Potencial Aceleración de Simulaciones de Autómatas Celulares utilizando Tensor Cores

Roberto Melita[1][0000−0003−3018−8801]

Universidad Austral de Chile, Valdivia roberto.melita@alumnos.uach.cl

Resumen Con el reciente auge de aplicaciones basadas en *Machine Learning*, en particular *Deep Learning*, NVIDIA introdujo en el 2017 los *Tensor Cores*. Estos son una nueva clase de núcleo de procesamiento específico para operar productos de matrices, a un rendimiento superior que el de los núcleos de procesamiento general de la GPU. Si bien originalmente los *Tensor Cores* fueron diseñados para aumentar el rendimiento del entrenamiento e inferencia de redes neuronales, existe la posibilidad de adaptar otros algoritmos para que se beneficien de esta aceleración adicional. Teniendo en cuenta esta premisa, en este trabajo se investiga la potencial aceleración que existiría al utilizar *Tensor Cores* para simular autómatas celulares, los cuales corresponden a simulaciones de tipo *Nearest-Neighbors* en espacios estructurados. Como caso de estudio, se analiza el autómata celular *Game of Life* de Conway que corresponde a una simulación en una grilla bidimensional. La estrategia para usar los *Tensor Cores* consiste en adaptar el cálculo de la función de transición en cada celda, a un cálculo de submatrices usando multiplicación y suma ($D = A \times B + C$). Utilizando un modelo teórico de costos simplificado para la GPU (basado en el modelo PRAM), se realiza un análisis del costo computacional del método propuesto con respecto a dos algoritmos comúnmente usados. Como resultado se obtuvo que el método propuesto basado en *Tensor Cores* logra un *Speedup* teórico de hasta 9× con respecto a una simulación tradicional en GPU y de hasta 2× con respecto a una simulación que usa caché programable. Los resultados de este trabajo sugieren que existe una potencial aceleración en la simulación de autómatas celulares si se emplea el uso de tensor cores.

Keywords: *Computación de Alto Rendimiento · Tensor Cores · Autómata Celular· Vecinos Cercanos.*

1. Introducción

En la ultima decada, la revolución de la inteligencia artificial (IA) ha sido en parte gracias a la gran cantidad de datos que nos provee el fenómeno de *Big Data* y también al gran rendimiento que entregan los procesos paralelos como la GPU (Graphics Processing Unit), haciendo que tareas como la de entrenar una red neuronal profunda (*Deep Neural Network* o *Deep Learning*), se vuelva posible en tiempos acotados. Debido a esto la demanda de procesamiento potente ha aumentado, por lo que NVIDIA, a fines del año 2017 y junto al lanzamiento de la

arquitectura *Volta* presentó un nuevo circuito integrado de aplicación específica (ASIC) denominado *Tensor Core*, el cual viene incluido dentro de sus GPUs. Este nuevo ASIC permite un aumento de rendimiento de hasta 10 veces [1] en el entrenamiento de redes neuronales. Con la primicia de esta gran mejora de rendimiento, es donde surge una gran oportunidad, ya que, si no estamos utilizando estas GPU para IA, estamos desaprovechando valiosos recursos *hardware*. Este trabajo busca extender la utilidad de los *Tensor Cores* y aprovechar este rendimiento extra en otras áreas de la ciencia distintas a IA. Más exactamente al área de las simulaciones de autómatas celulares, que pertenecen a la clase de simulaciones de tipo *Nearest-Neighbors* en espacios discretos estructurados. Las simulaciones en espacios estructurados utilizando vecinos mas cercanos son muy usadas en la ciencia y tecnología, tal como en el caso de la simulación de cristales, materiales ferromagnéticos (spin models), simulación de PDEs con diferencias finitas, convoluciones, entre otros. Para evaluar el método propuesto utilizamos el autómata celular de John Horton Conway [3] , mayormente conocido con el nombre Juego de la Vida (o *Game of Life*, en inglés), en donde se propone un nuevo método de simulación por *Tensor Cores* y se analiza su costo bajo un modelo de computo de GPU similar al PRAM[18]. Como resultado del análisis, se obtuvo una mejora de rendimiento teórico aproximadamente de $9\times$ con respecto a una implementación básica en GPU y $2\times$ con respecto a una implementación avanzada que usa cache programable.

El resto del artículo se estructura de la siguiente forma: Sección 2 cubre los trabajos relacionados, la Sección 3 explica los conceptos esenciales necesarios para entender el documento, la Sección 4 muestra el método propuesto, la Sección 5 presenta el análisis del costo computacional del método propuesto y el *Speedup* teórico, en la Sección 6 se realiza una discusión del análisis y la relevancia de los resultados, y finalmente la Sección 7 contiene las principales conclusiones del trabajo.

2. Trabajo Relacionado

En la literatura de los ultimos años se pueden encontrar ya varios trabajos enfocados en comprobar la eficiencia que generan los *Tensor Cores*, realizando pruebas [4,6,8,12] de rendimiento, de precisión, de programabilidad, entre otros. Los autores de estos trabajos concluyen que en gran parte si pueden extraer el rendimiento prometido de los *Tensor Cores* en las aplicaciones indicadas para estos núcleos, es decir *Deep Learning* y álgebra lineal.

Adicionalmente se pueden encontrar trabajos que se alinean con el trasfondo de este documento que es utilizar los *Tensor Cores* para acelerar tareas distintas a la inferencia y entrenamiento de inteligencia artificial y/o algebra lineal, como por ejemplo, trabajos orientados a la reducciones aritméticas [5,17,23], donde en el trabajo de R. Carrasco *et al.* se logra un aumento de rendimiento de aproximadamente $\frac{4}{5}\log_2(m^2)$ (con m siendo el tamaño lineal de la matriz aceptada por un *Tensor Core*) en la teoria, y hasta $3{,}2\times$ de aceleración en la practica. Otro buen ejemplo es el trabajo de C. Navarro *et al.* [19], acá se analiza el mapeo

eficiente de hilos (*threads*) en fractales 2D reportando una mejoría de hasta un 40 % con la utilización de los Tensor Cores.

