

Acelerando ecuaciones de mapeo de hilos de GPU para fractales 2D empotrados utilizando Tensor Cores

Felipe A. Quezada¹

Universidad Austral de Chile
felipe.quezada01@outlook.com

Abstract. Este trabajo propone el uso de Tensor Cores para acelerar el mapeo de hilos de computo GPU en dominios fractales 2D empotrados. Se proponen 3 variantes $\lambda_{tc}(\omega)$ que aceleran el rendimiento de un mapeo del estado del arte denominado $\lambda(\omega)$, el cual presenta potenciales oportunidades para ser mejorado vía el uso de tensor cores. Cada variante intenta aprovechar de distinta forma la operación matricial que realizan los tensor cores. Los resultados experimentales indican que $\lambda_{tc}(\omega)$ puede alcanzar hasta un $\sim 40\%$ extra en rendimiento con respecto a $\lambda(\omega)$. Además, estos resultados indican que el uso de tensor cores en cargas ajenas a álgebra lineal o machine learning puede ser beneficioso, y que dependen de que tan bien se puede aprovechar la operación matricial de un tensor core.

Keywords: GPU · Tensor Cores · Mapeo de hilos · Fractal. · Sierpinski

1 Introducción

Vivimos en un universo fractal, insertos entre conglomeraciones de materia que presentan pequeñas copias del todo en niveles sucesivamente mas pequeños, desde las agrupaciones de estrellas para formar galaxias, galaxias para formar clusters y clusters para formar super-clusters, hasta el crecimiento de arboles y plantas [19][20], la formación de terrenos [10][21], patrones de dinámica molecular [25], cristalización de copos de nieve [4], generación de capilares [3], entre muchos otros ejemplos que presentan una parte del todo en copias completas. Un fractal puede ser definido como un objeto geométrico cuya estructura fundamental se repite a diferentes escalas o niveles, fenómeno que se conoce como auto-similitud [9]. Los fractales son especialmente útiles cuando se quiere modelar aquellas estructuras que cumplen con la propiedad de auto-similitud y cuando resulta difícil describirlas en términos de geometría euclidiana. Las aplicaciones de computación relacionadas a estos temas pueden utilizar una versión empotrada del fractal en un dominio euclidiano discreto, el cual actúa como un "Bounding-box"¹ de los datos en memoria. Utilizar una versión empotrada del fractal permite realizar cálculos eficientes en términos de patrones de acceso a memoria,

¹ Se define a "Bounding-box" como una caja en 1, 2 o 3 dimensiones, lo suficientemente grande para cubrir a todos los datos.

ya que los cálculos en paralelo pueden realizar accesos alineados a memoria. La figura 1 muestra el ejemplo de el fractal **H** empotrado en un espacio de $n \times n$, con n el numero de elementos de un lado. Es importante aclarar que n también representa el tamaño de un lado del fractal.

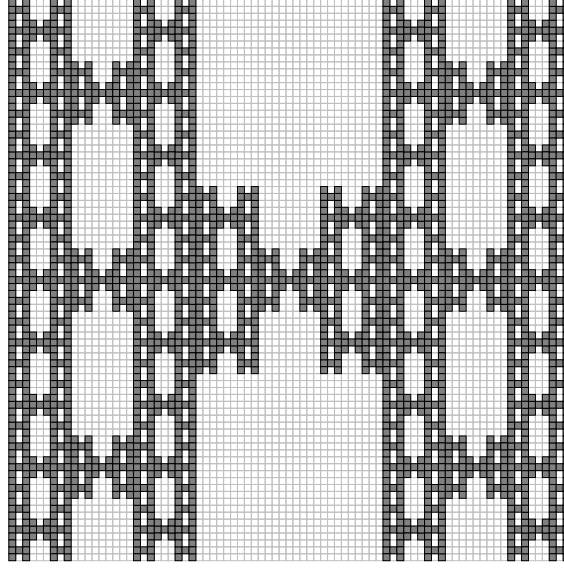


Fig. 1. El fractal **H** empotrado en un dominio euclidiano discreto de tamaño $n \times n$. En este caso $n = 81$

Eventualmente, cuando el fractal es lo suficientemente grande, utilizar un empotramiento en un espacio euclidiano discreto genera 2 problemas que vale la pena mencionar. El primero es que se podría requerir de una cantidad excesiva de memoria para almacenar el fractal completo dentro del espacio euclidiano, la cual crece en función de $\mathcal{O}(n^2)$. El segundo es que utilizar una técnica de computo secuencial podría llegar a ser demasiado lenta para efectos prácticos, especialmente si hay restricciones que requieran obtener resultados en tiempo real. Para solucionar el segundo problema, el computo en GPU se vuelve una herramienta atractiva si se quiere acelerar estas tareas [16]. Un concepto importante a la hora de cuantificar la aceleración de una tarea es el *Speedup* (S) en rendimiento, el cual es un numero que representa la razón de mejora en rendimiento de un tiempo de ejecucion con respecto a otro.

La unidad de procesamiento de gráficos (GPU por sus siglas en ingles) se ha transformado en una herramienta que habilita la practica de muchas aplicaciones que presentan cargas con trabajo paralelizable y demandan una alta capacidad de computo, por lo que su alto tiempo de procesamiento las haría inviables.[16]. Las 2 motivaciones principales para utilizar una GPU son (1) un alto ancho de

banda y TFLOPS², y (2) la ventaja en eficiencia energética con respecto a CPU [8][23]. Sin embargo, es importante aclarar que para hacer uso de estas ventajas, se requiere de un diseño e implementación algorítmica dedicada, ya que las GPUs son mas limitadas que la CPU en términos de lógica de control y patrones de acceso a memoria.

Para realizar computo de propósito general utilizando una GPU (lo que se conoce como GPGPU) comúnmente se utiliza una API. Las mas usadas son CUDA de NVIDIA (exclusivo para GPUs NVIDIA) y OpenCL de Khronos Group. En este trabajo se utilizara la terminología propuesta por CUDA. Las GPUs utilizan un modelo de ejecución conocido como SIMD (single instruction, multiple data) en el que el procesador ejecuta las mismas instrucciones en diferentes elementos del espacio de datos \mathcal{D} . Tradicionalmente las CPU tienen entre 1 y 64 núcleos, dependiendo de su uso, mientras que el chip de una GPU normalmente tiene miles de núcleos, llamados CUDA cores. Por ejemplo, la micro-arquitectura Volta de GPUs NVIDIA tiene 5120 núcleos en una tarjeta. Los CUDA cores son especialmente útiles a la hora de realizar operaciones aritméticas de punto flotante y lecturas/escrituras en paralelo, sin embargo, son mas limitados, pues no poseen una robusta lógica de control y patrones de acceso a memoria como los presentes en una CPU.

El programa que contiene la lista de instrucciones que se ejecutaran en paralelo en una GPU se conoce como *kernel*. Antes de poder ejecutar un *kernel* y dependiendo de las características del problema, es necesario definir el tamaño del espacio paralelo \mathbf{P} que contiene los hilos de ejecución a utilizar. El espacio paralelo contiene los distintos hilos de ejecución y tiene una jerarquía de hilos dividida en 3 niveles: grid, bloque e hilo. Un hilo representa la lista de instrucciones que ejecutara un núcleo de la GPU, *i.e.*, es una unidad que realizara trabajo concurrente sobre los datos. Estos hilos están agrupados en bloques que se ejecutan concurrentemente en la GPU, los cuales pueden ser 1, 2 o 3-dimensionales. Por diseño de la arquitectura, actualmente (2020) un bloque no pueden tener mas de 1024 hilos en total. El bloque es el nivel mas importante en la jerarquía de hilos de una GPU, pues permite que sus hilos intercambien datos utilizando una memoria cache extremadamente rápida conocida como memoria compartida (o shared memory, en ingles). Finalmente el grid es un grupo de bloques, este puede ser indexado como una caja de 1, 2 o 3 dimensiones. La figura 2 presenta un diagrama con un bloque y grid en 2 dimensiones.

