Optimización de Modelo de Heisenberg para GPU

Facundo Molina Heredia

Directores: Orlando V. Billoni, Nicolás Wolovick

Proyecto final de grado Lic. en Ciencias de la Computación.

Facultad de Matemática, Astronomía, Física y Computación.

Argentina
14/12/2020

Resumen

El modelo de Heisenberg de espines clásicos se emplea en la mecánica estadística para estudiar las propiedades magnéticas de diversos sistemas materiales. En el trabajo de Billoni, 2016 se plantea una caracterización de este modelo para representar perovskitas ortorrómbicas y la propiedad de magnetización reversa que estas presentan. A lo largo de este trabajo se cuenta como se hizo una primera implementación usando un hilo en C/C++ para CPU basada en la simulación planteada por Billoni, 2016, originalmente en Fortran. El segundo paso fue agregar paralelismo en CPU, esto se hizo usando las bibliotecas de "OpenMP", 2020. Luego se pasó las funciones paralelizables a kernels de CUDA. En la primera implementación se mantuvo el esquema de una matriz actualizando celdas intercaladas siguiendo la estrategia del tablero de ajedrez. En una segunda optimización para GPU se dividió varias de las matrices usadas en 2 sub-matrices, cada una conteniendo todas la celdas de un color. También se llevó cabo una visualización gráfica 3D de los espines en base a coordenadas (x,y,z) usando OpenGL, la cual serviría para observar las orientaciones de los diferentes espines. Por ultimo se comentan diferentes lecciones aprendidas y conclusiones, entre ellas la mejora que se obtuve partiendo de la primera versión sin optimizaciones de CPU tardando mas de 1500 ns para actualizar una celda, llegando a una versión optimizada para GPU que tarda 0.18 ns para actualizar una celda.

Palabras Claves: Modelo de Heisenberg, GPU, CUDA, Red Black, OMP.

Summary

The Heisenberg model of classical spins is commonly used in statistical mechanics to characterize magnetic properties or phase transitions in material systems.Billoni, 2016 proposes an implementation of this model to represent orthorhombic perovskites and the phenomenon observed in them where the magnetization reverses at a certain temperature. In this report you will read about a first single-thread implementation using C/C++ for CPU based on the simulation presented by Billoni, 2016, which was originally in Fortran. The second step was to add parallelism to the CPU implementation, this was achieved using the "OpenMP", 2020 libraries. Then the functions that presented the most parallelization potencial were moved to CUDA kernels. In the first implementation, we used the same storage scheme where we have one matrix and update first one subnet and then the other, following the Red/Black strategy. For a second optimization for the GPU version the matrices used were split into 2 sub-matrices, each containing all the spins of one colour. A graphic implementation was also developed using the OpenGL, which serves to observe the direction of each spin. Finally we show some of the lessons learned and conclusions, among which we mention the improvement from the first unoptimized CPU version, which takes over 1500 ns to update one cell, to the GPU optimized version which can update a cell in 0.18 ns.

Key words:
Heisenberg Model, GPU, CUDA, Red Black, OMP.

Dedicatoria

Para mi familia y amigos, que me apoyaron durante este largo proceso.

Agradecimientos

A mis directores de proyecto Orlando y Nicolás por responder la incontable cantidad de mails y dudas.

Índice general

1 Introducción

Algunos sistemas magnéticos, cuando son enfriados en presencia de campos magnéticos de baja intensidad, muestran magnetización reversa. A altas temperaturas, la magnetización apunta en la dirección del campo, pero a medida que la temperatura baja, la magnetización se revierte, oponiéndose al campo magnético en un determinado rango de bajas temperaturas. Este fenómeno se ha observado especialmente en las perovskitas ortorrómbicas, como por ejemplo RMO_3, donde R es alguna tierra rara, por ejemplo, itrio, y M hierro, cromo o vanadio.

Para investigar correctamente algunos sistemas moleculares o atómicos y sus comportamientos, se deben llevar a cabo un gran número de operaciones numéricas para evaluar su dinámica microscópica en tiempos macroscópicos y además, en sistemas grandes y complejos. En particular para la simulación planteada en el trabajo de Billoni, 2016 se usa un Modelo de Heisenberg de espines clásicos, en el cual para la dinámica en tiempos microscópicos de los espines debemos tener en cuenta la interacción con los espines adyacentes. Para realizar dicha simulación se utiliza el Método de Monte Carlo. Esto nos deja con una cantidad de cálculos considerable en cada actualización. Dado que la actualización de la matriz consume la mayor cantidad de tiempo de cada avance de tiempo, aquí es donde más beneficio se puede sacar de una buena optimización.

Para lograr esto usaremos GPUs (Graphics Processing Units - Unidad de Procesamiento de Gráficos - Placas de Vídeo) cuya arquitectura las hace muy buenas para este tipo de cálculos. Dados los avances que se han logrado en el hardware de las GPUs, cada día hacen falta mejores algoritmos que saquen el mayor provecho del hardware. Dada la diferencia de velocidad entre los diferentes tipos de memorias que traen, si no se hace un correcto manejo de datos, se termina desperdiciando mucho poder de cálculo. Previamente se han escrito publicaciones abarcando este uso de las GPUs para diferentes Modelos de Heisenberg o Dinámica de Moléculas, con modelos de Heisenberg más simples, como por ejemplo M. Bernaschi, 2011, o con el muy conocido modelo de Ising en Tobias Preis y Schneider, 2009. Pero las implementaciones que se encontraron son de versiones desactualizadas de

CUDA.

El objetivo de este trabajo es mostrar el desempeño de código optimizado tanto para CPU y la ganancia de agregar unidades de procesamiento, como código optimizado para GPU y el uso de las diferentes memorias que estas contienen.

La organización de los capítulos es la siguiente: Capítulo 2 se da una breve descripción del modelo de Heisenberg descripto en el trabajo de Billoni, 2016 el cual se desea simular. En el Capítulo 3 hay una detallada descripción de las arquitecturas modernas de las GPUs de NVIDIA, sus memorias y la jerarquía de unidades de cómputo presentada en CUDA. Durante el Capítulo 4 se pasa a explicar las diferentes estrategias que se implementaron para sacar la mayor ventaja de las diferentes memorias o que se mencionan en otras publicaciones. El Capítulo 5 lleva una comparación entre los resultados de performance de los 2 equipos disponibles tanto en CPU como GPU y una breve descripción de las mediciones comparativas que se usaron. Finalmente en el Capítulo 6 analizaremos los resultados presentados en el Capítulo anterior y presentaremos algunas ideas para posibles extensiones si alguien quisiera retomar esta temática.

2 Modelo de Heisenberg

Usamos el modelado descripto por Billoni, 2016 el cual será brevemente explicado en esta sección. El modelo de Heisenberg es usado en la mecánica estadística para describir sistemas magnéticos. Uno de los usos es para estudiar de transición de fase en esta clase de sistemas. En este caso analizamos un sistema ferromagnético, en el cual se desea estudiar el fenómeno de reversión de la magnetización a temperaturas cercanas a cero. Este fenómeno se puede observar en perovskitas ortorrómbicas, como por ejemplo RMO_3, con R siendo alguna tierra rara, por ejemplo, itrio, y M hierro o cromo. Estos materiales presentan comportamiento de ferromagnetismo débil a temperaturas menores a la temperatura de Néel (T_N).

El ferromagnetismo débil (WFM - Weak Ferromagnetism) que se observa en estos materiales puede ser efecto de dos tipos de interacciones magnéticas distintas, intercambio antisimétrico o interacción de Dzyaloshinskii–Moriya (DM), y anisotropía magnetocristalina de un solo ión. En particular, para ortocromitas $RCrO_3$ y ortoferritas $RFeO_3$, el mayor efecto lo tienen las interacciones DM. Estas últimas fueron analizadas en aproximaciones de Campo Medio (Mean Fields - MF) como bien muestras N Dasari y Vidhyadhiraja, 2012, para explicar las curvas de enfriamiento del campo en policristalinos como $YFE_{1-x}Cr_xO_3$ para $0 \leq x \leq 1$. La dependencia de la magnetización en función de la temperatura se explica mediante la competencia de interacciones DM para enlaces Fe-O-Fe, Cr-O-Cr y Cr-O-Fe.

Para modelar el comportamiento magnético, podemos restringirnos a la red de iones de los metales de transición (TM), cuya estructura puede ser descripta como una red cúbica simple con alta precisión. Se usan direcciones (x,y,z) y una serie de rotaciones para simplificar la descripción del sistema de spines.

Para modelar la perovskita $RFe_{1-x}Cr_xO_3$, con R= Lu o Y usando un Hamiltoniano de Heisenberg con espines clásicos montados en una red cúbica con

$$N = (L \times L \times L) \tag{2.1}$$

$$\mathbf{H}=-\frac{1}{2}\sum_{<i,j>}[J_{ij}\vec{S}_i\cdot\vec{S}_j+\vec{D_{ij}}\cdot(\vec{S}_i\times\vec{S}_j)]-\sum_i K_i(S_i^x)^2-H\sum_i m_i S_i^z \tag{2.2}$$

donde $< ... >$ representa la suma sobre los vecinos adyacentes y $\vec{S}_i$ son vectores unitarios. $J_{i_j} < 0$ toma en cuenta las interacciones de súper intercambio y $\vec{D_{i_j}}$ la interacción anti-simétrica de Dzyalshinskii Moriya. H corresponde al campo externo aplicado y es expresado como, $H = B\mu_{Fe}/k_B$, donde B es el campo externo y $\mu_{Fe} = g\mu_B S_{Fe}$ con $g = 2$ el factor constante giromagnética, μ_B el magnetón de Bohr y $S_{Fe} = 5/2$ el spin total de los iones Fe^{3+} - equivalentemente $S_{Cr} = 3/2$ para el ion Cr^{3+}. Entonces $m_i = 1$ para el ion de Fe^{3+} y $m_i = S_{Cr}/S_{Fe} = 0{,}6$ para los iones de Cr^{3+}. Ambas interacciones, J_{ij} y $D_{ij} = |\vec{D_{ij}}|$, dependen del tipo de ion (Fe^{3+} o Cr^{3+}) que ocupa la posición i y j, entonces cada interacción puede tomar 3 valores diferentes. Supongamos que la posición 1 esta ocupada por Cr^{3+} y la posición 2 por Fe^{3+}, entonces los acoplamientos de interacción de súper intercambio son $J_{22} = 2S_{Fe}^2 J_{FeFe}/k_B, J_{12} = J_{21} = 2S_{Fe}S_{Cr}J_{FeCr}/k_B$ y $J_{11} = 2S_{Cr}^2 J_{CrCr}/k_B$ donde $J_{\alpha\beta}$- con $\alpha, \beta =$ Cr o Fe- son las integrales del intercambio y k_B es la constante de Boltzmann. En el caso de los módulos del vector DM tenemos $D_{22} = S_{Fe}^2 D_{FeFe}/k_B$; $-D_{12} = D_{21} = S_{Fe}S_{Cr}D_{FeCr}/k_B$ y $D_{11} = S_{Cr}^2 D_{CrCr}/k_B$. Las interacciones de sitio correspondientes a la anisotropía uniaxial son $K_1 = S_{Cr}K_{Cr}/k_B = K_2 = S_{Fe}K_{Fe}/k_B > 0$. El término anisotrópico asegura el orden AFM en la dirección x. Por simplicidad consideramos la misma anisotropía para los iones Cr^{3+} y Fe^{3+}.