Posiblemente el trabajo más relevante para esta investigación es el de Shalyapina N.A. y Gromov M.L [2]. En donde logran obtener un *Speedup* en el Juego de la Vida planteando un nuevo método basado en matrices y convoluciones. Este método conociste en realizar una convolución entre la matriz que describe al autómata y un kernel de 3×3, donde cada vecino tiene un peso de 1 y la célula misma un peso de 0,5, obteniendo como resultado entero la suma de los vecinos y en los decimales un 0,5 o un 0,0 dependiendo si la célula centra estaba viva o muerta respectivamente. Principalmente destacamos este trabajo porque comparte la intuición de utilizar operaciones matriciales para provocar un paso de simulación. La principal diferencia de este trabajo con el propuesto, es que en el trabajo de Shalyapina y Gromov los autores aplican una convolución a cada celda del autómata celular, utilizando librerías de alto nivel como *TensorFlow*, y no se considero la posibilidad de utilizar *Tensor Cores*. En cambio, en este trabajo se emplea un algoritmo de GPU diseñado tal que cada operación *Tensor Core* va a procesar múltiples celdas de una región del autómata celular, en paralelo.

Otros trabajos mas tradicionales son las implementaciones eficientes de autómatas celulares utilizando el modelo de computo clásico de la GPU junto con CPUs multi-core [20,21], como por ultimo mencionar que frecuentemente se proponen autómatas celulares para simular nuevos fenómenos científicos [22].

3. Conceptos Esenciales

Nota: La sección Apéndice˜A ofrece una explicación sobre los conceptos básicos de la programación de GPUs.

En esta Sección se van a definir dos conceptos importantes para el resto del articulo, i) *Tensor cores* y ii) Autómatas celulares.

3.1. Tensor Cores

Los *Tensor Cores* son un tipo de ASIC (*Application Specific Integrated Circuit*) que existe dentro de las GPUs de NVIDIA, el cual se especializa solo en un tipo de operación, denominada MMA (*Matrix Multiply-Accumulate*) que es la multiplicación y acumulación de matrices, pudiendo realizarlas en un ciclo de GPU. Actualmente las GPUs pueden tener cientos de *Tensor Cores*, y cada *Tensor Core* trabaja con cuatro matrices de tamaño reducido $r \times k \times q$ que operan de la forma

$$D_{r\times q} = A_{r\times k} \times B_{k\times q} + C_{r\times q} \tag{1}$$

Para el programador, se pueden escoger dimensiones donde $r \times k$, $k \times q$ y $r \times q = 256$. Sin embargo, a nivel de *hardware*, se realizarán operaciones sobre matrices de 4×4 (*Volta* y *Turing*) u 8×4 (*Ampere*). Para efectos de análisis, es importante mantener la representación general $r \times k \times q$ ya que en futuras generaciones

de GPUs estas dimensiones pueden de 4×4 u 8×4 pueden aumentar. Una restricción que tienen los tensor cores, es que la matrices A y B tienen una precisión numerica menor que las matrices de acumulacion C y D. Un uso común es generar las matrices A y B en punto flotante de 16 bits (FP16), o 19 bits (conocido como TF32 en *Ampere*) y las matrices C y D en punto flotante de 32 bits (FP32) o 64 bits (FP64). Una funcionalidad útil que nos permiten hacer los Tensor Cores, es que podemos reutilizar la misma matriz C como la matriz D

$$C_{r\times q} = A_{r\times k} \times B_{k\times q} + C_{r\times q} \tag{2}$$

lo que nos permite tener un mejor rendimiento de memoria, evitando traspasos innecesarios o duplicación de datos. En el algoritmo 1 , se ilustra el procedimiento típico que se realiza para programar una multiplicación de matrices con *Tensor Cores*. Primero se crean fragmentos los cuales son submatrices en cache con la que operan los *Tensor Cores*, posteriormente se cargan los valores de las matrices a los fragmentos. A continuación se procede a hacer la operación de multiplicación y suma de matrices, y finalmente se almacena la información del resultado de vuelta en memoria, listo para poder ser manipulado por el resto de la rutina.

Algorithm 1: Multiplicación de matrices utilizando Tensor Cores

Entrada: Matrices A, B, C y D
Salida : Matriz D con el resultado
fragment Af, Bf, Cf, Df;
$Af \leftarrow load\_matrix\_sync(A)$;
$Bf \leftarrow load\_matrix\_sync(B)$;
$Cf \leftarrow load\_matrix\_sync(C)$;
$Df \leftarrow mma\_sync(Af, Bf, Cf)$;
$D \leftarrow store\_matrix\_sync(Df)$;
$return(D)$;

Es relevante mencionar que un *warp* completo se encarga de la ejecución de un *Tensor Core*, por lo tanto en un bloque CUDA que tiene hasta 32 *warps*, pueden haber hasta 32 operaciones *Tensor Core* ocurriendo concurrentemente. Si luego consideramos que un kernel puede ejecutar un *grid* con miles de bloques CUDA, entonces se tendrá una solución GPU con una gran cantidad de operaciones *Tensor Core* concurrentes.

3.2. Autómata Celular

El autómata celular es un modelo matemático de un sistema dinámico discreto, cuya evolución para el paso de tiempo siguiente es guiada por una función de transición que se aplica a cada celda, considerando los estados de los vecinos. Este tipo de modelo de simulación ha sido utilizado ampliamente desde los 50, hasta la época actual, para simular distintos fenómenos complejos, cómo la resistencia de un diseño arquitectónico [7], bioinformática [9,10,11] , criptografía [13] , propagaciónón de incendios [16], entre otros [14,15]. Los componentes que definen un automata celular son:

- Posee un espacio estructurado o grilla que tiene una dimensión con $d \in \mathbb{Z}^+$. A cada una de las celdas del espacio se le denomina célula.
- Cada célula posee un valor dentro de un numero finito de estados.
- Cada célula posee una vecindad, la cual se define como un conjunto finito de células cercanas a la misma.
- Para el cálculo del siguiente estado de una célula se define una función de transición que tiene como argumentos el valor de dicha célula y la vecindad, devolviendo el siguiente valor de esta.
- Para el cálculo del siguiente estado del Autómata Celular se debe aplicar la función de transición a todas las células, obteniendo un nuevo valor para cada una de ellas.
- En el caso de ser finitos, se considera una condición de borde.

Típicamente, un automata celular comienza con una condición inicial arbitraria, y a medida que evoluciona la simulación, pueden emerger fenómenos de mayor complejidad, no posibles de contemplar con anterioridad de forma analítica. Dentro de las vecindades de los autómatas celulares de dos dimensiones, las más comunes son la de von Neumann y la de Moore. La primera hace referencia a las células que están inmediatamente arriba, abajo, izquierda y derecha de la célula escogida. La segunda considera las ocho células que rodean a la célula central, ambas vecindades se pueden ver de manera gráfica en la Figura 1.

