontrarla en computadores a prueba de fallos. Los sistemas heterogéneos también son MISD, ejemplo de ello es el sistema de control de vuelo del Transbordador STS (Space Transport System - Sistema de Transporte Espacial)(Spector & Gifford, 1984).

- Múltiples Instrucciones, Múltiples Datos(Multiple Instruction, Multiple Data - MIMD): arquitectura de múltiples cores independientes que ejecutan simultáneamente varios flujos de instrucciones sobre diferentes flujos de datos. Muchas de sus implementaciones incluyen ejecuciones SIMD en sus subcomponentes. Ejemplos de esta arquitectura son los sistemas distribuidos.

Figura 2.5: Diferencias de arquitecturas según Taxonomía de Flynn.



CPU VS GPU : DIFERENCIAS FUNDAMENTALES

Para ver las diferencias fundamentales entre estos dos tipos de hardware, tenemos que hacer un poco de historia sobre la creación de cada uno de ellos, el por qué se fueron moldeando con una determinada arquitectura y el papel que cumplen en el procesamiento de la información en los computadores modernos

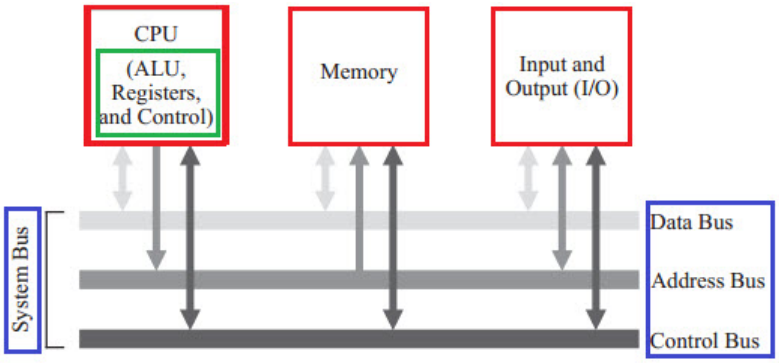
Historia de la CPU

La historia de la Unidad central de procesamiento o CPU por sus siglas en inglés viene de la mano con la historia de los ordenadores personales o PC por sus siglas en inglés, y ésta comienza a mediados de 1960, antes de este año, los computadores eran sólo accesibles por instituciones gubernamentales, universitarias o de investigación con alto poder monetario, ya que para ese entonces los computadores eran de tamaños exageradamente grandes y demasiado costosos para una persona normal (incluso para un pequeño grupo de personas), por lo tanto quienes quisieran utilizar una de estas máquinas debían hacerlo junto a los demás usuarios del mismo sistema computacional, tomando turnos para probar cada uno de sus algoritmos . Todo esto cambió el 15 de noviembre de 1971, cuando Intel lanzó el microprocesador intel 4004(Intel,

1971) , el cual tenía la característica única de contener todos los transistores que constituían un procesador en un único circuito integrado,lo que marcaba una revolución para la computación en esa época. Desde la creación de éste microprocesador, los computadores personales comenzaron a disminuir su tamaño y principalmente su costo, quedando accesible cada vez más a usuarios con menor poder adquisitivo.

Las partes esenciales de un procesador son la unidad de control, la unidad aritmética lógica (ALU). los registros y un co-procesador matemático que luego pasaría a llamarse unidad de cálculo en punto flotante.

Figura 2.6: Estructura de un Micro-procesador.



La arquitectura de la CPU está regida por la Ley de Moore, creada por Gordon Moore, cofundador de Intel, el 19 de abril del año 1965(Moore, 1965) que explica el aumento de la capacidad de los transistores con los cuales están hechos los microprocesadores cada año y el precio de los mismos se reduce de la misma forma, la sentencia fue cambiada o ajustada por (Moore, 1975) al doblaje de la capacidad de los transistores cada dos años.

El primer computador personal como tal fue lanzado el año 1974, es el llamado Altair 8800, que contenía el procesador de 8 bits Intel 8080, a quien la misma compañía llamó oficialmente el primer microcomputador(Intel, 1975), éste computador sembró las raíces para lo que sería la historia de los PC, ya que apuntaba a los usuarios que sólo querían tener un computador y no tener que ensamblarlos (era lo que se hacía comúnmente en esos tiempos), por el contrario se vendían kit armados con distintas especificaciones(MITS, 1975).

Así, siguiendo la ley de Moore, la siguiente generación los Intel 8086, eran procesadores de 16 bits, y marcaron la arquitectura hasta ese entonces e incluso hasta el día de hoy, ya que implementaron un nuevo juego de instrucciones llamado x86, el cual existe hasta los procesadores actuales y una de las características principales es que lograba retro compatibilizar el

software creado para los modelos anteriores, como el modelo 8086 era catalogado como muy costoso, Intel sacó otro modelo, el 8088 que básicamente tenía las mismas características que el 8086 pero mucho más barato.

Por otro lado y al mismo tiempo, IBM buscaba sacar al mercado lo que sería considerado el primer computador personal (PC por sus siglas en inglés) de venta masiva de la historia, el IBM PC 5150, que rondaba los 3000 dólares. Intel aprovechó el éxito de ésta computadora para sitiar la arquitectura x86, así nació la familia de los procesadores x86(Medina, 2014), la cual continúa hasta el día de hoy, pero adaptándose a los nuevos usos de la tecnología.

Es importante señalar que en estos momentos de la historia de la CPU, se cree que se está alcanzando el límite máximo de velocidad en los núcleos de los procesadores, ésto debido principalmente a dos factores, el primero y el más importante, es el calor generado por los transistores del procesador cuando estos son encendidos y apagados millones de veces por segundo para cumplir las tareas solicitadas. La explicación a ésto se debe a que los transistores en sí han reducido notoriamente su tamaño, a tal punto que en el mismo espacio físico, se ha avanzado desde cincuenta y cinco millones de transistores en el Pentium 4(Intel, 2017b) hasta el día de hoy, que están alrededor de los cuatro mil ochocientos millones de transistores (AMD, 2017). El segundo factor tiene que ver directamente con el primero, a mayor calor, mayor grado de error en los procesadores, es por esto que Intel, por ejemplo, creó los transistores de tres dimensiones, ya que éstos tienen menos poder de consumo, mayor procesamiento, generan menor calor y prueban ser una mejora significativa con respecto a sus pares de dos dimensiones(Intel, 2017a).

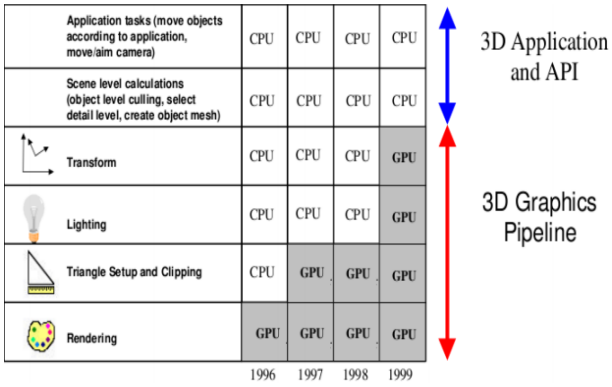
Historia de la GPU

El término Unidad de Procesamiento Gráfico o GPU por sus siglas en inglés fue creado por NVIDIA el año 1999 (McClanahan, 2010) en la presentación de lo que hasta el momento era un hito para la época, la empresa lanzaba la primera GPU tal como la conocemos el día de hoy, el modelo fue llamado NVIDIA GeForce 256(NVIDIA, 2017g), el cual marcó un antes y un después en lo que a cómputo gráfico se refiere, hasta ese entonces el procesamiento de la información era compartido por las unidades gráficas y por el procesador (CPU) de los computadores, el trabajo realizado en conjunto se puede ver reflejado en la figura 2.7, la cual es llamada línea de cómputo gráfico, y muestra los pasos necesarios para la muestra de un objeto en tres dimensiones en un espacio de dos dimensiones como es la pantalla de los computadores.

Si bien esta línea permitía a la GPU hacerse cargo de todo el proceso gráfico, tam-

bién impedía la modificación de la información por parte de los programadores una vez enviada a la línea de cómputo. Ésto dio origen a un gran problema, el cuál era la rigidez de los efectos visuales o gráficos ya que la paleta de efectos estaba incorporada en el hardware de la GPU y no a través de software.

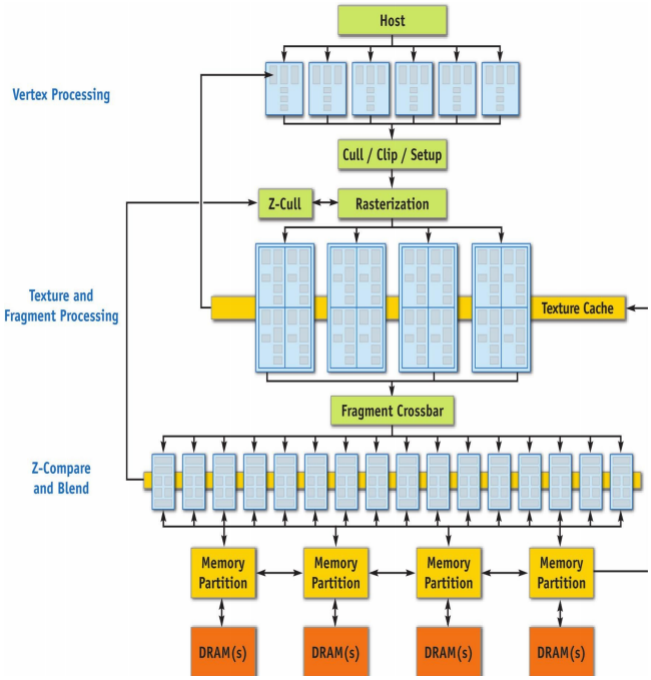
Figura 2.7: Evolución del proceso gráfico a través de los años.



Debido a éste problema de rigidez en las GPU, por el año 2001 NVIDIA lanzó la GeForce 3, junto con esta tarjeta se lanzó lo que sería la primera línea de cómputo gráfico programable en la GPU, gracias a ésto, los programadores al fin podían incorporar pequeños programas computacionales llamados shaders, los cuales daban flexibilidad a la paleta de transformaciones y a los efectos visuales que se podía utilizar hasta el momento, directamente en la GPU.

Ya en los años 2002-2003, se lanzaron las primeras GPU totalmente programables, la GeForce FX y la ATI Radeon 9700, las cuales tenían muchas ventajas versus sus antecesores, como el soporte completo de programación de shaders, operaciones por-pixel, el mapeo de la entrada y salida de datos, entre otros; Además, con el lanzamiento de DIrectX 9 se logró la primera oleada de programación en GPU pero en el ámbito computacional general, escapando de lo estrictamente gráfico/visual, creando una nueva rama de la ciencia de la computación, llamada General Purpose Computing Graphics Processing Units (GPGPU) y básicamente se dedica al estudio y el aprovechamiento de las capacidades de cómputo de las GPU(GPGPU, 2017).

Figura 2.8: Arquitectura de la NVIDIA GeForce 6



La arquitectura de la GPU hasta ese entonces podía describirse como un motor totalmente programable de vertex, uno totalmente programable de fragmentos, uno de texturas y un motor de escritura muy potente, todas éstas cualidades podrían ser vistas, desde un punto de vista no-gráfico, como un gran y potente caballo de carreras totalmente programable y con un ancho de banda poderoso los cuáles era cuestión de tiempo para ser descubiertos.

Así, en el año 2006 se dió el siguiente paso, dar a conocer a la GPU como un procesador masivamente paralelo (McClanahan, 2010), la GeForce 8800, que incorporaba la primera arquitectura unificada de gráficas y cómputo en GPU(NVIDIA, 2017a) llamado CUDA, programable en C y con diferente número de cores de procesamiento para así abarcar todo el mercado de usuarios posibles, también fue la primera en tener multiprocesadores de streams, los cuales fueron los pilares para las siguientes arquitecturas hasta la que tenemos actualmente.

Ésto abrió un mercado nuevo para NVIDIA, quienes en el año 2007 percibieron una gran demanda por sistemas computacionales basados en GPU, en respuesta la empresa lanzó una nueva gama de tarjetas GPU llamadas Tesla, las cuales se basaban en el modelo GeForce 8800 antes mencionado pero con un enfoque y configuración plena en el ámbito del cómputo paralelo.

Gracias a ésto, dos campos totalmente distintos como lo son el de la ciencia de la

computación y el del desarrollo de videojuegos se vieron beneficiados con la forma de procesamiento que iba tomando la GPU, la cual se puede denominar el modelo de procesamiento de stream, lo cual significa que la GPU está altamente optimizada para el cómputo en streams. Aún cuando el cómputo en GPU es generalmente simple y requiere mucha menos memoria que los cómputos en CPU, la necesidad de procesar la mayor cantidad de cómputo muy rápido y en paralelo es algo que debe estar siempre presente en el desarrollo de las tarjetas de video, aquí es donde entra la arquitectura de procesamiento de stream, la cual requiere poca sino nula memoria, en contraste con la CPU, que basada en la arquitectura de Von Neumann, el traspaso de memoria hacia el procesador ha sido el gran cuello de botella para la velocidad de cómputo a lo largo de los años.

El cómputo en stream es distinto al cómputo tradicional, ya que utiliza streams y kernels para lidiar con el cuello de botella del ancho de banda, un stream es básicamente una cola, la cual se crea añadiendo elementos al final de la misma, mientras los primeros elementos son los que se ejecutan primero; los kernels son pequeños trozos de programas que operan en el entorno de los streams, con lo cual pueden utilizar uno o varios streams de entrada y asimismo varios streams de salida.

