Aplicación de *Paralelismo Dinámico* para el mapeo eficiente de threads de GPU en problemas de dominio tetraédrico

Gabriel A. González
Instituto de Informática,
Universidad Austral de Chile,
Valdivia, Chile
Email: gabriel.gonzalez.uribe@alumnos.uach.cl

Abstract

In GPU computing there is a stage in the execution pipeline where threads are mapped to the data domain. Under certain situations such as triangular domain problems, the traditional programming would generate threads that will be discarded at runtime. This is a problem, since it makes the programming in GPU inefficient in terms of wasted execution time and space in execution threads. The main purpose of this article is to study the *dynamic parallelism* feature applied to tetrahedral domain problems. First, we study the case where the mapping is performed using dynamic parallelism, i.e. blocks of execution threads are recursively generated. Then, we consider the brute force method where threads are discarded at runtime. Based on the results, we concluded that the dynamic parallelism approach did not offer any speedup for Kepler and Pascal GPU architectures, but for Volta it offered a remarkable improvement of up to $\sim 3\times$. The results obtained in this article make the mapping through Dynamic Parallelism an alternative to be considered in Volta GPUs for the solution of tetrahedral domain problems.

Keywords

GPU computing; Dynamic Parallelism; Mapping threads ;

I. Introducción

La programación en GPU es un recurso que en este último tiempo ha contribuido en el avance de varias áreas de la informática. Donde más se destaca este avance es en el área de los video juegos, animación 3D y todo lo que respecta a procesamiento gráfico en general. Pero no solo en este aspecto ha contribuido la GPU, sino que también en el área de la computación de alto rendimiento (HPC). La razón de este crecimiento es debido a la arquitectura de la GPU, que a diferencia de la CPU que tiene uno o varios núcleos, de gran potencia, la GPU está constituida de muchos núcleos (de un orden de miles o más) por lo que se puede aprovechar bastante el paralelismo masivo en este tipo de arquitectura, logrando, en gran parte de los casos, tiempos de ejecución significativamente más bajos, que en una CPU convencional. De esta forma la computación de alto rendimiento puede beneficiar áreas como la inteligencia artificial, cálculos financieros, análisis de datos, investigaciones científicas, entre otras.

Una de las principales características que posee el modelo de computo de las GPUs, es que están diseñadas para trabajar con dominios lineales, cuadrados, rectangulares o paralelepípedos en el caso de dominios en tres dimensiones. Al ejecutar un cálculo en paralelo, todos los *threads* primero deben agruparse geométricamente como una lista, matriz o paralelepípedo dependiendo si el problema es 1D, 2D o 3D, respectivamente. Idealmente el dominio de datos del problema debiese seguir la misma geometría que la que contiene los *threads* agrupados. Cuando esto se cumple, la GPU logra utilizar todos sus *threads* de manera eficiente ya que todos estos participan haciendo parte del trabajo necesario. Sin embargo, cuando la geometría del dominio de datos es distinta a la geometría que contiene los *threads*, surge un problema de rendimiento, ya que la geometría que contiene los *threads* no coincidirá e inevitablemente existirán *threads* innecesarios que serán descartados en tiempo de ejecución. Un caso que se presenta en algunos problemas de la ciencia y tecnología es el de dominio tetraédrico. Este problema geométricamente se puede representar como un tetraedro. Para este problema particular surge la siguiente pregunta de investigación. ¿Pueden las últimas tecnologías de GPU, tales como *Dynamic Parallelism*, reducir la cantidad de threads para dominios de tetraedro en tres dimensiones y entregar una mejora en rendimiento como consecuencia?

Este trabajo busca estudiar *Dynamic Parallelism* o paralelismo dinámico en problemas donde el dominio del problema modelado en $\mathcal{R}^3$ puede resultar un tetraedro. Estos problemas se llamarán de ahora en adelante *problemas de dominio tetraédrico*. Este tipo de problemas pueden existir en áreas como la física, química y ciencias en general tales como la sistemas dinámicos, simulaciones y ecuaciones en derivadas parciales (PDEs) [1] [2]. El desafío desde la ciencia de la computación es buscar formas eficientes para mapear *threads* a espacios de tipo tetraedro.

La problemática presentada en este trabajo es análogo a *td-problem* [3] donde se trata el espacio de mapeo en dos dimensiones sobre dominios triangulares en los cuales los *threads* que están por sobre la línea diagonal (cuando $x > y$) se descartan. Algo

similar ocurre en tres dimensiones donde los *threads* que están por sobre el plano diagonal $x + y > z$ se descartan. El principal hallazgo de este estudio, es que no existe una mejora notable en la eficiencia al utilizar *dynamic parallelism* excepto para el caso en que se usa la GPU Tesla V100 donde se encontró una mejora de hasta $\sim 3\times$ en comparación al método fuerza bruta.

El artículo se encuentra estructurado, para un mejor entendimiento, comenzando por la sección "Modelo De Programación", donde se explican algunos conceptos para entender el artículo, como CUDA, *dynamic parallelism* y problemas de dominio tetraédrico. Luego sigue la sección "Trabajos Relacionados", donde se contextualizará este trabajo describiendo brevemente algunos trabajos relacionados a este. La sección "Solución Propuesta", explica con detalle el desarrollo de la solución propuesta. En "Resultados de Rendimiento", se da a conocer el resultado de este algoritmo recursivo probado en cuatro distintas máquinas y comparando este algoritmo con una técnica más convencional. En "Discusión" se analizan los gráficos con más detalle y se plantean posibles trabajos futuros sobre los resultados arrojados en este estudio. Finalmente, en la sección "Conclusión" se obtendrán conclusiones de este trabajo en general.

II. Modelo de programación en GPU

En esta sección se abordan algunos conceptos fundamentales para la solución propuesta. Se explica el modelo de programación CUDA, el paradigma utilizado para la resolución del problema que es *dynamic parallelism* y el concepto de problema de dominio tetraédrico.

A. CUDA

Para el desarrollo de este estudio se utilizó CUDA, cuya sigla en ingles significa *Compute Unified Device Architecture* [4], lenguaje basado en C que funciona mediante una jerarquización espacial para distribuir el computo en los núcleos de una GPU. Dentro de la terminología básica de CUDA se encuentran los términos *Host* y *Device*, cuando se menciona el termino *host* se refiere a la CPU. Mientras que el termino *device* se refiere a la GPU. En la GPU existe una memoria que almacena los datos a utilizar en los cálculos de la GPU, es por esto que existe un tiempo de traspaso de datos entre *device* y *host* que se puede percibir cuando la matriz de datos a tratar es muy grande y el cálculo se hace una sola vez, pero que, en este caso, se omite dado a que este tiempo se vuelve imperceptible a medida que la cantidad de computo supera a la cantidad de traspasos entre *host* y *device*.