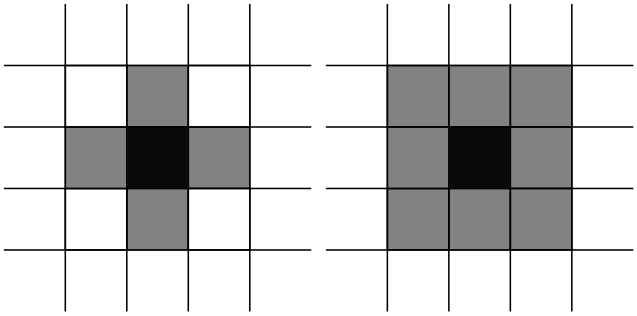


Figura 1: Vecindad de von Neumann y de Moore respectivamente. En gris se muestran las células del vecindario de la célula elegida (negra).

En cuanto a las condiciones de borde, las mas usadas son

- Frontera abierta: Se considera que todas las células fuera del espacio definido en la simulación toman un estado fijo e inmutable.
- Frontera periódica: Se considera que las células de los extremos se conectan con su lado opuesto. Por ejemplo, para el caso de una dimensión se visualiza como un cinturón y para las dos dimensiones su superficie se visualiza como un toroide.
- Frontera reflectora: Se considera que las células del exterior toman los valores del interior como si de un espejo se tratase.

- Sin frontera: Se utiliza memoria dinámica que varía según el tamaño actual del autómata, además de técnicas para ocupar menos espacio, pero finalmente existe un límite que no se podrá superar.

Autómata Celular de Conway En el año 1970 el matemático británico John Horton Conway diseña un tipo de autómata celular llamado Juego de la Vida [3]. Este consiste en un autómata celular de dos dimensiones, con dos estados denominados "vivo/"muertoz que utiliza la vecindad de Moore. Para efectos de este trabajo usaremos una frontera abierta con valor 0 ("muerto"). La función de transición se codifica basado en las siguientes reglas:

- Una célula muerta cobra vida si tiene exactamente 3 células vecinas vivas.
- Una célula viva con 2 o 3 células vecinas vivas sigue viva, en otro caso muere ya sea por falta de células para reproducirse o sobrepoblación.

En las siguientes dos secciones, se presenta la principal contribución de este trabajo, que es la propuesta de un nuevo método de simulación de autómata celular y el análisis del costo computacional.

4. Método propuesto

En esta sección se propone un método de simulación de autómata celular basado en *Tensor Cores*, utilizando como caso de estudio el juego de la vida. La parte más importante para poder utilizar los *Tensor Cores* para esta operación, es la de transformar nuestro problema a un problema que se exprese como una operación MMA, por lo que definiremos el autómata celular como una matriz, en donde cada uno de los valores de las células corresponden a un valor en la matriz,

$$A_{n\times n} = \begin{pmatrix} a_{1,1} & a_{2,1} & \dots & a_{1,n} \\ a_{2,1} & \ddots & & \vdots \\ \vdots & & \ddots & a_{n-1,n} \\ a_{n,1} & \dots & a_{n,n-1} & a_{n,n} \end{pmatrix} \tag{3}$$

además utilizaremos una matriz tridiagonal con 1s

$$T_{n\times n} = \begin{pmatrix} 1 & 1 & & 0 \\ 1 & \ddots & \ddots & \\ & \ddots & \ddots & 1 \\ 0 & & 1 & 1 \end{pmatrix} \tag{4}$$

para realizar las operaciones sobre las vecindades de cada celda.

El procedimiento consiste en multiplicar por la izquierda y por la derecha la matriz A con la matriz T,

$$N_{n\times n} = (T_{n\times n} \times A_{n\times n}) \times T_{n\times n} \tag{5}$$

obteniendo la matriz N donde tenemos la cantidad de vecinos (en vecindad Moore) para cada una de las células del Autómata Celular. Cabe destacar que, si la célula central estaba viva, entonces el valor que contiene N, es el número de vecinos aumentado en 1. El resultado de N se debe a que al multiplicar por la tridiagonal por la izquierda se suman los tres valores contiguos de forma vertical. Ahora si hacemos la multiplicación de la tridiagonal por la derecha es la suma de los tres valores contiguos, pero de forma horizontal. Teniendo así que si hacemos primero una operación y después la otra lo que haremos es tener en la celda, la suma de los nueve valores que acompañan a una célula (el vecindario de Moore y la célula en si misma). Una vez obtenida esta matriz N podemos aplicar las reglas que definen al Juego de la vida.

En caso de querer utilizar la vecindad de von Neumann lo que debemos hacer es multiplicar la matriz del autómata celular por la derecha con la tridiagonal y después sumar el A multiplicado por tridiagonal por la izquierda

$$N_{n\times n} = T_{n\times n} \times A_{n\times n} + A_{n\times n} \times T_{n\times n} \tag{6}$$

Al realizar la suma de ambas multiplicaciones lo que obtenemos, como vimos anteriormente, es la suma de la cuenta vertical y la cuenta horizontal, por lo que obtendremos la vecindad de von Neumann, en caso de que la célula se encuentre viva el resultado en N tendrá la cuenta de los vecinos +2.

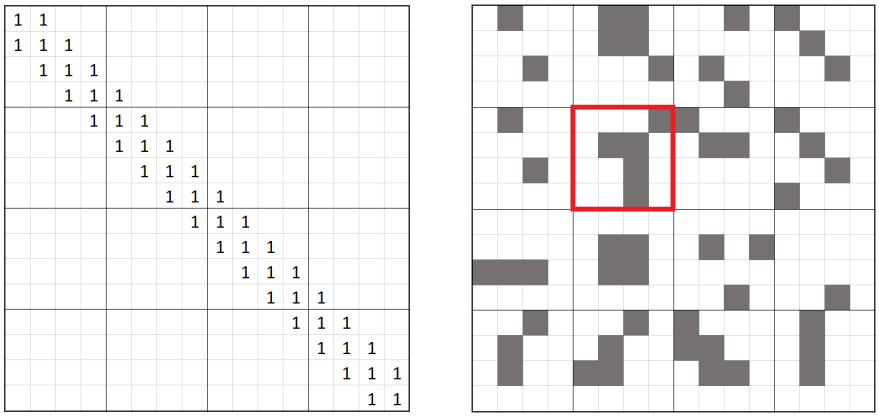
Un gran desafío que se presenta es que las matrices que pueden operar en los *Tensor Cores* tienen un tamaño reducido (actualmente matrices de hasta 4×4 o bien 8×4). Por lo que debemos dividir las matrices originales de $n \times n$ en submatrices, dejándonos una multiplicación de matrices por bloques. Esta división nos entrega una oportunidad ya que nos permite realizar menos trabajo al ahorrar cálculo en las zonas nulas de la matriz tridiagonal (matriz *sparse*). De esta forma, el proceso se realiza sobre 2 trios de productos matriciales, pero te tamaños significativamente menores que una de $n \times n$ en la práctica. En la siguiente subsección se presenta un ejemplo de paso de simulación utilizando el algoritmo propuesto.