Las interacciones DM surgen, en las perovskitas, por la inclinación del octaedro (Fe, Cr)O_6 Igor Solovyev y Terakura, 1996. Esta inclinación colectiva resulta en un alternamiento de las interacciones DM en la dirección z. Pero para este trabajo se asumió un modelo simplificado. En este modelo los vectores DM están orientados en la dirección y y alternados en todas las otras direcciones.

Para las simulaciones se usara el método de Monte Carlo con un algoritmo de Metrópolis y dinámicas de actualización de un solo spin. Se consideró una red cúbica de $N = 40 \times 40 \times 40$ sitios y condiciones de borde abiertas. Para simular $RFe_{1-x}Cr_xO_3$ cada sitio de la red es ocupado por un ión Cr^{3+} con probabilidad x y con probabilidad $(1-x)$ por un ion Fe^{3+}. Como todas las interacciones de súper intercambio que son anti-ferromagnéticas se puede dividir en dos sub-grillas A y B, cada una ordenada ferromagnéticamente y opuestas entre ellas. Gracias a la simulación MC podemos seguir el estado magnético de cada spin en la grilla, y calcular la magnetización de la sub-grilla,

$$\tilde{\mathbf{m}}_\alpha = \frac{1}{N} \sum_{\vec{S_i} \in \{\vec{S_\alpha}\}} \vec{S_i}, \tag{2.3}$$

donde $\{\vec{S_\alpha}\}$ con $\alpha = A, B$ es un conjunto de spines que pertenecen a la sub-grilla A o B. Usando esto podemos calcular la susceptibilidad como,

$$\mathbf{X}_\alpha = (\frac{N}{k_B T})(< m_\alpha^2 > - < m_\alpha >^2) \tag{2.4}$$

donde $< ... >$ es un promedio de termodinamico. De esta forma podemos obtener la magnetización y la susceptibilidad en función de la temperatura, comenzando a temperaturas altas y bajando en pasos de 10K. Para cada temperatura equilibramos el sistema usando 10^5 pasos de MC (MCS - Paso de Monte Carlo), tomando varias observaciones (e.j. magnetización, temperatura) cada 10^2 MCS. Por convención actualizamos la configuración magnética de cada spin en cada MCS. La temperatura crítica se puede obtener del pico de susceptibilidad en función del contenido de Cromo para $x = 0, 0.1, 0.2, ..., 1$. Como las interacciones DM son considerablemente más bajas que las de súper intercambio, pequeñas variaciones de estas interacciones no afectan sustancialmente las temperaturas de ordenamiento anti-ferromagnético.

3 Arquitecturas Paralelas Modernas

En esta sección se comentarán nociones básicas de arquitectura de CPU y GPU, para tener una mejor comprensión de los motivos que impulsaron diferentes decisiones a lo largo de las distintas implementaciones.

3.1. Arquitecturas de CPU

Los lenguajes de programación son traducidos a código de máquina de la arquitectura subyacente. Este código es cargado en una memoria especial para este propósito, decodificado, ejecutado y por último su resultado (si hubiera alguno) es guardado en los registros, más adelante se hablará de ellos. Todas estas tareas se llevan a cabo en el CPU (Central Processing Unit).

3.1.1. Jerarquía de Memoria

Con los avances de tecnología que se han logrado, hoy en día tenemos acceso a memorias que son muy rápidas pero muy caras. Por esto se ha recurrido a usar varios niveles de memoria, para tener pequeñas memorias muy rápidas usando de soporte a diferentes niveles de memorias cada vez más lentas pero también más grandes.

Registros

Los registros son los espacios de memoria más cercanos al procesador y vienen integrados en el mismo núcleo. El uso de estos está optimizado por el sistema ya que algunos tienen propósitos muy definidos, como guardar instrucciones, y otros son más generales. Como se puede observar en Fig. (3.1) , los registros son la memoria más pequeña, más rápida y más cara de la que se dispone. Estos van a la velocidad del procesador.

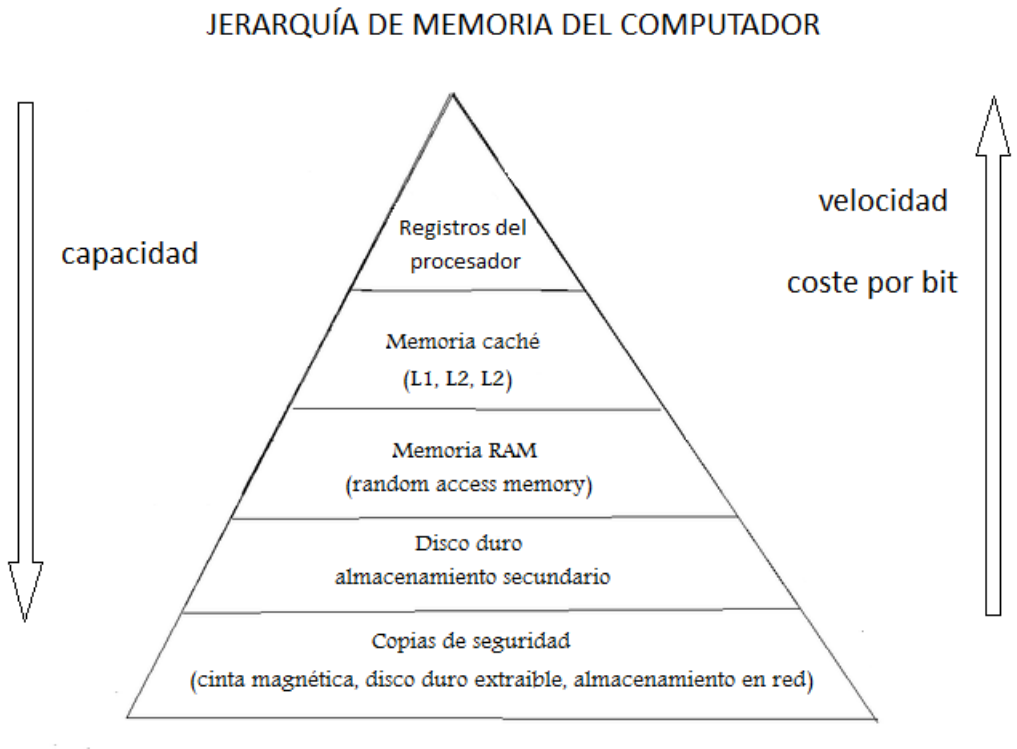


Figura 3.1: Esta pirámide representa la jerarquía de memoria usual en un sistema con CPU. Los 2 pisos de la base son los únicos que se consideran memoria persistente ya que todas las memorias superiores pierden su contenido cuando se quita la fuente de poder.

Cachés

Las memorias caché son más lentas que los registros pero su función principal es ser bancos de memoria intermedios entre los registros y la memoria principal (RAM) que suele ser muchísimo más lenta que estos. Usualmente hay 3 niveles de cachés, L1, L2 y L3, estas son cada una más grande en tamaño y aumentando también en latencia (tiempo que tardan en pedir un dato y recibirlo). Por ejemplo en un CPU moderno (AMD Ryzen 5 2600X"AMD Ryzen 5 - 2600X", 2020) la L1 tiene 576KB, L2 3MB y L3 16MB (casi 6 veces el tamaño de la memoria anterior).

- La L1 viene integrada con cada núcleo y tardan entre 4 y 10 ciclos en responder.
- La L2 aunque no suele estar incorporada en el núcleo, está muy cerca y sigue siendo exclusiva a cada núcleo. Como ya dijimos suele ser varias veces más grande que la L1, lo cual ayuda a

reducir los *cache misses* (esto se profundiza a continuación) pero a cambio introduce algo de latencia.

- Y por última la L3, como puede observarse en Fig. (3.2) es común a todos los núcleos del CPU y varias veces más grande que la L2, por los mismos motivos que L2 es mayor a L1. Además, varias generaciones atrás, L3 solía ser parte de las placas madres, no de los CPUs, esto se cambió con los años probando mejoría en tiempos de latencia al ser parte del CPU, pero todavía depende de la comunicación con la RAM.

La función principal de estas memorias es reducir los accesos a la RAM. Cuando alguna instrucción necesita usar un dato que no está entre los registros, va a buscarlo a la caché L1. Si este dato está en la L1, se trae a registros y se usa; si no está, se produce lo que se conoce como un *cache miss*, entonces se va a buscarlo a L2. Análogamente de L2 a L3, y de L3 a RAM.

Otros mecanismos que se usan para reducir los *cache misses*, es cuando se hace un pedido del dato i, también se traen los n datos adyacentes. Esta estrategia a probado ser de mucha utilidad ya que cuando se usa un dato, es muy posible que se vaya a usar alguno de los datos adyacentes y, de esta forma, ya está en caché para futuros llamados. Esto se denomina localidad espacial.

Un problema muy importante cuando se trabaja con varios niveles de cachés, es la *coherencia de datos*. Este problema consiste de mantener el mismo valor para el mismo dato en diferentes lugares de la jerarquía de memoria, ya que al tener cachés L2 y L1 que son privadas a cada núcleo, dos de estos podrían pedir el mismo dato y modificarlo. Hay diferentes formas de garantizar esta coherencia de datos en caché, y si no se usan mecanismos tanto en hardware como software, la corrección del algoritmo puede peligrar. Además, asegurar la coherencia de datos genera una sobrecarga en todo el subsistema de memoria.

RAM

La memoria RAM (Random Access Memory - Memoria de Acceso Aleatorio) o DRAM (D por Dynamic) es la que se conoce como memoria principal. Aunque en Fig. (3.1) está justo debajo de las cachés es mucho más lenta que estas y en comparación ronda las decenas de GBs, en comparación a los pocos MB de la L3. La función principal de la RAM es tener disponibles los datos que están siendo usados por los programas que están activos. Sirve tanto para escribir como para leer.

Al igual que las cachés es memoria volátil, esto significa que cuando el sistema se apaga, pierden su contenido. Como se puede ver en Fig [3.2] es la memoria volátil más alejada de los núcleos. Pero a diferencia de las cachés, donde los programadores *usualmente* no tienen control explícito, en la RAM si, cuando se crea una variable o un objeto, este es almacenado en RAM.

Discos

Hoy en día tenemos dos grandes clasificaciones: SSD (Solid State Drive - Disco de Estado Sólido) y HDD (Hard Disk Drive - Discos Duros). Ambos son memorias persistentes, y rondan desdes los cientos de GB hasta varios TB.

Los SSD son memorias flash que no incurren en penalización de *seek time*, porque son aleatorias. Esto reduce mucho la latencia y los tiempos de espera, pero a cambio de esto tienen una vida útil bastante corta comparados con los HDD, ya que tienen una cantidad limitada (aunque bastante elevada) de lecturas y escrituras.

Por otro lado los HDD son platos o discos que guardan los datos a través de mecanismos magnéticos y mecánicos. Son considerablemente más baratos que los SSD pero al tener varias partes mecánicas en movimiento si alguna se rompe o falla se puede perder todo el contenido del disco. Además la locación de memoria es un factor que afecta fuertemente a los tiempos de carga de disco a RAM, en los HDD de forma negativa y en los SSD no tiene efecto, dadas las lecturas en simultáneo previamente mencionadas.