En CUDA el código que se ejecuta y es invocado a través de otro lenguaje como en este caso es C, se llama *kernel*, en cada *kernel* se puede especificar si el código se ejecuta en *host* o *device*. Si el código se ejecuta en *device*, cada *kernel* recibe el número de *threads* en donde se ejecutará, estos *threads* se ejecutan de forma paralela en la GPU, organizados mediante una jerarquización espacial. Como se puede ver en la Figura 1, esta jerarquización comienza con los *threads* de ejecución. Luego, le siguen los bloques o *blocks*, donde cada *block* está compuesto por N *threads* de ejecución [1]. En el último nivel se encuentra lo que se conoce como *grid*, que es la agrupación de los *blocks* en una malla o grilla. Para ejecutar un *kernel* en la GPU se debe especificar el tamaño del *block* (número de *threads* por *block*) y el tamaño de la grilla o *grid* (número de *blocks* por *grid*). Cada *thread* tiene un índice o identificador dentro del *block* y cada *block*, tiene un índice dentro del *grid*.

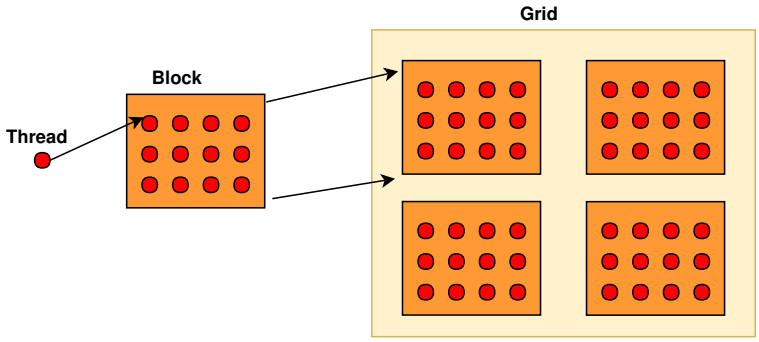


Fig. 1: Representación de jerarquía espacial en CUDA

B. Dynamic Parallelism

Dynamic Parallelism [5] [3], es una técnica utilizada en programación en GPU, que se ha promocionado como una técnica que extiende el modelo de programación y a la vez puede mejorar el rendimiento. Sin embargo, esta última propiedad no se ha podido confirmar claramente en la literatura actual. Esta técnica consiste en hacer llamadas recursivas de *kernels* de CUDA, donde cada *kernel* ejecuta un cierto número *threads* de forma paralela en los núcleos de la GPU. Es decir, como se ve en la Figura 2, desde un *kernel* en ejecución con múltiples *threads*, es posible lanzar otro *kernel* de ejecución que a su vez también

[1] El tamaño y la elección de *block* se explican en sección IV-C.

puede lanzar más *kernels* en ejecución. Esta técnica se puede utilizar en los siguientes casos: (1) cuando la estructura de datos es jerárquica, (2) cuando existe la posibilidad de utilizar recursividad o (3) el problema se puede dividir en lotes que se pueden resolver paralelamente.

Esto es importante para este estudio, ya que lo que busca *dynamic parallelism* es que la GPU evite hacer trabajo que no es necesario para la resolución del problema haciendo el proceso más eficiente, por lo que se utiliza *dynamic parallelism* para que no sea necesario descartar tantos *threads* en ejecución ya que no se crean más *threads* de los necesarios durante el proceso[2].

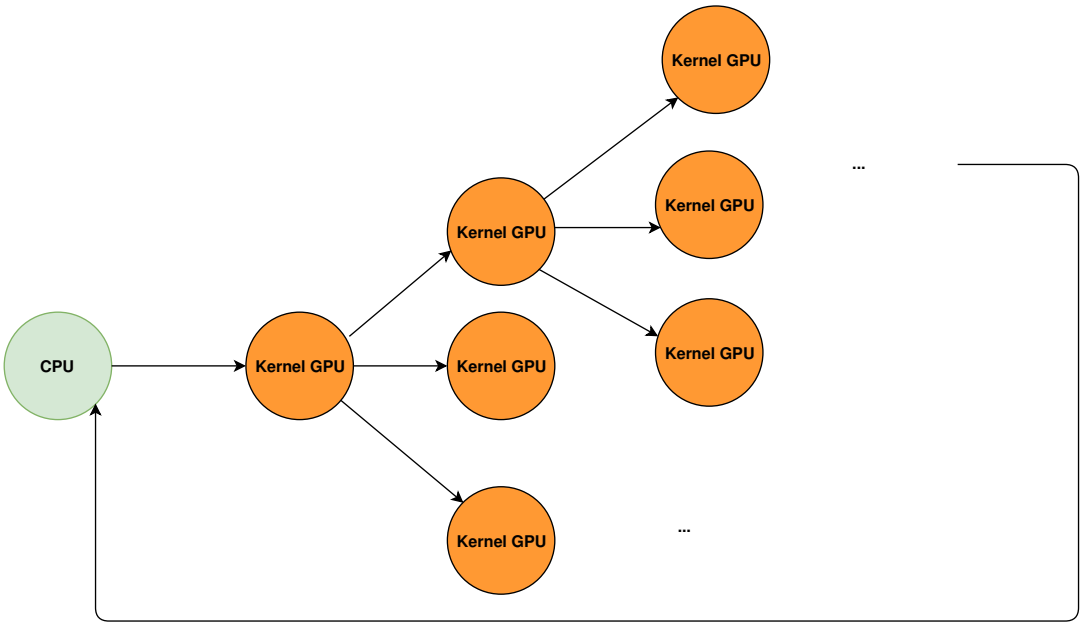


Fig. 2: Esquema conceptual del paradigma *Dynamic Parallelism*

C. Problemas de dominio tetraédrico

En múltiples áreas científicas como la química, física y biología, existe un tipo de problema donde el dominio de datos en donde el computo se realiza puede ser, por un lado, el área bajo la linea diagonal interceptada por los ejes x e y formando un área triangular y que se puede modelar en superficies de dos dimensiones. Sin embargo, también existe un problema donde su dominio puede formar un tetraedro de forma análoga al caso de 2D, el cual se puede modelar en superficies de 3D. Aquí es donde nacen los problemas de dominio tetraédrico [6] [2]. Cada vez crece la necesidad de solucionar estos problemas de forma más eficiente, dado a que al momento de mapear los datos existentes se genera una redundancia, esto se produce debido a que los datos que están bajo de la diagonal, en el caso dos dimensiones, se repiten con los datos que están por sobre la diagonal, haciendo que exista un cálculo innecesario ocupando más recursos de los necesarios en la GPU. Esta redundancia de datos también aparece cuando se buscan todos los tríos posibles de un conjunto de n elementos, haciendo que en algunos de estos tríos generen el mismo resultado, provocando la redundancia anteriormente mencionada. Un ejemplo de este tipo de problemas es el de las ecuaciones diferenciales parciales, cuyas funciones de frontera pueden formar una superficie de dominio tetraédrico y al hacer la simulación por diferencias finitas se genera la necesidad de descartar los cálculos que están por sobre el plano diagonal. Por lo que resultaría interesante encontrar una forma más eficiente de trabajar con este tipo de problemas.