4.1. Ejemplo de paso de Simulación con *Tensor Cores*

El siguiente ejemplo muestra cómo se codifica un paso de simulación para una submatriz 2D de 4×4 celdas de un autómata celular. En una arquitectura paralela con múltiples *Tensor Cores*, muchas submatrices se estarán procesando simultáneamente. En el modelo de programación de CUDA, estas submatrices se refieren como fragmentos.

Dado un autómata celular de $n \times n$, con $n = 16$, y *Tensor Cores* que trabajan con matrices de 4×4, se tiene el siguiente escenario ilustrado en la Figura 2.

Al observar la matriz tridiagonal en Figura 2a, notamos que a priori tenemos muchas submatrices que contienen solo $0's$, por lo que podemos descartarlas al momento de realizar las multiplicaciones, ya que, no aportan al cálculo. Si extendemos el tamaño de esta matriz, notaremos que el ahorro de cálculo se vuelve cada vez más grande. Por ejemplo para una matriz de $n^2 = 256 \times 256$



(a) Matriz Tridiagonal. (b) Autómata Celular.

Figura 2: Matriz Tridiagonal y Autómata Celular utilizados como ejemplo. Para (a) las celdas vacías corresponden a ceros. Para (b) las celdas blancas corresponden a células muertas y las grises a células vivas.

(64×64 submatrices de tamaño 4×4) solo se utilizaran $64+63+63$ submatrices de la matriz diagonal (3 submatrices por cada fila de submatrices del automata celular, excepto que en la primera y última fila solo son 2), lo que corresponde aproximadamente a un 5 % del total. Generalizando, para un n muy grande, el trabajo a realizar será de $O(n)$ en comparación a un $O(n^2)$ manejando la matriz completa.

Para el ejemplo de simulación tomaremos en cuenta la matriz destacada en rojo de la Figura 2b, correspondiente a A_{22}, debemos primero interpretarlo como un matriz numérica, donde las células vivas corresponderán a 1 y las células muertas a 0 obteniendo la matriz

$$A_{22} = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{7}$$

Utilizaremos la estrategia de multiplicación de matrices por bloque, por lo que para la primera operación $T \times A$, multiplicaremos los bloques de la misma fila de T por la columna de A que corresponden a A_{22}, en este caso no es necesario operar T_{14} ya que es una matriz nula

$$T_{21} = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}, T_{22} = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix}, T_{23} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \tag{8}$$

.

$$A_{12} = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}, A_{22} = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}, A_{23} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \tag{9}$$

Obteniendo la matriz transitoria,

$$A'_{22} = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 1 & 2 & 1 \\ 0 & 1 & 3 & 0 \\ 0 & 0 & 2 & 0 \end{pmatrix} \tag{10}$$

donde podemos notar que se obtiene como resultado una submatriz en donde cada uno de los elementos indica el número de vecinos verticales (uno arriba, uno abajo y si mismo). Ahora procederemos al siguiente paso, que es sincronizar las submatrices para así asegurarse de obtener el resultado transitorio en todos los bloques. Luego procedemos a realizar la segunda operación matricial, que supone multiplicar la matriz transitoria A' con la matriz diagonal tridiagonal por la derecha, que en este caso corresponden a

$$A'_{12} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}, A'_{22} = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 1 & 2 & 1 \\ 0 & 1 & 3 & 0 \\ 0 & 0 & 2 & 0 \end{pmatrix}, A'_{23} = \begin{pmatrix} 1 & 1 & 2 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \tag{11}$$

$$T_{21} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}, T_{22} = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix}, T_{23} = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}. \tag{12}$$

Así finalmente

$$N_{22} = \begin{pmatrix} 1 & 2 & 3 & 3 \\ 1 & 3 & 4 & 4 \\ 1 & 4 & 4 & 3 \\ 0 & 2 & 2 & 2 \end{pmatrix} \tag{13}$$

obteniendo como resultado N_{22}. En donde se sumaron los vecinos horizontales (izquierda, derecha y si mismo). Si tenemos en cuenta que en el paso anterior se sumaron los vecinos verticalmente, entonces la suma final cada elemento de N_{22} es la suma de la vecindad de Moore de una célula y sí misma. Con esta matriz obtenida podemos aplicar las reglas del autómata celular de Conway y calcular la iteración para el bloque, teniendo en cuenta que en caso de que la célula este viva, el elemento de la N tendrá sumado un 1 adicional.

5. Análisis del costo computacional

A continuación se presenta un análisis teórico del tiempo de ejecución entre tres implementaciones GPU a) algoritmo clásico usando memoria global y b) algoritmo avanzado con utilización del caché programable y c) algoritmo propuesto

utilizando *Tensor Cores*. Para analizar el costo computacional de cada algoritmo, utilizaremos un modelo de GPU simplificado, similar al modelo PRAM [18] pero con cantidades finitas para los *Cores* regulares, denotado como R_c, y *Tensor Cores*, denotado como R_{tc}, además de una memoria caché programable accesible por todos los hilos. Para simplificar el análisis, tanto la memoria global como la caché programable se considerarán global e ilimitadas, extendiendo el concepto PRAM a dos niveles de memoria (en la discusión se analiza que pasaría al considerar la memoria caché finita y distribuida en los bloques CUDA). En cuanto a costos, en este modelo el costo para leer/escribir en memoria es de C para todos los núcleos que lo hacen síncronamente, para leer/escribir en caché es c, con $c << C$. La operación MMA proveniente de un *Tensor Core* cuesta 1 unidad de tiempo. Cada *Tensor Core* puede hacer su MMA en paralelo con los otros, manteniendo el costo de 1 unidad de tiempo. Las dimensiones para un MMA son

$$D_{r\times q} = A_{r\times k} \times B_{k\times q} + C_{r\times q} \tag{14}$$

donde $r \times q \times k << n \times n$. Adicionalmente nuestro problema a resolver será un autómata celular de Conway cuadrado de tamaño $n \times n$. A continuación, se analiza el costo de los tres algoritmos para una posterior comparación.