Normalmente cuando uno se enfrenta a la paralelización de un programa, se utilizan bloques de tamaño máximo para aprovechar hacer mejor uso de todos los recursos de una GPU, y en base a este tamaño, definir el largo del grid para cubrir por completo el espacio de datos con el espacio paralelo. Para lograr que los hilos puedan realizar el mismo trabajo sobre diferentes elementos del espacio de datos, estos son inicialmente otorgados con una coordenada única dentro del bloque, mientras que el bloque, a su vez, recibe una coordenada análoga dentro del grid. Existe además, un nivel jerárquico implícito en la arquitectura que de-

² TFLOS: Tera operaciones de punto flotante por segundo; es un indicador simple de la velocidad de computo de un procesador.

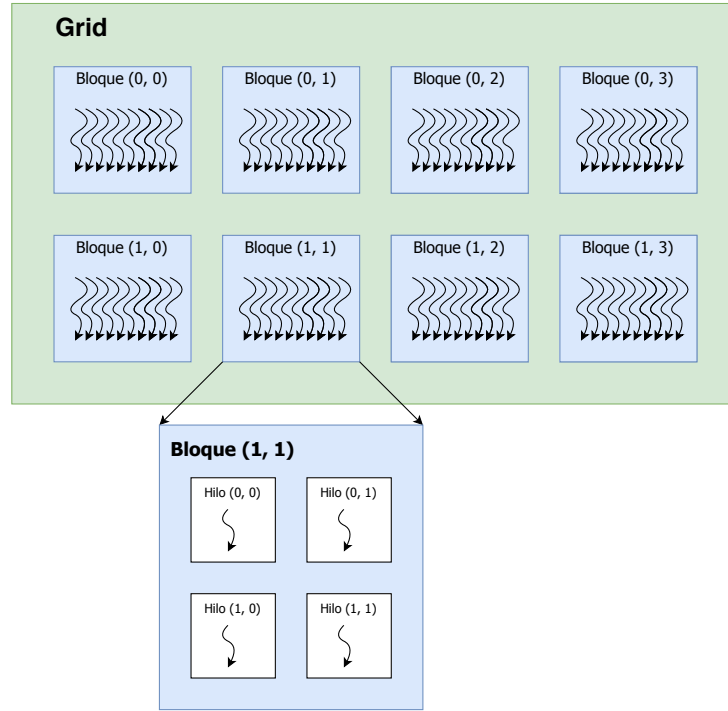


Fig. 2. Arquitectura de hilos en GPU. La zona verde representa al grid. La zona azul representa un bloque. La zona blanca un hilo. En este caso se presenta un grid de tamaño 2×4 y un bloque de 2×2 . Es importante notar que cada estructura presenta una coordenada única dentro de su estructura padre. El espacio paralelo es de tamaño 16×8 .

pendiendo del problema puede ser transparente al programador, esta se conoce como *warp* y es un grupo de 32 hilos consecutivos (según su coordenada) que se ejecutan simultáneamente en paralelo y están siempre en sincronía. Realizar una programación considerando este nivel de jerarquía puede reducir considerablemente la divergencia entre líneas de ejecución de los hilos y los accesos a memoria. También le permite al programador hacer uso de funciones exclusivas para warps, como el manejo de Tensor Cores.

Los Tensor Cores son el resultado de recientes investigaciones por parte de NVIDIA que les han llevado a desarrollar esta nueva unidad hardware de cómputo integrada en las GPU desde la micro-arquitectura Volta en adelante. Cada tensor core es capaz de realizar una operación del tipo $D = A * B + C$, donde A , B , C y D son matrices de 4×4 , utilizando solo un ciclo de reloj de GPU. Esto se traduce en un incremento significativo de TFLOPS cuando se compara la misma operación utilizando CUDA cores. Actualmente las GPUs desde el 2018 pueden contener en el orden de 640 tensor cores en conjunto a sus CUDA cores. Para hacer uso de los tensor cores, CUDA provee una interfaz al programador donde

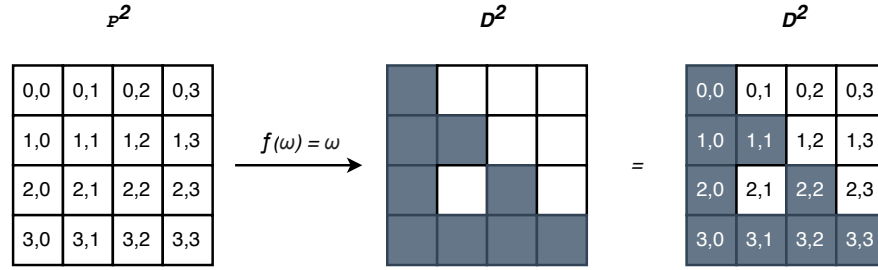


Fig. 3. Un mapeo de hilos utilizando la función identidad. A la izquierda esta el 2-ortotopo Π^2 de 4×4 generado para abarcar el problema. Al centro el espacio de datos \mathcal{D}^2 con un triángulo de Sierpinski empotrado. Los cuadros pintados representan espacios en memoria con datos útiles, mientras que los blancos representan datos basura. A la derecha se presenta la correspondencia de cada hilo con un dato en particular, generado por la función identidad.

es capaz de declarar, cargar y operar matrices de 16×16 , llamadas fragmentos³. Estos fragmentos son enviados a las rutinas de operación de tensor core, las cuales internamente lo dividen en matrices de 4×4 para realizar el calculo utilizando un warp completo, *i.e.* 32 hilos. Actualmente en 2020, no se tiene clara la correspondencia de cada warp con los campos del fragmento o como se realiza la operación matricial dentro del tensor core, pues no ha sido revelado por NVIDIA, además no se tiene garantía que una implementación tensor core que obtenga buenos resultados en la micro-arquitectura Volta, vaya a tenerlos en Turing, o vice versa, pues los detalles de la implementación no son públicos. Lo que si se sabe es que la mejora en rendimiento potencial proveída por los tensor cores va a depender de que tan bien se pueda aprovechar esta operación matricial. Estas interrogantes referentes a tensor cores introducen una motivación adicional para saber si los TFLOPS teóricos extra de la operación matricial usando tensor cores pueden ser aprovechados para aplicaciones distintas para las cuales fueron diseñados, como álgebra lineal o machine learning.

Al comienzo de la ejecución de un kernel en GPU, existe un paso en el pipeline de ejecución donde los hilos son mapeados desde su coordenada local dentro del espacio paralelo hasta su coordenada correspondiente en el espacio de los datos. Realizar un mapeo eficiente es un aspecto de las GPUs que ha sido recientemente estudiado, cuyos beneficios podrían ser minimizar el numero necesario de hilos cuando no se trabaja con dominios no triviales de datos, y en consecuencia optimizar el uso de recursos. Un mapeo puede ser definido formalmente como $f: \mathbb{Z}^k \rightarrow \mathbb{Z}^m$, que transforma cada punto k -dimensional $x = (x_1, x_2, \dots, x_k)$ en el espacio paralelo P^k en un punto m -dimensional único $f(x) = (y_1, y_2, \dots, y_m)$ el cual yace dentro del dominio de los datos \mathcal{D}^m . El espacio paralelo, que es donde residen los hilos dentro de su correspondiente jerarquía, se puede definir como

³ La API de tensor core también permite otros tamaños de fragmento similares, sin embargo en este trabajo se utilizara exclusivamente 16×16 .

un ortotopo (también llamado hipercubo) $\Pi^k \in P^k$ con $k = 1, 2, 3$ dimensiones. Esta notación es especialmente útil para entender de una forma mas general la geometría que puede tomar el espacio paralelo [17][12][15]. Particularmente este trabajo se enfoca en utilizar el 2-ortotopo.