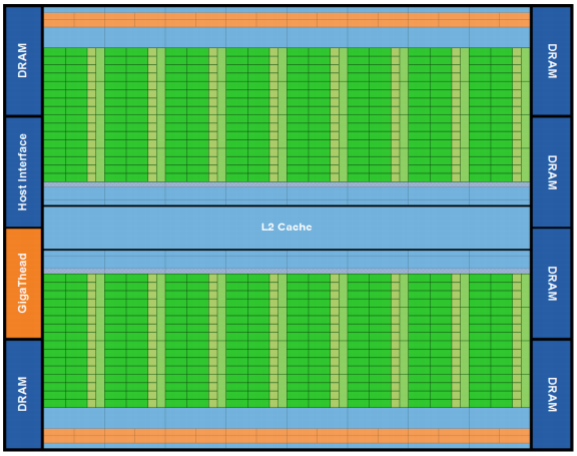
Podríamos mencionar varias razones por los cuales la arquitectura de procesamiento basada en streams tiene ventaja sobre la arquitectura CPU, la más notoria es la idea de paralelismo[asdasfd], el tomar ventaja del paralelismo les permite a las GPU un altísimo nivel de cómputo, que son necesarios para el tipo de aplicaciones enfocadas en audio y gráficas, entre otras. Tres niveles de paralelismo podemos ver incorporado en cómo trabajan las tarjetas, a nivel de instrucción, un kernel puede ejecutar cientos de instrucciones en cada elemento de un stream, muchas de estas instrucciones pueden ser ejecutadas en paralelo. Como los kernels ejecutan la misma instrucción en cada elemento del stream, el paralelismo a nivel de datos puede ser alcanzado realizando éstas instrucciones en varios elementos del stream al mismo tiempo, por último, para el llamado nivel de paralelismo de tareas, es la habilidad de tener múltiples procesadores de streams dividiéndose el trabajo de un kernel o tener diferentes kernels corriendo en diferentes streams(Scott, 2004).

Arquitecturas GPU NVIDIA

Arquitectura Fermi

En el año 2001 se lanzó la segunda microarquitectura de NVIDIA luego de la arquitectura Tesla, la arquitectura Fermi, que se puede apreciar en la figura 2.9.

Figura 2.9: Arquitectura de Fermi.



La complejidad de la arquitectura de Fermi está controlada por un modelo de programación multinivel que permite a los programadores enfocarse en el diseño de su algoritmo más que en el cómo se distribuirá el algoritmo en la GPU. El software de CUDA permite a los programadores crear elementos computacionales que representan los algoritmos llamados kernels. Una aplicación de CUDA consiste en uno o varios kernels.

Una vez compilado, un kernel consiste en muchos threads que ejecutan el mismo programa en paralelo, básicamente un thread cumple la función de una iteración en un ciclo (un for, un while, por ejemplo.).

En cualquier momento determinado, la arquitectura Fermi utiliza todos sus recursos para una pura aplicación, pero ésta puede tener muchos kernels. Fermi soporta la ejecución de más de un kernel en simultáneo de múltiples kernels de la misma aplicación.

Arquitectura Kepler

Con el fuerte avance de la programación en GPU, NVIDIA lanza en 2012 una nueva arquitectura, llamada Kepler. Esta arquitectura es una mejora notoria a la arquitectura anterior (Fermi).

Figura 2.10: Potencia de cómputo de las arquitecturas Fermi y Kepler.

	FERMI GF100	FERMI GF104	KEPLER GK104	KEPLER GK110	KEPLER GK210
Compute Capability	2.0	2.1	3.0	3.5	3.7
Threads / Warp	32				
Max Threads / Thread Block	1024				
Max Warps / Multiprocessor	48		64		
Max Threads / Multiprocessor	1536		2048		
Max Thread Blocks / Multiprocessor	8		16		
32-bit Registers / Multiprocessor	32768		65536		131072
Max Registers / Thread Block	32768		65536		65536
Max Registers / Thread	63			255	
Max Shared Memory / Multiprocessor	48K				112K
Max Shared Memory / Thread Block	48K				
Max X Grid Dimension	2^16-1		2^32-1		
Hyper-Q	No			Yes	
Dynamic Parallelism	No			Yes	

Como se puede ver en la figura 2.10, la diferencia de potencia de cómputo entre las dos arquitecturas es significativa.

Kepler, además de ser más potente incorpora nuevas herramientas que permiten un provecho mayor de los recursos de la GPU, entre ellos el Hyper-Q, Grid Management Unit y NVIDIA GPUDirect(NVIDIA, 2017h), pero la más importante característica que incorpora esta arquitectura es la llamada Dynamic Parallelism.

Arquitectura Maxwell

La primera generación de ésta arquitectura fue lanzada por el año 2014, NVIDIA se enfocó principalmente en la mejora del consumo de energía de las GPU, además de la introducción de un nuevo Multiprocesador de Streams, que gracias a su nueva forma de organizar los datos y la mejora del agendador de instrucciones lograron obtener una mejora de más del $40\,\%$ de eficiencia por cada núcleo de CUDA, además de las ya mencionadas, se mejoró el balance de carga entre otras especificaciones.

Otro de los cambios significativos que vinieron con ésta nueva arquitectura, fue el

enfoque que se le dió a la misma por parte de NVIDIA, y éste enfoque es el llamado diseño Mobile-first (ZURB, 2017), el cual tiene como dogma el progressive enhancementGremilllion (2017) se define como "diseño móvil, al ser el más difícil, debe ser el primero en ser terminado, así, una vez terminado, el diseñar para las otras plataformas será más fácil. Lo que se deduce de esto es que el diseño más pequeño debe tener las características más esenciales, para así tener el corazón de lo que se está desarrollando". Llevado al modelo Maxwell, los ingenieros se enfocaron en crear un arquitectura eficiente versus el gasto energético de las GPU, ésto era una parte crítica, ya que su objetivo era crear GPUs para teléfonos inteligentes y después escalar el mismo modelo lo necesario para que pueda satisfacer las necesidades de un computador de escritorio o un laptop, en otras palabras, si podías crear un módulo lo suficientemente pequeño para ser insertado en un chip, simplemente bastaba unir ese módulo dentro de un chip con muchos módulos más para escalar al tamaño deseado, lo que en teoría es la filosofía de mobile first.

Gracias a las mejoras mencionadas anteriormente y que se observan en la figura 2.11, NVIDIA logró mejorar su eficiencia por watt vs Kepler, utilizando el mismo tipo de procesador Smith (2014).

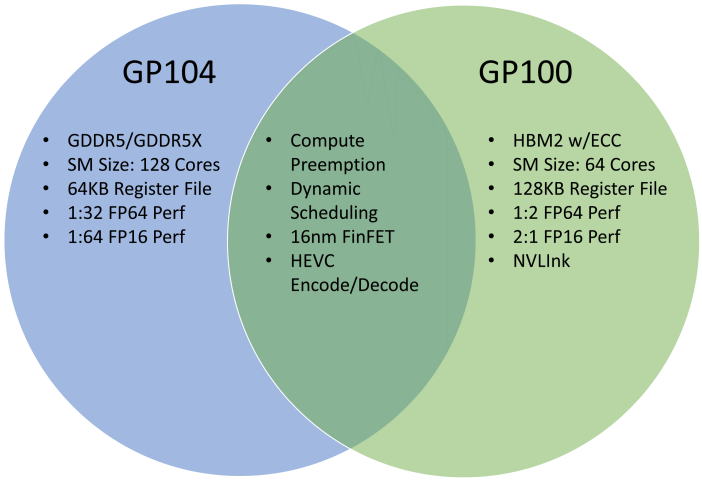
Figura 2.11: Arquitecturas NVIDIA Kepler SMX y NVIDIA Maxwell SMM

Arquitectura Pascal

Pascal es la última arquitectura lanzada por NVIDIA en abril del 2016 y es utilizada por la nueva gama de tarjetas de video que son la familia de las GeForce 10 series (NVIDIA, 2017f), luego de largos años con Maxwell, Pascal trae consigo mejoras a nivel de manufactura, siendo por primera vez compatible con los transistores finFET (De Borja & Garcia, 2010) de 16 nm, con ésto se enfocan en solucionar uno de los problemas principales de las tarjetas de video, el consumo de energía.

Por otra parte, en ésta arquitectura se presenta una división marcada (a diferencia de las arquitecturas anteriores, que si bien existían, no eran tan notables) entre los dispositivos enfocados en el HPC y los dispositivos enfocados en el gaming.

Figura 2.12: Diferencias en la arquitectua Pascal.



Cómo se puede ver en la figura 2.12, el modelo GP100, que está enfocado en el HPC, contiene por ejemplo más capacidad de memoria compartida y a pesar de tener menos cores, éstos tienen mayor capacidad de cómputo y de registro que los de la GP104 que es un dispositivo enfocado al usuario común.

GPU vs CPU, Diferencias

Ya que hablamos de la historia de las arquitecturas y cómo fueron tomando su forma en base a las necesidades, es hora de mostrar alguna de las diferencias concretas entre las dos

piezas de hardware:

Una forma simple de entender una de las diferencias fundamentales es el comparar cómo es que procesan las tareas. Una CPU consiste en unos pocos Núcleos optimizados para el procesamiento secuencial en serie, mientras que la GPU tiene una arquitectura que consiste en cientos de pequeños núcleos diseñados para manejar de forma más eficiente múltiples tareas en simultáneo. Así mismo, los núcleos de la CPU sólo pueden manejar unos pocos hilos de procesamiento en simultáneo, mientras que los núcleos de la GPU están diseñados para manejar cientos de hilos en simultáneo.

Por otra parte, la CPU utiliza núcleos MIMD(Giles, 2017), que significa que sus núcleos tienen la capacidad de operar independientemente y cada uno de ellos puede estar trabajando con un código diferente, realizando diferentes operaciones con datos totalmente diferentes, mientras que la GPU utiliza núcleos SIMD(Giles, 2017), lo que significa que todos los núcleos realizan la misma instrucción al mismo tiempo, pero trabajando en diferentes datos. Aunque existen avances en la emulación de MIMD en GPU (Dietz & Dalton, 2009). La GPU tiene por definición más unidades ALU que la CPU, lo que se traduce en una mejor capacidad para procesar operaciones paralelas aritméticas. La CPU está diseñada para una gran variedad de aplicaciones distintas y provee una respuesta rápida a una tarea, por otro lado la GPU, está construida específicamente para el renderizado y otras aplicaciones gráficas que tienen un largo grado de datos en paralelo, lo que es ideal para una arquitectura que cuenta con mucho núcleos.

Con estas diferencias tan marcadas, es fácil encontrar el tipo de problema en el que la GPU podría llevar todas su características al límite, estos problemas son aquellos que tienen una solución paralela, un ejemplo de ello es la multiplicación de matrices, la cual puede ser distribuida en muchas combinaciones de sumas y multiplicaciones independientes unas de las otras. Por otra parte, existen problemas que no tienen solución en paralelo, que en su mayoría son problemas con dependencia de resultados, lo que quiere decir es que necesitan un resultado de la iteración anterior para completar el actual, un ejemplo de este tipo de algoritmos es la aproximación de la raíz cuadrada de x utilizando el método Newton-Rhapson (Peelle, 1974).

CUDA

Introducción

CUDA es una plataforma de cómputo paralelo y un modelo de programación desarrollado por (NVIDIA, 2017c), el cual ayuda a mejorar el desempeño de algoritmos paralelos utilizando el gran poder de cómputo paralelo a través de utilización de GPUs.

Desde sus inicios, CUDA ha sido utilizado por científicos para desarrollar aplicaciones paralelas y publicar papers, aprovechando el poder de procesamiento de los miles de millones de CUDA cores disponibles en muchas de sus plataformas: desde notebooks hasta supercomputadores. Todo esto se realiza utilizando una API de programación que se mantiene con una documentación constantemente actualizada, además de una comunidad activa que permite obtener ayuda a través de su foro de desarrolladores[1] y en sitios web pregunta-respuesta como StackOverflow[2].

El nombre CUDA proviene de **C**ompute **U**nified **D**evice **A**rchitecture y es una extensión del lenguaje C, que permite compilar código que será ejecutado directamente en la GPU.

La masificación de las GPUs de NVIDIA ha permitido que muchos ingenieros de software, científicos e investigadores, busquen formas de poder utilizar esta tecnología y así mejorar sus procesos. La NASA obtuvo mejoras importantes en los resultados del análisis del flujo del tráfico aéreo utilizando esta tecnología, logrando identificar nuevos caminos para aliviar congestiones, permitiendo mantener el tráfico aéreo más despejado para que funcione eficientemente. Los resultados fueron alentadores, reduciendo el tiempo de análisis desde los 10 minutos a los 3 segundos de ejecución(NVIDIA, 2017d).

Recursos disponibles

NVIDIA proporciona un conjunto de herramientas que facilitan el desarrollo de aplicaciones paralelas a los programadores. Algunas de las herramientas son:

1. CUDA Toolkit: ambiente de desarrollo proporcionado por NVIDIA, el cual sirve principalmente para programar aplicaciones paralelas. Dentro de las herramientas más importantes

[1] https://devtalk.nvidia.com/text

[2] https://stackoverflow.com

incluidas se encuentra el compilador NVCC. Además, viene con ejemplos de algoritmos paralelos implementados utilizando CUDA, los cuales sirven de guía sobre todo para mejorar la performance de los programas. La última versión disponible es la 8.0.61[3]. Otras de las utilidades incluidas son:

- Visual Profiler: es una herramienta gráfica que permite hacer perfilamiento de las aplicaciones, medir el desempeño de cada una gracias a una línea de tiempo generada tras la ejecución, en donde es posible ver la actividad completa tanto en la CPU como también en la GPU. El profiler incorpora análisis automático de código para identificar oportunidades de optimización. Esta herramienta viene incluida en el CUDA Toolkit[4].
- NVIDIA Nsight: entorno de desarrollo integrado (IDE) impulsado por la plataforma Eclipse. Proporciona un entorno todo incluido para editar, construir, hacer debug y perfilamiento. También soporta una gran cantidad de plugins privados y gratis[5].
- Debugger: proporciona control total de la ejecución de una aplicación CUDA, incluyendo la posibilidad de fijar puntos de quiebres y paso-a-paso en la ejecución del código, permitiendo inspeccionar variables, escritura y lectura de memoria, registros, entre otros. Estas sólo son algunas de las características del debugger[6].

2. NVIDIA Developers site: foro oficial de NVIDIA, creado para discutir sobre algoritmos, optimizaciones y posibles soluciones a problemas utilizando CUDA[7].

Kernel

La guía de programación en C de CUDA (NVIDIA, 2017b), menciona lo siguiente respecto a los kernels de CUDA:

CUDA C es una extensión de C que permite al programador definir funciones de C, llamadas kernels, que cuando son llamadas, se ejecutan paralelamente N veces en diferentes hilos CUDA, diferente a las funciones comunes de C que sólo permiten ejecutar una sola función

El kernel se define utilizando el especificador __global__ en la declaración de la función, además se debe especificar como función void acompañado de un nombre para identificarlo y llamarlo posteriormente. Los argumentos de la función son declarados de igual forma que en

[3]https://developer.nvidia.com/cuda-downloads
[4]http://docs.nvidia.com/cuda/profiler-users-guide/index.html#axzz4qSGiHfdN
[5]https://developer.nvidia.com/nsight-eclipse-edition
[6]https://developer.nvidia.com/cuda-gdb
[7]https://devtalk.nvidia.com/

una función en C. Cada hilo posee un identificador único y se puede acceder mediante la variable threadIdx:

Algoritmo 1: Ejemplo kernel hijo en CUDA

```
__global__ Kernel_1(float * vector){
    int id = threadIdx.x
    vetcor[id] = vector[id] + 1
    ...
}
```

El número de hilos a ser utilizados se define al momento de lanzar el kernel siguiendo la sintaxis a continuación:

$$kernel\_1 \lll 1, N \ggg (vector) \tag{2.11}$$

Los kernels se ejecutan como una función pero en la GPU, es un arreglo de hilos ejecutados en paralelo. Todos los hilos ejecutan el mismo código, pudiendo tomar diferentes caminos. En el caso del ejemplo recién expuesto, N hilos aumentarán en una unidad cada uno de los elementos del vector.