5.1. Algoritmo #1: Utilizando Memoria global

Este método es la implementación más común de la literatura, y la más intuitiva para simular un autómata celular. Antes de calcular, necesitamos los datos, por lo que cada hilo necesita los datos de la célula correspondiente y del vecindario, por lo que debemos extraerlos desde la memoria global, lo que nos cuesta C por cada dato, siendo $9 \cdot C$ en total la lectura. Posteriormente con los datos obtenidos calcularemos la suma del vecindario para poder calcular el valor de la célula en la siguiente iteración, como resultado necesitaremos realizar 7 sumas. Con el resultado de la suma aplicaremos la regla del autómata celular de Conway, la cual nos cuesta 1 unidad de tiempo. Este procedimiento descrito debemos realizarlo para las $n \times n$ células, dejándonos

$$T_c(n) = \frac{n^2(7 + 9 \cdot C + 1)}{R_c} \tag{15}$$

5.2. Algoritmo #2: Utilizando Caché Programable

Este algoritmo toma la idea principal del Algoritmo #1, pero diferenciándose en la gestión de lecturas/escrituras en la memoria global utilizando la caché programable y obteniendo mejor rendimiento. En la literatura, este algoritmo se considera una versión avanzada y mucho más eficiente que la primera. Primero cada hilo realiza la lectura de la célula correspondiente desde la memoria global para ser guardada en la memoria caché programable, costando esta operación C y c respectivamente. Con este paso tendremos cargado el autómata celular en una memoria más rápida, optimizando tiempos. Posteriormente procederemos

a leer los datos de los vecinos, que con este método están en memoria caché programable por lo que su costo es de $8 \cdot c$, realizamos las 7 sumas y realizamos la comparación (costo 1) con la regla del autómata celular para finalmente guardar el resultado de vuelta a la memoria global (costo C), resultando

$$T_{cp}(n) = \frac{n^2(7 + 2 \cdot C + 8 \cdot c + 1)}{R_c} \tag{16}$$

5.3. Algoritmo #3: Utilizando *Tensor Cores* (método propuesto)

El método propuesto cambia la forma de abordar el problema con respecto a los dos algoritmos previos, redefiniendo la simulación como un cálculo matricial acelerada por *Tensor Cores*. Para cuando $n \times n > r \times q$ debemos subdividir el problema en $\frac{n^2}{r \cdot q}$ submatrices debido a la limitación de tamaño en los *Tensor Cores*, por cada fila o columna de submatrices solo se realizarán 3 multiplicaciones para ahorrarnos las submatrices nulas como vimos anteriormente. Como el autómata se encuentra alojado en memoria nos cuesta C llevarlo al *Tensor Core* (y otro costo C para guardar el resultado de regreso), donde operamos y obtenemos una matriz de transición, que es guardada en caché (c). Ahora se procede con la segunda operación, donde se inicia cargando desde caché (c) y se opera con 3 unidades de costo de tipo *Tensor Core*, obteniendo finalmente la matriz N. Finalmente, para la matriz N se utiliza 1 ciclo más para realizar una comparación según las reglas de transición del autómata y obtener el nuevo estado de la célula. El proceso recién descrito consiste en un paso de simulación completo en un autómata celular de $n \times n$, lo que tiene un costo total de

$$T_{tc}(n) = \frac{n^2}{(r \cdot q) \cdot R_{tc}} \left(2 \cdot 3 + 2 \cdot C + 2 \cdot c + 1 \right). \tag{17}$$

5.4. Speedup del Algoritmo #3 (propuesto)

Para poder operar los costos C y c en una misma ecuación, se propone una relación $C = \alpha \cdot c$, donde $\alpha > 1$ es el factor de costo de C en términos de c para las operaciones de lectura/escritura en memoria global y en caché, respectivamente. El *Speedup* teórico del algoritmo propuesto con respecto al Algoritmo #1 es

$$S_c = \frac{T_c(n)}{T_{tc}(n)} = \frac{\frac{n^2}{R_c}(7 + 9\alpha c + 1)}{\frac{n^2}{(r \cdot q) \cdot R_{tc}}(2 \cdot 3 + 2\alpha c + 2c + 1)} \tag{18}$$

$$S_c = \frac{(r \cdot q) \cdot R_{tc}}{R_c} \cdot \left(\frac{9\alpha c + 8}{2\alpha c + 2c + 7} \right) \tag{19}$$

y el *Speedup* teorico con respecto al Algoritmo #2 es

$$S_{cp} = \frac{T_{cp}(n)}{T_{tc}(n)} = \frac{\frac{n^2}{R_c}(7 + 2\alpha c + 8c + 1)}{\frac{n^2}{(r \cdot q) \cdot R_{tc}}(2 \cdot 3 + 2\alpha c + 2c + 1)} \tag{20}$$

$$S_{cp} = \frac{(r \cdot q) \cdot R_{tc}}{R_c} \cdot \left(\frac{2\alpha c + 8c + 8}{2\alpha c + 2c + 7} \right) \tag{21}$$

Un escenario representativo de la arquitectura de un GPU es uno donde la diferencia de acceder a memoria local vs global es muy significativa $c << C$. Para efectos de análisis esto se analizará aplicando limites $\alpha \to \infty$ solo en el cuociente que involucra al algoritmo los costos de las operaciones en cada celda. El otro factor, se considera posteriormente. Aplicando los límites, y considerando $c > 1$, tenemos que el factor de mejora en el costo de simular una celda se aproxima a

$$\lim_{\alpha \to \infty} \frac{9\alpha c + 8}{2\alpha c + 2c + 7} \sim 4{,}5 \tag{22}$$

$$\lim_{\alpha \to \infty} \frac{2\alpha c + 8c + 8}{2\alpha c + 2c + 7} \sim 1 \tag{23}$$

para los *Speedups* sobre los Algoritmos #1 y #2, respectivamente. Con estos factores, ahora el *Speedup* final dependerá del tamaño de matriz del *Tensor Core* $(r \times q)$, la cantidad de *Tensor Cores* (R_{tc}) y la cantidad de CUDA *Cores* (R_c).

$$S_c \sim 4{,}5 \cdot \frac{(r \cdot q) \cdot R_{tc}}{R_c} \tag{24}$$

$$S_{cp} \sim 1 \cdot \frac{(r \cdot q) \cdot R_{tc}}{R_c} \tag{25}$$

En la siguiente sección se analizara el *speedup* obtenido asignando valores a los parámetros en base a las GPUs actuales.