Un enfoque común para el mapeo de hilos de GPU desde el espacio paralelo a espacio de datos, es utilizar la técnica *bounding-box*. Esta técnica consiste en generar un 2-ortotopo lo suficientemente grande para abarcar todos los datos y luego mapear los hilos utilizando la función identidad $f(\omega) = \omega$. Es importante aclarar que el 2-ortotopo generado es el espacio paralelo, no el espacio de datos al empotrar el fractal como el presente en la figura 1. A pesar de que ambos espacios tengan el mismo tamaño (utilizando el mapeo identidad), los elementos de uno son hilos y los del otro son datos. Este tipo de mapeo es particularmente eficiente y conveniente para aquellos problemas cuyo espacio de datos puedan ser definidos por un ortotopo; vectores, tablas y cajas son ejemplos de un ortotopo en 1, 2 y 3 dimensiones respectivamente. Un descriptivo ejemplo es presentado en la figura 3. Sin embargo, para un problema con un espacio de datos con forma de un fractal 2D empotrado, este enfoque deja de ser eficiente, pues existen hilos a los que les correspondería una coordenada dentro del ortotopo en el cual no hayan datos, y por lo tanto, este hilo no efectuaría trabajo efectivo, desaprovechando rendimiento potencial. Para estos casos es conveniente utilizar un ortotopo con una cantidad de hilos que mejor se ajuste a los datos disponibles, y a su vez sea asintoticamente inferior a lo que un enfoque de tipo *bounding-box* generaría. En base a este problema, Navarro et al. (2017) proponen una función de mapeo eficiente al nivel de bloques: $\lambda(\omega)$, para una familia de fractales 2D empotrados [12]. La figura 4 ilustra un ejemplo de la situación descrita.

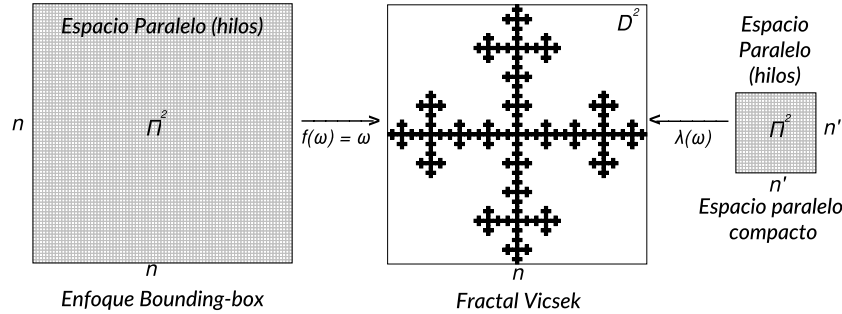


Fig. 4. Mapeo de hilos de GPU para el fractal Vicsek. El mapeo de izquierda al centro ilustra un enfoque *bounding-box* y su costo asociado en cuanto a cantidad de hilos (tamaño espacio paralelo). El mapeo de derecha a centro muestra un enfoque utilizando $\lambda(\omega)$ desde un espacio paralelo compacto, *i.e.* utilizando una menor cantidad de hilos.

Se definen dos preguntas de investigación relacionadas con la eficiencia en mapeo de hilos de GPU. La primera pregunta es *¿Es posible acelerar la ecuación*

de mapeo eficiente de bloques $\lambda(\omega)$ utilizando tensor cores?, y de ser así ¿Cual es el Speedup máximo obtenido?. Se espera también, mejorar la intuición de cual es la dirección correcta para aprovechar el rendimiento extra provenientes de la utilización de tensor cores. Resultados preliminares indican que los tensor cores pueden ser utilizados exitosamente para acelerar tareas ajenas a aquellas aplicaciones para lo cual fueron diseñados, como machine learning y algebra lineal. El resto de este documento se distribuye de la siguiente manera: la sección 2 presenta en resumen los estudios recientes mas relevantes para este trabajo. La sección 3 define la familia de fractales sobre la cual se trabajara. La sección 4 formula la ecuación de mapeo $\lambda(\omega)$. La sección 5 presenta los resultados de las pruebas y la sección 6 las conclusiones y discusión.

2 Trabajo relacionado

Jung *et al.* (2008) exploraron las posibilidades de mejorar aquellas aplicaciones cuyos dominios presentan una forma triangular utilizando un mapeo de hilos de GPU. Algunas de estas aplicaciones son descomposición LU y Cholesky. Su estrategia consiste en construir una caja rectangular para acceder y almacenar este dominio triangular. Las estructuras de datos utilizando esta técnica disminuyen su tamaño a la mitad con respecto a la matriz completa. Originalmente, el objetivo de esta estrategia era disminuir la memoria utilizada (*i.e.*, la memoria utilizada por la matriz). Sin embargo uno puede aprovechar el concepto y utilizarlo para ahorrar espacio paralelo [6]. Ries *et al.* (2009) desarrollaron un método paralelo para calcular el inverso de una matriz triangular utilizando GPU[24]. Los autores identificaron que el espacio paralelo puede ser utilizado de manera mas óptima al particionar recursivamente el grid, utilizando un enfoque *divide y vencerás*. El mapeo de bloques tiene complejidad $O(\log_2(n))$, con n siendo el tamaño de un lado de la matriz. Navarro, Hitshfeld and Bustos (2014) propusieron un mapeo a nivel de bloques de GPU para 2-simplex⁴ y 3-simplex [15][13][17] basado en la solución de una ecuación de orden m , la cual esta formulada a partir de la enumeración lineal de elementos discretos. Los autores reportan una mejora en el rendimiento para el caso de 2-simplex. Para el caso de 3-simplex, se realizo una extensión de la técnica donde se encontró que el uso del espacio paralelo puede ser hasta 6 veces mas eficiente. Sin embargo, los autores clarifican que es complejo traducir tal mejora en espacio paralelo en una mejora en rendimiento, ya que el calculo del mapeo requiere realizar una cantidad significativa de raíces cuadradas y cubicas, lo que se traduce en mayor penalización al rendimiento.

Explorar los beneficios de un mapeo eficiente del espacio paralelo de GPU para fractales 2D empotrados es un tópico de investigación relevante ya que su geometría deja de ser euclidiana como en los trabajos previamente mencionados. Entonces, encontrar una función eficiente $\lambda(\omega)$, produciría una mejora asintótica

⁴ Un k -simplex es la generalización de un triangulo en k dimensiones. Un 2-simplex corresponde a un triangulo, mientras que un 3-simplex corresponde a un tetraedro.

en el espacio paralelo y potencialmente un aumento en el rendimiento que eventualmente pueda ser aprovechado. Por otro lado, el calculo de estas funciones de mapeo puede hacer uso de herramientas diseñadas para aprovechar al máximo los recursos disponibles, como lo son los tensor cores. Esto permitiría que las funciones de mapeo dejen de apoyarse únicamente en un ahorro del espacio paralelo para obtener una mejora en rendimiento. El uso de Tensor Cores para acelerar aplicaciones que estén fuera del dominio de machine learning y álgebra lineal es un tópico relativamente nuevo, con poca investigación asociada. La cantidad de investigaciones destinada a aprovechar el rendimiento de los tensor cores se aminora a medida que uno se aleja de aquellas aplicaciones que no se basen en operaciones matriciales, como es el caso de mapeo de espacio paralelo en GPU. Sin embargo, algunos estudios recientes validan la idea de aplicar tensor cores en estas tareas e introducen motivación adicional para estudiar su impacto al ser aplicados a diversas aplicaciones. Carrasco, Vega y Navarro (2018), estudiaron el Speedup potencial que puede tener una reducción al aplicar tensor cores, en relación a la versión clásica en GPU. Los autores concluyen que la reducción basada en Tensor Cores es, en teoría, mas rápida que su contra parte. Continuando con su trabajo, Navarro et al. (2020a) comprobaron los resultados empíricamente, alcanzando un Speedup de 3.2 con respecto a la versión CUDA core en GPU [11].

3 Fractales 2D Empotrados

Esta sección se presenta la familia de fractales 2D empotrados discretos definidos por Navarro et al. (2020b)[14]. Además, se presentan 2 propiedades características de esta familia, relacionadas a su dimensión y volumen, los cuales proveen pistas útiles sobre como formular un mapeo de espacio paralelo GPU a esta familia de fractales.

3.1 La familia de fractales de construcción ascendente y sin traslape (NBB⁵)

La familia de fractales NBB son fractales 2D discretos cuya geometría irregular yace en \mathbb{Z}^2 cuando están contenidos en una caja euclidiana discreta. Están definidos por un nivel de escala r y una función de transición de un nivel a otro (paso de construcción). El nivel de escala r es cuantas veces se aplico la función de transición, y por lo tanto, esta relacionada directamente con el tamaño del fractal. Al ser estructuras discretas, estos fractales tienen una unidad fundamental (*i.e.* una unidad de espacio), y están formados por la agrupación de 1 o mas de estas unidades. Por este motivo, resulta conveniente construir estos fractales utilizando un enfoque ascendente, partiendo desde su unidad básica.