3.1.2. Multi-Core

En los CPUs tenemos los núcleos o cores que son donde se efectúa el cálculo o ejecución de las instrucciones. Pero cuando se tiene que traer un dato desde RAM se pierden muchos ciclos de ejecución esperando, entonces se empezó a simular paralelismo ejecutando instrucciones de un proceso distinto mientras se espera que se traigan estos datos. Esto se conoce como *SMT* (Simultaneous Multi Threading). En este contexto se puede explicar el concepto de los "hilos" que son subprocesos. Cada núcleo ejecuta un hilo a la vez, pero gracias al *SMT*, se pueden tener varios hilos a la vez en un núcleo y, así, ir intercalando entre uno y otro si hubiera que esperar algún dato.

Hoy en día los CPUs cuentan con varios núcleos y estos, a su vez, ejecutan varios hilos a la vez, logrando un paralelismo que nos permite

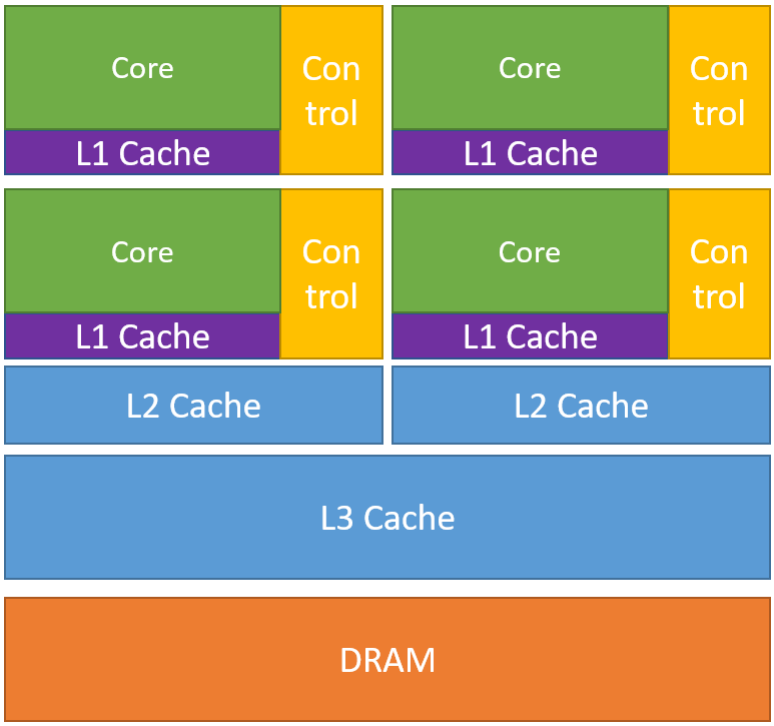


Figura 3.2: En esta figura se ve una simplificación de lo que seria un CPU con 4 núcleos, sus respectivas cachés.

tener varios programas corriendo en simultaneidad real y multiplexados. Por esto cada núcleo tiene su propio Program Counter, como se puede ver en Fig (3.2), para poder llevar ya sea un hilo de otro programa u otro hilo del mismo programa pero de manera totalmente paralela e independiente.

Con tantos niveles de paralelismo surgen nuevos problemas, como la sincronización entre hilos, dado que si no se coordinan se pueden leer o escribir datos *erróneos* y así terminar con comportamientos no deseados o peor, deadlocks[1].

3.2. Arquitecturas de GPUs

En el 2006 NVIDIA publicó el primer SDK de CUDA (Compute Unified Device Architecture), proveyendo un entorno que facilitó y promovió fuertemente la migración de muchos algoritmos numéricos de sus implementaciones en CPU a lo que se conoció como el GPGPU (General Propuse Graphics Processing Unit). CUDA consiste de una extensión al lenguaje C++, que permite interactuar con la GPU. Todas las generaciones de placas de video desde entonces se apegaron al

[1]Estado de un sistema donde tenemos diferentes partes en espera inactiva con dependencia circular.

paradigma concurrente SIMT (Single Intruction Multiple Thread) que yace entre SIMD (Single Instruction Multiple Data) y MIMD (Multiple Intruction Multiple Data).

3.2.1. Jerarquía de ejecución

A diferencia de los CPUs, que cuentan con una pequeña cantidad de núcleos (entre 6 y 10, o los CPUs para servidores con hasta 64), las placas de vídeo cuentan con cientos o hasta miles de núcleos. Estos están organizados en una particular jerarquía de ejecución que es clave para usar eficientemente una GPU. Esta jerarquía se conforma por una grilla de multiprocesadores (Streaming Multiprocessors - SM) donde cada SM está subdividido en warps, y estos, a su vez, están compuesto por 32 hilos cada uno. Como podemos ver en Fig. (3.5), al compartir Program Counter en todo un warp, todos los hilos de este ejecutan el mismo código en simultáneo[2], la clave está en procesar diferentes porciones de datos con cada hilo.

Las funciones que se ejecutan en la GPU se llaman kernels y al momento de llamar a un kernel, además de los parámetros que tendría una función usual en C/C++, se le agrega <<< , >>> donde se le especifica cuantos bloques y cuantos hilos (threads) por bloque se desean lanzar con ese kernel. Los hilos se agrupan en bloques y estos bloques conforman una grilla. Tanto la organización de los bloques como la de los hilos se pueden especificar en 3 dimensiones, esto sirve para poder dar un identificador (id) único a cada hilo a lo largo de toda la grilla y con este, poder definir que porción de los datos va a procesar cada hilo.

[2]Salvo cuando tenemos divergencia de hilos. Esto quiere decir que tenemos alguna herramienta de control de flujo que modifica el comportamiento de algunos hilos.

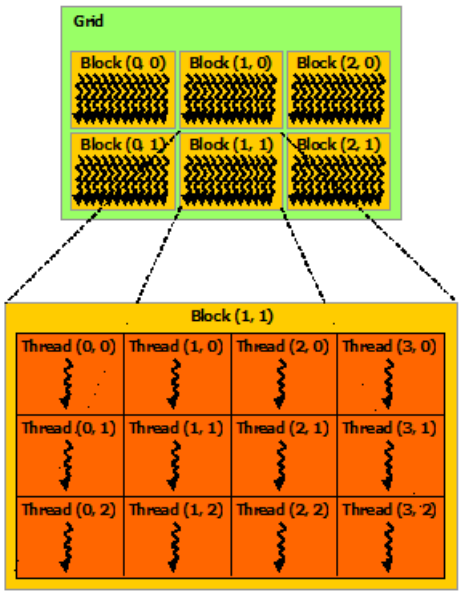


Figura 3.3: Una grilla de bloques donde podemos observar cada bloque con X,Y hilos por bloque

Cuando se ejecuta un kernel, cada bloque corre en un warp cuando los recursos para ese bloque están disponibles (registros, datos de memoria, memoria caché). Ese mismo warp puede correr otros bloques a medida que se tienen los recursos necesarios y de esta forma incrementar el paralelismo de ejecución. Aunque cada warp tenga solo 32 hilos, el máximo de hilos que se pueden pedir para un bloque es de 1024 dado que cada warp puede ejecutar varias porciones de un bloque.

Como bien dice la Ley de Amdahl (Amdahl, 1967), las mejoras de rendimiento por paralelización están limitadas por las partes secuenciales. Esto nos pone un límite a que tanto podemos paralelizar. A las secciones de un programa que no sean paralelizables es mejor ejecutarlas en CPU ya que al usar uno o pocos hilos terminarían reduciendo performance por toda la sobrecarga de usar un kernel. Asi es como terminamos con programas como se puede observar en Fig. (3.4) donde se puede ver claramente que secciones que corren en un solo hilo y las partes paralelizables en kernels. Otra detalle no menor que podemos observar es que cada kernel puede ser llamado con una cantidad de bloques e hilos distinta, tanto diferentes kernels con diferentes tamaños, como el mismo kernel ser llamado en diferentes partes, con diferentes tamaños.

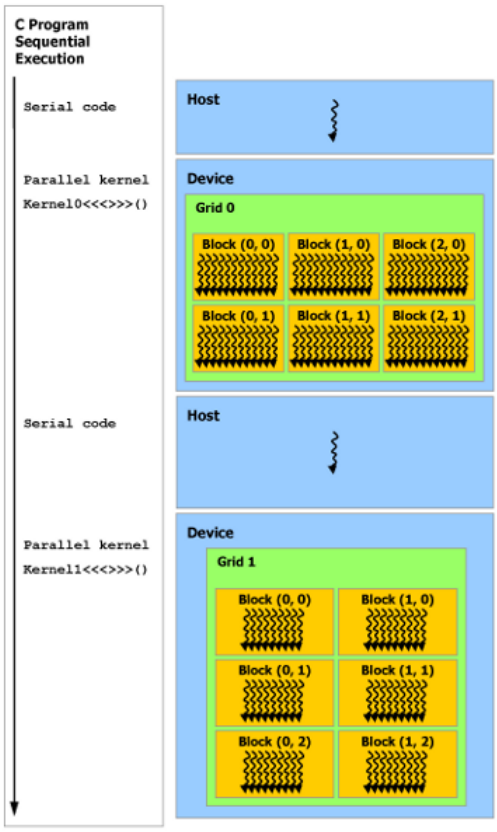


Figura 3.4: Para las secciones de código que sean altamente paralelizables se usa un kernel de CUDA con grillas y bloques de diferentes tamaños. Mientras que las partes secuenciales siguen corriendo en Host o CPU.

3.2.2. Jerarquía de Memoria

Dada la inmensa cantidad de unidades de ejecución con que cuentan las GPUs el problema se traslada a tener suficientes datos a ser procesados para mantenerlas ocupadas y la rápida disponibilidad de dichos datos. Para esto nos harían faltan memorias grandes muy rápidas, pero grandes memorias muy rápidas son demasiado caras, por esto se usan pequeñas memorias muy rápidas y memorias más grandes pero más lentas.

Host RAM

Las memorias más *alejadas* serían la memoria del Huésped (HM - Host Memory), la RAM o DRAM; estas son las memorias que usan los CPUs y no pueden ser accedidas por los núcleos de la GPU. En la 6ta versión de CUDA introdujeron lo que nombraron como Memoria UnificadaHarris, 2013. Previo a esta versión, uno tenía que declarar

punteros y pedir memoria tanto en la memoria Host como en la memoria de la GPU de manera independiente; y también hacer un manejo explícito de los copiados de memoria de Host a GPU como de GPU de vuelta al Host. Con este nuevo concepto de Memoria Unificada, lograron abstraer a los desarrolladores de este manejo dando la ilusión de una sola memoria, y manejando los copiados entre Host y GPU en base acceso a los datos.

Memoria Global

Luego tenemos las más grande y lenta de las memorias en la GPU, la Memoria Global (GM - Global Memory). Una de las primeras cosas que se debe hacer para usar CUDA es pedir memoria. Dadas las facilidades que provee CUDA con la memoria unificada, con solo pedir memoria para un objeto, se auto-genera tanto en memoria Host, como en memoria del dispositivo. Todos los hilos de todos los bloques tienen acceso a los datos de GM, pero estas memorias suelen tardar cerca de 400 ciclos para traer los datos. Como dijimos antes, memorias lentas pero grandes, las GM suelen rondar 2GB a 12GB en las GPUs de uso común y hasta 24GB en las últimas generaciones. Algo clave para tener en cuenta cuando hacemos uso de la memoria global es que cada acceso ya sea grande o pequeño tarda mucho, pero si coordinamos que hilos del mismo bloque acceden a posiciones de memoria contiguas se puede hacer una gran lectura en lugar de muchas pequeñas. Esto se conoce como accesos *coalesced* y el no hacerlos tiene un impacto negativo en la performance del programa ya que cada warp debe esperar por todos los accesos pequeños en lugar de un gran acceso.