III. Trabajo Relacionado

Los trabajos relacionados a este estudio tienen como eje central la búsqueda de formas eficientes de trabajar con GPU en cuanto a tiempo de ejecución y espacio en memoria.

C. A. Navarro y N. Hitschfeld, estudiaron *td-problem (problema de dominio triangular)* en $\mathcal{R}^2$ [7]. Este trabajo se centra en hacer un análisis comparativo de cuatro formas de mapeo de *threads* de ejecución: *bounding box (BB), upper-triangular mapping (UTM), rectangular box (RB) y recursive partition (REC)*. Además proponen un nuevo método denominado $g(\lambda)$, este mapeo se basa en la propiedad de matriz triangulo inferior. Para este estudio se utilizaron tres máquinas de Nvidia (GTX765M, GTX680, Tesla K40). De este estudio se llegó a la conclusión que este último tiene un 17% de mejora con respecto a *bounding box (BB)*. Otro trabajo, presentado por Jin Hyuk Jung [8] mostró como se puede explotar la rasterización triangular de matrices simétricas en GPU, resultando beneficioso en cuanto a rendimiento en comparación a algoritmos que son diseñados para CPU.

[2]La determinación del limite de recursiones y *threads* a ejecutar en este estudio se explica en la sección IV-B.

C. A. Navarro et al., plantearon y analizaron la posibilidad de realizar mapeos eficientes de forma recursiva (donde se podría aplicar *dynamic parallelism*) en uno, dos, tres o n dimensiones [9]. En este trabajo se planteó un incremento de entre dos y seis veces de eficiencia utilizando algoritmos recursivos, sin embargo no se comprobó esta teoría empíricamente.

Cristóbal A. Navarro et al. [10] explicaron los conceptos básicos de la computación paralela y en especial en GPU. Explicó algunos ejemplos de aplicación que son ideales para sacarle provecho a la computación de alto rendimiento en GPU. También mencionó los problemas de mapeo de *threads* en GPU dando a entender los conceptos con algunos ejemplos.

Penporn Koanantakool et al., estudiaron los problemas en donde interactúan 3 cuerpos [1] y la necesidad para las ciencias como la física, química o de los materiales para realizar algoritmos eficientes para realizar cálculos de gran escala que se produce gracias a cierta redundancia de datos (considerando que para el caso de dos dimensiones $f(i,j) = -f(j,i)$).

D. Man et al. [11] propusieron mejorar la eficiencia de la GPU desde el punto del vista de la velocidad de acceso a memoria, proponiendo un algoritmo basado en el mapa de distancias euclidiano donde logran una mejora de aceleración en un factor de 52 veces por sobre la implementación de un algoritmo secuencial. Otro trabajo mostrado por Z. Ying et al. [12] buscó acelerar el programa DNADIST utilizando openCL, con el fin de explotar la gran capacidad de computo de una GPU. En este trabajo se logró una aceleración de 12 veces.

Es importante mencionar que cada uno de estos trabajos manifiestan un interés por encontrar formas eficientes de trabajar en GPU, tanto en tiempo y espacio. A pesar de las mejoras que se consiguen en rendimiento en cuanto a cálculos de gran escala, nuestra hipótesis es que aún es posible seguir mejorando ya sea optimizando ciertos algoritmos o encontrando nuevas formas de explotar los núcleos de la GPU.

IV. Solución propuesta

Esta sección se enfoca en explicar el desarrollo de un algoritmo usando el paradigma de *dynamic parallelism* el cual de ahora en adelante se llamará *método DP*. Para el desarrollo de este algoritmo se utilizó NVidia CUDA [13].

A continuación se explica el método DP, con el estudio de sus respectivos parámetros.

A. Método DP

El algoritmo propuesto (ver VII-B) se basa en la alternativa de mapeo *3-simplices* [9], considerando la dimensión del problema o cubo de tamaño de arista n. Se aplicó *dynamic parallelism* es para formar un tetraedro constituido por cubos, el primer *kernel* de recursión forma un cubo de tamaño $n/2$ (cubo de color rojo en Figura 3), del cual se desprenden tres *streams*[3], un *stream* hacia el eje *x*, otro hacia el eje y y el último hacia el eje z. De cada una de estas ramas recursivamente se formarán cubos de la mitad de su tamaño anterior, hasta completar en la condición $x + y > z$.

Dado a las características de la geometría de los datos y el modelo de computo que posee la GPU, se debieron elegir ciertos parámetros para el correcto funcionamiento del algoritmo. Debido a un problema de la geometría de un tetraedro, hay cubos que sobresalen del plano diagonal. Para contrarestar esto, se llenan los espacios vacíos que quedan en la diagonal lanzando un último *kernel* dedicado a llenar estos espacios y a eliminar los *threads* sobrantes por el plano diagonal.

[3]En CUDA las ramas de recursión se asocian a un *stream* distinto, esto permite lanzar varios kernel de recursión paralelamente, a través de *pipeline* de forma concurrente en la GPU.