Jerarquía en hilos y en memoria

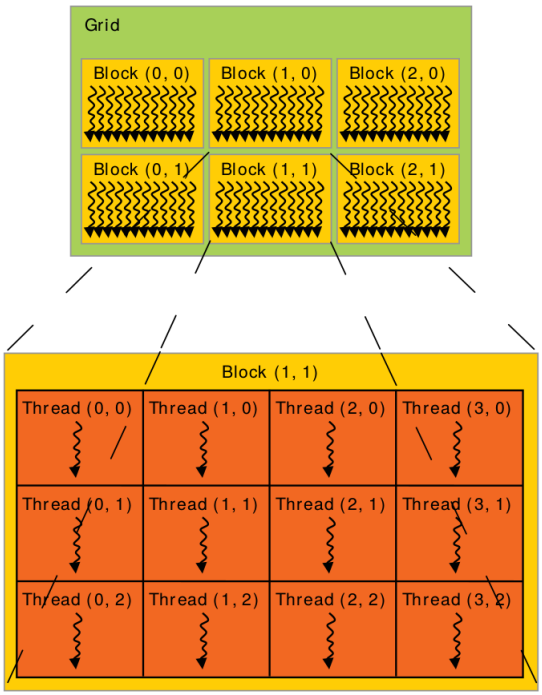
La variable threadIdx es un vector de tres elementos, en donde cada uno representa una dimensión, permitiendo agrupar conjuntos de hilos en bloques.

Los bloques también están organizados en más dimensiones a través de grillas (de ahora en adelante grid) de una, dos y tres dimensiones. El número de blocks por grid necesarios está determinado por la cantidad de datos a ser procesados o la cantidad de procesadores disponibles, los cuales pueden variar mucho dependiendo del problema.

Cuando se utilizan bloques de dos dimensiones de tamaño (D_x, D_y), el identificador del hilo de cada índice (x, y) está determinado por $(x + yD_x)$; Cuando se utilizan bloques de tres dimensiones de tamaño (D_x, D_y, D_z), el identificador del hilo de cada índice (x, y, z) está determinado por $(x + yDx + zDxDy)$.

De forma análoga, la variable blockIdx corresponde al vector con los identificadores de cada bloque de hilos. Además es posible conocer la dimensión de cada bloque a través de la variable blockDim.

Figura 2.13: Diagrama paradigma programación CUDA (Grid y Block)



Cada hilo posee una memoria privada local. Cada bloque de hilos tiene una porción de memoria compartida, la que es visible entre todos los hilos del mismo bloque y con el mismo ciclo de vida del bloque. Todos los hilos tienen acceso a los datos que se encuentren en la memoria global del dispositivo.

CUDA también considera espacios de memoria adicionales especiales para que sean accedidos por todos los hilos. Estos espacios están destinados para almacenar constantes y texturas, pudiendo incluso manipular datos o filtrarlos de forma sencilla.

Dynamic Parallelism

Introducción

El paralelismo dinámico o (DP) por sus siglas en inglés, es soportado por una extensión del modelo de programación que le otorga a los kernels de CUDA la habilidad de poder crear y sincronizar trabajo anidado ¿Qué quiere decir ésto? En términos muy simples, le permite a un

kernel padre invocar otro kernel (llamado comúnmente kernel hijo), controlar la sincronización del mismo y utilizar la salida de datos de él, todo sin involucrar la CPU.

Algoritmo 2: Ejemplo kernel hijo en CUDA

```
__global__ childKernel(void * data){
    //do some work
}
```

Algoritmo 3: Ejemplo llamada kernel hijo desde kernel padre en CUDA

```
__global__ parentKernel(void *data){
    childKernel <<< 16,1 >>>(data);
}
```

Invocando desde el host:

$$parentKernel\_1 \lll 256, 64 \ggg (data) \tag{2.12}$$

Además, otra característica importante es la habilidad de un kernel de llamarse recursivamente:

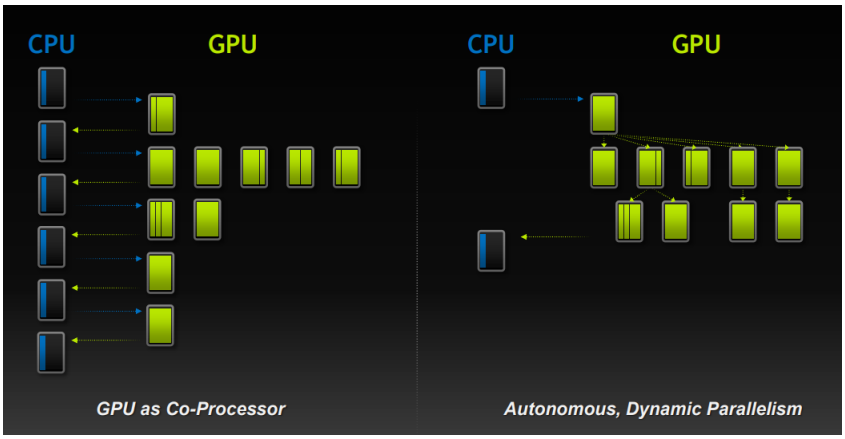
Algoritmo 4: Kernel recursivo

```
si continueRecursion == true entonces
    recursiveKernel <<< 64,16 >>>(data)
en otro caso
    detenerRecursion
fin
```

Antes de la aparición de DP, los programadores tenían que conformarse con un modelo de programación donde simplemente se le enviaban datos a la GPU para que ésta trabajara aquellos datos y retornarlos a la CPU, si luego se debía realizar otra operación con ellos que involucre cómputo en la GPU, se debía cargar nuevamente los datos a la GPU, trabajarlos y enviarlos de regreso a la CPU, todo ésto las veces que sea necesaria. Así, la carga de archivos de CPU a GPU y viceversa era inmensa, por el contrario, con DP el kernel padre puede llamar a los kernels que sean necesarios en orden de terminar el trabajo necesario y luego enviar los datos de regreso

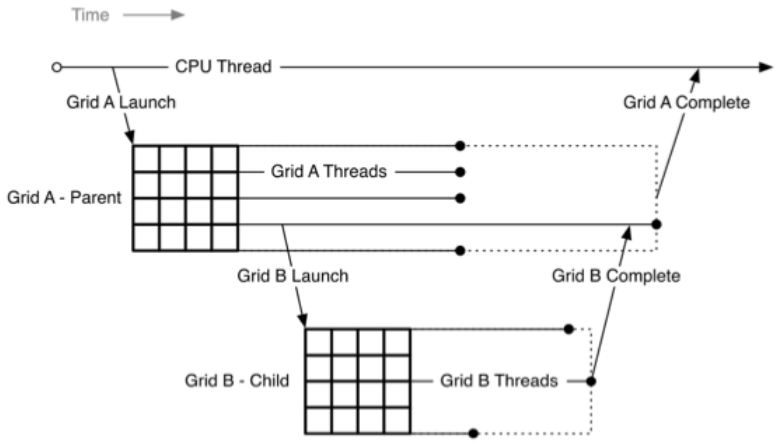
a la CPU.

Figura 2.14: CUDA normal versus CUDA Dynamic Parallelism.



El modelo de ejecución de CUDA trabaja en base a primitivas, que son los threads y blocks, el conjunto de ellas en un kernel se llama grid, CUDA DP trabaja de la misma forma, pero con la diferencia que los kernels lanzados por el kernel padre son inherentes de él, lo quiere decir es que el padre comparte cierto límites y atributos a los kernel hijos, como el cache L1 y la configuración de la memoria compartida. Las grid en Dp tienen lo que se llama un diseño completamente anidado, lo que quiere decir es que la grid hija siempre completa su ejecución antes que la grid padre que la lanzó.

Figura 2.15: Diseño anidado de las Grids en Dynamic Parallelism.

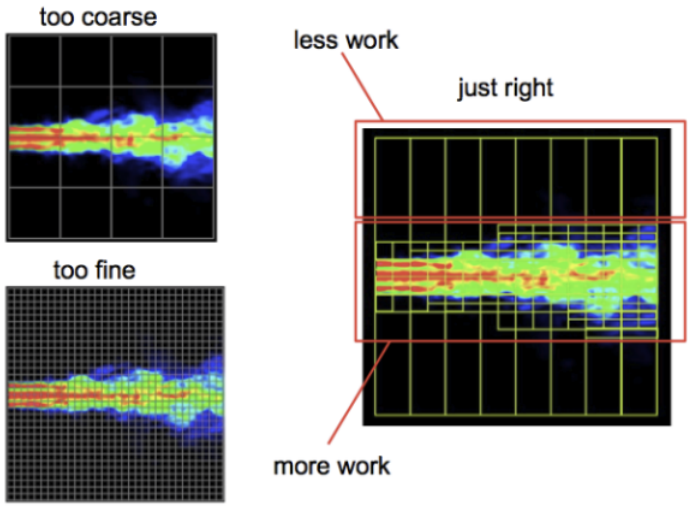


Es importante mencionar que incluso cuando el comportamiento descrito de las grids

podría tomarse como una sincronización implícita, si un kernel padre necesita del cómputo realizado por el kernel hijo, debe asegurarse que la grid del hijo fue ejecutada completamente, para ésto es que CUDA ha implementado una función llamada cudaDeviceSyncronize(void), la cual espera el término de la ejecución de todos los grids de ese nivel antes de continuar, ésta función se considera altamente costosa y se aconseja utilizarla sólo cuando es necesario, para no perjudicar el desempeño del algoritmo implementado. Otra de las características importantes de DP es la consistencia de memoria entre kernel padre e hijo, ya que si un padre escribe un valor en memoria y luego lanza un kernel hijo, está garantizado que el kernel hijo puede ver el valor, así mismo, si un kernel hijo escribe en memoria y luego el kernel padre realiza una sincronización, está asegurado que el padre podrá ver el valor escrito por el padre(NVIDIA, 2012).

Una de los problemas que ha ayudado a solucionar en parte DP es cuando los espacios de información que involucran al problema no pueden ser fácilmente almacenados por las estructuras de CUDA, dando como resultado una pérdida importante de cómputo localizando los puntos de información de interés del problema.

Figura 2.16: Ejemplos mapeos a un dominio de problema poco común.



En la figura 2.16 podemos observar que la forma más optima de hacer un mapeo es utilizando Dynamic Parallelism, asi, el tamaño de los blocks se puede adaptar al al dominio del problema, por otro lado, también se puede observar que si utilizamos un mapeo con tamaños de blocks muy grandes, no se puede aprovechar el poderío cómputo de la GPU, al mismo tiempo, si utilizamos muchos blocks más pequeños, la GPU desperdicia mucha potencia en espacios donde

no existe dominio del problema (cubos donde sólo se ve el fondo negro)

Un ejemplo clásico que puede ser solucionable con DP es el del conjunto de Mandelbrot, uno de los fractales más estudiados y conocidos del mundo(Adinetz, 2014).

Restricciones y limitaciones

Como restricciones generales de DP, en Cheng et al. (2014a) se mencionan tres:

- Se necesita un dispositivo de capacidad de cómputo 3.5 o superior.
- Los kernels invocados a través de DP no pueden ser lanzados en dispositivos físicamente separados. Es permitido, sin embargo, obtener algunas propiedades de cualquier dispositivo CUDA en el sistema.
- El límite máximo de profundidad de la recursión es 24, pero en la realidad la mayoría de los kernels estarán limitados por la cantidad de memoria requerida en tiempo de ejecución para cada nuevo nivel, ya que se reserva memoria adicional para el manejo de la sincronización entre cada padre y sus hijos en cada nivel de anidado.

Problemas triangulares

INTRODUCCIÓN

En el proceso para realizar una implementación que solucione algún problema matemático, científico u otros, uno de los pasos esenciales es el mapeo del espacio de computo al dominio del problema a resolver, para que este pueda ser perfectamente reconocible por el dispositivo computacional. Así, conociendo la forma del dominio del problema, se puede buscar la tecnica mas óptima para amoldar el espacio de cómputo con la menor cantidad de desperdicio de recursos computacionales. Las estructuras con las que trabaja CUDA están desarrolladas en una dimensión (array), dos dimensiones (matriz) y tres dimensiones (vector), de forma que si un problema tiene un dominio con una figura simple, digamos un cuadrado, podemos utilizar una estructura de dos dimensiones para cubrir el problema sin mayores inconvenientes, de hecho, basta con el mapeo por defecto que realiza la GPU, ya que la pérdida de recursos es mínima, sin embargo, existen problemas que tienen una forma más complicada, un ejemplo de ellos son los problemas de dominio triangular. Un problema triangular puede definirse como un espacio de computo con forma de triángulo, también en algunos casos, puede ser expresado como un sistema triangular de la forma:

Siendo A una matriz escalonada inferior, tenemos $Ax + b$

Figura 3.1: Gaussian Elimination.

$$\begin{bmatrix} \ell_{11} & 0 & 0 \\ \ell_{21} & \ell_{22} & 0 \\ \ell_{31} & \ell_{32} & \ell_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

Al mapear de la forma tradicional un problema triangular, se genera en GPU un gasto innecesario de recursos, ya que existe una gran cantidad de threads y bloques que se crearan innecesariamente y deben ser descartados por ellos mismos con condiciones de corte, lo que lleva a una penalización de performance importante.

Técnicas de mapeo

MÉTODO BOUNDING BOX

El método Bounding Box o BB, por sus siglas en inglés, es uno de los métodos más simples para el mapeo de espacios triangulares. La estrategia utilizada es simple, ya que considera todos los elementos de una matriz de tamaño N, distribuye en blocks los threads disponibles y descarta aquellos que queden sobre la diagonal, pero durante su ejecución. Lo que realiza es un mapeo 1:1, con $f(x) = x$. Si bien el mapeo es trivial, su desempeño no es el mejor, ya que por simple inspección se observa que se podrían estar utilizando más recursos de los necesarios, en el caso en que el dominio de datos del problema sea triangular, por ejemplo, ya que este método siempre considera formas cuadradas o rectangulares.