3.2.3. Caché L2

Al igual que en los CPUs en la GPU tenemos una caché L2. Esta cumple la misma funcionalidad que otras cachés, reducir *cache misses* entre L1 y GM. Su uso es totalmente automático y no se puede pedir memoria específicamente en ella. Como podemos observar en Fig. (3.5), a diferencia de los CPUs donde cada núcleo tiene una L2 privada, en las GPUs la L2 es global a todos los bloques.

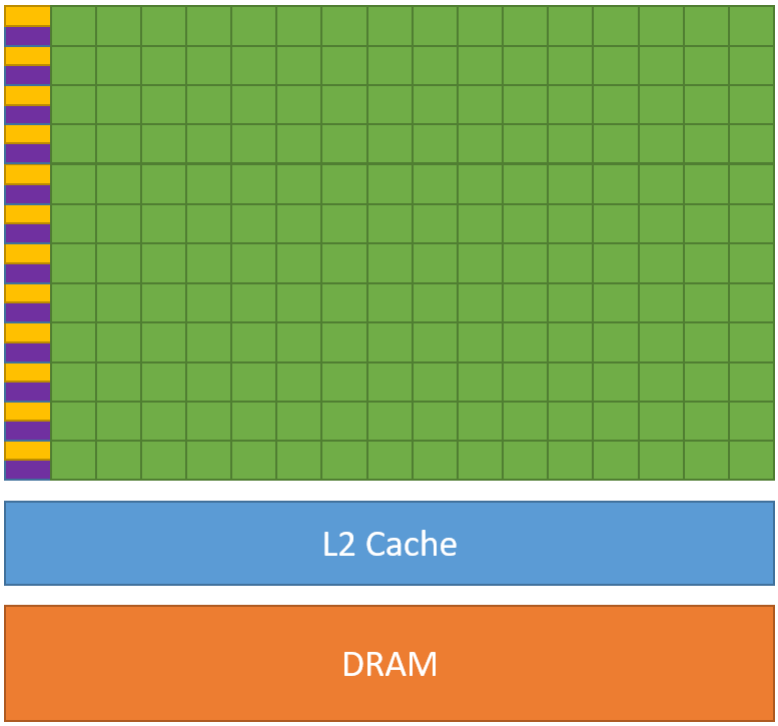


Figura 3.5: Representación simplificada de un SM. Los rectángulos amarillos son Program Counters y los violetas cachés L1. Cada fila representa un warp con varias unidades de ejecución cada uno.

Unified Cache

A nivel teórico, hay 3 memorias intermedias diferentes que se deberían tener en cuenta, y en arquitecturas anteriores, tenían un gran impacto si se las usaba correctamente. Estas 3 memorias intermedias son la shared, la memoria de texturas y la caché L1.

- La Shared o Memoria Compartida, es memoria compartida entre todos los hilos de un bloque. Tiene un tiempo de acceso muchísimo menor a la GM, pero solo cuenta con unos pocos MB de tamaño. Estas solían usarse como una suerte de caché para los bloques, coordinando para traer datos entre todos los hilos y usándolos más de una vez, pero desde una memoria de menor latencia. En contrapartida hay que tener mucho cuidado cuando se maneja esta memoria ya que al accederla todos los hilos del bloque se los debe sincronizar para evitar valores erróneos o comportamientos inesperados.

- La Memoria de Texturas es una memoria de alta velocidad y solo lectura. Originalmente pensadas para almacenar las textu-

ras para aplicarle a un modelo o imagen, se las suele usar para almacenar valores constantes que no vayan a cambiar a lo largo de todo el programa. Cualquier hilo de cualquier bloque puede acceder a este espacio de memoria. Además tiene capacidad de interpolar espacialmente para aumentar la resolución del bitmap.

- Al igual que en los CPU, tenemos una caché L1 como respaldo de los registros. Es la memoria de más alta frecuencia luego de los registros y la más chica. Su uso no es explícito sino que está manejado por los controladores.

Desde la arquitectura Volta, y en adelante, se introdujo la caché unificada, que son un gran bloque de caché L1 que, dependiendo de cuanta memoria de texturas o shared se pida, se va subdividiendo en base a las necesidades del programa. Estas memorias constan de 3 bloques de 32KB (en Turing "Turing Tunning Guide", 2020). Pero solo acepta 2 configuraciones, ya sea 1 bloque para shared y 2 para L1/Texturas o 2 bloques para shared y 1 para L1/texturas. Estas cachés, al igual que las cachés del CPU, tienen mecanismos optimizados tanto por hardware como software para garantizar la coherencia de datos en caché.

Registros

Por último tenemos los registros, estos son privados para cada hilo, una vez que son asignados, y al igual que los registros en CPU son muy pocos. En la arquitectura Turing "Turing Tunning Guide", 2020 hay 64K en registros de 32 bits para cada SM, con un límite de hasta 255 registros por hilo. Algo no menor a tener en cuenta es que mientras más registros use cada hilo, menos hilos podremos tener corriendo en simultáneo.

4 Implementaciones

Para este trabajo se desarrollaron diferentes implementaciones tomando en cuenta diferentes estrategias para procesar datos en GPU y una primera implementación en CPU.

4.1. Pseudo Código

En esta sección brindaremos un pseudo código de lo que hace la simulación para facilitar la comprensión de las futuras secciones. Donde TempMax, tmax y tmic son argumentos de entrada del programa. El resto de las matrices y funciones se explican en las siguientes secciones.

```
pedido de memoria
inicializacion de OC
inicializacion de matrices de constantes en base a OC
inicializacion de mx, my y mz
for TempMax <= temp <= 0 {
    // termalizacion previa
    for i = 0, i < tmax, i++ {
        update(mx, my, mz)
    }
    // actualizaciones y mediciones - Montecarlo
    for i = 0, i < tmax, i++ {
        update(mx, my, mz)
        if i mod t_mic == 0 {
            calcular magnetizacion
            calcular energias
        }
    }
    guardar mediciones
}
liberar memoria
```

4.2. Almacenamiento de Matrices

Para simular un espacio ocupado por partículas adyacentes se puede pensar como una matriz 3D donde cada posición es ocupada por

una partícula. Pero guardar esto en memoria puede ser complicado, ya que la memoria no es 3D sino 1D. Para afrontar esta problemática hay diferentes estrategias que se pueden considerar. Antes de explicar las diferentes opciones me parece necesario explicar las diferentes estructuras que se podrían considerar, en particular para los lenguajes C/C++:

- **Arreglo**: es un puntero a una dirección de memoria, donde hay un dato del tipo del que definimos este arreglo. Son de tamaño fijo y se pueden definir su tamaño al momento de declararlo o se puede pedir memoria de manera dinámica. En los arreglos todos los elementos están contiguos, reduciendo los tiempos de acceso, ya que con aritmética de punteros podemos simplemente hacer un llamado al puntero + la posición del elemento deseado.

- **Lista**: cada elemento de una lista consiste de dos partes, el dato que nos interesa y un puntero, al siguiente elemento de la lista. Estas van creciendo a medida que se lo necesita, y se pueden agregar elementos intermedios, pero tiene un tiempo de acceso bastante lento ya que para ir a hasta el elemento N debemos pasar por los N-1 punteros anteriores.

- **Vector**: estos son introducidos por C++ como una alternativa a los arreglos. Consisten de bloques de memoria contigua, pero a diferencia de los arreglos, pueden aumentar o disminuir su tamaño; esto es bastante costoso, ya que podría involucrar copiar todo el vector a otra posición de memoria.

Dado que:

- el objetivo es buscar la organización de datos más eficiente,
- que sabemos desde un principio la cantidad de elementos que tendremos en cada matriz,
- que esta cantidad no varía a lo largo de toda la ejecución,
- que se hace un extensivo acceso a lugares intermedios de cada matriz,

se eligió trabajar con un arreglo para cada matriz. Una vez elegido esto, todavía tenemos diferentes maneras de almacenar cada matriz, ya que en teoría estas tienen 3 dimensiones, mientras que en las direcciones de memoria solo tienen 1 dimensión. A continuación se explican diferentes formas de almacenar varias dimensiones en memoria.

4.2.1. Arreglos anidados

Se puede tomar una arreglo de arreglos de arreglos, donde directamente usamos variables x,y,z para acceder a la posición del arreglo que corresponda, por ejemplo myArray[x][y][z], para acceder al valor en la posición (x,y,z).

Hay 2 problemas con esta implementación. El primero es que, como mencionamos antes, cada arreglo es un puntero a un bloque de memoria contigua, pero cada arreglo puede estar apuntando a lugares muy alejados en memoria, entonces si tomamos un j en Y, para un x,z fijos, tenemos que myArray[x][j][z] y myArray[x][j+1][z] muy posiblemente estén en bloques distintos de memoria y esto impactaría en las cachés teniendo que ir a buscar diferentes bloques de memoria. El segundo es que para llegar a un dato se tienen que hacer 3 accesos, el primero a la posición myArray+x (llamémosle p1), el segundo a p1+y (llamémosle p2) y el tercero a p2+z.

4.2.2. Matriz aplanada

Las matrices con las que trabajaremos tiene 3 dimensiones, (x,y,z), tomaremos solo matrices cúbicas, esto es, el valor máximo para las 3 dimensiones es L-1. Para escribirlas en memoria (1D) lo que haremos será escribirlas por "filas", entonces tendríamos como primer elemento el (0,0,0), luego (1,0,0), y así para las primeras L de X. Luego la "fila de abajo" esto sería, (x,1,0) y así sucesivamente hasta haber cubierto la primera sub-matriz (x,y,0); y luego repetir esto para los diferentes z. Escribir las matrices de esta forma tiene como gran ventaja que para acceder a cada posición de la matriz es una sola lectura a memoria. Pero tiene 2 desventajas;

1. Que vecinos en Y y en Z quedan en posiciones de memoria no adyacentes, a L posiciones de memoria o L^2;

2. Que para acceder a un elemento de la matriz debemos realizar un pequeño cálculo previo para tener la posición exacta en la matriz aplanada. Aunque este, por suerte, es un cálculo bastante sencillo

$$IX(x, y, z) = x + y * dimX + z * dimY * dimX \tag{4.1}$$

4.2.3. Matrices

Para facilitar la comprensión del problema y del código, se listarán a continuación las matrices que se usan a lo largo de la simulación, y que representan. En Fortran se usaba el tipo de datos REAL de 32 bits por lo cual se decidió usar float para C.

- mx my mz: estas 3 matrices representan el valor del vector del spin de la partícula que esté en cada posición, esto es, para saber el valor del spin de la partícula (x,y,z) tenemos que mirar en ese momento a mx(x,y,z), my(x,y,z), mz(x,y,z), y con esos 3 valores podremos dibujar el vector spin de esa partícula.

- oc: esta matriz contiene la distribución de que elemento tiene cada partícula, ya sea Cr o Fe.

- mafe, macr: estas dos matrices tienen 0 o 1 en sus casillas, dependiendo de si una partícula es Fe o Cr. Si para la posición (x,y,z) tenemos que en mafe hay un 1, entonces en esa misma posición para macr habrá un 0, y viceversa.