6. Discusión

Considerando los resultados obtenidos en la sección anterior, con las ecuaciones (24) y (25) podemos dilucidar el rendimiento teórico que pueden tener las actuales generaciones del hardware de NVIDIA, ya que conocemos sus parámetros R_c, R_tc y $r \times q$. En el caso de las arquitecturas *Volta* y *Turing*, la relación entre los *Tensor Cores* y CUDA *Cores* es de 1 es a 8 respectivamente, además el tamaño de $r \times q$ es de 4×4, por lo que

$$S_c \sim 4{,}5 \cdot \frac{(4 \cdot 4) \cdot 1}{8} \sim 9 \tag{26}$$

$$S_{cp} \sim 1 \cdot \frac{(4 \cdot 4) \cdot 1}{8} \sim 2 \tag{27}$$

Para el caso de la arquitectura *Ampere* la relación entre los *Tensor Cores* y CUDA *Cores* es de 1 es a 16 y el tamaño de $r \times q$ es de 8×4, entregando *Speedups* de

$$S_c \sim 4{,}5 \cdot \frac{(8 \cdot 4) \cdot 1}{16} \sim 9 \tag{28}$$

$$S_{cp} \sim 1 \cdot \frac{(8 \cdot 4) \cdot 1}{16} \sim 2 \tag{29}$$

Por este motivo podemos decir que con el método propuesto podemos obtener un rendimiento extra, respecto de una implementación básica en GPU de $9\times$ y de $2\times$ para el caso de una implementación más avanzada. Al aplicar $\alpha \to \infty$, se consideró el escenario mas adverso para el método propuesto. En la práctica, se suele tener entre uno o dos ordenes de magnitud entre la velocidad de acceso a memoria cache y global, es decir entre $\alpha \sim 10$ y $\alpha \sim 100$. En esos casos, el *Speedup* para el método propuesto se favorece aún mas.

Sobre una eventual implementación en GPU, un aspecto importante a considerar es que la memoria caché programable es distribuida en los bloques CUDA, y en el modelo teórico esta se consideró compartida por todos e ilimitada, por lo tanto, habría que considerar un costo de acceso a memoria global para leer los vecinos de las fronteras de cada región. Esto significaría leer vecinos fuera del caché en los costados y esquinas. Para las esquinas existen al menos dos maneras de solucionar el problema. La primera solución es crear una barrera de sincronización global para que los bloques CUDA terminen de realizar la primera operación MMA, así con la operación 100 % realizada podemos guardarla en memoria global para el intercambio de información entre todos los bloques CUDA, teniendo la información completa. La segunda opción corresponde a que cada *warp* asociado a una submatriz de la frontera haga un acceso a la submatriz que esta fuera de la región de caché, y sumado a esto, utilizar 4 *threads* en paralelo para leer los valores singulares de las esquinas fuera de caché. Con esto se obtienen todos los datos faltantes sin necesidad de sincronizar entre las dos operaciones MMA.

Si bien manejar las fronteras implicaría un aumento en el costo de los algoritmos, estos no serían tan significativos mientras el área de las regiones de caché GPU contengan varias submatrices y provoque una diferencia importante entre el área y perímetro (frontera) de la región de cache. Es importante notar también que el costo de las fronteras se debería incorporar no solo a la solución propuesta, sino que también al algoritmo con caché programable con el que se comparó, lo que al final podría resultar en un *Speedup* similar al obtenido aquí. De todas formas asumir una caché programable de gran tamaño y global no está muy alejado de la realidad, ya que la última versión de la arquitectura de NVIDIA (*Ampere*), contempla un caché global de 40MB, frente a los 9MB de la revisión anterior. Además, recientemente AMD anuncio GPUs con 128MB de caché, lo cual va a presentar nuevas oportunidades en el diseño de algoritmos en GPU.

7. Conclusiones

Con el lanzamiento de los *Tensor Cores* como aceleradores del rendimiento de las redes neuronales, NVIDIA puso a disposición un nuevo hardware para investigar y estudiar su uso en otras aplicaciones distintas a Deep Learning. En este trabajo decidimos estudiar el cómo usar los *Tensor Cores* para aumentar el rendimiento en simulaciones de autómatas celulares, que son procesos computacionales que utilizan la información de los vecinos más cercanos, como en el caso del autómata celular de Conway conocido como *Game of Life*. Para poder aprovechar este rendimiento y debido a que los *Tensor Cores* trabajan con operaciones de multiplicación y acumulación de matrices, transformamos el paso de simulación en muchos productos de submatrices que ocurren en paralelo, entre fragmentos del autómata celular y fragmentos de la matriz tridiagonal. De esta forma, los *Tensor Cores* se encargan de realizar estas operaciones en paralelo, produciendo un *Speedup* teórico de hasta 9× con respecto a una implementación básica de GPU y hasta 2× con respecto a una implementación avanzada que usa caché programable. Este resultado teórico es una gran motivación para continuar esta investigación por el lado experimental, y estudiar la forma de lograr la implementación adecuada en GPU que pueda manifestar estos beneficios observados en la teoría. El trabajo futuro trata justamente de esto. Además, es relevante mencionar que el juego de la vida fue un caso de estudio particular, y la técnica puede ser aplicada a cualquier autómata celular que en su función de transición necesite contabilizar sus vecinos mas cercanos.

Un aspecto a considerar cuando utilizamos *Tensor Cores*, es que el tipo de dato que utilicemos juega un importante papel en cuanto al *Speedup* del algoritmo. De momento hemos realizado una implementación que está utilizando datos de tipo FP16 (*Floating Point* de 16 bits) y en donde el algoritmo se comporta marginalmente mejor que la solución clásica de GPU, por lo que planeamos realizar una versión del algoritmo utilizando tipos de datos mas veloces, como INT4 (entero de 4 bits) o binario (1 bit), estos últimos disponibles en las arquitecturas *Turing* y *Ampere*.

Por ultimo, se identifico la posibilidad de superponer las submatrices para evitar las operaciones que corresponden a la frontera, lo que podría aumentar el rendimiento de la solución propuesta.