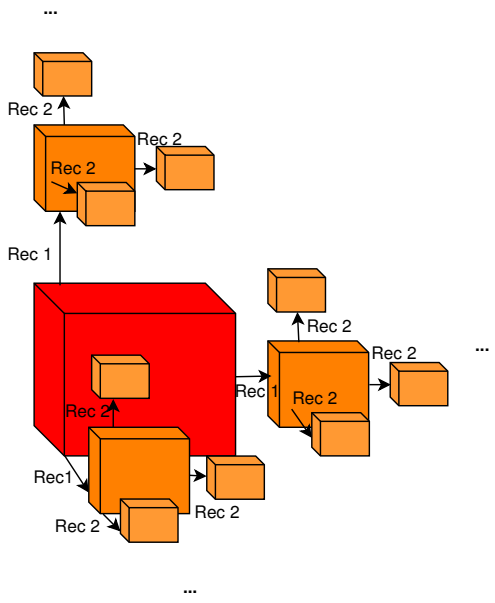


Fig. 3: Representación gráfica de métodos DP

Tomando como referencia la Figura 3 y considerando un problema de tamaño n, se puede modelar una ecuación que representa la recursividad de este método. Sea $V(D_n)$ el volumen del dominio de tamaño n o tetraedro y $V(C_n)$ el volumen del cubo de tamaño n, entonces se puede considerar que el volumen del tetraedro se puede modelar en la ecuación 1.

$$V(D_n) = V(C_{n/2}) + 3V(D_{n/2}) \tag{1}$$

$$V(D_{n_c}) = V(C_{n_c}) \tag{2}$$

Si se analiza la ecuación 1, vemos que el primer termino corresponde al volumen del cubo inicial de tamaño $n/2$ para luego sumar el volumen de tres de los cubos pequeños que se forman de tamaño $(n/2)/2$ (como también se puede ver en Figura 3) con condición de corte en la ecuación 2, donde n_c corresponde a un n de corte ideal, para no continuar ejecutando recursiones. La elección de este parámetro, se explica a continuación.

B. Elección de n_c

En teoría, el número de recursiones es $log_2 n$, sin embargo, a partir de cierto valor $n < n_c$ los sub-problemas son muy pequeños y no alcanzan a generar suficiente trabajo para la GPU. Para ello debe existir una condición para detener la ejecución de las recursiones. En teoría la ejecución de las recursiones debería detenerse cuando el tamaño del cubo sea igual o menor a uno. Al detener la recursión tempranamente, quedan espacios en el tetraedro que aún no fueron accedidos por algún *thread*. Este fenómeno se convierte en un problema que debe ser solucionado. La solución propuesta consiste en que al finalizar la última recursión se generen cubos de igual tamaño a los de la última recursión y se ubiquen a los lados de los cubos de la última recursión. Se debe encontrar un n de corte ideal (denominado de ahora en adelante n_c como se ve en ecuación 2) que alcance a cubrir la totalidad del problema y que afecte en lo más mínimo el rendimiento de la GPU. Es decir, una vez que la recursión llega a $n = n_c$ se invoca otro *kernel* que se encargue de cubrir la diagonal. Esto sucede en paralelo en todas la ramas de la recursión.

Para la elección de n_c se consideró hacer pruebas en dos máquinas (ver tablas I y II), obteniendo los resultados en la Figura 4. En este gráfico se puede notar que el eje x corresponde tamaño de problema n. Para cada n, se midió el tiempo dado el parámetro de n_c considerando el tiempo por cada tamaño de problema.

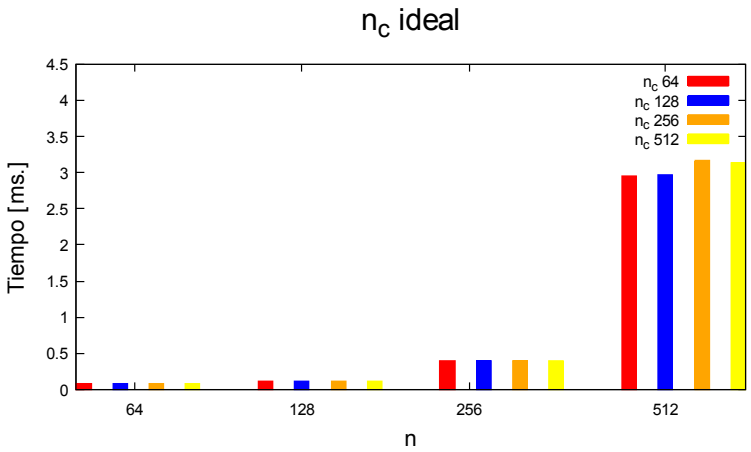


Fig. 4: Tiempo de ejecución para distintas combinaciones de n y n_c. La mayor reducción de tiempo se obtiene en $n_c = 128$ para todo n.

TABLE I: Máquina de prueba uno para determinar n_c.

Dispositivo	Modelo
GPU	GTX 1050 Ti Pascal, GP107, 640 cores, 2GB
CPU	AMD FX-8350 8-core
RAM	8GB RAM

TABLE II: Máquina de prueba dos para determinar n_c.

Dispositivo	Modelo
GPU	NVIDIA Tesla K40c,2880 cores, 12 GB
CPU	Procesador Intel Xeon E5-2640
RAM	128 GB

Con los resultados arrojados por el gráfico, se decidió tomar en consideración las últimas potencias de dos (64, 128, 256 y 512) que son las más altas y donde más se perciben los espacios en blanco. Se seleccionó 128 como n_c, a pesar de que 64 tiene valores similares con un margen muy estrecho de diferencia con 128, se eligió por ser un n alto que asegura el llenado de los espacios en blanco y que no afecta con creces el rendimiento del método.

C. Elección de BlockSize ideal

Como se mencionó anteriormente en la sección II-A, en CUDA, los *threads* de ejecución se organizan en una jerarquización espacial, donde la unidad básica es el *thread*. Estos *threads* se organizan en *blocks* de tamaño $BlockSize \times BlockSize \times BlockSize$. Los *blocks* se organizan en un *Grid* o grilla.

El tamaño de *block* ideal es aquel que maximiza grado de paralelismo por *kernel* ejecutado, para esto se debe entender que CUDA organiza los *threads* como se describió anteriormente (ver Figura 1). Para la elección del *BlockSize* ideal se tomó en consideración el n_c cuyo resultado fue 128. Se hicieron mediciones donde se tomaron los *BlockSize* de tamaño dos, cuatro y ocho *threads*, que son los tamaños admitidos por CUDA.

Las mediciones realizadas dieron como resultado un *BlockSize* ideal de ocho *threads* como se puede ver en el gráfico de la Figura 5. Esto es comprensible, dado a la arquitectura de una GPU mientras más *threads* se ejecutan por *block*, más *threads* se paralelizan, ya que los *streaming multiprocessor* de la GPU ejecutan un *block* en un instante dado y para que sea saturado estos deben ser del orden de 256, 512 o 1024 *threads* por *block*. Este comportamiento en el tamaño de *BlockSize* aplica para todos los casos ya que se comprobó que en las cuatro máquinas donde se realizó este estudio (ver tablas III, IV, V, VI), el ideal es un *BlockSize* de ocho. Esto implica que cada *block* ejecutándose en GPU, ejecuta $8 \times 8 \times 8 = 512$ *threads* de forma paralela.