La construcción con un enfoque ascendente consiste en definir el fractal como replicas de si mismo a partir del nivel anterior, aplicando traslaciones diferentes

⁵ NBB por sus siglas en ingles: non-Overlapping Bottom-up Boxes

para cada replica. Adicionalmente, al trasladar cada replica se tiene que cumplir que su bounding-box asociado no se traslape con el bounding-box de otra en el espacio. En otras palabras, el espacio no puede ser ocupado por mas de un bounding-box de una replica. Cada fractal tiene además, 2 parámetros asociados a su paso de construcción: k y s , donde k representa el numero de replicas del nivel siguiente con respecto al anterior y s el aumento en tamaño del eje mas grande. La tabla 1 presenta una lista con algunos ejemplos de fractales NBB, su función de transición asociada y su dimensión Hausdorff.

Navarro et al. (2020b) definen 2 propiedades importantes sobre los fractales NBB. La primera propiedad tiene que ver con la dimensión y el espacio ocupado por el fractal. Esta establece que el espacio ocupado por un fractal NBB esta en correspondencia con su dimensión Hausdorff. Por lo tanto, el espacio discreto ocupado por un fractal NBB es $n^{\mathcal{H}}$, donde n corresponde al tamaño de un lado del fractal en un nivel r y el exponente \mathcal{H} corresponde a la dimensión Hausdorff. *i.e.* el cociente entre el logaritmo de k y el logaritmo de s . Esta propiedad garantiza que un fractal NBB, que solo puede crecer escalando hacia arriba, demuestre su dimensión Hausdorff cuando n tiende a infinito, y por lo tanto se tenga garantía de que el espacio ocupado sera menor o igual al de un bounding-box.

La segunda propiedad se relaciona con la geometría del espacio paralelo para dominios fractales NBB. Esta propiedad dice que un fractal NBB puede ser empaquetado en un 2-ortotopo de tamaño $k^{\lceil \frac{\mathcal{H}}{2} \rceil} \times k^{\lfloor \frac{\mathcal{H}}{2} \rfloor}$ para cualquier nivel de escala r . Es importante mencionar que esta propiedad entrega la posibilidad de declarar un espacio paralelo con la cantidad de hilos exacta a los elementos del fractal *i.e.* $n^{\mathcal{H}}$, lo que produciría una reducción en el desperdicio de recursos y potencialmente el tiempo de ejecución. Como se menciono anteriormente, el paradigma GPU solo permite definir el espacio paralelo como cajas en 1, 2 y 3 dimensiones, lo que se traduce en una limitante cuando se procesan dominios de datos que son irregulares, como los fractales NBB. En este contexto, la propiedad 2 entrega pistas de como se podría mapear espacio de su representación euclidiana a una fractal.

4 La función de mapeo $\lambda(\omega)$

Navarro et al. (2020b) definieron la función $\lambda : \mathbb{Z}_{\mathbb{E}}^2 \mapsto \mathbb{Z}_{\mathbb{F}}^2$ la cual mapea coordenadas de bloque de GPU desde espacio paralelo \mathcal{P}^2 , el cual yace en un espacio euclidiano $\mathbb{Z}_{\mathbb{E}}$, hacia coordenadas de bloque en el espacio empotrado del fractal $\mathbb{Z}_{\mathbb{F}}$. Es importante destacar que $\lambda(\omega)$ mapea coordenadas de bloque de GPU a elementos del fractal NBB empotrado. Por lo tanto ω representa a coordenadas de bloque de GPU (zona azul en figura 2). Utilizar un mapeo al nivel de bloque y no al nivel de hilos tiene 3 ventajas: Primero, a nivel de bloques el fractal se vuelve una versión menos fina del original, requiriendo menos elementos a ser mapeados. Segundo, ya que el fractal es una versión mas simplificada, es posible llegar a tamaños mayores de n antes de alcanzar los limites en tamaños de grid impuestos por CUDA o alcanzar limites numéricos. En tercer lugar, permite la posibilidad de que los hilos dentro del bloque conserven su localidad, lo que es


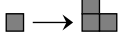

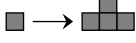

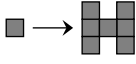
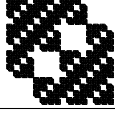
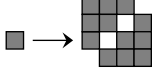
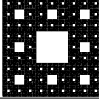




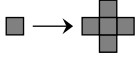

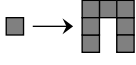

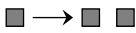
Nombre del Fractal	Ilustración	Paso de Construcción	Dimension Hausdorff ($\mathcal{H} = \frac{\log(k)}{\log(s)}$)
Triangulo de Sierpinski			$\frac{\log(3)}{\log(2)} \approx 1.58$
Candelabro			$\frac{\log(4)}{\log(3)} \approx 1.26$
Fractal \mathcal{H}			$\frac{\log(7)}{\log(3)} \approx 1.77$
Golosina			$\frac{\log(12)}{\log(4)} \approx 1.79$
Alfombra de Sierpinski			$\frac{\log(8)}{\log(3)} \approx 1.89$
Fractal \mathcal{X}			$\frac{\log(5)}{\log(3)} \approx 1.46$
Fractal Vicsek			$\frac{\log(5)}{\log(3)} \approx 1.46$
Botellas Vacías			$\frac{\log(7)}{\log(3)} \approx 1.77$
Set de Cantor			$\frac{\log(2)}{\log(3)} \approx 0.63$

Table 1. Ejemplos de fractales pertenecientes a la familia NBB.

esencial a la hora de realizar eficientes lecturas alineadas a memoria de GPU. La versión simplificada del fractal que surge al utilizar un mapeo al nivel de bloque de GPU tiene como nuevo tamaño $n_b = n/b$ con $b = \rho$, para un bloque $|B| = \rho \times \rho$. La figura 5 presenta una descripción de la situación.

Una propiedad importante de $\lambda(\omega)$ es que esta demostrado que su complejidad al mapear un espacio paralelo de tamaño $|\mathcal{P}^2| = \mathcal{O}(n^{\mathcal{H}})$ a cualquier fractal NBB en tiempo $\mathcal{O}(\log_2 \log_2(n_b))$ utilizando solo $|B| = \theta(\frac{\log_2(n_b)}{\log_2 \log_2(n_b)})$ hilos por

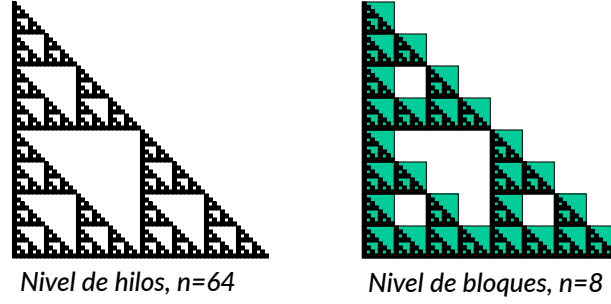


Fig. 5. Diferencia de un mapeo a nivel de hilos vs a nivel de bloques utilizando el Triangulo de Sierpinski en nivel de escala $r = 6$. A la izquierda el fractal a mapear utilizando un mapeo a nivel de hilos. Notese que no hay hilos que caigan fuera de los datos. A la derecha el mismo fractal utilizando un mapeo a nivel de bloque de $|B| = 8 \times 8$. La versión simplificada es un fractal de $n_b = 64/8 = 8$. Las zonas verdes representan hilos desperdiciados.