Referencias

1. NVIDIA Tensor Cores, https://www.nvidia.com/es-la/data-center/tensor-cores/. Last accessed 10 Oct 2020
2. Shalyapina N.A., Gromov M.L.: “Lifeïn Tensor: Implementing Cellular Automata on Graphics Adapters. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 3, 2019. pp. 217-228. DOI: 10.15514/ISPRAS-2019-31(3)-17.
3. Gardner M: The Fantastic Combinations of John Conway’s New Solitaire Game “Life”. Scientific American, vol. 223, no 4, 1970, pp. 120–123.
4. S. Markidis, S. W. D. Chien, E. Laure, I. B. Peng and J. S. Vetter: “NVIDIA Tensor Core Programmability, Performance & Precision,”2018 IEEE International Parallel

and Distributed Processing Symposium Workshops (IPDPSW), Vancouver, BC, 2018, pp. 522-531, doi: 10.1109/IPDPSW.2018.00091.
5. Abdul Dakkak, Cheng Li, Jinjun Xiong, Isaac Gelado, and Wen-mei Hwu: Accelerating reduction and scan using tensor core units. In Proceedings of the ACM International Conference on Supercomputing (ICS '19). Association for Computing Machinery, New York, NY, USA, 2019, pp. 46–57. doi: 10.1145/3330345.3331057
6. Martineau Matt, A Atkinson Patrick et al: Benchmarking the NVIDIA V100 GPU and Tensor Cores, Euro-Par 2018: Parallel Processing Workshops, 2019 Springer International Publishing, Cham pp. 444-455, doi: 10.1007/978-3-030-10549-5-35
7. Krawczyc, Robert: Architectural Interpretation of Cellular Automata. Generative Art 2002. Chicago, U.S.A. 2002.
8. Norman P Jouppi et al.: In-datacenter performance analysis of a tensor processing unit. In Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on. IEEE, 1–12.
9. Lahoz-Beltrá, Rafael. Bioinformática: simulación, vida artificial e inteligencia artificial. Diaz de Santos. Madrid, España. 2004.
10. Segura, Antinoo. La Bioinformática: una ciencia de riesgo. Revista Omnis Cellula. Cataluña, España. 2007.
11. Victor, Jonathan David: What can Automaton Theory tell us about the brain. Elsevier Science Publishers. Amsterdam, Holanda. 1990.
12. Jia, Zhe & Maggioni, Marco & Smith, Jeffrey & Scarpazza, Daniele: Dissecting the NVidia Turing T4 GPU via Microbenchmarking. 2019.
13. Wolfram, Stephen: Cryptography with Cellular Automata. SpringerVerlag. Heidelberg, Alemania. 1986
14. Hernández Encinas, L. et al.: Aplicaciones de los autómatas celulares a la generación de bits. Bol. Soc. Esp. Mat. Apl. No. 21, 65-87. Salamanca, España. 2002.
15. Reyes David: Descripción y Aplicaciones de los Autómatas Celulares, España. 2011
16. Peredo, Marco J. y Ramallo, Ramiro: Aplicación de Autómatas Celulares a Simulación Básica de Incendios Forestales. Acta Nova Vol. 2 No. 3. Cochabamba, Bolivia. 2003.
17. R. Carrasco, R. Vega, C. A. Navarro: Analyzing gpu tensor core potential for fast reductions, in: 2018 37th International Conference of the Chilean Computer Science Society (SCCC), 2018, pp. 1–6. doi:10.1109/SCCC.2018.8705253.
18. SCHEERER, Dieter: On the physical design of PRAMs. The Com, 1993.
19. Cristobál A. Navarro, Felipe A. Quezada, Nancy Hitschfeld, Raimundo Vega, Benjamin Bustos: Efficient GPU thread mapping on embedded 2D fractals, Future Generation Computer Systems, Volume 113, 2020, pp. 158-169, doi: 10.1016/j.future.2020.07.006.
20. Guan Qingfeng, Shi Xuan, Huang Miaoqing, Lai Chenggang: A hybrid parallel cellular automata model for urban growth simulation over GPU/CPU heterogeneous architectures, International Journal of Geographical Information Science 2016, doi: 10.1080/13658816.2015.1039538
21. S. Rybacki, J. Himmelspach and A. M. Uhrmacher: Experiments with Single Core, Multi-core, and GPU Based Computation of Cellular Automata, 2009 First International Conference on Advances in System Simulation, Porto, 2009, pp. 62-67, doi: 10.1109/SIMUL.2009.36.
22. Gobron, S., Devillard, F. & Heit, B.: Retina simulation using cellular automata and GPU programming. Machine Vision and Applications 18, 331–342 (2007). doi: 10.1007/s00138-006-0065-8

23. C. A. Navarro, R. Carrasco, R. J. Barrientos, J. A. Riquelme and R. Vega, "GPU Tensor Cores for Fast Arithmetic Reductions,ïn IEEE Transactions on Parallel and Distributed Systems, vol. 32, no. 1, pp. 72-84, 1 Jan. 2021, doi: 10.1109/TPDS.2020.3011893.

Apéndice A. Programación en GPU

La GPU (*Graphics Processing Unit*), es una unidad de coprocesamiento que complementa a la CPU (*Central Processing Unit*). En términos simples, la GPU es lo que se usa principalmente para aplicaciones de gráficos, ya sea soporte de video general o juegos. Es realmente muy rápida para hacer cálculos matemáticos, esto se logra mediante lo que se denomina multiproceso. La GPU tiene del orden de miles de pequeñas unidades de procesamiento (hilos) que trabajan en diferentes datos simultáneamente, lo que proporciona un alto rendimiento que se necesita para la representación de gráficos en tiempo real. Al ser en esencia las GPUs tan rápidas, es que se comenzaron a utilizar en distintos tipos de aplicaciones, las que no estaban relacionadas con los gráficos, aprovechando el gran entorno paralelizado con el que se dispone. Así es como desde el año 2006 nacen distintos *frameworks* de desarrollo para fomentar el uso de las GPUs con un propósito más general (GPGPU). Algunos de los *frameworks* más populares son OpenCL o CUDA.

Apéndice A.1. Arquitectura de GPU (Chip y SM).

La GPU se compone principalmente de múltiples SMs (*Streaming Multiprocessor*), en donde podemos encontrar todos los núcleos dedicados a realizar cálculos, además tenemos memorias y registros que nos apoyan al momento de la ejecución, también se visualizan unidades dedicadas a la coordinación y ejecución de instrucciones de la GPU.

En la Figura 3 podemos ver un SM perteneciente a la arquitectura Ampere de NVIDIA, lanzada en el año 2020. Aquí podemos ver que además de los núcleos dedicados a los cálculos de punto flotante o entero, también existen otros, como por ejemplo los *Tensor Cores* y *RT Cores* (*Ray Tracing Cores*) que comenzaron a aparecer desde hace ya algunos años en las nuevas arquitecturas de NVIDIA.