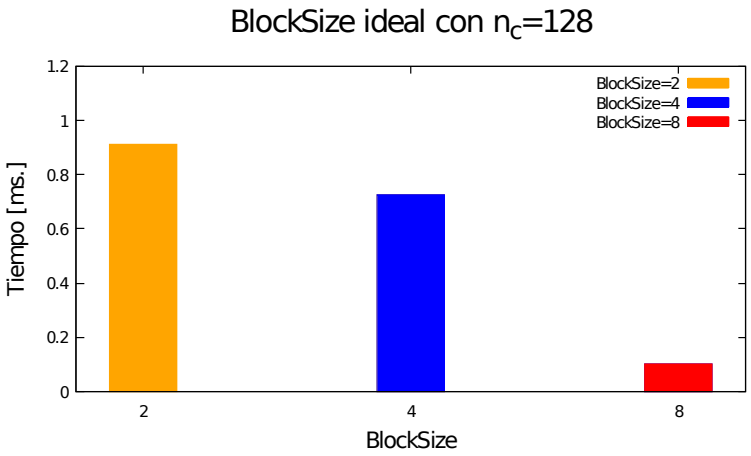


Fig. 5: Tiempo de ejecución en función del *BlockSize*, el valor ideal es ocho.

D. Implementación

En esta subsección se explica la implementación del algoritmo que se encuentra en la sección Anexos VII-B correspondiente al método DP.

El *kernel* que se programó en CUDA recibe como parámetro de entrada una matriz de tamaño $n \times n \times n$, la coordenada de inicio del primer cubo en $\mathcal{R}^3$, el lado n del tetraedro, n_c y el *BlockSize*. Los *threads* son mapeados a una región cúbica dentro del tetraedro. En este punto se crean dos identificadores de *threads*, uno para saber la coordenada absoluta en los datos y la segunda para saber el identificador del thread relativo a su *kernel*. En este momento se puede escribir el primer cubo de tamaño de arista $n/2$ mapeado al dominio de datos. Luego pueden pasar dos situaciones prueba: (1) se termina la recursión porque el tamaño del sub-problema es muy pequeño y se procede a llenar la diagonal pendiente con un último *kernel* especial que no es recursivo o (2) se crean tres ramas de recursión (usando *streams* distintos) y para ejecutar sub-problemas de $n/2 \times n/2 \times n/2$. Se definió que cada *stream* sea *Non-Blocking* ya que esto permite generar concurrencia en los kernels a ejecutar. Cabe destacar que los *streams* son llamados por un solo *thread* del *kernel* el que tiene identificador 0, ya que de lo contrario generarían una cantidad enorme e innecesaria de *kernels* recursivos.

V. Resultados de rendimiento

En esta sección se describen los resultados obtenidos en las distintas pruebas de rendimiento realizadas. Para obtener los resultados se consideraron cuatro máquinas (ver tablas III, IV, V, VI), con distintas GPUs y arquitecturas de estas mismas. Para estudiar el método propuesto en este trabajo, se compara con el método fuerza bruta (*ver VII-C*) basado en *bounding box* [9], el cual consiste en mapear los *threads* de GPU de forma directa. Es decir, se recorre cada uno de los puntos del cubo, descartando aquellos que cumplen con la condición de $(x + y > z)$. No se consideró el tiempo de traspaso de datos entre *device* y *hosts*, ya que pasa a ser despreciable cuando la GPU reutiliza varias veces los datos alojados en la memoria de GPU. Es decir, que el traspaso entre *host* y *device* sucede una sola vez, mientras que el cálculo o computo en los núcleos de la GPU sucede muchas veces.

TABLE III: Especificaciones de hardware de la máquina 1.

Dispositivo	Modelo
GPU	Titan-X Pascal, GP102, 3584 cores, 12GB
CPU	Intel i7-6950X 10-core Broadwell
RAM	128GB DDR4 2400MHz

TABLE IV: Especificaciones de hardware de la máquina 2.

Dispositivo	Modelo
GPU	GTX 1050 Ti Pascal, GP107, 640 cores, 2GB
CPU	AMD FX-8350 8-core
RAM	8GB RAM

TABLE V: Especificaciones de hardware de la máquina 3.

Dispositivo	Modelo
GPU	NVIDIA Tesla K40c,2880 cores, 12 GB
CPU	Procesador Intel Xeon E5-2640
RAM	128 GB

TABLE VI: Especificaciones de hardware de la máquina 4.

Dispositivo	Modelo
GPU	Tesla V100-SXM2, 5120 cores, 16 GB
CPU	Intel(R) Xeon(R) CPU E5-2686 v4 @ 2.30GHz
RAM	128GB

Se utilizan los parámetros anteriormente encontrados en cada una de las máquinas por igual. Cada *kernel* de la recursión escribe un valor constante en su región correspondiente en el tetraedro. Cuando todos los *kernels* recursivos terminan, el resultado es un tetraedro escrito completamente. Para el caso de la fuerza bruta se escribe la misma constante utilizando el *bounding box*. Independientemente del método escogido el tiempo de ejecución de una instancia se obtiene promediando 180 ejecuciones de kernel. Como métrica final se saca el tiempo promedio de 10 instancias lo que equivale a un promedio de promedios. Este proceso de medición entrega tiempos estables y con bajo error estándar. Este *benchmark* se realiza por cada n. El proceso se iteró haciendo variar el tamaño de *n* del problema comenzando desde $n = 8$ y terminando en $n = 512$ con intervalos de 8 unidades.

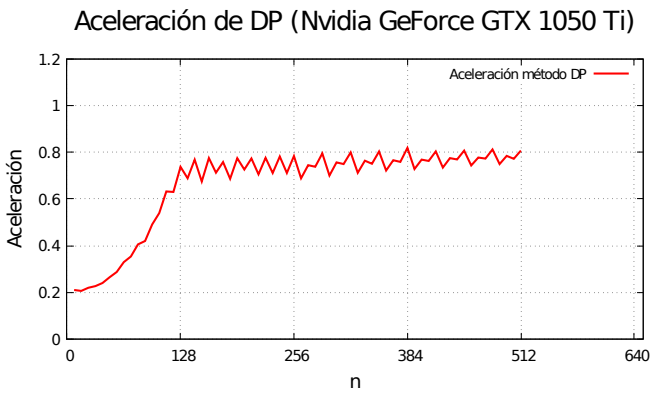
Como métrica de rendimiento se utiliza el tiempo de ejecución arrojado en cada una de las iteraciones para posteriormente graficar la aceleración (A_{DP}) [10], cuyo valor se define como:

$$A_{PD} = \frac{T_{FB}}{T_{DP}}, \tag{3}$$

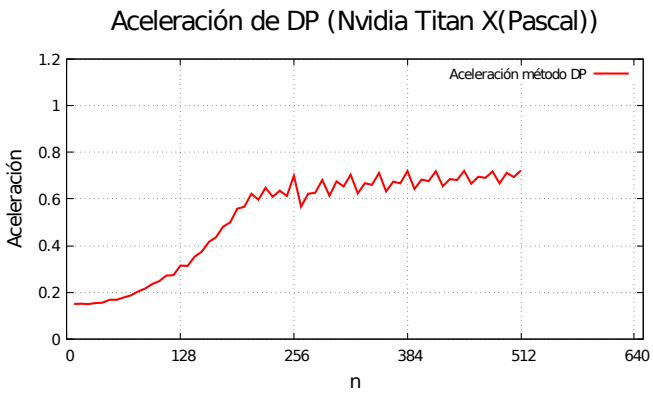
Que corresponde a la razón entre el tiempo obtenido del *Método DP* (T_{DP}) y el método *Fuerza bruta* (T_{FB}).
En la Figura 6a, donde se utiliza la máquina descrita en la Tabla IV, se observa que el método *fuerza bruta*, considerando que es una máquina de gama media, es mejor que el método DP logrando una aceleración aproximada de entre 0.7 y 0.8. Volviéndose relativamente constante a partir de $n = 128$. En la Figura 6b, donde se utiliza la máquina descrita en III, se puede observar una situación similar a la mencionada anteriormente. La aceleración comienza a permanecer constante en $n = 220$ aproximadamente y los valores de este oscilan entre 0.6 y 0.7. En la Figura 6c, donde se utiliza la máquina descrita en la Tabla V, no se presenta una situación muy lejana a las anteriormente mencionadas, los valores de la aceleración comienzan a volverse constantes a partir de $n = 140$ aproximadamente y sus valores oscilan entre 0.7 y 0.9.

En la Figura 6d, donde se utiliza la máquina descrita en la Tabla VI, es donde se presenta la mayor diferencia y variaciones de aceleración frente a las otras máquinas estudiadas. A partir de $n = 128$ se observa un notorio incremento en la aceleración. Hasta $n > 256$ este valor oscila entre $1\times$ y $2\times$, sin embargo a partir de 256 este tiene una abrupta crecida oscilando entre $2.4\times$ y $3.1\times$ aproximadamente.

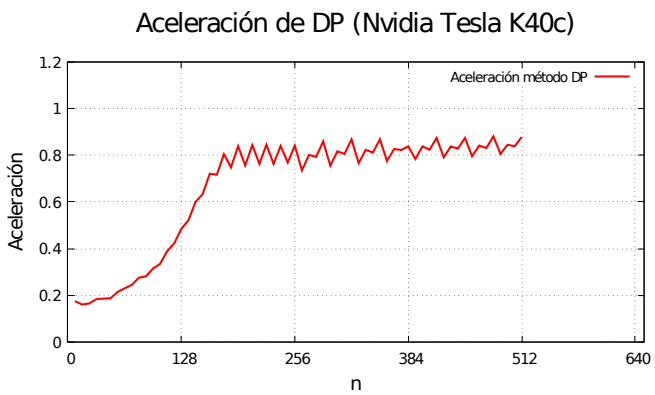
Los resultados de rendimiento arrojados por cada máquina permiten conocer y analizar el comportamiento del método PD frente a fuerza bruta. Por otro lado, es posible inferir que existe una influencia importante de la arquitectura de la GPU en el comportamiento del método.



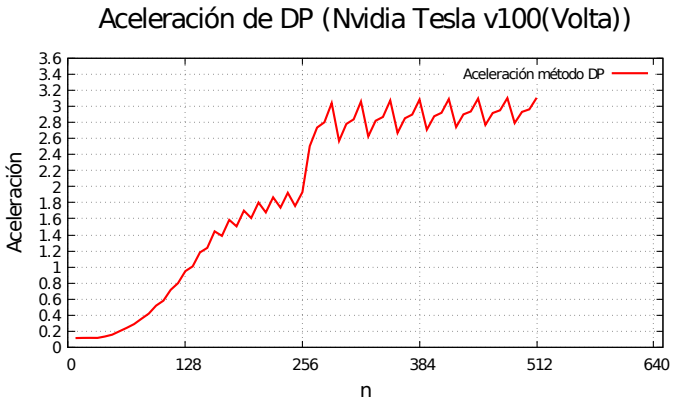
(a) Gráfico de aceleración en NVidia GeForce GTX 1050 Ti



(b) Gráfico de aceleración en Titan X(Pascal)



(c) Gráfico de aceleración en Tesla K40c



(d) Gráfico de aceleración en Tesla V100(Volta)

Fig. 6: Gráficos de aceleración de las cuatro máquinas estudiadas.

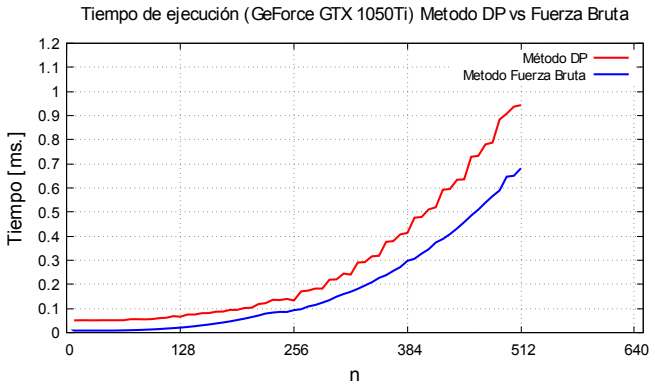
VI. Discusión

De la sección anterior (V) se pudo observar que en tres de las máquinas estudiadas no hay grandes diferencias de comportamiento entre GPUs con respecto a la utilización de *dynamic parallelism*, sigue sobresaliendo el método de mapeo de *threads* fuerza bruta. Esto se debe a que quizás hay una menor cantidad de cálculos que cada *thread* mapeado debe hacer con respecto a la cantidad de *threads* que se deben descartar. Donde se pudo notar un comportamiento similar pero con mayor variación de tiempo, fue en la GPU Titan X con arquitectura Pascal, donde el valor de la aceleración osciló entre $0.6\times$ y $0.7\times$. Cabe destacar que esta oscilación se podría deber a que hay valores de n donde se deben ajustar los cálculos dado a una inexactitud en la división que no coincidían con el algoritmo planteado. Es decir, dentro del algoritmo hay divisiones que no son exactas por el hecho de que su resultado es un número impar o no entero, para ello se hicieron cálculos de redondeos para ciertos n que presentan ese problema. Esto, en alguna medida debió afectar el rendimiento del algoritmo, por lo tanto quizás sea posible optimizar el código para encontrar una solución que minimice este cálculo.