- Jx, Jy, Jz, Dx, Dy, Dz: estas son las matrices de constantes de interacciones. Dependiendo del valor que tenga que elemento tenga una partícula dada, en base a su vecino en x, se fijan los valores de Jx y Dx. Análogamente se llenan los valores de Jy, Dy, Jz y Dz.

- mc, KA: son dos matrices de constantes, dependiendo del elemento de una celda tendrán un valor constante u otro. Ambas se usan para calcular diferentes valores de interés, entre ellos la magnetización y la energía.

Además en las diferentes implementaciones se han usado diferentes matrices auxiliares para almacenar valores temporales o sumas parciales.

4.3. Funciones

Como se mencionó previamente el objetivo de este trabajo es adaptar una simulación existente en Fortran a C/C++ para lograr una versión optimizada para CPU y luego extenderla con CUDA para lograr una versión optimizada para GPU. Dado este objetivo, el primer paso fue familiarizarse con el código Fortran, identificar los algoritmos

implementados, que secciones podían ser paralelizadas y cuales no. A continuación daremos una breve descripción de las funciones en las que se trabajo para todas las implementaciones.

4.3.1. Inicializaciones

Hay un pequeño grupo de funciones donde se inicializan todas las matrices. Estas se ejecutan solo una vez, al principio de la simulación, por ende no impactan al tiempo total de ejecución del programa. Sin embargo se optimizaron en todas las versiones para lograr los mejores resultados posibles.

Init_oc

En esta función se realiza el sorteo para la distribución de partículas de Cr o Fe con probabilidad *frac* (constante definida según que proporción se desee simular). Además de llenarse los valores de oc se llenan macr y mafe, según correspondan con 1 y 0.

Interact_init

Como bien indica el nombre en esta función, se inicializan las matrices de constantes de interacciones KA, Jx, Jy, Jz, Dx, Dy y Dz, similarmente a mafe y macr.

mc_init

Análogamente a la función anterior, esta simplemente llena la matriz mc con dos valores constantes según cual elemento corresponda a cada posición.

init_fill

La última de las inicializaciones llena las matrices mx, my y mz [4.2.3] con valores basados en números aleatorios[1] uniformes en (0,1).

4.3.2. Funciones de Cálculos

Estas funciones tiene un alto impacto en el tiempo total de la simulación ya que son llamadas numerosas veces. En particular la función

[1] se le aplican una serie de operaciones a cada n.a. según se explica en Billoni, 2016.

update consistía de aproximadamente el 98 % del tiempo de la simulación previa a todas las optimizaciones.

Update

Esta es la función que se encarga de actualizar el valor de las 3 direcciones para cada spin de la matriz, en otras palabras, actualizar todos los valores mx, my y mz. Es la que más tiempo ocupa dado que tiene un alto número de accesos a memoria y que debe realizar muchos cálculos con estos. Para actualizar una celda se llevan a cabo los siguientes pasos:

1. Cálculo de índices de los vecinos en base al índice correspondiente al hilo.
2. Cálculo de los candidatos a nuevos valores en base a un valor generado de forma aleatoria (con distribución uniforme en (0,1))
3. Cálculo de los delta de energía en base a los spines de los vecinos y los nuevos candidatos.
4. Si la energía decrece el candidato es aceptado. Pero si la energía incrementa o se mantiene, se acepta con una probabilidad que depende de el delta de energía y la temperatura actual del sistema.
5. Guardar valor.

Calculate

Esta función no estaba en el código original en Fortran, sino que se agrego en todas las adaptaciones para facilitar la lectura del main y para poder evitar duplicidad de código, ya que se realiza la misma tarea para 3 mediciones distintas. La misma consiste de realizar una sumatoria con pesos de todas las matrices mx, my, mz, usando como pesos mc, macr, y mafe, respectivamente. De esta forma, podemos aproximar un promedio de la temperatura de toda la matriz, o de solo las partículas de Cr o de solo las de Fe.

Energy

Tal y como indica el nombre en esta función se usan los valores de mx, my, mz para calcular la energía en base a las diferentes interacciones.

4.4. Estrategia Red/Black

Esta es una estrategia ampliamente usada en simulaciones donde se tiene que calcular un valor en base a las celdas vecinas. En particular, cuando las interacciones se dan con los vecinos más cercanos. Si fijamos z, tenemos una matriz 2D. La analogía más comúnmente usada es un tablero de ajedrez, donde tenemos casilleros rojos y negros intercalados. Entonces, como para calcular cada celda solo usamos los vecinos (que son todos del color opuesto), podemos calcular todas las celdas de un color y luego todas las celdas del otro color, y de esta forma usar siempre los valores más "nuevos" de una red para calcular la otra. Ahora, si pensamos esto en un contexto paralelo, esta estrategia nos sirve por que podemos calcular todas las celdas de un color sin tener que preocuparnos por si los vecinos tienen valores actualizados o viejos o fueron actualizados en medio del procesamiento.

Además, a nivel optimización de memoria, si usamos las matrices de forma usual, como las lecturas a memoria se hacen por bloques, se usan la mitad de los datos que se traen. Lo que esta estrategia permite es hacer lecturas de bloques contiguos más grandes, ya que se lee o escribe en posiciones de memoria de cada bloque.

Como teníamos las matrices de 2D $L \times L$, si las dividimos a la mitad, para tener en una matriz todas las celdas de un color y en la otra todas las del otro color, nos resultan 2 submatrices de tamaños $L/2 \times L$, pero si a esto lo aplicamos para todo z nos resultan 2 matrices con 3 dimensiones de tamaño

$$RBN = L/2 \times L \times L \tag{4.2}$$

Pero no olvidemos que en la matriz original, cada celda tenía 6 vecinos, 2 en cada dirección. Entonces para dados z y *z+1*, si tenemos que en la posición (x,y) de z es red, entonces *z+1* es black. Con esta aclaración debería ser fácil ver que para dos valores en Z continuos, se invierte el color de cada posición (x,y).

Ahora si solo observamos un color, supongamos rojo, en la matriz completa, podemos ver que queda la matriz *ahuecada*. Si la *comprimimos* en el eje X tendremos una media matriz con todas celdas del mismo color.

Para mantener la relación con la posición original (para no perder a los vecinos del otro color), se puede observar que para las filas que empezaban con la primera celda del color rojo, cumplen que su posición en la matriz completa es 2 veces la posición en la media matriz en el eje de las X (ya que ahora todas las posiciones de este color son

pares), mientras que su posición en Y y en Z son las mismas. Por otro lado para las filas que empezaban con el otro color, veremos que sus posiciones en la matiz completa son el doble más 1 de la posición nueva en el eje X (ya que las posiciones de la matriz original eran impares), pero de nuevo mantienen mismo índice en Y y en Z.

Si consideramos los párrafos anteriores tenemos que para un color dado, supongamos rojo, si tenemos que la primera fila ($y = 0$) empieza con una celda de este color, entonces la posición en la matriz completa será $2 * x$ y para la fila siguiente ($y = 1$) tenemos que será $2 * x + 1$. Esto podemos expandirlo tal que todos los y pares tienen que $2 * x$ es la posición en la matriz completa y $2 * x + 1$ para los y impares. Pero como también dijimos, los colores se invierten en z consecutivos. Entonces para $z+1$ tendríamos que con y par, la posición en la matriz completa es $2 * x + 1$, mientras que para para y impar es $2 * x$. A su vez, para el otro color, todo esto se invierte.

Para diferenciar los colores se uso un `enum` para los colores RED/-BLACK, que esto asigna valores numéricos a estas variables, en particular 0 y 1. Con esta implementación y lo especificado en el párrafo anterior, si tenemos $(x, y, z) \in (L/2 \times L \times L)$ y $color \in (RED, BLACK)$, la siguiente fórmula nos define ese +1 de diferencia en la matriz completa:

$$STRF(color, y, z) = ((color + y\&1 + z\&1)\&1) \tag{4.3}$$

Donde & es la función AND binaria, es otra forma sacar `mod 2`, osea, si el valor es par o no.

Entonces en las diferentes versiones se terminan utilizando siempre que se acceda a una media matriz y/o matriz completa las siguientes fórmulas

$$\begin{aligned} ixf &= 2 * x + STRF(color, y, z) \\ index &= IX(ixf, y, z) \\ rbx &= RB(x, y, z) \end{aligned} \tag{4.4}$$

Donde STRF es (4.3), IX es (4.1) con $dimX = dimY = L$ y RB es (4.1) pero con $dimX = L/2$ y $dimY = L$.

Un detalle no menor para tener en cuenta es que para acceder a los vecinos que están en la matriz del otro color, podemos ver en Fig. (4.1) que los vecinos en Y y en Z, mantienen el mismo valor para las otras dos dimensiones. Osea, los vecinos en Y de *(x, y, z)* serían *(x, y+1, z)* y *(x, y-1, z)*. Análogamente para Z. Pero para los vecinos en

X podemos ver que, por ejemplo, para $x = 0$ rojo, los vecinos tienen las mismas coordenadas para Y y Z, y en X son las posiciones 0 y 1, osea x y $x + 1$.

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

rojo

0	2	4	6
9	11	13	15
16	18	20	22
25	27	29	31
32	34	36	38
41	43	45	47
48	50	52	54
57	59	61	63

negro

1	3	5	7
8	10	12	14
17	19	21	23
24	26	28	30
33	35	37	39
40	42	44	46
49	51	53	55
56	58	60	62

Figura 4.1: Representación de una matriz unificada y dos medias matrices, manteniendo la numeración original

Pero si vemos los vecinos de $x = 1$ rojo (número 2), serían las celdas en $x = 0$ y $x = 1$ en negro. O sea, los vecinos de $x = 1$ son $x-1$ y x en negro. Esta consideración para los vecinos en X se tiene en cuenta y aplica en todas las implementaciones con 2 matrices, en las funciones Update y Energy.

4.5. CPU

4.5.1. SingleThread

Esta fue la primera implementación que se realizó. En una primera instancia, se llevo a cabo solo una traducción, sin grandes optimizaciones. Solo se probaron diferentes flags de compilación con gcc, y se ordenaron los `for` anidados.

Como se mencionó previamente en Almacenamiento de Matrices, los arreglos son bloques contiguos de memoria plana (1D). Entonces al recorrer una matriz cúbica, para mantener la estructura de vecinos, se decidió usar 3 variables (i,j,k) para simular las 3 dimensiones de la matriz. Si queremos recorrer la matriz primero por las X (o sea cada fila), luego por las Y (columnas) y al último por Z (siguientes sub matrices), en lugar de escribir:

```
for i=0; i<L; i++:
    for j=0; j<L; j++:
        for k=0; k<L; k++:
```

Que esto en realidad primero fija la posición en X, luego la posición en Y y varía siempre en Z. Para poder aprovechar la localidad de los datos e ir accediéndolos en un orden favorable para las lecturas de memoria, conviene hacerlo de la siguiente manera:

```
for k=0; k<L; k++:
    for j=0; j<L; j++:
        for i=0; i<L; i++:
```

de esta forma accedemos a memoria en el orden en que se fueron guardando los datos según se explicó en 4.2.2

Random Number Generator

Los RNG son una parte vital de este tipo de simulaciones ya que, tanto para generar el nuevo valor de cada celda en la función *Update*, como para la distribución de elementos y los valores originales de cada spin, se usan valores generados aleatoriamente, con distribución uniforme en (0,1), sobre los cuales se realizan una serie de operaciones.