Los threads son descartados en tiempo de ejecución utilizando la siguiente validación:

Algoritmo 5: Condición de término recursión método Bounding Box

```
si blockIdx.x >blockIdx.y entonces
    fin recursión
fin
```

Si la condición fuese correcta se termina el cómputo en ese kernel, en caso contrario, se obtiene la coordenada de los demás threads y se realiza trabajo en ellos. Además, se realiza una validación:

Algoritmo 6: Condición de corte diagonal método Bounding Box

```
i, j <= coordenada global en x,y
si i >= j entonces
    ...
fin
```

Para determinar si el thread está bajo la diagonal o no, ésto por los problemas en la diagonal que han sido observados por (Navarro & Hitschfeld, 2013).

Algoritmo en pseudocódigo:

Algoritmo 7: Pseudocódigo algoritmo método Bounding Box

```
si coordenada_bloque_x >coordenada_bloque_y entonces
    fin recursión
en otro caso
    i, j <= coordenada global en x, y
    si i>=j entonces
        index <= i*N +j
        matrix[index] <= cost_function()
    fin
fin
```

MÉTODO RECTANGLE BOX

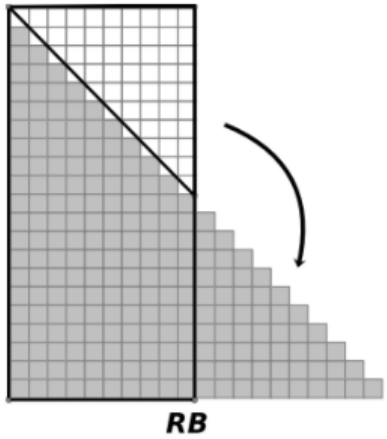
El método Rectangle Box o RB fue propuesto por (Hyuk & OLeary, 2008) en el año 2008. Una de las características principales de este algoritmo, es que utiliza la N/2 espacio de la matriz y no N, como la mayoría de los métodos. Esto ocurre porque mapea un espacio rectangular de dimensiones N de largo por N/2 de ancho. A diferencia del método BB, el método RB considera el espacio de mapeo óptimo, independiente si debe realizar una traslación de algunas coordenadas. Este método, descarta en tiempo real los threads que no procesarán trabajo y lo realiza con la siguiente condición:

$$if(i >= N || j <= N/2) \tag{4.1}$$

Luego queda pendiente trasladar la parte sobrante de la diagonal inferior o superior del mapeo, la que posee la forma de un triángulo rectángulo, y reposicionarlo bajo la diagonal [ver figura X]. Esta traslación se realiza con la condición de que el thread en la coordenada x sea mayor a N/2, y los threads que superan esa marca son trasladados con la función:

$i = t_y - 1$, con t_y coordenada del thread en y; i nueva coordenada del thread en x.

Figura 4.1: Método Rectangle Box.



El programa en pseudocódigo:

Algoritmo 8: Pseudocódigo algoritmo método Rectangle Box

```
i, j <= coordenada global en x, y
si i>=N || j>= N/2 entonces
  | fin recursión
fin
si j>i entonces
  | j <= N-j-1 i <= N-i-1
en otro caso
  | i <= i-1
fin
C <= i*N+j si C <N/2 entonces
  | matrix[C] <= cost_function()
fin
```

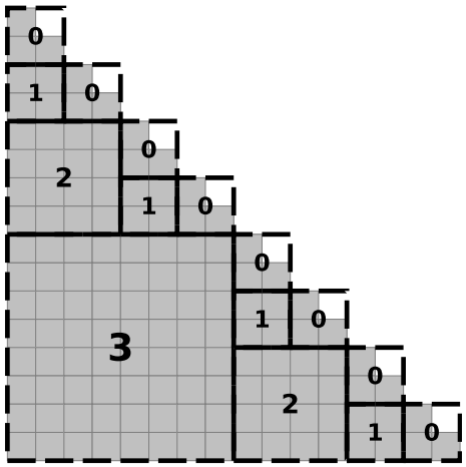
MÉTODO RECURSIVE COMPUTATION

El método de cómputo recursivo (REC siglas de su nombre en inglés Recursive Computation) es una estrategia propuesta para la inversión de la matriz en GPU (Ries et al., 2009), donde el tamaño del problema está definido por:

$$N = m2^k \tag{4.2}$$

Donde k y m son enteros positivos y m es un múltiplo del blocksize. La idea es hacer una recursión binaria (al igual que el método de búsqueda Merge-sort) de tamaño k, en donde cada nivel lanza una serie de bloques de recursión, un total de k veces. Cabe destacar que éste método requiere una pasada adicional para trabajar con los casos que están cerca de la diagonal o en ella misma- También es importante señalar que el mapeo es presentado como una solución a un problema concreto (que es la optimización de la triangulación de la matriz inversa) utilizando el principio divide et impera (divide y vencerás).

Figura 4.2: Método Recursive Computation



El mapeo de los blocks a sus respectivas posiciones en cada nivel de recursión es dado por una tabla guardada en memoria. El algoritmo en pseudocódigo:

Algoritmo 9: Pseudocódigo algoritmo método Recursive Computation

```
rec_index <= calcula el bloque recursivo (∀ blockIdx.x / gridDim.y ∈ N)
offset_x <= blockIdx.x % gridDim.y
tx <= (bx+(gridDim.y*rec_index*2)+ dbx)*blockDim.x + threadIdx.x
ty <= (by+(gridDim.y*rec_index*2)+ blockIdx.y)*blockDim.y + threadIdx.y
C <= ty*N+tx
si c <N entonces
    matrix[C] <= cost_function()
fin
```

MÉTODO LOWER TRIANGULAR MAPPING

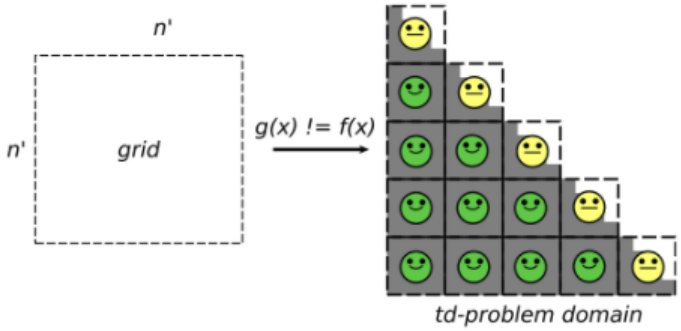
El método LTM propuesto por (Navarro et al., 2016), ataca el principal problema de la pérdida de recursos computacionales que se puede ver en el BB. El análisis hecho en el paper demuestra que para un problema triangular de tamaño N, se necesitan $n(n+1)/2$ blocks para cubrir completamente el dominio del problema, para ésto se agrega una condición que descarta todos los blocks que están sobre la diagonal.

Así, el mapeo consiste en buscar las coordenadas i y j para cada uno de los blocks, como se explica en las ecuaciones cinco, seis, ocho y nueve en (Navarro et al., 2016), éstas coordenadas quedan de la siguiente forma:

$$(i, j) = \left(\sqrt{\frac{1}{4} + 2\lambda} + \frac{1}{2}, \lambda - \frac{i(i+1)}{2} \right) \tag{4.3}$$

El factor de mejora de este método, radica en que realiza una menor cantidad de operaciones con punto flotante respecto a otros métodos como el UTM (Upper-Triangular Mapping) (Avril et al., 2012). Además de que se mapea a nivel de blocks y no de threads, y para grandes valores de N el resultado de la raíz cuadrada posee menor porcentaje de error en la aproximación.

Figura 4.3: LTM usa sólo la cantidad de blocks necesarias para cubrir el dominio del problema.



El algoritmo en pseudocódigo:

Algoritmo 10: Pseudocódigo algoritmo método LTM

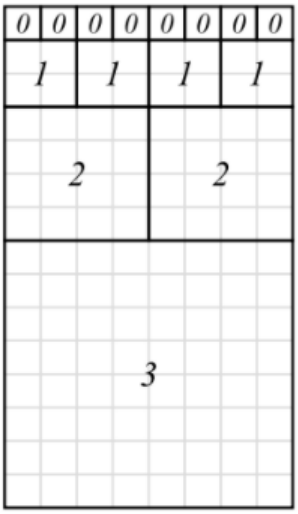
```
bc <= obtener posición global del bloque
bi <= rsqrtf(0.25+2*bc)+0.50
bj <= bc - bi(bi´1)/2
i <= bi * blockDim.y + threadIdx.y
j <= bj * blockDim.x + threadIdx.x
index <= i*N+j
si i >=j && index <N*N entonces
    matrix[index] <= cost_function()
fin
```

IMPLEMENTACIÓN MÉTODO FLAT RECURSIVE

La teoría detrás de éste método está explicada en (Navarro et al., 2016), y si bien el paper explica el mapeo hasta en tres dimensiones, éste escapa de los límites de esta memoria y se realiza la implementación sólo en 2 dimensiones.

El objetivo del algoritmo es crear una figura de tamaño $N/2 * N$, como se muestra en la siguiente figura:

Figura 4.4: Mapeo del método Flat Recursive.



Cada cuadrado pequeño gris en el interior de cada cuadrado con bordes negros, es un block. Para lograr esto es necesario obtener una coordenada i y j, las que son de la forma:

$$(W_x + Q_b, W_y + 2Q_b) \tag{4.4}$$

Donde $b = 2^{\log W_y}; q = \frac{blockIdx.x}{b}$

Podemos destacar que se hicieron dos implementaciones de éste algoritmo, una fue llamada Flat Recursive Basic, que como su nombre lo indica es la muestra más básica del método de mapeo (hasta la ecuación 14 del paper), La segunda es una versión completa llamada flat recursive, la cual implementa las optimizaciones de las ecuaciones 15 y 16 de (Navarro et al., 2016). Cabe señalar que este método sólo ha sido expuesto teóricamente y el presente trabajo incorpora la primera implementación real del mismo, haciendo interesante a la hora de contrarrestar la teoría expuesta en el mismo documento con la implementación real y práctica del método.

El algoritmo Flat Recursive Basic en pseudocódigo:

Algoritmo 11: Pseudocódigo algoritmo método Flat Recursive Básico

```
//blockIdx.x es la coordenada de block en x
//blockIdx.y es la coordenada de block en y
si blockIdx.y == 0 entonces
    si threadIdx.y >= threadIdx.x entonces
        //ty y tx corresponden a las coordenadas globales
        matrix[ty*N + tx] = cost_function()
        matrix[(ty+halfN)*N+tx] = cost_function()
    fin
fin
//utilizamos bitwise operator para realizar la potencia de 2
b = 1 «trunc(log2f(blockIdx.y))
alpha = blockIdx.x/b
q = trunc(alpha)
Wx = blockIdx.x + q*b
Wy = blockIdx.y + 2*q*b
index = ((Wx * blockDim.x)+threadIdx.x) + ((Wy*blockDim.y)+ threadIdx.y)*N
matrix[index] = cost_function()
```

El algoritmo Flat Recursive en pseudocódigo:

Algoritmo 12: Pseudocódigo algoritmo método Flat Recursive

```
//siendo blockIdx.x la coordenada de block en x
//siendo blockIdx.y la coordenada de block en y
si blockIdx.y == 0 entonces
    si threadIdx.y >= threadIdx.x entonces
        //Siendo ty y tx las coordenadas globales
        matrix[ty*N + tx] = cost_function()
        matrix[(ty+halfN)*N+tx] = cost_function()
    fin
fin
//utilizamos bitwise operator para realizar la potencia de 2
w <= blockIdx.x, blockIdx.y
b <= 1 ≪ (31 - __clz(w.y))
q <= w.x/b
m <= w.x + q*b, w.y + ((q*b) «1)
index <= (threadIdx.y + m.y*blockDim.y)*N + threadIdx.x + m.x*blockDim.x
matrix[index] <= cost_function()
```

Método utilizando CUDA Dynamic Parallelism

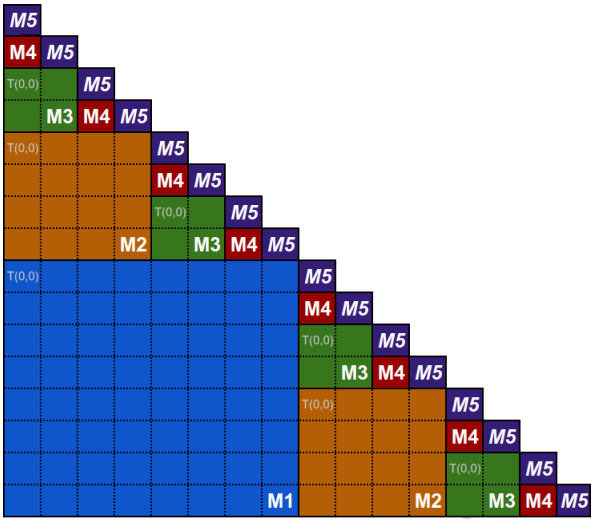
INTRODUCCIÓN

El método que desarrollamos utilizando la tecnología de CUDA llamada Dynamic Parallelism, como su nombre lo indica permite programar algoritmos utilizando paralelismo dinámicamente, utilizando recursos a medida que se necesita y no asumiendo nada previa ejecución haciéndolo masivamente paralelo. El proceso del método consiste en esparcir el dominio del problema a nivel de blocks, porcionando la cantidad de blocks por nivel de recursión. Cada ejecución es lanzada desde la GPU, similar a cuando se recorre recursivamente un árbol binario(Othman & Arshad, 2011). Un kernel padre, se encarga de lanzar dos kernels hijos: hijo izquierdo e hijo derecho. Este proceso se repite hasta cubrir con blocks pequeños la diagonal, en un proceso totalmente dinámico y sin tener que pasar por una etapa de sincronización con la CPU.

Como ya sabemos, los blocks pueden ser de dos dimensiones y son geométricamente cuadrados. Nuestro problema se basa en resolver problemas que necesitan la mitad diagonal del cuadrado, por lo tanto sólo necesitamos considerar la diagonal superior o inferior, desperdiciando la mitad de los recursos si se mantuviera el esquema cuadrado. Dado que parte del objetivo de este trabajo es encontrar un método con un mejor desempeño, se necesita resolver el problema de mapeo óptimamente, y es por esto que nuestro método se basa en una estrategia que utiliza los recursos dinámicamente, hasta el nivel más bajo de recursión, posibilitando así resolver el problema de mapeo gracias a la tecnología desarrollada por NVIDIA.