bloque. Además también esta comprobado que procesar un fractal NBB utilizando $\lambda(\omega)$ requiere una cantidad de trabajo asintoticamente menor que al usar un enfoque bounding-box. La importancia de esta ultima propiedad es que garantiza la existencia de un fractal de tamaño $n > n_0$ donde el Speedup entregado por $\lambda(\omega)$ se vuelve monótonamente creciente. Mientras mas pequeña es la dimensión de Hausdorff, mas notorio es este comportamiento. La intuición detrás de $\lambda(\omega)$ es calcular las coordenadas resultantes utilizando como base el método de construcción de un fractal NBB descrito en la sección anterior, desde su primer elemento hasta el nivel r_b , con $r_b = \log_s(n_b)$ (*i.e.* el nivel de escala del fractal a nivel de bloque). Para calcular $\lambda(\omega)$, primero se debe obtener el valor de la función auxiliar $\beta_\mu(\omega)$, la cual entrega el índice que identifica a una región dentro del fractal correspondiente a un bloque de GPU. Este valor esta en el rango $\beta_\mu(\omega) \in [0, k - 1]$. La función auxiliar $\beta_\mu(\omega)$ esta definida como

$$\beta_\mu(\omega) = \left(\frac{\omega_x(\mu \bmod 2) + \omega_y((\mu + 1) \bmod 2)}{k^{\lceil \frac{\mu}{2} \rceil - 1}} \right) \bmod k \quad (1)$$

la cual luego es utilizada para acceder a una tabla hash $H[\]$ de tamaño k con la traslación correspondiente a la región denotada por $\beta_\mu(\omega)$. Esta traslación esta denotada por (τ_x^μ, τ_y^μ) , definida como

$$\tau^\mu = H[\beta_\mu(\omega)] = (\tau_x^\mu, \tau_y^\mu), \quad \tau_x^u, \tau_y^u \in [0..s - 1]. \quad (2)$$

Las traslaciones (τ_x^μ, τ_y^μ) en conjunto con el largo de la región del fractal correspondiente $s^{\mu-1}$, entrega la magnitud de la traslación en el espacio empotrado

$$\Delta^\mu = (\tau_x^\mu(s)^{\mu-1}, \tau_y^\mu(s)^{\mu-1}) = (\Delta_x^\mu, \Delta_y^\mu) \quad (3)$$

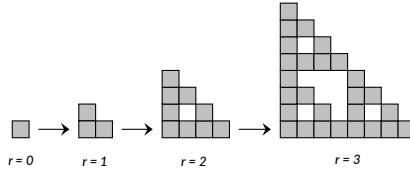


Fig. 6. Construcción con un enfoque ascendente del triángulo de Sierpinski discreto.

el cual contribuye a la coordenada final mapeada. La sumatoria de todas las coordenadas parciales entonces produce el mapeo

$$\lambda(\omega) = (\lambda_x(\omega), \lambda_y(\omega)), \quad (4)$$

$$\lambda_x(\omega) = \sum_{\mu=1}^{r_b} \Delta_x^\mu \quad (5)$$

$$\lambda_y(\omega) = \sum_{\mu=1}^{r_b} \Delta_y^\mu \quad (6)$$

Es importante mencionar que la tabla hash $H[\]$ puede ser reemplazada por una función aritmética que produzca los mismos resultados. Además, esta tabla es la misma para cada nivel de escala de fractal y por lo tanto puede ser reutilizada.

5 Adaptando $\lambda(\omega)$ a tensor cores

Esta sección propone tres algoritmos que potencialmente puedan acelerar el cálculo de la función $\lambda(\omega)$ utilizando tensor cores. Para demostrar su factibilidad se utiliza el fractal triángulo de Sierpinski como caso de estudio.

El triángulo de Sierpinski, ilustrado en la figura 6, fue descrito por Waclaw Sierpinski en 1915. Este fractal aun es objeto de estudio en distintas áreas como la construcción de antenas[1][22], simulaciones de autómatas celulares [18][26], ensamblajes moleculares con forma de fractal [5], auto-organización de ADN [25], teoría de auto-ensamblajes [2][7], entre otros. El fractal de Sierpinski está identificado por $k = 3$ y $s = 2$.

Adaptando la función $\lambda(\omega)$ al triángulo de Sierpinski se obtienen las siguientes ecuaciones:

Para la función auxiliar de índices $\beta_\mu(\omega)$ se tiene

$$\beta_\mu(\omega) = \left(\frac{\omega_x(\mu \bmod 2) + \omega_y((\mu + 1) \bmod 2)}{3^{\lceil \frac{\mu}{2} \rceil - 1}} \right) \bmod 3. \quad (7)$$

Las regiones del fractal son arbitrariamente enumeradas como 0 la de arriba, 1 la del medio y 2 la de la derecha. La tabla hash para este fractal es $H[0] =$

$(0,0), H[1] = [0,1], H[2] = [1,1]$, la cual puede ser sustituida por la función aritmética que arroja los mismos resultados

$$h(\beta_\mu) = (\tau_x^\mu, \tau_y^\mu) = \left(\left\lfloor \frac{\beta_\mu}{2} \right\rfloor, \beta_\mu - \left\lfloor \frac{\beta_\mu}{2} \right\rfloor \right) \quad (8)$$

de la cual se obtienen los mismos resultados independiente del escala transitorio μ . La magnitud de las traslaciones en cada nivel están dados por

$$\Delta^\mu = (\Delta_x^\mu, \Delta_y^\mu) = (\tau_x^\mu 2^{\mu-1}, \tau_y^\mu 2^{\mu-1}) \quad (9)$$

finalmente la función de mapeo resulta ser

$$\lambda(\omega) = \left(\sum_{\mu=1}^{r_b} \Delta_x^\mu, \sum_{\mu=1}^{r_b} \Delta_y^\mu \right) \quad (10)$$

con $r_b = \log_2(n_b)$.

Para lograr una aceleración en el calculo de $\lambda(\omega)$ utilizando tensor cores, su ecuación debe ser codificada en una operación MMA de la forma $D = A \times B + C$, donde A, B, C y D son fragmentos de 16×16 . Existen un sin fin de formas distintas de realizar esta codificación y esta sección presenta 3 variantes a una codificación en tensor cores. La adaptación de $\lambda(\omega)$ a tensor cores se define como $\lambda_{tc}(\omega)$.

5.1 Variante 1: operación tensor core simple y por bloque

Esta variante ejecuta el calculo de una operación tensor cores por cada bloque de GPU para obtener 2 coordenadas y se aprovecha de la similitud entre la función $\lambda(\omega)$ y una operación MMA. Para lograr esto, se realiza una expansión de la sudatoria de la ecuación 10, además de descomponer los términos $\Delta_x^\mu, \Delta_y^\mu$ de la ecuación 9 en un producto de sus factores. Esto resulta en dos sumatorias de productos, una para calcular λ_x y la otra para λ_y . Este comportamiento es el mismo que se obtiene en cada elemento de la matriz resultante al multiplicar 2 matrices. Luego cada factor izquierdo de los productos en sumatoria se ubica como el primer elemento de una columna en el fragmento A, mientras que cada factor izquierdo es ubicado como primer y segundo valores en cada fila del fragmento B, para λ_x y λ_y , respectivamente. Como el factor izquierdo de cada multiplicación corresponde a potencias de 2 desde $2^0 \rightarrow 2^{\mu-1}$, y son iguales para λ_x y λ_y , estos factores pueden se codificados solo utilizando una fila de la matriz A, reutilizada para el calculo de ambas coordenadas. La codificación final puede ser visualizada en la figura 7.

Una importante nota técnica es que el fragmento B fue definido en memoria como una matriz indexada por columna (o "column-major" en ingles) para mejorar el alineamiento de los accesos a memoria durante el calculo y minimizar la divergencia de los datos. Una vez ejecutada la operación tensor cores, los resultados λ_x y λ_y se ubican en el primer y segundo elemento del fragmento D, respectivamente.

$$A = \begin{pmatrix} 2^0 & 2^1 & \dots & 2^{\mu-1} \\ 0 & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 0 \end{pmatrix} \quad B = \begin{pmatrix} \tau_x^1 & \tau_y^1 & 0 & \dots & 0 \\ \tau_x^2 & \tau_y^2 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \tau_x^\mu & \tau_y^\mu & 0 & \dots & 0 \end{pmatrix}$$

Fig. 7. Codificación de la Variante 1 en una operación MMA. Notar que estas matrices son de $\mu \times \mu$ que serán empotradas en los fragmentos de 16×16 , alineándose con el primer elemento.