Para la primera versión se utilizo el RNG básico que viene con C, `rand()`. Este generador no es particularmente rápido, ni paralelizable, pero para una primera implementación fue más que suficiente.

Red/Black unificado

Como bien se contó en Estrategia Red/Black esta estrategia consiste de dividir la matriz en dos sub matrices, cada una de un color, y procesarlas de forma independiente. En esta primera implementación, se usó una sola matriz para ambos colores y lo que se utilizó para independizar las subredes fue usar una combinación de índices para acceder solo celdas de un color por cada paso, como se puede ver en Fig. (4.2). Para los de un color se usaba la siguiente combinación de índices, con $(i, j, k) \in L \times L \times L$

$$
\begin{aligned}
x_0 &= 2 * i & y_0 &= 2 * j & z_0 &= 2 * k \\
x_3 &= 2 * i + 1 & y_3 &= 2 * j + 1 & z_3 &= 2 * k \\
x_5 &= 2 * i + 1 & y_5 &= 2 * j & z_5 &= 2 * k + 1 \\
x_6 &= 2 * i & y_6 &= 2 * j + 1 & z_6 &= 2 * k + 1
\end{aligned}
$$

y para la el otro color, la siguiente combinación

$$
\begin{array}{lll}
x_1 = 2 * i + 1 & y_1 = 2 * j & z_1 = 2 * k \\
x_2 = 2 * i & y_2 = 2 * j + 1 & z_2 = 2 * k \\
x_4 = 2 * i & y_4 = 2 * j & z_4 = 2 * k + 1 \\
x_7 = 2 * i + 1 & y_7 = 2 * j + 1 & z_7 = 2 * k + 1
\end{array}
$$

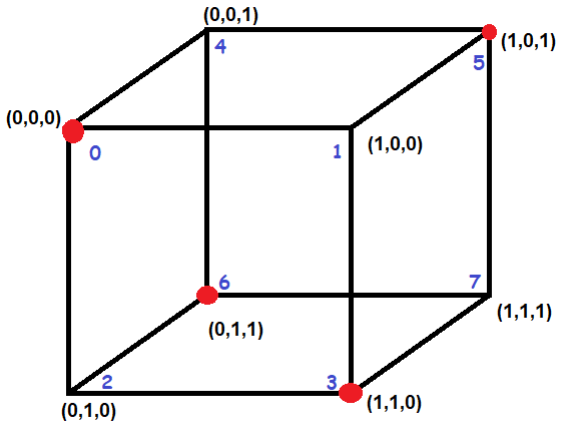


Figura 4.2: pequeña matriz para ilustrar los índices descriptos anteriormente

Para la función Update se hacen 2 pasadas, cada una tomando los diferentes índices para los diferentes colores, de esta forma asegura que se actualiza todo un color antes de modificar valores del otro color.

En la función Calculate como usan los valores de cada color para calcular una variable distinta se pueden leer todos juntos en la misma pasada, pero usando los índices descriptos previamente para sumarlos a la variable correspondiente.

Por último, en Energy, al calcular valores basados en la matriz completa no solo un color u otro, se recorre la matriz de forma normal, accediendo solo a los vecinos.

4.5.2. Multi-Hilos - OMP

Para la implementación multi-hilos se decidió utilizar OMP (Open Multi Processing), que es una API para el lenguaje C/C++ que facilita ciertas modificaciones al comportamiento del programa en tiempo de ejecución a través de directivas para el compilador y funciones de su biblioteca. Para dar las directivas al compilador se utilizan lo que

se llaman *pragmas*, estos son como comentarios para el compilador, que serán descriptos a continuación. Por otro lado también ofrece diferentes funciones, relacionadas al manejo o identificación de los hilos. En la siguiente lista se dará una breve descripción de las principales funcionalidades de OMP que se utilizaron.

- **parallel**: indica que el siguiente bloque debe ser tratado como una sección paralela, esto significa que todos los hilos disponibles deben ejecutar cada uno el código del bloque.

- **for**: especifica que hay un bucle de tipo `for` que se debe dividir entre los hilos, osea, cada hilo toma un intervalo diferente de la variable sobre la que se cicla.

- **collapse**: se usa cuando se tienen `for` anidados. De esta forma podemos decirle que cada hilo ejecute usando una combinación particular de las variables de los for, similar a la directiva anterior.

- **shared**: denotando una serie de variables entre paréntesis, por ej `shared(const1, const2, result1, result2)` le decimos al compilador que estas variables deben estar disponibles para todos los hilos y que son compartidas entre todos. Esto no nos obliga a que todos los hilos deban leerlas o escribirlas, pero el compilador debe asegurar que esto sea posible si se lo desea.

- **private**: en contraposición a `share`, esta directiva se usa para indicar que variables son privadas a cada hilo. Todas las que se declaren `private(x,y,z)`, tendrán una instancia para cada hilo, y la forma en que cada hilo la modifique solo afecta a su propia ejecución.

- **default**: especifica el comportamiento por defecto que se le debe atribuir a todas las variables no especificadas en un `share()` o en un `private()`. Si se la usa con parámetro *share*, entonces todas las variables que no esten especificadas por `private()` serán compartidas entre todos los hilos. Pero si se usa como parámetro *none*, entonces si hubiera alguna variable no especificada dentro de `share()` o en un `private()`, esto lanzará un error ya que no se sabría como tratar esta variable dentro del bloque paralelo. En otras palabras, fuerza al desarrollador a especificar un comportamiento para todas las variables.

- **atomic**: le da un carácter atómico a la siguiente línea, esto significa que hasta cuando un hilo empiece a ejecutar las instrucciones que conforman esa línea de código, todos los otros hilos no podrán modificar o leer las variables que estén involucradas en dicha línea. De esta forma forma uno puede evitar algunos comportamientos inesperados dados por lecturas y escrituras múltiples de las mismas variables.

- **reduct**: se usa con un `for`. La idea es poder paralelizar tareas como sumatoria de un arreglo. Esta es una tarea que se realiza varias veces a lo largo del programa. Similarmente a los `paralel for` antes explicados, a cada hilo se le asigna una valor de la variable que se cicla, esta variable es considerada privada a cada hilo. Al final la ejecución del bloque, un hilo se encarga de realizar la operación especificada entre los resultados parciales.

Para las funciones de inicializaciones, Update, Energy y Calculate, donde se recorren las matrices completas, se utilizó `parallel for` para paralelizar los segmentos de `for` anidados definiendo como `share` las variables de resultados y los punteros a las matrices, y como `private` las variables relacionadas a índices o resultados parciales. En una primera iteración se uso `atomic` para las sumatorias donde varios hilos aportan al valor final, guardando estos resultados en variables. Esto ocurría en el contador de cambios de Update, las variables resultantes de la función Calculate y las de Energy

RNG Multi-Hilo

En cuanto al RNG para esta versión, se tenía que buscar un generador que no dependiera de estados, o que tuviera una forma altamente paralelizable de llevarlos, ya que si todos los hilos tuvieran que esperar a que el RNG les entregue los nuevos valores uno a uno, esto termmina serializando gran parte del código.

Por estos motivos se decidió usar un motor de los presentados en Tina's Random Number Generator Library (TRNG "Tina's Random Number Generator Library", 2020), que presenta un alto nivel de paralelización. En particular el motor lcg64_shift implementa un generador congruencial basado en el algoritmo de transición:

$$r_{i+1} = a \cdot r_i + b \; mod \; 2^{64}. \tag{4.5}$$

Su periodo es de 2^{64} solamente si b es impar y $a \; mod \; 4 = 1$. Además, se realiza la siguiente transformación lineal

$$
\begin{aligned}
t_{i,0} &= r_i \\
t_{i,1} &= t_{i,0} \oplus (t_{i,0} \gg 17) \\
t_{i,2} &= t_{i,1} \oplus (t_{i,1} \ll 31) \\
t_{i,3} &= t_{i,2} \oplus (t_{i,2} \gg 8) \\
q_i &= t_{i,3}
\end{aligned}
$$

Donde $\oplus$ es la suma binaria (disyunción exclusiva), $x \gg n$ es un shift binario de x hacia la derecha en n posiciones y $x \ll n$ es un shift binario a la izquierda.

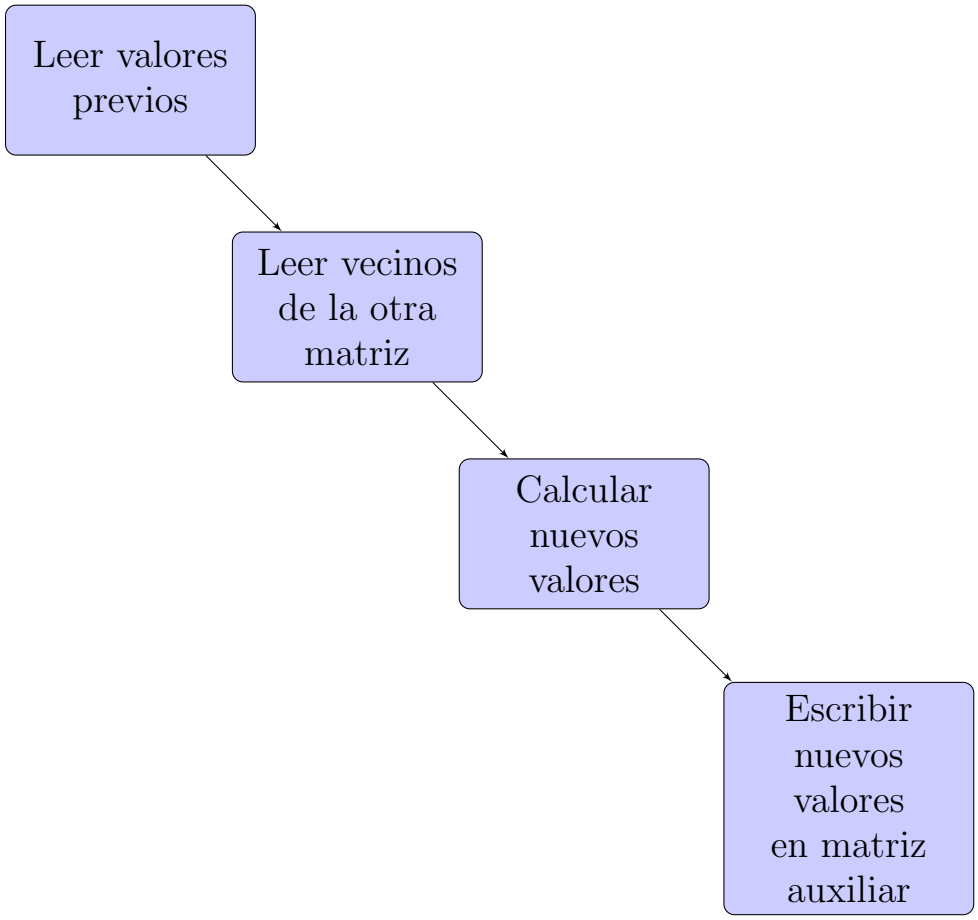
Para paralelizar este algoritmo se eligió usar el método leapfrog (salto de rana) que distribuye una secuencia r_i de números aleatorios en p procesos (en nuestro caso hilos) separando la secuencia base y permitiendo que cada hilo genere números de manera independiente a los otros. Es aquí donde podemos aprovechar una de las funcionalidades que ofrece OMP, esta es identificar a cada hilo con id único. Con la función `omp_get_thread_num()` se obtiene este id, identificando al hilo que este ejecutando. Con este id único, podemos definir P canales (siendo P la cantidad total de hilos que tenemos disponibles) y que cada hilo p acceda solo a su canal del RNG. En un primer momento se llamaba a `omp_get_thread_num()` por cada número aleatorio que se iba a calcular. Aquí fue cuando se notó el gran impacto y el costo que trae esta función. Tras un pequeño análisis fue claro que cada hilo esperaba mucho tiempo por esta función. Asi fue que se decidió reducir sus llamados al principio del bloque paralelo y guardarlo en un registro.