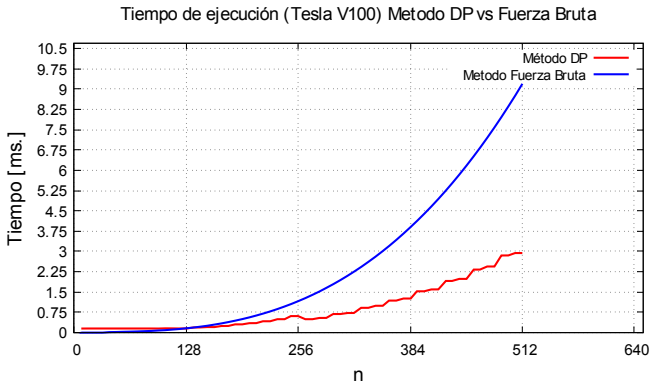
El principal hallazgo, es la mejora de rendimiento y eficiencia utilizando *dynamic parallelism* en la máquina con GPU Tesla V100. Mejora que puede ser hasta $\sim 3\times$. Se puede inferir que este crecimiento puede deberse a que dentro de las cuatro GPUs seleccionadas para este estudio, es la más reciente y su arquitectura está hecha para potenciar la computación de alto rendimiento. Algo que se puede observar en este gráfico, es que se produce un abrupto crecimiento en $n = 256$ aproximadamente. Hay muchos factores que podrían desencadenar este comportamiento, por ejemplo, podría ser atribuible a una elección de parámetros más exactos para cada máquina seleccionada, por ejemplo, quizás para esta máquina el valor de n_c debería ser 256 y no 128. Este comportamiento merece ser estudiado más a fondo a futuro.

Otro comportamiento observado en este estudio, es la diferencia que existe al ejecutar el método fuerza bruta en la máquina con GPU tesla V100 (Volta), en comparación con el resto de las máquinas analizadas. Como se puede observar en la Figura 7a donde se comparan los tiempos de ejecución, el método fuerza bruta gana con respecto al método DP, donde el tiempo de ejecución para $n = 512$ no alcanza a ser más 0.6 ms. En contraste con la GPU Tesla V100, como podemos ver en la Figura 7b pierde con creces, de hasta casi $10\times$. Ocurre una situación muy similar si comparamos el gráfico de la GPU Tesla V100 con

las otras GPUs analizadas, por lo que resultaría interesante descubrir por qué ocurrió esto solo en esta GPU y no en las otras estudiadas.



(a) Gráfico de tiempo vs. n en NVidia GeForce GTX 1050 Ti



(b) Gráfico de tiempo vs. n en Tesla V100

Fig. 7: Gráficos de comparación entre máquina Tesla V100 con GTX 1050Ti.

VII. Conclusión

Retomando la pregunta de investigación planteada en la introducción de este articulo ¿Pueden las últimas tecnologías de GPU, tales como *Dynamic Parallelism*, reducir la cantidad de *threads* para dominios de tetraedro en tres dimensiones y entregar una mejora en rendimiento como consecuencia? Respondiendo a esta pregunta, en este estudio, solo se lograron resultados positivos en una GPU con arquitectura Volta (Tesla V100), logrando una mejora de $\sim 3\times$, por lo que sería interesante descubrir por qué razón solo se logró una mejora en esta GPU. Además, del comportamiento curioso que tuvo esta arquitectura con el método fuerza bruta fue mucho más lento que en otras máquinas. Considerando los resultados se puede deducir que el funcionamiento de *dynamic parallelism* varía de acuerdo a la arquitectura de la GPU. Sin embargo también puede que esto sea debido a la variación de parámetros utilizados como n_c o *BlockSize*. Los resultados de este estudio no están muy lejos a los que propone y estima C. A. Navarro et al. [9], quien afirma que se podría lograr una posible mejora de rendimiento variando entre $2\times$ y $6\times$.

En este estudio se utilizaron valores iguales para cada equipo asumiendo un comportamiento similar en todas las máquinas donde se realizó este estudio. Por lo que cabe la posibilidad de que para cada arquitectura y GPU se deba realizar un estudio previo para determinar la existencia de los valores ideales de *BlockSize* y de n_c específicos. Si esto es así, podrían existir cambios en el rendimiento en lo que se refiere a tiempo de ejecución del algoritmo para la resolución de un problema de dominio tetraédrico.

Se demostró empíricamente que *dynamic parallelism* es una técnica que no puede ser descartada totalmente, se puede aprovechar bastante si se encuentra la arquitectura adecuada para su uso. Se pudo notar la gran ventaja que tiene esta técnica en arquitectura Volta de NVidia. La arquitectura Volta es una de las más recientes que desarrolló Nvidia, lo que demuestra que si en esta máquina se encontró que es conveniente utilizar *dynamic parallelism*, aún queda mucho por descubrir y mejorar en esta tecnología.

References

[1] P. Koanantakool and K. Yelick, "A computation- and communication-optimal parallel direct 3-body algorithm," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 363–374. [Online]. Available: https://doi.org/10.1109/SC.2014.35

[2] C. A. Navarro, M. Vernier, N. Hitschfeld, and B. Bustos, "Competitiveness of a non-linear block-space gpu thread map for simplex domains," *IEEE Transactions on Parallel and Distributed Systems*, pp. 1–1, 2018.

[3] C. A. Navarro and N. Hitschfeld, "Improving the GPU space of computation under triangular domain problems," *CoRR*, vol. abs/1308.1419, 2013. [Online]. Available: http://arxiv.org/abs/1308.1419

[4] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," *Queue*, vol. 6, no. 2, pp. 40–53, Mar. 2008. [Online]. Available: http://doi.acm.org/10.1145/1365490.1365500

[5] S. Jones, "Introduction to dynamic parallelism," in *GPU Technology Conference Presentation S*, vol. 338, 2012, p. 2012.

[6] D. A. Huckaby and L. Blum, "Effect of triplet correlations on the adsorption of a dense fluid onto a crystalline surface," *The Journal of Chemical Physics*, vol. 97, no. 8, pp. 5773–5776, 1992. [Online]. Available: https://doi.org/10.1063/1.463761

[7] C. A. Navarro and N. Hitschfeld, "Gpu maps for the space of computation in triangular domain problems," in *2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC,CSS,ICESS)*, Aug 2014, pp. 375–382.