5.2 Variante 2: operación tensor cores de sub-bloques

Esta variante comparte el principio de la variante 1, pero expande la idea al subdividir cada bloque de GPU en 4 sub-bloques lógicos, calculando más coordenadas por operación tensor core con respecto a la variante 1. Como cada fragmento tiene 16 filas y 16 columnas, utilizando la variante 1 existe la posibilidad de calcular 16 valores diferentes; 8 pares de coordenadas (λ_x, λ_y) , sin embargo, cada bloque solo necesita 1 par (λ_x, λ_y) , ya que son compartidas por todos sus hilos. Este enfoque intenta aprovechar mejor la operación tensor core al calcular 4 pares de coordenadas (λ_x, λ_y) por lo que divide el bloque de GPU en sub-bloques mas pequeños que son tratados lógicamente como bloques reales. De esta manera, a cada sub-bloque le correspondería un par de coordenadas (λ_x, λ_y) . Se asume un tamaño de bloque de GPU lo suficientemente grande para contener 4 sub-bloques lógicos de tamaño $b/2 \times b/2$ que aun sean lo suficientemente grandes para producir un calculo eficiente. La figura 8 demuestra un ejemplo de la subdivisión cuando se tiene originalmente un bloque de GPU de tamaño 32×32 y se subdivide en 4 sub-bloques lógicos de 16×16 .

Una vez que se realiza la operación tensor core, la coordenadas resultantes para cada sub-bloque se encuentran en la primera fila del fragmento D resultante. Es importante mencionar que este enfoque introduce grupos de hilos que no serán utilizados (marcados por una X en la figura 8), esto ocurre ya que al declarar un bloque lo suficientemente grande para almacenar los 4 sub-bloques, hay trozos que caen fuera del 2-ortotopo de espacio paralelo generado. Estos hilos extras no introducen una penalización significativa al rendimientos puesto que la cantidad es acotada superiormente por $O(\sqrt{n^{\mathcal{H}}})$ (*i.e.* el perímetro del fractal compacto).

5.3 Variante 3: aprovechamiento completo de los fragmentos A, B y C

Los 2 variantes descritos anteriormente solo utilizan los fragmentos A y B para el calculo de lambda, dejando de lado por completo el fragmento C y por ende, la suma de la operacional MMA. Además, la mayoría de los elementos de los fragmentos no contienen datos, lo que provoca que en teoría, no se este aprovechando por completo la operación MMA. Esta tercer variante mantiene la misma codificación para A y B, pero incluye a C en el calculo, llenando por completo todos los campos disponibles de los 3 fragmentos. La figura 9 muestra la configuración

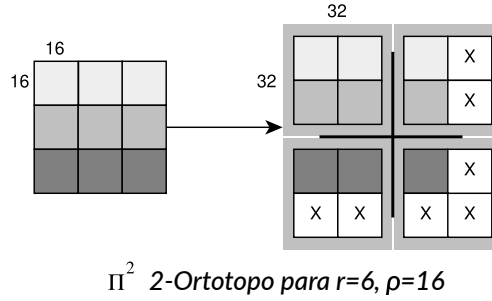


Fig. 8. A la izquierda un ejemplo del espacio paralelo generado para la variante 2. Se asume un tamaño de bloque 16×16 , donde cada uno de los 9 sub-bloques son mapeados de forma independiente. A la derecha el tamaño de bloque real ocurriendo en la GPU y su subdivisión correspondiente.

de cada matriz. El resultado de la operación tensor core, a diferencia de las variantes anteriores, es una coordenada de hilo de GPU en espacio de datos, mientras que en las variantes anteriores era una coordenada de bloque de GPU, que sería compartida por todos los hilos. Esto se traduce a que cada hilo ahora recibe directamente su coordenada desde fragmento D. Para ejecutar esta variante se utilizan 2 operaciones tensor core por bloque de GPU, una para el cálculo de λ_x y otra para λ_y de cada hilo. Esta variante fue desarrollada para tamaños de bloque de GPU de 16×16 para calzar con las 256 coordenadas disponibles en el fragmento D. La versión para bloques de tamaño 32×32 utiliza una estrategia de sub-bloques similar a la utilizada en la variante 2, para obtener 4 sub-bloques de 16×16 . Este caso requeriría la ejecución de 8 operaciones tensor cores en total por bloque. El acceso a espacio de datos por parte de los sub-bloques es contiguo, a diferencia del variante 2 donde los sub-bloques eran tratados como bloques independientes y ubicados en posiciones arbitrarias.

$$A = \begin{pmatrix} 2^0 & 2^1 & \dots & 2^{\mu-1} \\ 2^0 & 2^1 & \dots & 2^{\mu-1} \\ \vdots & \vdots & \ddots & \vdots \\ 2^0 & 2^1 & \dots & 2^{\mu-1} \end{pmatrix} \quad B_x = \begin{pmatrix} \tau_x^1 & \tau_x^1 & \dots & \tau_x^1 \\ \tau_x^2 & \tau_x^2 & \dots & \tau_x^2 \\ \vdots & \vdots & \ddots & \vdots \\ \tau_x^\mu & \tau_x^\mu & \dots & \tau_x^\mu \end{pmatrix}$$

$$C_x = \begin{pmatrix} t_{1,1}^x & t_{1,2}^x & \dots & t_{1,\mu}^x \\ t_{2,1}^x & t_{2,2}^x & \dots & t_{2,\mu}^x \\ \vdots & \vdots & \ddots & \vdots \\ t_{\mu,1}^x & t_{\mu,2}^x & \dots & t_{\mu,\mu}^x \end{pmatrix}$$

Fig. 9. Fragmentos A, B y C para calcular una coordenada de un bloque utilizando la variante 3. Los fragmentos B_y y C_y comparten la misma representación que B_x y C_x .

6 Resultados

La función $\lambda(\omega)$ adaptada para el triángulo de Sierpinski y sus variantes utilizando tensor cores fueron programados con la herramienta de trabajo CUDA C++ de NVIDIA. El programa recibe como entrada un nivel de escala r para el fractal y del enfoque a utilizar: $\lambda(\omega)$ clásico, variante 1, 2 o 3. Los benchmark experimentales incluyen una fase de trabajo aplicado una vez finalizada la fase de mapeo, con el fin de representar escenarios realísticos de ejecución. Tres pruebas diferentes fueron implementadas como fase de trabajo aplicado:

- Escritura simple (SW): Esta prueba consiste en escribir un valor constante en todos los elementos del espacio de datos que pertenezcan al triángulo de Sierpinski de nivel r , el cual está empujado en una matriz de $n \times n$.
- Reducción aritmética (RD): Consiste en realizar una reducción aritmética de suma de todos los elementos que pertenecen al fractal.
- Simulación de Autómata Celular (CA): Consiste en realizar una simulación de autómata celular utilizando una adaptación de las reglas del juego de la vida de Conway. Esta adaptación aun utiliza la vecindad de Moore euclidiana, pero solo considera como vecinos aquellas células que pertenezcan al fractal.

Los tamaños utilizados para las pruebas están en el rango $r = 0..16$ (hasta $r = 15$ para las pruebas CA y RD por limitaciones de memoria), y utilizando tamaños de bloque de GPU en el rango $\rho = 8, 16, 32$, esto es ya que para realizar una operación tensor cores se requieren como mínimo 32 hilos de GPU en un bloque, lo que se consigue a partir de $\rho = 8$. El resultado promedio fue obtenido promediando 100 tiempos donde cada uno corresponde a un promedio de 10 ejecuciones consecutivas del kernel. Para asegurar que no hayan sesgos en los resultados, cada llamado al kernel se realizó seguido de una sincronización de la GPU que actúa como un reinicio a mismo estado inicial. Además, la máquina de test y la GPU se encontraban sin cargas adicionales. El hardware para las pruebas está listado en la tabla 2.

Table 2. Hardware utilizado para las pruebas de rendimiento.