División de matrices

En la primera iteración se afronta la estrategia Red/Black con una matriz unificada. Pero como bien se explicó en Estrategia Red/Black, esta estrategia se puede aplicar a dos matrices de $L/2 \times L \times L$. Para la segunda iteración paralela con OMP se hizo esto mismo. Se dividieron las matrices *mx, my, mz* en 2 mitades, estas serán referidas como *mx_r, mx_b, my_r, my_b, mz_r y mz_b*. El principal cambio que implicó esto fue el manejo de índices que se hacia en las funciones de cálculos. Ya no se usan más las fórmulas de Red/Black unificado sino que se usan las funciones descriptas en 4.4, pero como por ahora solo se pasaron las 3 matrices principales, y las matrices de constantes de interacción (Jx, Jy, Jz, Dx, Dy, Dz), a medias matrices, se debe mantener la relación entre las posiciones en las medias matrices y las posiciones en

las matrices completas (RBX a IX), para todas las otras matrices de constantes.

En cuanto a la función Update, esta división de matrices por color, nos permite reformular la función para que calcula todo un color a la vez, y llamar a esta función 2 veces (ajustando los parámetros para cada color), en lugar de un llamado a una función que hace dos bucles casi idénticos, donde solo varían los índices que vamos accediendo. Entre los cambios implementados gracias a esta idea, se tomaron 2 (medio) matrices auxiliares, para guardar los nuevos valores de las matrices, cambiando la estructura de la función Update al siguiente:



Pasándole primero como parámetros *mx_r, my_r, mz_r* junto con *RED* y luego *mx_b, my_b, mz_b* junto con *BLACK*. Luego de los dos llamados a Update se realiza un `swap` (intercambio) entre los punteros a las matrices con los valores nuevos y los punteros a las matrices con los valores viejos.

En Calculate sería poco eficiente si cada hilo solo cargara 1 valor de cada matriz principal y un valor de matriz de constante correspondien-

te, y los multiplicara. Por este motivo, para darle más trabajo a cada hilo, y como ya no tenemos la complicación de los índices cruzados de Red/Black unificado, podemos agarrar varios elementos consecutivos. Así como en la versión de matriz unificada se tomaban 4 celdas del mismo color, en esta nueva implementación, se toman una celda y las 3 que le siguen, se multiplica cada valor por el correspondiente de la matriz de constantes según sea mc, macr o mafe, y luego se las suma con `atomics` a una variable de tipo `share` para que acumule la suma de todos los hilos.

Por otro lado para Energy se sigue una idea más parecida a Update, ya que aquí si se relaciona una cantidad más contundente de cálculos y accesos. Como el valor que se desea obtener es en base a toda la matriz, lo que se hace es sacar una suma parcial las mediciones de todo un color, luego sumar los valores que se van calculando del segundo color sobre este.

Luego de esto se agregó `default` en las secciones paralelas. Esto se considera principalmente una buena práctica, ya que fuerza a especificar con que alcance debe ser considerada cada variable.

En una otra iteración, se dividieron también las matrices de constantes, mc, KA, macr, mafe, dejando de lado el uso de IX para las funciones de cálculos, ya que solo queda oc como matriz de tamaño N.

Por último, se modificó la forma de procesar resultados parciales en Calculate y Energy. En lugar de ir agregando resultados parciales con `atomics` a la variable final, se tomó una matriz auxiliar de tamaño RBN, y cada hilo hace sus cálculos internos escribiendo a la matriz auxiliar en la posición de la celda que calculó. De esta forma no se necesitan los `atomics`, cada hilo puede calcular y escribir de manera indpendiente. El costo de esto es que después del llamado a Calculate o Energy para los dos colores, se debe hacer un `reduce` para sumar los valores de cada posición de las matrices auxiliares. Además para las funciones de cálculo, se cambio el `collapse` de 3 variables (x,y,z) a solo 2 variables (y,z). Esto quiere decir que cada hilo en lugar de hacer, 4 multiplicaciones y 4 sumas en Calculate, o calcular las mediciones correspondientes a una celda en Energy o Update, recorre lo que sería equivalente a una *cara* de la media matriz. Al realizar el trabajo de una grupo más extenso de datos contiguos, se consigue una mejora considerable gracias al uso de la caché.

Contador de cambios

Como parte de la función Update se lleva un contador de cambios aceptados. Este contador suma 1 o 0 por cada hilo según haya aceptado o no el nuevo valor para ese spin. En un principio era simplemente una variable compartida entre todos los hilos que se sumaba con un `atomic`. Luego se paso a un vector, con tantas posiciones como hilos haya, así cada hilo escribiría solo en su espacio del arreglo y no habría necesidad para coordinar, entre los hilos, el acceso al contador. Tras hacer los dos llamados a Update (uno por cada color) se debía hacer un pequeño `for` para sumar las sumas parciales que había hecho cada hilo.

Inspirado en la separación de matrices por color, en lugar de usar un arreglo para los dos llamados de Update, se agrego un segundo vector, y de esta forma tener un arreglo de contador para cada color, y tras los dos llamados sumar ambos.

Como última mejora en este aspecto, se extendió el tamaño de estos arreglos a RNB, de 4.2, y de esta forma cada lugar se escribía una sola vez, evitando tener que hacer lecturas de los contadores sino solo escrituras. El problema con este cambio es que ahora en lugar de tener que hacer P sumas parciales tras la ejecución de los dos Updates, hay que hacer RBN. Para optimizar esto se uso el `reduct` de 4.5.2, haciendo esta suma de una forma paralela.

4.6. GPU

En esta sección se detallarán las diferentes implementaciones que se realizaron para GPU explorando las alternativas que presenta la arquitectura de GPUs y el modelo CUDA.

4.6.1. RNG GPU

Para todas las implementaciones en GPU se decidió utilizar Philox de "Random123: a Library of Counter-Based Random Number Generators", 2016. Philox se basa en un cifrado de Fesitel en conjunto con multiplicación de enteros para generar múltiples números aleatorios a la vez. En particular se usa en las inicializaciones, para la distribución aleatoria de elementos y para los valores iniciales de mx, my y mz. Y en Update, tanto para los nuevos valores de como para la distribución de aceptación de cambios. Cada vez que se va llamar a una función de Philox, se le pasan dos parámetros, el tiempo, como factor incremental, y la posición de cada celda como factor de unicidad. Ambos

son tomados en cuenta para las operaciones internas que realiza este RNG.

4.6.2. Nociones generales de los kernels

Antes de dar detalles sobre las diferentes implementaciones, se contará sobre algunas estrategias o prácticas que se tomaron en todas las implementaciones, ya sea porque son buenas prácticas o porque fueron optimizaciones que no se veían afectadas por el tipo de memoria que se fuera a utilizar en las funciones de cálculo.

Manejo de índices

Como bien se menciona en Arquitecturas de GPUs, los hilos se organizan en bloques, y lo bloques en una grilla. Cada tanto los hilos como los bloques tiene coordenadas, estas se definen al llamar a un kernel. Estas coordenadas pueden tener 3 dimensiones. Para las coordenadas de los hilos se usa `threadIdx.x`, `threadIdx.y` y `threadIdx.z`. Análogamente para los bloques se usa `blockIdx`, pero estos cuentan además con `blockDim`, que tiene valor constante y es la igual a la cantidad de hilos en cada dimensión. Entonces para definir el *(i, j, k)* de la matriz de tamaño N que le corresponde calcular o acceder a un hilo, se deben seguir las siguientes expresiones:

$$\begin{aligned} i &= threadIdx.x + blockDim.x \times blockIdx.x; \\ j &= threadIdx.y + blockDim.y \times blockIdx.y; \\ k &= threadIdx.z + blockDim.z \times blockIdx.z; \end{aligned} \tag{4.6}$$

Con estos *(i,j,k)* ya podemos asignar cada hilo una posición de la matriz, ya sea con las funciones de Red/Black unificado o con las de Fig.(4.4). Estos índices también nos sirven para limitar los hilos, ya que `blockDim.x` × `gridDim.x` (esto es, cantidad total de hilos en la dimensión X), no necesariamente es igual L. Usualmente esto es superior a L, y usando estas variables *(i, j, k)* uno puede filtrar que hilos deben trabajar. En particular se usa una sentencia similar a la siguiente en todos los kernels para este propósito:

```
if (i < L && j < L && k < L)
```

para los kernels que tengan que pasar por las N posiciones de alguna matriz completa, y

```
if (i < L/2 && j < L && k < L)
```

para los kernels que solo tengan que trabajar con medias matrices.

Manejo de Resultados

A diferencia de las funciones normales, los kernels son siempre de tipo `void`, esto quiere decir que cualquier tipo de resultado que se desee sacar de ellos, debe hacerse mediante punteros.

Para las funciones de inicialización estos resultados estaban en las mismas matrices cuyos valores eran llenados. Para las funciones de cálculos requiere un poco más de trabajo.

De la función Update nos interesan dos resultados. Por un lado los nuevos valores de las matrices *mx, my* y *mz*, y por otro lado el contador de cambios aceptados. En las primeras versiones donde se utiliza la matriz unificada, en la misma matriz que se daba como argumento de entrada se devolvían los nuevos valores. Luego cuando se pasó a medias matrices, se decidió a utilizar matrices auxiliares para sacar los nuevos valores. En cuanto al contador de cambios aceptados, similarmente a lo que se hizo para CPU, en un primer momento se escribían los cambios en una matriz tamaño N y luego sumaban todas las posiciones de la matriz en un `reduce`. Después se pasó a 2 *medias* matrices, donde cada una se pasa a un llamado de Update, con un color diferente, guardando en ellas los cambios aceptados de ese color. Luego de los dos llamados, se usa un `reduce` para sumar todos los cambios de ambos colores.

En cuanto a Calculate, similarmente a lo ocurrido en Update, en la primera versión para GPU se usaba la matriz unificada, usando los índices cruzados para sumar las celdas de los diferentes colores de manera independiente. Luego se reestructuró para tomar una media matriz, todas las celdas de un color, sumar 4 posiciones contiguas, multiplicando cada una por la constante correspondiente, y escribiendo estas sumas parciales a una matriz auxiliar. Tras hacer este llamado para los dos colores, se suma cada matriz auxiliar con un `reduce` y se guardan en 6 variables cuyos punteros son provistos como argumentos de la función. Estas 6 variables son una por cada color, por cada dimensión, x-rojo, y-rojo, z-rojo, x-negro, y-negro, z-negro.