Nuestra solución se basa en la siguiente figura 5.1:

Figura 5.1: Mapeo del método Dynamic Parallelism



Una de las variables importantes a considerar es que la correcta forma de implementar una matriz de dos dimensiones en CUDA, es mediante un array de una dimensión, obligando a llevar un punto desde un sistema de dos coordenadas a uno de una. Esto se formaliza con la siguiente función:

$$S(x, y) = x * Matrix_w + y \tag{5.1}$$

Podemos pensar en este método como si se tratase de un sistema de coordenadas de 2-dimensiones, llevado a uno de 1-dimensión. Por tanto, cada (0,0) presente en la figura, necesitan de un offset en las coordenadas x e y, los que llamaremos offset-x y offset-y respectivamente. Estas variables permiten abstraer el saber si se debería lanzar el kernel dependiendo de su índice en 1-dimensión, dada su posición en el plano de 2-dimensiones. El cálculo del index se realiza utilizando la siguiente ecuación :

$$index <= (tidy + offsety) * N + (tidx + offsetx) \tag{5.2}$$

La lógica del método TCDP inicia con la invocación al primer kernel en el nivel M1, primer nivel de profundidad de la recursión, con un espacio de cómputo de N2, el que además posee la mayor cantidad de bloques respecto a los demás hijos. El thread T(0,0) del primer block de cada kernel es el encargado de invocar dos kernels hijos más, los dos del mismo tamaño pero siempre cumpliendose que deben ser N2 veces más pequeño que su padre. En la figura

corresponde al nivel llamado M2. A su vez, cada uno de estos podría eventualmente realizar una nueva invocación a otros kernels hijos, para cubrir más datos con otra porción de blocks. Este proceso se puede repetir recursivamente escalando a problemas con un límite teórico de 2^{∞}, posibilitando así mantener la proporción con la cantidad óptima de 32 hilos de un warp, para evitar cuellos de botella (bottlenecks) (Cheng et al., 2014a) en la optimización que pudiera realizar la GPU al encontrarse con hilos ociosos (idle) o en la sincronización necesario para acceder a datos ubicados en distintos wraps.

En la estrategia CDP, cada hijo es lanzado con la siguiente configuración de parámetros:

$$hijo\_izquierdo(..., offsetx, offsety - n/2, depth + 1, ...) \tag{5.3}$$

$$hijo\_derecho(..., offsetx + n, offsety + n/2, depth + 1, ...) \tag{5.4}$$

En nuestro ejemplo, la recursión se detiene en M3, ya que la porción de datos considerada en M4y M5 es de $N <= 1024$, lo cual es un valor muy pequeño y resulta más óptimo utilizar una implementación más simple. Este valor es arbitrario y se utiliza para optimizar el desempeño del algoritmo, pero de todas formas es bueno aclarar que depende del tamaño del problema. Pese a esto, se utiliza la implementación del método bounding box para cubrir los elementos cercanos a la diagonal y los que se encuentran en ella también y se utiliza principalmente por la simplicidad de su implementación, la cual mapea la diagonal inferior de una matriz considerando su diagonal, en el mismo nivel de profundidad para M4 y M5. La condición que determina el corte de cuántos son considerado pocos datos, depende del valor de una variable llamada NCORTE, la que permite realizar variaciones dependiendo del tamaño del problema, al momento de buscar el valor óptimo dependiendo de la situación. Junto con esta condición, se valida si se ha alcanzado el nivel máximo de profundidad en la recursión, es decir, si alguna de las condiciones se cumpliese, es necesario utilizar el método bounding box para resolver la porción trivial de datos.

Cuando se deben lanzar los kernels con bounding box, se realiza con los siguientes parámetros:

$$hijo\_izquierdo(..., offsetx, offsety - n) \tag{5.5}$$

$$hijo\_derecho(..., offsetx + n, offsety) \tag{5.6}$$

Además de dicha variable, se considera una adicional para controlar el nivel máximo de profundidad que puede alcanzar el algoritmo, la que llamaremos maxdepth. Esto está determinado por:

$$maxdepth <= log2(n) * a; 0 < \alpha < 1 \tag{5.7}$$

Esta variable permite delimitar la profundidad de los niveles dependiendo de la cantidad de datos de entrada, haciéndolos variar poco independientemente si el valor aumenta mucho, éste acotará logarítmicamente su valor. Además se multiplica por una constante, la cual permite disminuir aún más el tamaño de la profundidad.

Una eventual optimización podría considerar re mapear estos kernels creando otra función que permita reagrupar todos estos blocks más pequeños del mismo nivel, en uno más grande pero eso escapa del objetivo de este trabajo.

Ya se han nombrado algunas optimizaciones implementadas en el algoritmo. Sin embargo, existen detalles que dieron un boost favorable en el desempeño del algoritmo. Ejemplo de esto es la utilización de bitwise operator para realizar divisiones por dos. En la unidad lógica-aritmética de la CPU, existen operaciones matemáticas realizadas a nivel de bit. Desde C se pueden utilizar dichos operadores:

Operador	Operación
&	AND
|	OR
^	XOR
~	Complemento
$\ll$	Shiftleft
$\gg$	Shiftright

En este método se utiliza el operador shift right ($>>$), el cual permite realizar divisiones en potencias de dos. Formalmente el operador realiza un desplazamiento de todos los bits hacia la derecha, dependiendo del número especificado:

$$resultado = variable >> cantidadBits$$

Por ejemplo:

$$98 \ll 0 = 1100010 = 98 \tag{5.8}$$

$$98 \ll 2 = 0011000 = 24$$

$$98 \ll 3 = 0001100 = 12$$

$$98 \ll 4 = 0000110 = 6$$

...

$$98 \ll 7 = 0000000 = 0$$

En el caso de este método se utiliza para optimizar la división que se realiza en cada kernel invocado. La división de la variable halfn se realiza siempre por dos, utilizando el bitwise operator de la siguiente forma:

$$halfn = halfn \gg 1 \tag{5.9}$$

En nuestro caso, si se tuviese:

$$n = 8192 \tag{5.10}$$

En la primera llamada al kernel se recibe halfn como la mitad de n, pero en cada nivel de recursión se divide halfn por la mitad, en este caso:

$$n >> 1 = 4096 \tag{5.11}$$

Logrando el objetivo de optimizar la performance en un $30\,\%$, respecto a su versión sin esta optimización. Por otro lado, cada llamada de kernel requiere una configuración de las dimensiones del espacio de cómputo que se utilizará. En este caso, todos los kernels mantienen siempre constante la dimensión de sus bloques, en cambio, la dimensión de la grilla varía siempre. Dicha variación se da por la siguiente ecuación:

$$gridDim = (halfn + blockDim - 1)/blockDim \tag{5.12}$$

La invocación de cada kernel se realiza siempre de a dos hijos, en este trabajo los denominamos hijo izquierdo e hijo derecho. Cada uno hace referencia al comportamiento que posee la ejecución del algoritmo, el cual parte dividiendo M1 en dos hijos M2, cada uno con

una porción menor, pero cada nivel potencialmente podría pasar a ser padre de al menos dos hijos. Si se observa la figura 1, se puede apreciar que el thread (0,0) siempre debe estar en una coordenada en el plano x e y, por tanto es necesario generar dicho comportamiento dada la ecuación 1. Para ello simplemente se parametrizan los offsets dependiendo del tipo de hijo que era.

En el caso del hijo izquierdo, la llamada es la siguiente:

$$dynamicParallelism(..., offsetx, offsety - halfn, ...) \tag{5.13}$$

Por otro lado, la llamada al hijo derecho es de la siguiente forma:

$$dynamicParallelism(..., offsetx + n, offsety + halfn, ...) \tag{5.14}$$

Así, cuando se aplica la ecuación , se puede realizar un correcto mapeo según su posición. Lo mismo ocurre con el método bounding box, pero la diferencia es que los parámetros varían un poco.

En el caso del hijo izquierdo, la llamada es:

$$boundingBox(..., offsetx, offsety - halfn, ...) \tag{5.15}$$

Y la llamada al hijo derecho:

$$boundingBox(..., offsetx + n, offsety, ...) \tag{5.16}$$

ALGORITMO GENERALIZADO

Algoritmo 13: Pseudocódigo algoritmo método Dynamic Parallelism

```
tidx = obtiene coordenada lineal del hilo en el eje x
tidy = obtiene coordenada lineal del hilo en el eje y
// se obtiene índice del array que representa la matriz (2D) en una dimension (1D)
index = (tidy + offsety)*N + (tidx + offsetx)
si tidx & tidy <= n entonces
    matrix[index] <= cost_function()
fin
si tidx & tidy = 0 entonces
    halfn = se divide halfn a la mitad, utilizando bitwise operator (halfn≫1)
    newGrid = se obtiene el nuevo grid en relacion a halfn
    newBlock = blockSize
    boundingBox_Method (matrix, N, n, offsetx, offsety - n)
    //invocamos al hijo derecho BB
    boundingBox_Method (matrix, N, n, offsetx + n, offsety)
en otro caso
    //invocamos al hijo izquierdo DP
    dynamicParallelism(matrix, N, halfn, offsetx, offsety - halfn, depth+1, maxDepth,
     blockSize)
    //invocamos al hijo derecho DP
    dynamicParallelism(matrix, N, halfn, offsetx + n, offsety + halfn, depth+1, maxDepth,
     blockSize)
fin
```

Análisis de resultados

EXPERIMENTOS

Hardware utilizado

Para los experimentos se utilizó un workstation ubicado en el Departamento Matemáticas y Ciencias Computacionales (DMCC) de la Universidad de Chile. Además se realizaron experimentos en un computador de escritorio con características de hardware mucho más simples que el workstation, orientado principalmente al gaming. Cada pc cuenta con las siguientes características de hardware:

Workstation Apophis

- CPU: 2x Procesadores Intel Xeon E5-2640 V3 (x8) Hyperthreading
- RAM: 132 GB
- Sistema Operativo: Fedora Versión 21 (21 - Twenty One)
- Modelo Tarjeta GPU: NVIDIA Corporation GK110BGL [Tesla K40c] (rev a1) (x4)
    - CUDA Cores disponibles por GPU: 2880
    - Memoria disponible por GPU: 12 GB
    - Memoria total en GPU: 48 GB
    - Desempeño punto flotante con doble precisión: 1.43 Tflops
    - Desempeño punto flotante con precisión simple: 4.29 Tflops

PC Escritorio - GTX 960

- CPU: Intel Core i5-4460 CPU 3.20GHz (x4)
- RAM: 8 GB
- Sistema Operativo: Ubuntu 16.04 LTS

- Modelo Tarjeta GPU: NVIDIA Corporation GM206 [GeForce GTX 960] (rev a1)
    - CUDA Cores disponibles: 1024
    - Memoria disponible: 2 GB
    - Desempeño punto flotante con precisión simple: 2.1 Tflops

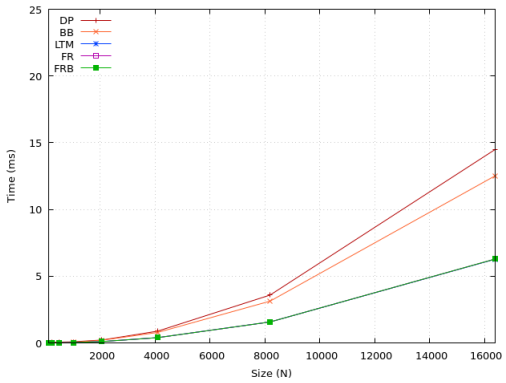
DESARROLLO DE ANÁLISIS

En este capítulo se exponen los resultados de los experimentos asociados a los métodos mencionados anteriormente, tomando el tiempo de ejecución correspondiente a cada método bajo las mismas condiciones, aumentando el tamaño de la matriz en múltiplos de 1024 en cada ejecución. Se realizan pruebas y ajustes, para determinar la configuración más óptima en términos del tiempo bruto empleado, además de determinar el error cuadrático medio en cada medición. Los experimentos se realizaron utilizando toda la capacidad de cómputo del hardware disponible, junto con buscar el óptimo de cada método por separado, comparándolos consigo mismo, pero variando una de las variables parametrizadas con mayor relevancia, dada la forma de mapeo empleada en este trabajo, el cual corresponde al tamaño de los blocks.

RESULTADOS DE LOS EXPERIMENTOS

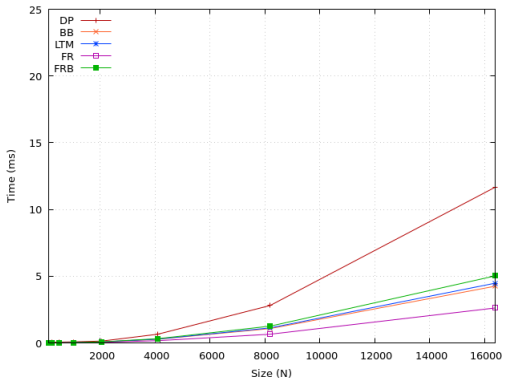
Comportamiento de todos los métodos, sujetos a las mismas condiciones, las cuales son modificar el blocksize con valores de 8, 16 y 32, aumentando el N en múltiplos de 1024, para buscar diferencias de desempeño en Workstation Apophis:

Figura 6.1: Comparación de todos los métodos con Blocksize 8 en Apophis



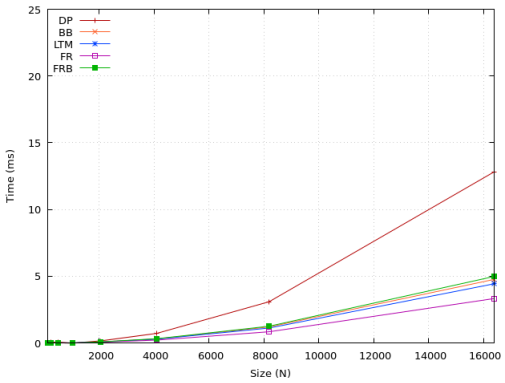
Como podemos apreciar en la figura 6.1, el método Dynamic Parallelism obtiene una performance muy deplorable con respecto a los otros métodos, con N = 16384, alcanza los 14.500981 ms , seguido por el método Bounding Box con un tiempo de 12.530691ms, y luego están los métodos lower triangular method (6.28231ms), flat recursive basic (6.297441ms) y el flat recursive (6.293232ms), con una diferencia de tiempo diminuta.