#	Dispositivo	Modelo
0	GPU	Titan V, 5120 cuda cores 12GB
	CPU	Intel i7-6950X 10-core Broadwell
	RAM	128GB DDR4 2400MHz
1	GPU	Titan RTX, 4608 cuda cores, 24GB
	CPU	Intel i7-6950X 10-core Broadwell
	RAM	128GB DDR4 2400MHz

Los Speedups obtenidos de las variantes tensor cores con respecto al λ clásico se presentan en la figura 10. Es importante considerar los resultados de

las arquitecturas Volta y Turing, considerando que las implementaciones interna de un tensor core pueden resultar en rendimientos distintos.

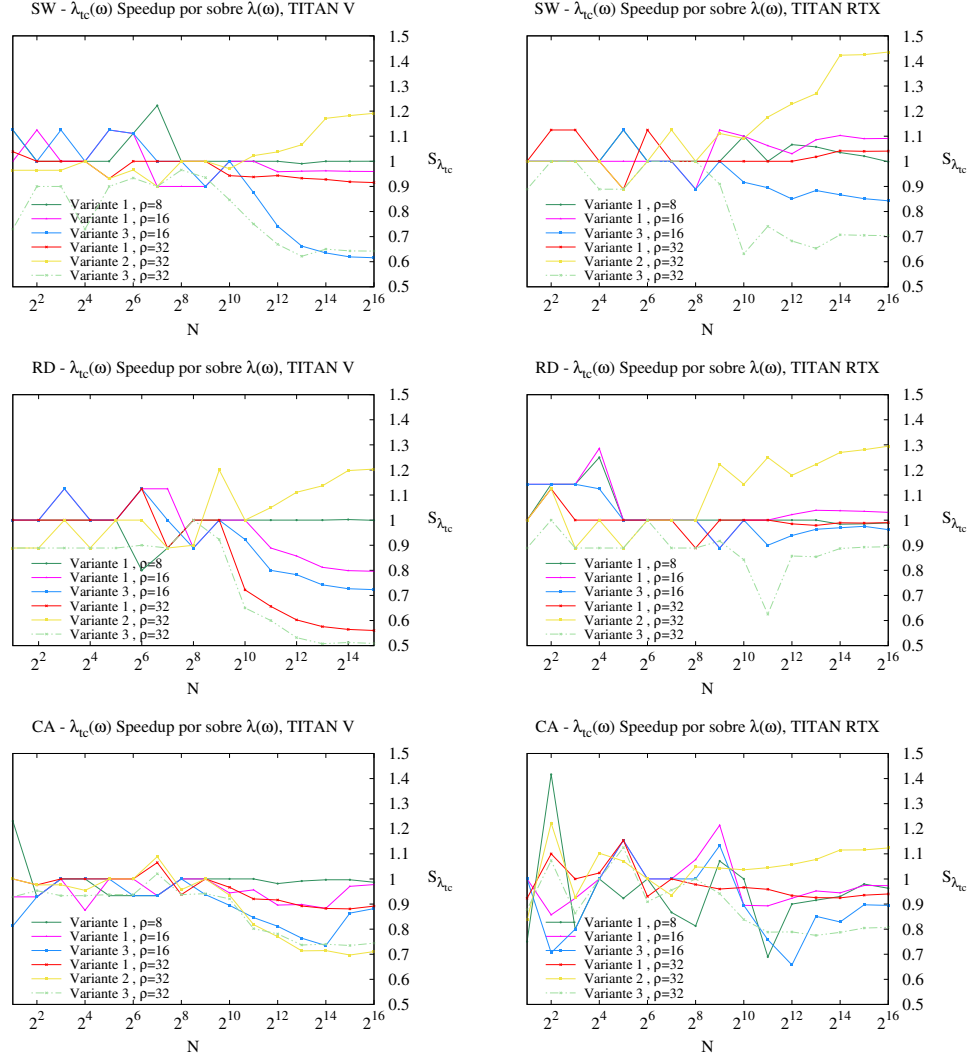


Fig. 10. Los gráficos muestran el Speedup de $\lambda_{tc}(\omega)$ con respecto a $\lambda(\omega)$. La columna izquierda muestra los resultados en la GPU TITAN V (arquitectura Volta). La columna derecha muestra los resultados en la GPU TITAN RTX (arquitectura Turing).

Comenzando con la prueba SW, los resultados en la TITAN V muestran que cuando $n \geq n_0 = 2^{10}$, las curvas de Speedup alcanzan un comportamiento estable, mientras que cuando $n < n_0$, las curvas muestran un comportamiento

inestable, oscilando al rededor de 1.0. Para tamaños grandes, la variante 2 alcanza un máximo de 20% extra de rendimiento por sobre $\lambda(\omega)$. Con la TITAN RTX, las curvas de Speedup muestran que nuevamente la variante 2 alcanza el máximo de rendimiento con un 40% extra. Además la variante 1 también se vuelve una opción viable, pues alcanza un Speedup de 1.1. En ambas GPUs la variante 3 obtiene Speedups menores a 1 cuando $n \geq n_0$, en ambas configuraciones. La prueba RD arrojaron resultados similares a las de SW, con la variante 2 obteniendo el mejor rendimiento con Speedups de 1.2 y 1.3 para la TITAN V y TITAN RTX, respectivamente. La variante 1 solo obtiene rendimiento extra en la TITAN RTX con tamaño de bloque $\rho = 16$. La variante 3 resulto ineficiente en ambas tarjetas. Finalmente, los resultados en CA muestran que cuando $n < n_0$, el comportamiento de las curvas Speedup es igual a las pruebas anteriores, pero cuando n crece a partir de n_0 los resultados en la TITAN V comienzan a disminuir bajo 1. Para la TITAN RTX ocurre una situación similar, con la excepción de la variante 2 que alcanza un extra positivo en rendimiento, con un 11%.

Resumiendo los resultados en tensor core, la variante que obtuvo los mejores resultados en general fue la variante 2, con un rendimiento máximo de 40% por sobre el lambda tradicional cuando $n > 2^{10}$. La variante 1 también se mostró superior en ciertas configuraciones con un 5 ~ 10% de rendimiento extra. La variante 3 resulto ser la peor, con Speedups menores a 1 en todas sus configuraciones.

7 Discusión y Conclusión

Este trabajo presenta 3 variantes a la adaptación de $\lambda(\omega)$ a tensor core, $\lambda_{tc}(\omega)$. Cada variante utiliza diferentes principios en torno a como se adapta el mapeo a un paradigma basado en tensor cores. De las tres variantes, la segunda resulto ser consistentemente mejor que lambda original una vez que el tamaño de problema es $n \geq 2^{10}$. La variante 1 resultó ser beneficiosa en algunas configuraciones mientras que la variante 3, en ninguna. La adaptación de $\lambda(\omega)$ a tensor cores alcanzó un máximo de ~ 40% en rendimiento extra.

Para lograr una adaptación a tensor core eficiente es importante utilizar de la mejor manera posible las comunicaciones entre los fragmentos de tensor core con la memoria compartida de GPU, y además, lograr una codificación simple, con la menor cantidad de datos redundantes en el fragmento, ya que de esta manera el costo extra de llenar los fragmentos es compensando con la velocidad de una operación tensor core que procesa mas elementos distintos. Mientras mas cercana es la tarea a álgebra lineal, será más fácil la adaptación. Todas las variantes de adaptación a tensor core presentadas utilizan fragmentos de tamaño 16×16 , lo que introduce una restricción en el tamaño máximo para el fractal con el que se puede trabajar, porque En este caso en particular, el tamaño máximo es cuando el nivel de escala del fractal utilizando un mapeo a nivel de bloque es $r_b = 16$. Sin embargo, ya que el mapeo se hace en el espacio de bloques, un fractal con este nivel de escala se traduce a manejar un espacio de datos

de tamaño $2^{19} \times 2^{19}$, es decir, 274877906944 elementos, lo que significaría un mínimo de 256GB de memoria para almacenar el fractal (utilizando 1 byte por dato). Es importante mencionar que la adaptación en tensor core presentada en este trabajo podría ser implementada por cualquier fractal perteneciente a la familia NBB.