Apéndice A.2. Modelo de programación en GPU.

El flujo de trabajo que se emplea en la GPU normalmente comienza en CPU, donde se obtienen y preparan los datos con los que se van a trabajar. Desde CPU se coordina y reserva la memoria de GPU necesaria, posteriormente se realizar el traspaso de los datos entre la RAM y la VRAM (memoria de la GPU). Una vez este todo preparado para realizar el cálculo en la GPU, la CPU lanza el *kernel*, que es una función que será ejecutada en GPU, este *kernel* recibe parámetros como una función normal, solo que los datos entregados deben estar en memoria de GPU. Adicionalmente a estos parámetros, el *kernel* recibe el número de hilos que se ocuparán.

Figura 3: SM de la arquitectura Ampere de NVIDIA.

Los hilos de programación de la GPU poseen una estructura que nos permite utilizar ciertas funcionalidades o limitaciones dependiendo del nivel en donde nos encontremos, por lo que es importante tomarlo en cuenta. Los hilos (*threads* en inglés) se agrupan en una estructura denominada bloque (*block* en inglés) los cuales a su vez pertenecen a una estructura denominada grilla (*grid* en inglés), tal como se muestra en la Figura 4. Adicionalmente existe una estructura que se encuentra en medio entre los hilos y los bloques denominada *warp*, esta estructura está más afianzada a un concepto de *hardware* que de programación, esta estructura viene dada por un SM y corresponde a los hilos de un bloque que efectivamente están siendo ejecutados en paralelo.

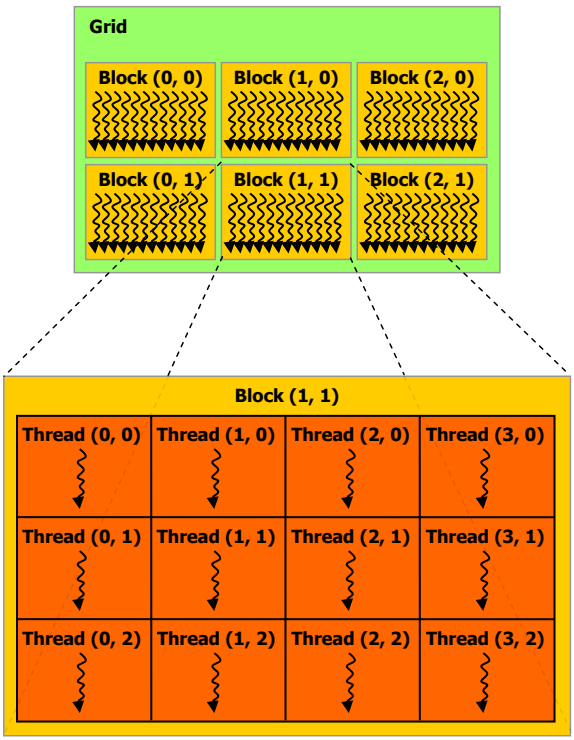


Figura 4: Jerarquía presente en el lenguaje de programación CUDA para el tratamiento de los hilos.

Apéndice A.3. Jerarquía de Memoria.

Adicionalmente y en conjunto con la estructuración de los hilos, existe una jerarquía de memoria en la GPU, las cuales tienen mayor o menor latencia dependiendo de los acotados que estén. Primero encontramos la denomiada *local memory*, la cual es una memoria privada que contiene cada uno de los hilos. Siguiendo en la jerarquía tenemos la *shared memory*, la cual es una memoria caché programable que es accesible por todos los hilos dentro de un mismo bloque. Por último, disponemos de la *global memory* la cual es accesible por todos los hilos, aquí es donde se almacenan los datos copiados desde la CPU.

Apéndice A.4. Ejemplo.

A continuación podemos encontrar un ejemplo programado en CUDA de SAXPY (*Single precision A X plus Y*), que corresponde a multiplicar un vector por un escalar y luego sumarle al resultado otro vector.

```
// Kernel que realiza la operacion SAXPY
__global__ void saxpy(int n, float a, float *x, float *y)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  y[i] = a*x[i] + y[i];
}

int main()
{

  ...

  // Declaracion de variables
  int N = ...;
  float *x, *y, *d_x, *d_y;

  // Asignacion de memoria CPU
  x = (float*)malloc(N*sizeof(float));
  y = (float*)malloc(N*sizeof(float));

  // Asignacion de memoria GPU
  cudaMalloc(&d_x, N*sizeof(float));
  cudaMalloc(&d_y, N*sizeof(float));

  // Traspaso de datos desde CPU a GPU
  cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);

  // Llamada al Kernel para realizar la multiplicacion de los
  // vectores en GPU
  saxpy<<<1, N>>>(N, 2.0f, d_x, d_y);

  // Traspaso de datos desde GPU a CPU
  cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);
}
```

Apéndice A.5. Consideraciones al programar en GPU

- Acceso a la memoria global (*coalesced*): Para realizar una lectura/escritura de datos eficiente en la *global memory*, lo que debemos hacer es realizarla de tal manera que los hilos de un *warp* accedan a la memoria de manera alineada, por ejemplo, que el hilo t_x acceda al dato A_y del arreglo, así el hilo t_{x+1} accederá al dato A_{y+1}, el hilo t_{x+2} al dato A_{y+2} y así sucesivamente para todos los hilos del *warp*.
- Divergencia de threads: Las instrucciones que procesa un *warp* se ejecutan al mismo tiempo, avanzando instrucción por instrucción. Por lo que si existe una bifurcación en el código, como sucede con una instrucción *if-else*, existe

la posibilidad de que el *warp* se divida, con hilos entrenando al *if* y los otros al *else*, esto produce que el warp pierda paralelismo, ya que por ejemplo, cuando se ejecute una instrucción del bloque *if* los hilos que entraron al *else* quedaran suspendidos perdiendo posibilidad de trabajo.

- Uso de la memoria caché programable (*shared memory*): Utilizar esta memoria es muy importante para generar mejoras de rendimiento, ya que, es muy rápida en comparación con la *global memory*, pero siempre se debe tener en consideración que esta memoria solo es accesible entre los hilos de un mismo bloque, por lo que se deben considerar las condiciones que se producen en los bordes.
- Mapeo eficiente de hilos: El mapeo de los hilos a los datos dependerá del problema, pero siempre es importante realizarlo tomando en cuenta las lecturas/escrituras que se realizaran en memoria.