Figura 6.2: Comparación de todos los métodos con Blocksize 16 en Apophis



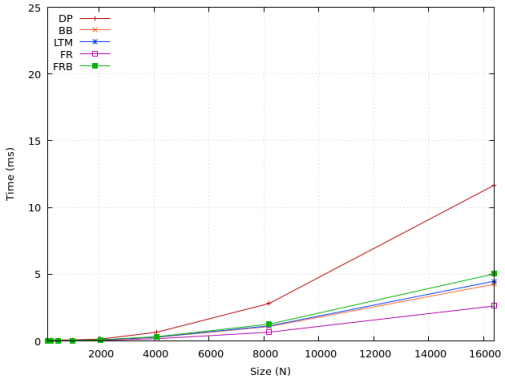
En la figura 6.2, se puede observar que el comportamiento de los métodos en general, con blocksize 16, es superior a los mismo con un blocksize de 8, cabe destacar que el método que más beneficio obtuvo de la modificación del tamaño del blocksize fue el método BB con un tiempo de 4.260016ms, que superó en rendimiento a los métodos LTM (4.484935ms) y FRB(5.033811ms) pero no así al método FR que obtuvo el mejor tiempo con (2.628616ms), si bien el método DP obtuvo una mejora de alrededor del 20 % con respecto a su rendimiento con blocksize de tamaño 8, sigue muy por debajo del nivel de los demás métodos.

Figura 6.3: Comparación de todos los métodos con Blocksize 32 en Apophis



En la figura 6.3 podemos observar el comportamiento de todos los métodos con un blocksize de 32, se puede apreciar un leve declive en el rendimiento de los métodos en comparación a los resultados con blocksize de 16, ésto se puede apreciar en los tiempos, ya que el método DP alcanzó un tiempo de 12.812818ms, mientras que el BB alcanzó un tiempo de 4.768753ms, LTM alcanzó 4.437811ms, FRB alcanzó 4.989709ms y FR alcanzó 3.332782ms

Figura 6.4: Comparación de todos los métodos con Blocksize 16 en Apophis



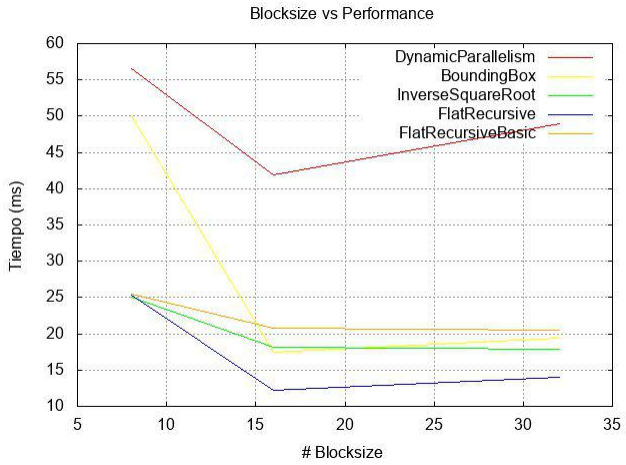
En la figura 6.4, es posible apreciar que para el blocksize 16, gran parte de los métodos sufren una mejora en su performance, lo cual llama la atención debido a que, en teoría, CUDA está optimizado para funcionar en conjuntos de 32 threads, denominados warps. El incremento en la performance de los métodos incluso alcanza el $10{,}35\,\%$ en algunos casos, como por ejemplo el método FR en el workstation obtuvo una mejora que va desde los 13.92ms a los 12.48ms.

Llama la atención que el método BB sea el segundo mejor en gran parte de la ejecución con blocksize 16, la razón de ésto no se puede conocer de una manera trivial, ya que existen

muchos factores involucrados en la ejecución de un kernel en GPU, tantos que es muy complejo el analizarlo teóricamente, por lo tanto y según (Cheng et al., 2014a), sólo existen directivas de buenas prácticas a seguir, como por ejemplo el múltiplo del warp por dimensión, pero todo se reduce a experimentar con las distintas ejecuciones de kernels en distintas condiciones, para poder conocer el comportamiento del kernel mismo.

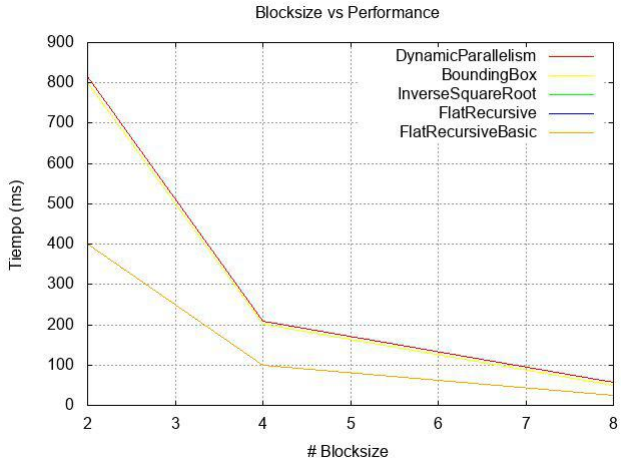
Por otro lado, se han comparado todos los métodos al mismo tiempo, utilizando como parámetro el tamaño del BLOCK SIZE versus el tiempo en milisegundos, esto para determinar el óptimo de performance en un blocksize determinado, como se observa en la figura 6.5:

Figura 6.5: Blocksize vs Performance 16



Una característica importante que se aprecia es que todos los métodos tienen su peak de performance cuando el tamaño del BLOCK SIZE está entre 16 y 32, que es el tamaño recomendado por las buenas prácticas de CUDA(Cheng et al., 2014a).

Figura 6.6: Blocksize vs Performance 32



En términos de performance, podemos apreciar en la figura 6.6 que el método menos eficiente, a pesar de lo que explicaba la teoría, es el método de DP. De acuerdo a lo presentado por(Yang & Zhou, 2015), esto puede deberse al overhead que implica el lanzamiento de un kernel hijo desde otro kernel.

Otra cosa que se puede apreciar es la diferencia entre los dos métodos de flat-recursive. Esto se debe principalmente a los problemas en una de las ecuaciones a resolver que es de la siguiente forma:

$$b = 2^{log2W_y c} \tag{6.1}$$

En la versión básica de flat-recursive se utiliza la función log2f para obtener el resultado, por el contrario, en la versión optimizada del mismo, se utiliza la fórmula 16 de (Navarro et al., 2016), que es equivalente a la fórmula mostrada, pero que es más rápida de computar en GPU, dando una mejora notable a la performance del método.

BUSCANDO LA MEJOR VERSIÓN DE CADA MÉTODO

En la búsqueda del mejor tiempo, o la mejor versión de cada uno de los métodos, comparados consigo mismo, nos encontramos con varias sorpresas.

Método DP obtenido en Workstation Apophis

N	BLOCKSIZE	tiempoDP(ms)	BLOCKSIZE	tiempoDP(ms)	BLOCKSIZE	tiempoDP(ms)
128	8	0.071731	16	0.072641	32	0.075676
256	8	0.073278	16	0.073062	32	0.074517
512	8	0.07578	16	0.07643	32	0.07651
1024	8	0.094713	16	0.09410	32	0.093252
2048	8	0.225772	16	0.146167	32	0.161828
4096	8	0.892404	16	0.660388	32	0.72339
8192	8	3.59841	16	2.815374	32	3.096529
16384	8	14.49944	16	11.672132	32	12.824244
32768	8	58.29832	16	47.897728	32	52.311298

Tabla 6.1: Tiempos Método Dynamic Parallelism en Apophis

Como se puede observar en la tabla 6.1, el método ronda por un milisegundo hasta N = 4096, después de éste punto se comienza a ver el overhead causado por el lanzamiento de muchos kernels, desde ahí el método se dispara en términos de tiempo, quedando fuera de cualquier resultado medianamente competente. Por otra parte, podemos observar que el rendimiento máximo en término de tiempo se logra cuando el tamaño del bloque es 16, logrando una mejora de un $18{,}40\,\%$ versus el blocksize de 8, mientras que la versión con blocksize = 32 tiene un descenso en rendimiento del $-9{,}33\,\%$, todas las diferencias obtenidas son obtenidas con el promedio de los tiempos totales para cada blocksize distinto.

Método DP obtenido del pc escritorio - GTX 960

N	BLOCKSIZE	tiempoDP(ms)	BLOCKSIZE	tiempoDP(ms)	BLOCKSIZE	tiempoDP(ms)
128	8	0.049653	16	0.049358	32	0.049944
256	8	0.051142	16	0.05066	32	0.050956
512	8	0.060255	16	0.058112	32	0.057825
1024	8	0.101952	16	0.083134	32	0.081544
2048	8	0.2978	16	0.214688	32	0.221049
4096	8	1.330902	16	0.872153	32	0.90680
8192	8	5.230913	16	3.725289	32	3.858215
16384	8	21.470209	16	15.4476	32	15.933224

Tabla 6.2: Tiempos Método Dynamic Parallelism en GTX 960

Podemos observar en la tabla 6.2, que la capacidad de cómputo del PC de escritorio sólo llega a un N = 16384, debido al tamaño de la memoria física de la tarjeta que es de 2 Gigabytes ésta no es lo suficientemente amplia para soportar un N = 32768. Enfocándonos en los resultados, podemos observar el mismo patrón existente en los tiempos del método ejecutado en Apophis, y es que el DP obtiene su mejor desempeño cuando el blocksize es 16, obteniendo una mejora de un $28{,}29\,\%$ versus su versión cuando N = 8, mientras que la versión con blocksize de 32, tiene un descenso en su rendimiento de un $-3{,}21\,\%$, todas las diferencias obtenidas son obtenidas con el promedio de los tiempos totales para cada blocksize distinto.

Método BB en Workstation Apophis

N	BLOCKSIZE	tiempoBB(ms)	BLOCKSIZE	tiempoBB(ms)	BLOCKSIZE	tiempoBB(ms)
128	8	0.013441	16	0.013959	32	0.013966
256	8	0.015677	16	0.013389	32	0.014738
512	8	0.024704	16	0.017696	32	0.017951
1024	8	0.06026	16	0.029015	32	0.030947
2048	8	0.207297	16	0.08154	32	0.08838
4096	8	0.795681	16	0.286047	32	0.312786
8192	8	3.143354	16	1.084077	32	1.207227
16384	8	12.529383	16	4.259798	32	4.771259
32768	8	50.07247	16	17.434311	32	19.120428

Tabla 6.3: Tiempos Método Bounding Box en Apophis

El caso del Bounding Box es algo similar al comportamiento del método DP cuando los N son pequeños, ésto es que rondan un milisegundo cuando N es inferior a 4096, con la diferencia que aquí se puede observar de una forma más notoria el impacto del cambio del tamaño del blocksize, ya que cuando blocksize es 16, obtiene una mejora de notable de un $65{,}27\,\%$ versus la versión con blocksize de 8, así mismo y al igual que en el método anterior, la versión de block size de tamaño 32, tuvo una baja en el rendimiento de un $-10{,}15\,\%$ con respecto a su ejecución con blocksize 16, todas las diferencias obtenidas son obtenidas con el promedio de los tiempos totales para cada blocksize distinto.

Método BB en PC escritorio - GTX 960

N	BLOCKSIZE	tiempoBB(ms)	BLOCKSIZE	tiempoBB(ms)	BLOCKSIZE	tiempoBB(ms)
128	8	0.010362	16	0.009862	32	0.00976
256	8	0.012992	16	0.010769	32	0.011177
512	8	0.024377	16	0.01487	32	0.016087
1024	8	0.070187	16	0.035182	32	0.037832
2048	8	0.211649	16	0.11747	32	0.121194
4096	8	0.870513	16	0.422811	32	0.435085
8192	8	3.265332	16	1.659175	32	1.71725
16384	8	13.361833	16	6.599985	32	7.273937

Tabla 6.4: Tiempos Método Bounding Box en GTX 960

Al igual que en Apophis, podemos observar en la tabla 6.4 que el rendimiento máximo del BB se obtiene con un block size igual a 16, aumentando su rendimiento un $50{,}24\,\%$, mientras que el rendimiento del método cuando tiene un block size de tamaño 32 es de $-8{,}48\,\%$ comparado con la ejecución con blocksize = 16, nuevamente todas las diferencias obtenidas son obtenidas con el promedio de los tiempos totales para cada blocksize distinto.

Método LTM en Workstation Apophis

N	BLOCKSIZE	tiempoLTM(ms)	BLOCKSIZE	tiempoLTM(ms)	BLOCKSIZE	tiempoLTM(ms)
128	8	0.013239	16	0.013678	32	0.014131
256	8	0.014754	16	0.013854	32	0.014907
512	8	0.019495	16	0.018683	32	0.018203
1024	8	0.037226	16	0.031135	32	0.030421
2048	8	0.111578	16	0.083721	32	0.084638
4096	8	0.407449	16	0.29747	32	0.292259
8192	8	1.584276	16	1.134726	32	1.127329
16384	8	6.280766	16	4.484449	32	4.440549
32768	8	25.061001	16	17.877754	32	17.668196

Tabla 6.5: Tiempos Método Lower Matriz Triangular en Apophis

En la tabla 6.5 podemos observar el mismo comportamiento que, al igual que los métodos anteriores, ronda un milisegundo de tiempo con N menor a 4096, pero a diferencia de los dos métodos vistos hasta el momento, podemos apreciar que si bien, el método sigue teniendo un rendimiento superior cuando su blocksize es 16 versus el rendimiento con block size de tamaño ocho, un notorio $28{,}55\,\%$, por primera vez la versión con block size de 32 es superior en rendimiento versus la versión con 16, aunque sea por un pequeño margen de $1{,}1\,\%$, todas las diferencias obtenidas son obtenidas con el promedio de los tiempos totales para cada blocksize distinto.

Método LTM en PC Escritorio - GTX 960

N	BLOCKSIZE	tiempoLTM(ms)	BLOCKSIZE	tiempoLTM(ms)	BLOCKSIZE	tiempoLTM(ms)
128	8	0.010098	16	0.009796	32	0.009926
256	8	0.01167	16	0.01073	32	0.010972
512	8	0.018781	16	0.014611	32	0.015332
1024	8	0.052238	16	0.032505	32	0.033219
2048	8	0.120132	16	0.103501	32	0.104686
4096	8	0.598898	16	0.385954	32	0.386298
8192	8	1.919059	16	1.515116	32	1.520081
16384	8	7.621097	16	6.032162	32	6.143716

Tabla 6.6: Tiempos Método Lower Matriz Triangular en GTX 960

En la tabla 6.6 podemos observar una anomalía en términos de rendimiento del método LTM, ésto porque a diferencia de los tiempos obtenidos en Apophis, el método obtiene su mejor rendimiento con un blocksize de 16, con una mejora versus su ejecución con block size de 8 de un $21{,}71\,\%$, por otro lado, con un blocksize de 32, obtiene una baja de rendimiento de un $-1{,}47\,\%$, todas las diferencias obtenidas son obtenidas con el promedio de los tiempos totales para cada blocksize distinto.