Como trabajo futuro es interesante considerar la existencia de una función $\lambda(\omega)^{-1}$ que pueda mapear los elementos desde el espacio de datos a un 2-ortotopo compacto, y la existencia de su homólogo en tensor cores $\lambda(\omega)_{tc}^{-1}$. Esto permitiría que, además de comprimir los datos pertenecientes al fractal, se pudiera realizar trabajo en el espacio compacto, sin empotrarlo en un espacio euclidiano, logrando así poder trabajar con tamaños superiores sin tener que fraccionar el fractal, del orden de n^H . Además es interesante considerar acelerar otras ecuaciones de mapeo utilizando tensor cores y consolidar su factibilidad a la hora de realizar tareas ajenas al álgebra lineal.

References

1. Baliarda, C.P., Borau, C.B., Rodero, M.N., Robert, J.R.: An iterative model for fractal antennas: application to the sierpinski gasket antenna. *IEEE Transactions on Antennas and Propagation* **48**(5), 713–719 (May 2000). <https://doi.org/10.1109/8.855489>
2. Doty, D.: Theory of algorithmic self-assembly. *Commun. ACM* **55**(12), 78–88 (Dec 2012). <https://doi.org/10.1145/2380656.2380675>, <http://doi.acm.org/10.1145/2380656.2380675>
3. Gamba, A., Ambrosi, D., Coniglio, A., de Candia, A., Di Talia, S., Giraudo, E., Serini, G., Preziosi, L., Bussolino, F.: Percolation, morphogenesis, and burgers dynamics in blood vessels formation. *Phys. Rev. Lett.* **90**, 118101 (Mar 2003). <https://doi.org/10.1103/PhysRevLett.90.118101>, <https://link.aps.org/doi/10.1103/PhysRevLett.90.118101>
4. He, K., Xu, C.Y., Zhen, L., Shao, W.Z.: Fractal growth of single-crystal α -Fe₂O₃: From dendritic micro-pines to hexagonal micro-snowflakes. *Materials Letters* **62**(4–5), 739 – 742 (2008). <https://doi.org/https://doi.org/10.1016/j.matlet.2007.06.082>, <http://www.sciencedirect.com/science/article/pii/S0167577X07006647>
5. Jian Shang, Wang Yongfeng, M.C., et al.: Assembling molecular Sierpiński triangle fractals. *Nat Chem* **7**(5), 389–393 (May 2015). <https://doi.org/10.1038/nchem.2211>, <http://dx.doi.org/10.1038/nchem.2211>
6. Jung, J.H., O’Leary, D.P.: Exploiting structure of symmetric or triangular matrices on a gpu. Tech. rep., University of Maryland (2008)
7. Lathrop, J.I., Lutz, J.H., Summers, S.M.: Strict self-assembly of discrete sierpinski triangles. *Theoretical Computer Science* **410**(4), 384 – 405 (2009). <https://doi.org/http://dx.doi.org/10.1016/j.tcs.2008.09.062>, <http://www.sciencedirect.com/science/article/pii/S030439750800724X>
8. Ma, K., Li, X., Chen, W., Zhang, C., Wang, X.: Greengpu: A holistic approach to energy efficiency in gpu-cpu heterogeneous architectures. In: 2012 41st International Conference on Parallel Processing. pp. 48–57 (2012)
9. Mandelbrot, B.B.: *Fractals*. John Wiley & Sons, Inc. (2004). <https://doi.org/10.1002/0471667196.ess0816>, <http://dx.doi.org/10.1002/0471667196.ess0816>

10. Milne, B.T.: Measuring the fractal geometry of landscapes. *Applied Mathematics and Computation* **27**(1), 67 – 79 (1988). [https://doi.org/http://dx.doi.org/10.1016/0096-3003\(88\)90099-9](https://doi.org/http://dx.doi.org/10.1016/0096-3003(88)90099-9), <http://www.sciencedirect.com/science/article/pii/0096300388900999>
11. Navarro, C.A., Carrasco, R., Barrientos, R.J., Riquelme, J.A., Vega, R.: Gpu tensor cores for fast arithmetic reductions. *IEEE Transactions on Parallel and Distributed Systems* **32**(1), 72–84 (2021)
12. Navarro, C.A., Vega, R., Bustos, B., Hitschfeld, N.: Block-space gpu mapping for embedded sierpiński gasket fractals. In: 2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS). pp. 427–433 (Dec 2017). <https://doi.org/10.1109/HPCC-SmartCity-DSS.2017.56>
13. Navarro, C.A., Bustos, B., Hitschfeld, N.: Potential benefits of a block-space GPU approach for discrete tetrahedral domains. In: CLEI-2016, XLII Conferencia Latinoamericana de Informática, Valparaíso, Chile, October 10-14, 2016 (2016)
14. Navarro, C.A., Canfora, F., Hitschfeld, N., Navarro, G.: Parallel family trees for transfer matrices in the potts model. *Computer Physics Communications* **187**(0), 55 – 71 (2015). <https://doi.org/http://dx.doi.org/10.1016/j.cpc.2014.10.011>, <http://www.sciencedirect.com/science/article/pii/S0010465514003464>
15. Navarro, C.A., Hitschfeld, N.: GPU maps for the space of computation in triangular domain problems. In: 2014 IEEE International Conference on High Performance Computing and Communications, HPCC/CSS/ICSS 2014, Paris, France, August 20-22, 2014. pp. 375–382 (2014). <https://doi.org/10.1109/HPCC.2014.64>, <http://dx.doi.org/10.1109/HPCC.2014.64>
16. Navarro, C.A., Hitschfeld-Kahler, N., Mateu, L.: A survey on parallel computing and its applications in data-parallel problems using GPU architectures. *Commun. Comput. Phys.* **15**, 285–329 (2014)
17. Navarro, C.A., Vernier, M., Bustos, B., Hitschfeld, N.: Competitiveness of a non-linear block-space gpu thread map for simplex domains. *IEEE Transactions on Parallel and Distributed Systems* **29**(12), 2728–2741 (2018)
18. Ohi, F., Takamatsu, Y.: Time-space pattern and periodic property of elementary cellular automata — sierpinski gasket and partially sierpinski gasket — . *Japan Journal of Industrial and Applied Mathematics* **18**(1), 59 (2001). <https://doi.org/10.1007/BF03167355>, <http://dx.doi.org/10.1007/BF03167355>
19. Oppenheimer, P.E.: Real time design and animation of fractal plants and trees. *SIGGRAPH Comput. Graph.* **20**(4), 55–64 (Aug 1986). <https://doi.org/10.1145/15886.15892>, <http://doi.acm.org/10.1145/15886.15892>
20. Palmer, M.W.: Fractal geometry: a tool for describing spatial patterns of plant communities. *Vegetatio* **75**(1), 91–102 (1988). <https://doi.org/10.1007/BF00044631>, <http://dx.doi.org/10.1007/BF00044631>
21. Pentland, A.P.: Fractal-based description of natural scenes. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **PAMI-6**(6), 661–674 (Nov 1984). <https://doi.org/10.1109/TPAMI.1984.4767591>
22. Puente-Baliarda, C., Romeu, J., Pous, R., Cardama, A.: On the behavior of the sierpinski multiband fractal antenna. *IEEE Transactions on Antennas and Propagation* **46**(4), 517–524 (Apr 1998). <https://doi.org/10.1109/8.664115>
23. Qasaimeh, M., Denolf, K., Lo, J., Vissers, K., Zambreno, J., Jones, P.H.: Comparing energy efficiency of cpu, gpu and fpga implementations for vision kernels. In:

- 2019 IEEE International Conference on Embedded Software and Systems (ICESS). pp. 1–8 (2019)
24. Ries, F., De Marco, T., Zivieri, M., Guerrieri, R.: Triangular matrix inversion on graphics processing unit. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis. pp. 9:1–9:10. SC '09, ACM, New York, NY, USA (2009)
 25. Rothmund PWK, Papadakis N, W.E.: Algorithmic self-assembly of dna sierpinski triangles. PLoS Biol **2**(12), e424 (2004). <https://doi.org/https://doi.org/10.1371/journal.pbio.0020424>
 26. Wolfram, S.: Statistical mechanics of cellular automata. Rev. Mod. Phys. **55**(3), 601–644 (Jul 1983). <https://doi.org/10.1103/RevModPhys.55.601>, <http://link.aps.org/doi/10.1103/RevModPhys.55.601>