[8] J. Hyuk Jung and D. O'leary, "Exploiting structure of symmetric or triangular matrices on a gpu," *Technical report, University of Maryland*, 01 2008.

[9] C. A. Navarro, B. Bustos, and N. Hitschfeld, "Possibilities of recursive GPU mapping for discrete orthogonal simplices," *CoRR*, vol. abs/1610.07394, 2016. [Online]. Available: http://arxiv.org/abs/1610.07394

[10] C. A. Navarro, N. Hitschfeld-Kahler, and L. Mateu, "A survey on parallel computing and its applications in data-parallel problems using gpu architectures," *Communications in Computational Physics*, vol. 15, no. 2, p. 285329, 2014.

[11] D. Man, K. Uda, Y. Ito, and K. Nakano, "A gpu implementation of computing euclidean distance map with efficient memory access," in *2011 Second International Conference on Networking and Computing*, Nov 2011, pp. 68–76.

[12] Z. Ying, X. Lin, S. C. See, and M. Li, "Gpu-accelerated dna distance matrix computation," in *2011 Sixth Annual Chinagrid Conference*, Aug 2011, pp. 42–47.

[13] Nvidia-Corporation, "Nvidia cuda c programming guide," 2016.

ANEXOS

A. Main

Algorithm 1 Código principal (Main)

function MAIN(Arg)
- $n \leftarrow Arg(1)$
- $met \leftarrow Arg(2)$
- $rep \leftarrow Arg(3)$
- $n_c \leftarrow Arg(4)$
- $BlockSize \leftarrow Arg(5)$
- A
- $xi \leftarrow 0$
- $yi \leftarrow 0$
- $zi \leftarrow n/2$
- $malloc..$ ▷ Reserva de espacio en memoria
- $Bloque \leftarrow dim3(BlockSize, BlockSize, BlockSize)$
- $NB \leftarrow n/(2 * BlockSize)$
- $Grid \leftarrow dim3(NB, NB, NB)$
- $gridBruto \leftarrow dim3((N + BSize - 1)/BSize, (N + BSize - 1)/BSize, (N + BSize - 1)/BSize)$
- CudaMemcpy() ▷ Traspaso de datos a memoria GPU
- **if** $met = 1$ **then**
  - $T1 \leftarrow obtenerTiempo()$
  - **for** $1 : rep$ **do**
    - METODOPD$<<< Grid, Bloque >>> (A, n, n/2, Xi, Yi, Zi + Nm, n_c, BlockSize)$
  - **end for**
  - $T2 \leftarrow obtenerTiempo()$
- **end if**
- **if** $met = 2$ **then**
  - $T1 \leftarrow obtenerTiempo()$
  - **for** $1 : rep$ **do**
    - MetodoFuerzaBruta$<<< Grid, Bloque >>> (A, n)$
  - **end for**
  - $T2 \leftarrow obtenerTiempo()$
  - $media \leftarrow (t2 - t1)/rep$
  - CudaMemcpy() ▷ Traspaso de datos de memoria GPU a memoria CPU
- **end if**

end function

B. Método DP

Algorithm 2 *Kernel* Método DP

function METODODP(n, A, Nm, xi, yi, zi, nrec, n_c, BlockSize)
  $x \leftarrow blockIdx.x * blockDim.x + threadIdx.x + xi$
  $x0 \leftarrow blockIdx.x * blockDim.x + threadIdx.x$
  $y \leftarrow blockIdx.y * blockDim.y + threadIdx.y + yi$
  $y0 \leftarrow blockIdx.y * blockDim.y + threadIdx.y$
  $z \leftarrow blockIdx.z * blockDim.z + threadIdx.z + zi$
  $z0 \leftarrow blockIdx.z * blockDim.z + threadIdx.z$
  $tid1 \leftarrow z * N * N + y * N + x$
  $tid \leftarrow z0 * N * N + y0 * N + x0$
  if $x < n$ AND $y < n$ AND $z < n$ **then**
    $A(tid1) \leftarrow 1$
    if $x + y > z$ **then**
      $A(tid1) \leftarrow 9$
    end if
  end if
  if $Nm <= n_c$ AND $tid = 0$ **then**
    $Nbloques \leftarrow (Nm + BlockSize - 1)/BlockSize$
    $b \leftarrow dim3(BlockSize, BlockSize, BlockSize)$
    $g \leftarrow dim3(Nbloques, Nbloques, Nbloques)$
    $Stream1$
    $Stream2$
    $Stream3$ ▷ Ramas de recursión
    diagonal$<<< g, b, 0, stream1 >>> (A, n, Nm, Xi + Nm, yi, zi, nrec)$
    diagonal$<<< g, b, 0, stream2 >>> (A, n, Nm, Xi, yi + Nm, zi, nrec)$
    diagonal$<<< g, b, 0, stream3 >>> (A, n, Nm, Xi + Nm, yi, zi - Nm, nrec)$ ▷ Llama a función que rellena los espacios vacíos en diagonal
    $break$;
  end if
  if Nm =!0 AND $tid = 0$ **then**
    $Nbloques \leftarrow (Nm + BlockSize - 1)/BlockSize$
    $b \leftarrow dim3(BlockSize, BlockSize, BlockSize)$
    $g \leftarrow dim3(Nbloques, Nbloques, Nbloques)$
    $Stream1$
    $Stream2$
    $Stream3$ ▷ Ramas de recursión
    $Nm \leftarrow Nm/2$
    METODOPD$<<< g, b, 0, stream1 >>> (A, n, Nm, Xi + Nm, yi, zi + Nm, nrec, n_c, BlockSize)$
    METODOPD$<<< g, b, 0, stream2 >>> (A, n, Nm, Xi, yi + Nm, zi + Nm, nrec, n_c, BlockSize)$
    METODOPD$<<< g, b, 0, stream3 >>> (A, n, Nm, xi, yi, zi - Nm, nrec, n_c, BlockSize)$ ▷ Llama a Kernel nuevamente (recursión)
    $break$;
  end if
end function

C. Método Fuerza Bruta

Algorithm 3 *Kernel* Método Fuerza Bruta

```
1: function METODOFUERZABRUTA(n,A)
2:     x ← blockIdx.x * blockDim.x + threadIdx.x + xi
3:     y ← blockIdx.y * blockDim.y + threadIdx.y + yi
4:     z ← blockIdx.z * blockDim.z + threadIdx.z + zi
5:     ind ← z * N * N + y * N + x
6:     if x < N AND y < N AND z < N then
7:         if x + y <= z then
8:             A(ind) ← 1
9:         end if
10:    end if
11: end function
```