Método FR en Workstation Apophis

N	BLOCKSIZE	tiempoFR(ms)	BLOCKSIZE	tiempoFR(ms)	BLOCKSIZE	tiempoFR(ms)
128	8	0.01288	16	0.013272	32	0.013631
256	8	0.014307	16	0.012884	32	0.014153
512	8	0.018712	16	0.015949	32	0.015893
1024	8	0.035973	16	0.023213	32	0.025281
2048	8	0.110078	16	0.053519	32	0.065412
4096	8	0.404909	16	0.176204	32	0.221306
8192	8	1.583034	16	0.663642	32	0.845609
16384	8	6.291693	16	2.627997	32	3.334375
32768	8	25.46637	16	12.4806	32	13.922712

Tabla 6.7: Tiempos Método Flat Recursive en Apophis

En la tabla 6.7 podemos observar la misma tendencia de los demás métodos, el máximo de rendimiento se puede alcanzar con block size de 16, que supera al tiempo de la ejecución con block size 8 en un $52{,}65\,\%$, mientras que la ejecución con blocksize de 32 obtiene una baja en rendimiento de un $-14{,}88\,\%$ versus la ejecución del mismo con blocksize de 16, denuevo todas las diferencias obtenidas son obtenidas con el promedio de los tiempos totales para cada blocksize distinto.

Método FR en PC escritorio - GTX 960

N	BLOCKSIZE	tiempoFR(ms)	BLOCKSIZE	tiempoFR(ms)	BLOCKSIZE	tiempoFR(ms)
128	8	0.009854	16	0.009513	32	0.009389
256	8	0.011635	16	0.01026	32	0.010412
512	8	0.01872	16	0.013319	32	0.013723
1024	8	0.051192	16	0.032244	32	0.033078
2048	8	0.119897	16	0.103475	32	0.104038
4096	8	0.661377	16	0.385463	32	0.386066
8192	8	1.924572	16	1.518809	32	1.516029
16384	8	7.752954	16	6.058421	32	6.142078

Tabla 6.8: Tiempos Método Lower Flat Recursive en GTX 960

En la tabla 6.8 podemos observar el mismo patrón que en la tabla de comparación del método ejecutado en Apophis, donde se obtiene el mayor rendimiento con blocksize de 16, que tiene un rendimiento positivo de un $22{,}92\,\%$ versus su ejecución con block size de 8, y al igual que en Apophis, el rendimiento con block size de 32 es menor al de 16 por un $-1{,}02\,\%$, todas las diferencias obtenidas son obtenidas con el promedio de los tiempos totales para cada blocksize distinto.

Método FRB en Workstation Apophis

N	BLOCKSIZE	tiempoFRB(ms)	BLOCKSIZE	tiempoFRB(ms)	BLOCKSIZE	tiempoFRB(ms)
128	8	0.012903	16	0.013241	32	0.013641
256	8	0.014366	16	0.013377	32	0.014663
512	8	0.01883	16	0.017479	32	0.016629
1024	8	0.036044	16	0.032072	32	0.016629
2048	8	0.110067	16	0.090987	32	0.091222
4096	8	0.405087	16	0.328894	32	0.325136
8192	8	1.584149	16	1.268816	32	1.26145
16384	8	6.296079	16	5.03266	32	4.990633
32768	8	25.494463	16	20.591171	32	20.267815

Tabla 6.9: Tiempos Método Flat Recursive Basic en Apophis

En la tabla 6.9, podemos apreciar que al igual que el método LTM, ejecutado en Apophis, el peak de rendimiento se obtiene con blocksize de 32, con una mejora de un $1{,}37\,\%$ con respecto a la ejecución con blocksize 16, el cual a su vez es superior a la ejecución de con blocksize de 8 por un $19{,}37\,\%$, todas las diferencias obtenidas son obtenidas con el promedio de los tiempos totales para cada blocksize distinto.

Método FRB en PC escritorio - GTX 960

N	BLOCKSIZE	tiempoFRB(ms)	BLOCKSIZE	tiempoFRB(ms)	BLOCKSIZE	tiempoFRB(ms)
128	8	0.009458	16	0.00922	32	0.009201
256	8	0.011586	16	0.010422	32	0.010507
512	8	0.018681	16	0.013846	32	0.01446
1024	8	0.050782	16	0.032561	32	0.03305
2048	8	0.11986	16	0.103442	32	0.104013
4096	8	0.659803	16	0.38543	32	0.386068
8192	8	1.926623	16	1.517213	32	1.51616
16384	8	7.760622	16	6.057889	32	6.142551

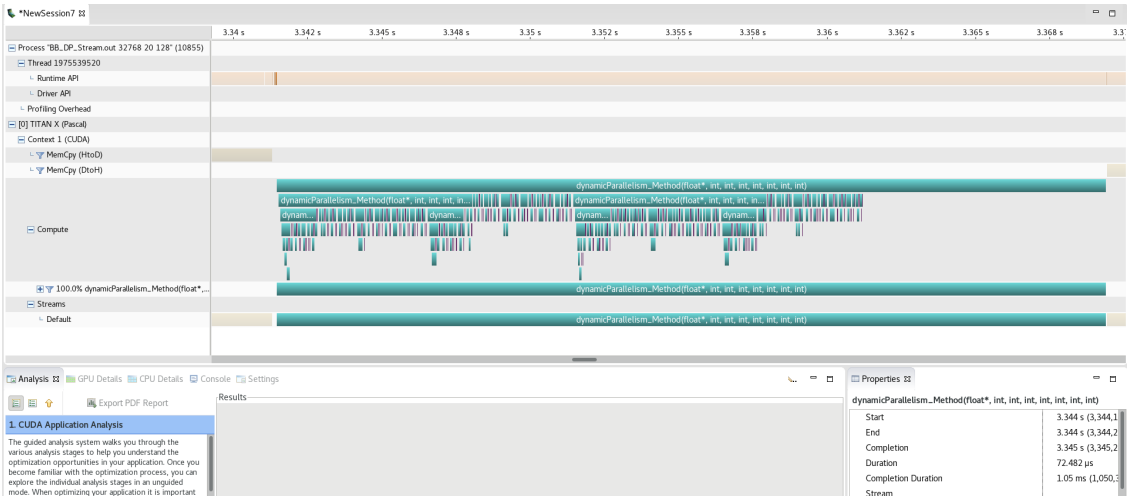
Tabla 6.10: Tiempos Método Lower Flat Recursive Basic en GTX 960

Al igual que con el método LTM, el comportamiento de FRB cambia cuando se ejecuta en el PC de escritorio, ya que a diferencia de la ejecución en Apophis, se alcanza el peak de rendimiento con blocksize de 16, el cual es superior a la ejecución de blocksize 8 por un $22{,}99\,\%$, por otro lado la ejecución con block size 32 tiene una disminución de rendimiento de un $-1{,}05\,\%$ con respecto a la ejecución del mismo con blocksize 16, todas las diferencias obtenidas son obtenidas con el promedio de los tiempos totales para cada blocksize distinto.

Análisis del método DP

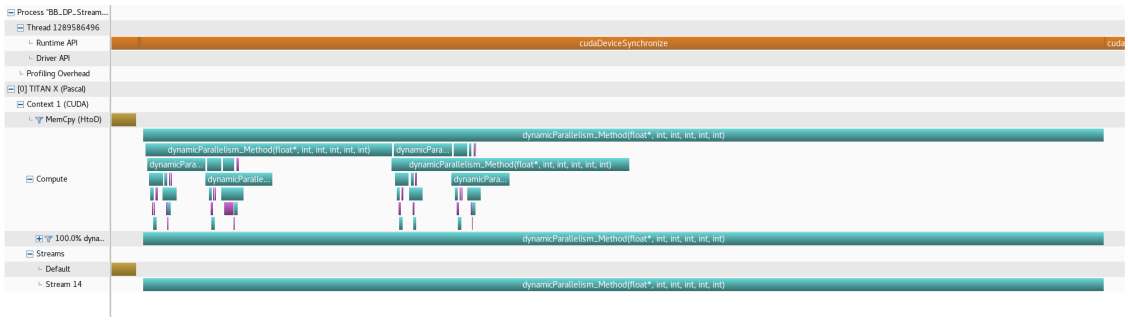
Como se puede notar en los experimentos, el método Dynamic Parallelism obtiene un rendimiento muy inferior en comparación a los demás métodos, así que utilizamos una de las herramientas de CUDA llamada NVIDIA Visual Profiler, la cual muestra gráficamente el comportamiento de los algoritmos ejecutados y los resultados del método se pueden observar en la siguiente figura:

Figura 6.7: Método Dynamic Parallelism con N = 32768 en NVIDIA Visual Profiler.



En una ejecución más simple se puede observar mejor el problema:

Figura 6.8: Método Dynamic Parallelism con N = 4096 en NVIDIA Visual Profiler.

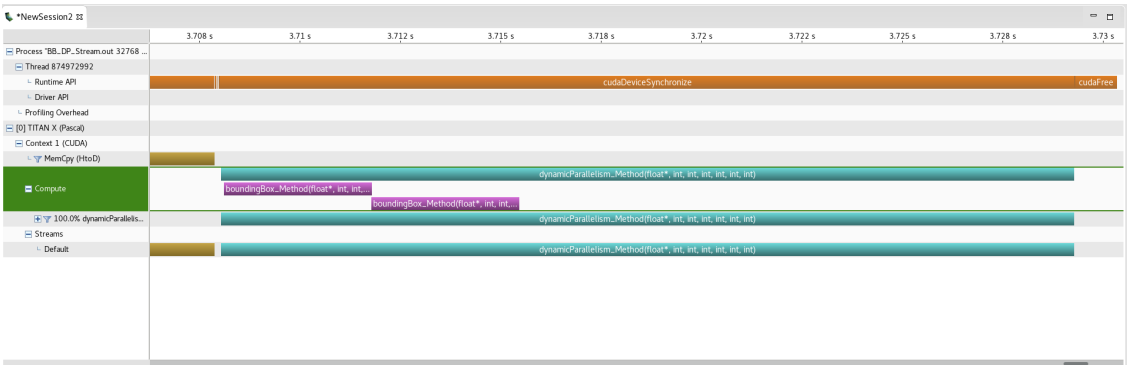


La figura 6.8 muestra el método Dynamic Parallelism ejecutado con blocksize de 32 en Apophis, se puede observar al comienzo de la ejecución la copia de memoria de Host al Device MemCpy (HtoD), luego comienza lo interesante, en la sección Compute se puede observar la ejecución principal de Dynamic Parallelism, donde en la parte superior color azul, se encuentra la ejecución del kernel principal o el denominado kernel padre, luego inmediatamente abajo se pueden observar las recursiones del método (demostrando de paso el cómputo en paralelo), pero lo interesante es que aproximadamente la mitad del tiempo de vida de cada kernel, éstos no ejecutan acción alguna, como se explica en (Yang & Zhou, 2015), es sabido que existe un overhead en el lanzamiento de kernels desde otros kernels y ésto podría explicar el porqué del comportamiento deplorable del método, lamentablemente determinar el porqué de éste comportamiento

no es algo trivial, buscando respuestas nos encontramos con Robert Crovella [1], trabajador de NVIDIA, especialista en CUDA, quien nos comentó que efectivamente las llamadas dentro de un kernel son costosas, así como son costosas las llamadas en CPU.

Para asegurarnos de que el comportamiento se daba para todos los niveles, decidimos ejecutar el método DP pero con un N igual al límite de corte, que llama al bounding box, lo que se traduce en un kernel padre que realiza dos llamadas a kernels, que en éste caso son los métodos BB, obteniendo el resultado que se observa en la siguiente figura:

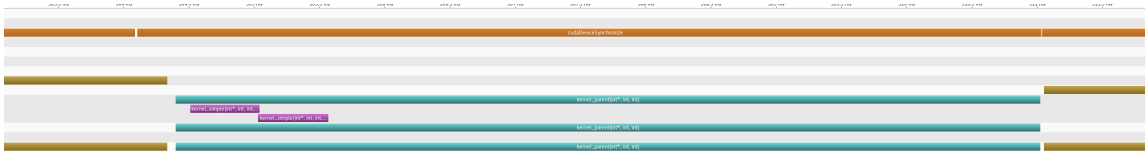
Figura 6.9: Método Dynamic Parallelism con N = 4096 en NVIDIA Visual Profiler.



Como se puede apreciar en la figura 6.9, el comportamiento es igual a la ejecución con más de una recursión, la ejecución del kernel permanece en éste caso más tiempo sin realizar trabajo alguno, es importante señalar que los métodos BB no se ejecutan en paralelo a propósito.

Entonces, es completamente válido hacerse la pregunta, ¿Es problema de la implementación? Para descartar esto, realizamos un problema trivial en DP, que básicamente convierte un arreglo de ceros a unos y lo analizamos con el Visual Profiler, obteniendo lo siguiente:

Figura 6.10: Dummy en NVIDIA Visual Profiler.



En la figura 6.10 podemos apreciar el mismo comportamiento visto anteriormente,

[1] Perfil de StackOverflow: https://stackoverflow.com/users/1695960/robert-crovella

lo que descarta completamente que la implementación del método DP sea el causante de su comportamiento.

Conclusiones

Una de las conclusiones más importantes que podemos deducir de los experimentos realizados, es el rendimiento del método Dynamic Parallelism, ya que como se pudo observar, su rendimiento es mucho más bajo de lo esperado. Teóricamente hablando, el método debiera estar rondando el rendimiento del Bounding Box, e incluso ganarle bajo condiciones normales, pero como se constantó anteriormente, existe un overhead demasiado notorio en el procesamiento del kernel padre, e incluso de los kernels hijos, lo que afecta notoriamente el rendimiento que podría haber alcanzado gracias a la paralelización.

Además queda en manifiesto a través de los experimentos que si bien Dynamic Parallelism es una herramienta poderosa, no es la ideal para todos los problemas de la ciencia de la computación. En el caso que interesa en esta memoria, si bien realiza el mapeo del espacio del cómputo, el rendimiento del método DP no alcanza a ser competente al ser comparado con los demás métodos, especialmente cuando éstos hacen un mapeo directo, sin necesitar de la llamada a múltiples kernels. Lo anterior deja en evidencia que la sobrecarga que conlleva la invocación de más de un kernel recursivamente, que si bien existe y ha sido estudiada por algunos investigadores (Yang & Zhou, 2015), afecta negativamente el rendimiento del método, al punto de no tener un rendimiento medianamente satisfactorio.

Podemos decir que si bien Dynamic Parallelism es una herramienta poderosa, el compararlo con métodos que no utilizan recursividad, sino que realizan el mapeo de los blocks *en una pasada*, resulta por decirlo menos, injusto. Aunque se sabe de la existencia de la latencia a la hora del lanzamiento de kernels en Dynamic Parallelism, la teoría apuntaba a que no debería ser tan notoria. Pero al contrastar dicho ese escenario frente a la no latencia de los demás métodos, que sólo están restringidos por el costo de cómputo de las operaciones matemáticas involucradas, evidentemente no tuvo punto de comparación medianamente satisfactorio.

Por otra parte, se puede mencionar que algunas optimizaciones críticas, como la que se le hizo al método flat-recursive, influyen significativamente en los resultados de cada técnica. Así, se infiere que el costo de las operaciones matemáticas, a este nivel de experimentación, es demasiado importante y crítico, tomando en cuenta que se busca optimizar no sólo la técnica en sí, sino que la resolución de problemas complejos que pueden tardar horas e incluso días de cómputo en resolverse.

Es importante señalar que en casi todos los métodos se logra el máximo de rendimiento con un blocksize de 16. Según el CUDA Programming Guide (Cheng et al., 2014b), se sugiere probar con varias configuraciones hasta encontrar el máximo de rendimiento. Pero es interesante señalar que se menciona que el tamaño del blocksize debería ser un múltiplo de

warpsize, que es de tamaño 32, pero el blocksize es menor a éste número.

Gracias al experimento de comparación de todos los métodos, podemos concluir que el método flat recursive es el completo ganador, ya que es superior a todos los demás métodos en todas las comparaciones realizadas, obteniendo peak de performances de alrededor de un milisegundo con un promedio de 1.785 ms para blocksize de 16 versus un 2.5799 ms para el Bounding Box en las mismas condiciones, seguidos por el Flat Recursive Basic con un tiempo de 2.6617 ms.

Trabajos futuros

Como trabajo futuro, sería interesante comparar el método Dynamic Parallelism con su par en CPU, que es la comparación estándar de CPU vs GPU, ya que si bien el comportamiento del DP no fue el esperado versus los demás métodos de mapeo, podría llegar a ser mejor que su versión en CPU, lo cual tiene un valor agregado.

Los experimentos se realizaron en la arquitectura Kepler y Maxwell, sería interesante observar el comportamiento de todos los métodos (en especial el de Dynamic Parallelism) en la última arquitectura lanzada por NVIDIA hasta la fecha, que es la arquitectura Pascal, más aún, en una tarjeta orientada al HPC, que es un tipo de tarjeta óptima para el cómputo en GPU.

Si bien el flat-recursive es el método más rápido encontrado en la comparación de esta memoria según el propio autor, éste método puede mejorarse aún más y alcanzar tiempos más acotados.

También sería interesante optimizar al máximo el DP. Según (Cheng et al., 2014a), se puede modificar el tamaño de los blocks para que no sean de la forma NxN, para así tratar de alcanzar mayor paralelismo y aumentar el ancho de banda de la tarjeta, para analizar el comportamiento del método DP en dichas condiciones, y quizás se podría alcanzar una mejora de rendimiento.

Muchos papers estudian la posibilidad de utilizar lo mejor de los dos mundos en sistemas CPU-GPU, que realizan el trabajo compartido entre los dos dispositivos, quizás convertir la técnica del Dynamic Parallelism en una que utilice CUDA + MPI o CUDA + OpenMP sería interesante de analizar.

Bibliografía

Adinetz, A. (2014). Adaptive parallel computation with cuda dynamic parallelism. https://devblogs.nvidia.com/parallelforall/introduction-cuda-dynamic-parallelism/. Accessed: 2017-07-15.

AMD (2017). Amd ryzen release.
URL https://www.youtube.com/watch?v=1v44wWAOHn8)

Amdahl, G. M. (1967). *Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities*, (pp. 483–485). AFIPS '67 (Spring). New York, NY, USA: ACM.
URL http://doi.acm.org/10.1145/1465482.1465560

Avril, Q., Gouranton, V., & Arnaldi, B. (2012). Fast collision culling in large-scale environments using gpu mapping function. *Research*, (pp. 71–80).

Breshears, C. (2009). *The Art of Concurrency: A Thread Monkey's Guide to Writing Parallel Applications*. O'Reilly Media, Inc.

Cao, Y. (2011). *Many-Core Architecture Oriented Parallel Algorithm Design for Computer Animation*, (pp. 180–191). Berlin, Heidelberg: Springer Berlin Heidelberg.
URL https://doi.org/10.1007/978-3-642-25090-3_16

Cheng, J., Grossman, M., & McKercher, T. (2014a). *Professional CUDA C programming*. Indianapolis: Wrox.

Cheng, J., Grossman, M., & McKercher, T. (2014b). *Professional CUDA C programming*. Indianapolis: Wrox.

De Borja, F., & Garcia, E. (2010). Transistores finfet. *Research*, (p. 1).
URL http://ibdigital.uib.es/greenstone/collect/enginy/index/assoc/Enginy_2/010v02p0/05.dir/Enginy_2010v02p005.pdf

del Rio, F. D., Garcia, J. S., & Sevillano, J. L. (2016). Extending amdahl's law for the cloud computing era. *Computer*, (pp. 14–22).

Dietz, H., & Dalton, y. (2009). Imd interpretation on a gpu. *Research*.
URL http://aggregate.ee.engr.uky.edu/EXHIBITS/SC09/mogsimlcpc09final.pdf

DiMarco, J., & Taufer, M. (2013). Performance impact of dynamic parallelism on different clustering algorithms. *Proc.SPIE*.
URL http://dx.doi.org/10.1117/12.2018069

Flores Carapia, R., Silva-GarcÃa, V., & RenterÃa, C. (2012). Monte carlo scheme: Cryptography application. , *6*, 6761–6767.

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

Foster, I. (1995). *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Giles, M. (2017). Lecture 0: Cpus and gpus. http://people.maths.ox.ac.uk/gilesm/cuda/lecs/lec0.pdf. Accessed: 2017-07-10.

GPGPU (2017). About gpgpu.org. http://gpgpu.org/about. Accessed: 2017-06-22.

Grama, A. (2003). *Introduction to Parallel Computing*. Pearson Education. Addison-Wesley.
URL https://books.google.cl/books?id=B3jR2EhdZaMC

Gremilllion, B. (2017). Nvidias next generation cuda compute architecture. *Research*, (p. 1).
URL https://www.uxpin.com/studio/blog/a-hands-on-guide-to-mobile-first-design/

Gustafson, J. L. (1988). Reevaluating amdahl's law. *Commun. ACM*, *31*(5), 532–533.
URL http://doi.acm.org/10.1145/42411.42415

Gustafson, J. L. (1990). *Fixed Time, Tiered Memory, and Superlinear Speedup*, (pp. 1255–1260). .

Hyuk, J., & OLeary, D. (2008). Exploiting structure of symmetric or triangular matrices on a gpu. *Research*.
URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.124.7313&rep=rep1&type=pdf

Intel (1971). Intel's first microprocessor. https://www.intel.com/content/www/us/en/history/museum-story-of-intel-4004.html. Accessed: 2017-06-13.

Intel (1975). 8080 microcomputer systems users manual. http://www.hartetechnologies.com/manuals/Intel/Intel\%208080\%20System\%20Manual_a.PDF. Accessed: 2017-06-15.

Intel (2017a). ntel announces new 22nm 3d tri-gate transistors. https://www.intel.la/content/www/xl/es/silicon-innovations/standards-22nm-3d-tri-gate-transistors-presentation.html. Accessed: 2017-06-21.

Intel (2017b). Procesador intel pentium 4. https://ark.intel.com/es/products/27504/Intel-Pentium-4-Processor-supporting-HT-Technology-3_40-GHz-512K-Cache-800-MHz-FSB. Accessed: 2017-06-18.

L Harrison, R. (2010). Introduction to monte carlo simulation. , *1204*, 17–21.

McClanahan, C. (2010). Cramming more components onto integrated circuits. *Research*, (p. 4). URL http://disi.unal.edu.co/~gjhernandezp/HeterParallComp/GPU/gpu-hist-paper.pdf

Medina, M. (2014). Familia de procesadores intel x86. http://mondrian.die.udec.cl/~mmedina/Clases/ProgPar/Apuntes2014-2/04-Familia\%20Intel\%20x86.pdf. Accessed: 2017-06-18.

Michalakes, J., & Vachharajani, M. (2008). *GPU acceleration of numerical weather prediction*, (pp. 1–7). .

MITS (1975). Altair 8800 operators manual. http://www.classiccmp.org/dunfield/altair/d/88opman.pdf. Accessed: 2017-06-17.

Moore, G. (1965). Cramming more components onto integrated circuits. *Research*. URL https://www.cis.upenn.edu/~cis501/papers/mooreslaw-reprint.pdf

Moore, G. (1975). Progress in digital integrated electronics. *Research*. URL http://ieeexplore.ieee.org/document/1478174/

Navarro, C., Bustos, B., & Hitschfeld, N. (2016). Possibilities of recursive gpu mapping for discrete orthogonal simplices. *Research*. URL https://scholar.google.com/citations?view_op=view_citation&hl=en&user=RNYDdjEAAAAJ&sortby=pubdate&citation_for_view=RNYDdjEAAAAJ:YsMSGLbcyi4C

Navarro, C., & Hitschfeld, N. (2013). Improving the gpu space of computation under triangular domain problems. *Research*. URL https://www.dcc.uchile.cl/TR/2013/TR_DCC-20130806-004.pdf

Navarro, C. A., Hitschfeld-Kahler, N., & Mateu, L. (2014). A survey on parallel computing and its applications in data-parallel problems using gpu architectures. *Communications in Computational Physics*, *15*(2).

NVIDIA (2012). Dynamic parallelism in cuda. http://developer.download.nvidia.com/assets/cuda/files/CUDADownloads/TechBrief_Dynamic_Parallelism_in_CUDA.pdf. Accessed: 2017-07-13.

NVIDIA (2013). Cuda toolkit 5.0 - archive.
URL https://developer.nvidia.com/cuda-toolkit-50-archive)

NVIDIA (2017a). About cuda. https://developer.nvidia.com/about-cuda. Accessed: 2017-06-22.

NVIDIA (2017b). *CUDA C PROGRAMMING GUIDE*. NVIDIA.
URL http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

NVIDIA (2017c). Cuda faq.
URL https://developer.nvidia.com/cuda-faq

NVIDIA (2017d). Cuda zone.
URL http://www.nvidia.com/object/cuda_home_new.html

NVIDIA (2017e). Dynamics (md) on gpus. http://images.nvidia.com/content/tesla/pdf/Molecular-Dynamics-July-2017-MB-slides.pdf. Accessed: 2017-06-12.

NVIDIA (2017f). Geforce gtx 10 series. https://www.nvidia.com/en-us/geforce/products/. Accessed: 2017-06-30.

NVIDIA (2017g). Nvidia geforce 256 ddr.
URL https://videocardz.net/nvidia-geforce-256-ddr/

NVIDIA (2017h). Nvidias next generation cuda compute architecture. *Whitepaper*, (p. 5).
URL http://international.download.nvidia.com/pdf/kepler/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf

Othman, N. F., & Arshad, H. (2011). *Binary search tree traversal for Arrick Robot virtual assembly training module*. .

P., D., & H., D. (2012). *Java for Programmers*. Prentice Hall; 9th edition (March 7, 2011).

Peelle, H. (1974). To teach newtons square root algorithm. *Research*.
URL https://dl.acm.org/citation.cfm?doid=585882.585889

Ries, F., De mArco, T., & Zivieri, M. (2009). Triangular matrix inversion on graphics processing unit. *Research*.
URL https://dl.acm.org/citation.cfm?id=1654069

Sahni, S., & Thanvantri, V. (2002). Parallel computing: Performance metrics and models. .

Schatz, M., Trapnell, C., Delcher, A., & Varshney, A. (2007). High-throughput sequence alignment using graphics processing units. *Research*.
URL https://bmcbioinformatics.biomedcentral.com/track/pdf/10.1186/1471-2105-8-474?site=bmcbioinformatics.biomedcentral.com

Scott, T. (2004). Cramming more components onto integrated circuits. *Research*, (p. 49).
URL https://pdfs.semanticscholar.org/8f68/c39e3925275e1d5a881619fb37944b0c4e31.pdf

Smith, R. (2014). The nvidia geforce gtx 980 review: Maxwell mark. https://www.anandtech.com/show/8526/nvidia-geforce-gtx-980-review/2. Accessed: 2017-06-30.

Spector, A., & Gifford, D. (1984). The space shuttle primary computer system. *Commun. ACM*, *27*(9), 872–900.
URL http://doi.acm.org/10.1145/358234.358246

Stallings, W. (2009). *Operating Systems: Internals and Design Principles*. GOAL Series. Pearson/Prentice Hall.
URL https://books.google.cl/books?id=dBQFXs5NPEYC

Stone, J. (2010). Heterogeneous gpu computing for molecular modeling. *Research*.
URL http://www.nvidia.com/content/PDF/sc_2010/CUDA_Tutorial/SC10_Heterogeneous_GPU_Computing_for_Molecular_Modeling.pdf

Tang, X., Pattnaik, A., Jiang, H., Kayiran, O., Jog, A., Pai, S., Ibrahim, M., Kandemir, M. T., & Das, C. R. (2017). Controlled kernel launch for dynamic parallelism in gpus. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, (pp. 649–660).

Trapnell, C., & Schatz, M. C. (2009a). Optimizing data intensive gpgpu computations for dna sequence alignment. *Parallel Comput.*, *35*(8-9), 429–440.
URL http://dx.doi.org/10.1016/j.parco.2009.05.002

Trapnell, C., & Schatz, M. C. (2009b). Optimizing data intensive gpgpu computations for dna sequence alignment. *Parallel Computing*.
URL http://www.sciencedirect.com/science/article/pii/S0167819109000714

Yang, Y., & Zhou, H. (2015). Cuda-np: Realizing nested thread-level parallelism in gpgpu applications. *Research*.
URL https://link.springer.com/article/10.1007/s11390-015-1500-y

ZURB (2017). Mobile first : An adaptive, future-friendly solution for website design. https://zurb.com/word/mobile-first. Accessed: 2017-06-22.