En la función Energy, como se desea calcular 3 valores de energía en base a toda la matriz, en la primera implementación se recorría toda la matriz sin diferenciar por colores. Para cada celda se calculan sus 3 valores, considerando los vecinos, y luego se escriben en 3 matrices auxiliares de tamaño N. Tras pasar por toda la matriz se realizaba un `reduce` para cada matriz, resultando los 3 valores deseados. Cuando se dividió las matrices, se modificó esta función para reflejar ese cambio. Esto consistió en realizar 2 llamados al kernel de Energy, cada uno para un color, y pasarle medias matrices para guardar las sumar parciales

de cada posición. Finalizando, al igual que antes, con un `reduce` para cada color, para cada valor.

Reduce

El objetivo del reduce es paralelizar tareas como sumatorias, donde cualquier hilo y en cualquier orden, puede aportar una parte. Ahora para aprovechar las características de la arquitectura GPU lo que se plantea es iterar en cada bloque para ir acumulando los valores de sus celdas de a 2, como se Fig. (4.3), terminando con la sumatoria de todo un bloque en el primer hilo de dicho bloque. Gracias a una nueva funcionalidad de CUDA esto se puede lograr en un kernel usando `__shfl_down_sync()`, que permite comunicación directamente con los registros de otro hilo, siempre que ambos estén en el mismo warp. Una vez que todos los bloques hayan terminado este cálculo deben escribirlo a memoria global, ya que es la única forma de comunicación entre bloques. En un segundo kernel se leen estos resultados y se usa un único bloque para repetir esto y sumar los resultados parciales de cada bloque del kernel anterior.

La librería "CUB", 2020 provee componentes de software reutilizables, de vanguardia, para diferentes capas del modelo CUDA. Una de las funcionalidades es, en particular, un reduce optimizado tanto para bloques como para trabajar con toda una GPU. Dadas la excelente performance que esta presenta, se decidió utilizarla en todas las diferentes implementaciones de GPU, en todas las ocasiones que se requiere hacer un `reduce`.

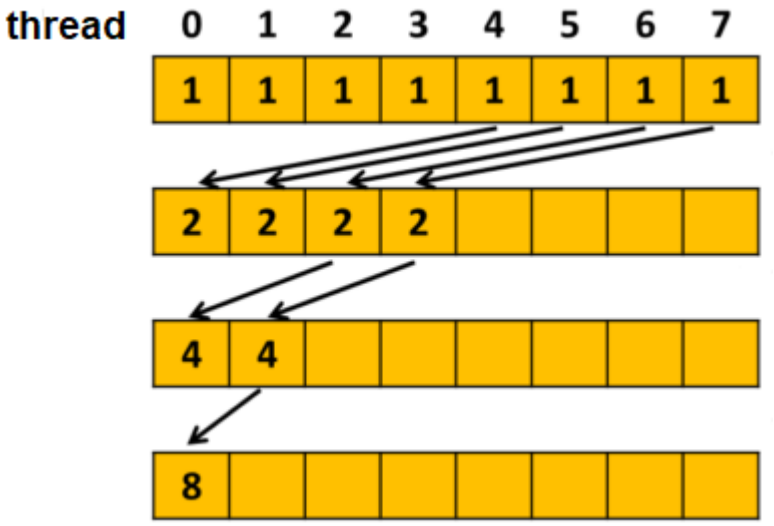


Figura 4.3: pequeño ejemplo de un reduce usando __shfl_down_sync.

4.6.3. Memoria Global - GM

La primera implementación que se hizo fue bastante similar a Red/-Black unificado, con una sola matriz de tamaño N, con ambos colores, calculando con los índices cruzados. La principal diferencia fue que, como se explicó previamente, los índices de cada matriz eran dados en base al id de cada hilo y no a los 3 `for` anidados.

Lo primero que se hizo fue modificar el pedido de memoria de `calloc` a `cudaMallocManaged`. Con este llamado, haciendo uso de la memoria unificada, ya contamos con los espacios de memoria necesarios tanto en memoria Host, como memoria global de GPU. Ya con memoria pedida para todas las matrices, se pasa por las diferentes funciones de inicialización. Estas no varían a través de las diferentes implementaciones de GPU ya que al ser llamadas una sola vez, su impacto en performance es nulo. Una vez inicializadas las matrices se pasa a Update.

En Update, una vez pasado a dos medias matrices, al igual que en la versión multi hilos, se separa en dos kernels, uno para cada color, escribiendo los nuevos valores en matrices auxiliares. Para hacer el manejo de índices se usaron las fórmulas de (4.6), y con estas cada hilo de cada bloque, trae de memoria los valores de las celdas vecinas. Se los guarda en variables locales de cada hilo, y con estas se hacen los cálculos necesarios para determinar si los nuevos valores serán aceptados o no. Ya sea el viejo valor o el valor anterior para esa celda, se escribe en la matriz auxiliar que contiene los nuevos valores y, según corresponda, se escribe en la matriz del contador de cambios. Tras llamarse Update para los dos colores, se realizan un `reduce` para los contadores, y se intercambian los punteros entre las matrices auxiliares con los nuevos valores y *mx_r, mx_b, my_r, my_b, mz_r y mz_b*.

Para Calculate, se hacen también 2 llamados, uno para cada color, donde se leen las matrices *mx_r, mx_b, my_r, my_b, mz_r y mz_b*, según corresponda, junto con la matriz de constantes correspondientes y se suman de a 4 celdas contiguas por hilo. Luego se escriben estas sumas parciales en una matriz auxiliar. Tras los llamados para los dos colores se hace un reduce para cada color para cada dirección.

Por ultimo en Energy, al igual que las dos anteriores, se hacen 2 llamados, uno para cada color, donde cada hilo lee una posición de las 3 matrices (del color que corresponda), calcula los 3 valores de energía y escribe cada uno en una matriz auxiliar, en la posición de la celda. Luego de los dos llamados a Energy se llama un reduce para cada variable y así se acumula los valores de toda la matriz.

4.6.4. Tiling - Share

Para esta implementación solo se modificó la función Update. La idea del *tiling* es traer, entre todos los hilos de un bloque, a memoria *share* los datos que van a necesitar todos los hilos, ya que al tener los bloques 3 dimensiones todos los los vecinos son usados más de una vez.

Supongamos que los bloques son de tamaño $\mathcal{BX} \times \mathcal{BY} \times \mathcal{BZ}$, entonces los bloques de share, o tiles, serán de tamaño $(\mathcal{BX}+2) \times (\mathcal{BY}+2) \times (\mathcal{BZ}+2)$, para poder tener en share todos los datos del bloque, más los vecinos de las celdas de los bordes del bloque. Estos bloques se manejan como se explica en Matriz aplanada y para el manejo de índices se implemento una modificación de (4.4) en donde se diferencian dimX de dimY, ya que antes se trabajaba con matrices cúbicas y ahora con bloques que no son necesariamente cúbicos. Entonces se definió la función `tdx` como:

```
__device__ __host__ static inline size_t tdx(const size_t i, const
    size_t j, const size_t k, const size_t dim_x, const size_t
    dim_y) {
    return (i + dim_x*j + dim_x*dim_y*k);
}
```

Donde `__device__` y `__host__` indican que la función debe ser compilada tanto para correr en GPU como en CPU.

Una vez pedida la memoria para el tile en share, cada hilo carga su posición de GM al tile y los hilos de los bordes cargan adicionalmente su correspondiente vecino que esta fuera del bloque, que sería el borde del tile. En este punto se usa `__syncthreads()`, que es una de las pocas funciones de CUDA que permiten algún tipo de sincronización entre hilos. Esta función en particular le indica a todos los hilos de un bloque que deben esperar a que todos los hilos lleguen a ese punto. De esta forma nos aseguramos que todos los hilos hayan cargado a share en su posición del *tile* antes de que se los lea para hacer los cálculos. El resto de la función es bastante similar a la implementación para GM, salvo que cuando se haría un llamado a algún valor de GM se usa el índice correspondiente y se lo llama del tile.

En cuanto a los nuevos valores y el contador de cambios, siguen escribiendo a GM, al igual que antes.

4.6.5. Texturas

Las memorias de Texturas son memorias especiales de solo lectura y todos los hilos de un bloque tienen acceso a ellas. Para pasar datos a ellas, se utilizó la interfaz provista por "CUB - Texture Iterator", 2020. Esta clase provee el método `BindTextures` con el cual podemos pasar datos a memoria de texturas; para limpiar esa sección de memoria y liberar memoria de texturas se usa `UnbindTexture`.

En primer lugar, tras inicializar las matrices de constantes, se crearon las correspondientes matrices en texturas, de esta forma las tenemos a todas en memoria de rápido acceso y solo lectura, ya que sus valores son constantes a lo largo de toda la simulación.

En cuanto a las funciones de cálculo, su comportamiento es análogo al de la implementación para GM, con la diferencia que las matrices de constantes son de tipo texturas. Respecto a las matrices *mx_r, mx_b, my_r, my_b, mz_r* y *mz_b*, en una primera implementación se las dejó como arreglos de `float`. Luego se las cambió a textura, pero para esto había que llamar a un `BindTextures` para cada matriz antes de las llamadas a Update y `UnbindTexture` después de las mismas. De esta forma, podíamos leer los valores viejos de texturas y escribir en las matrices auxiliares los nuevos valores.

4.7. Representación Gráfica

Uno de los aspectos de corrección que se tuvo en cuenta, fue la orientación de los spines una vez ordenados. Como se especifica en Billoni, 2016, a bajas temperaturas los spines se acomodan intercalando su signo en el eje X. De esta forma quedan todos los spines pertenecientes a una sub matriz (de las 2 medias matrices de Estrategia Red/Black) con valores en X positivos, y todos spines de la otra sub matriz con valores negativos en X.

Para observar esta condición y observar la gradual organización de toda la matriz se implementó una representación gráfica usando OpenGL ("Open Graphics Library", 2020). Para esto se tomó los valores de las matrices *mx, my* y *mz*, y con estos 3 se gráfica una flecha, desde el origen al punto *(x,y,z)*.

Para representar estas flechas se armo un modelo con 2 vértices. Uno que está siempre en el (0,0,0) siendo la base de la flecha, y otro que mueve a la posición *(x,y,z)* de cada celda de las matrices. Este segundo valor se actualiza tras cada Update, reflejando la nueva posición y cambiando de color según su orientación en X. Con este cambio

de color se puede observar a simple vista como las sub-matrices se ordenan.

Además de pasar la nueva posición del segundo vértice, para cada modelo de flecha se modifica la ubicación. Teniendo en cuenta que si miramos la pantalla de frente el eje X es horizontal con los valores positivos hacia la derecha; el eje Y es vertical con los valores positivos hacia arriba; y el eje Z es perpendicular a la pantalla con los valores positivos acercándose al observador. Las ubicaciones de definen según las 3 siguientes expresiones:

$$x = ixf - L$$
$$y = j - L$$
$$z = k - L$$

Donde *ixf, j* y *k* son las posiciones en la matriz completa. De esta forma se ubica cada flecha según la posición de la celda que representa de la matriz. Estando (0,0,0) estando lo mas alejado de la pantalla, a la izquierda abajo.; y (L,L,L) estando lo mas cerca a la derecha arriba.

Para aplicar al modelo de la flecha las diferentes modificaciones como traslación, rotación y estalación, se la multiplica por las respectivas matrices. Para hacer operaciones entre matrices, se usa la biblioteca GLM ("OpenGL MAthematics", 2020)

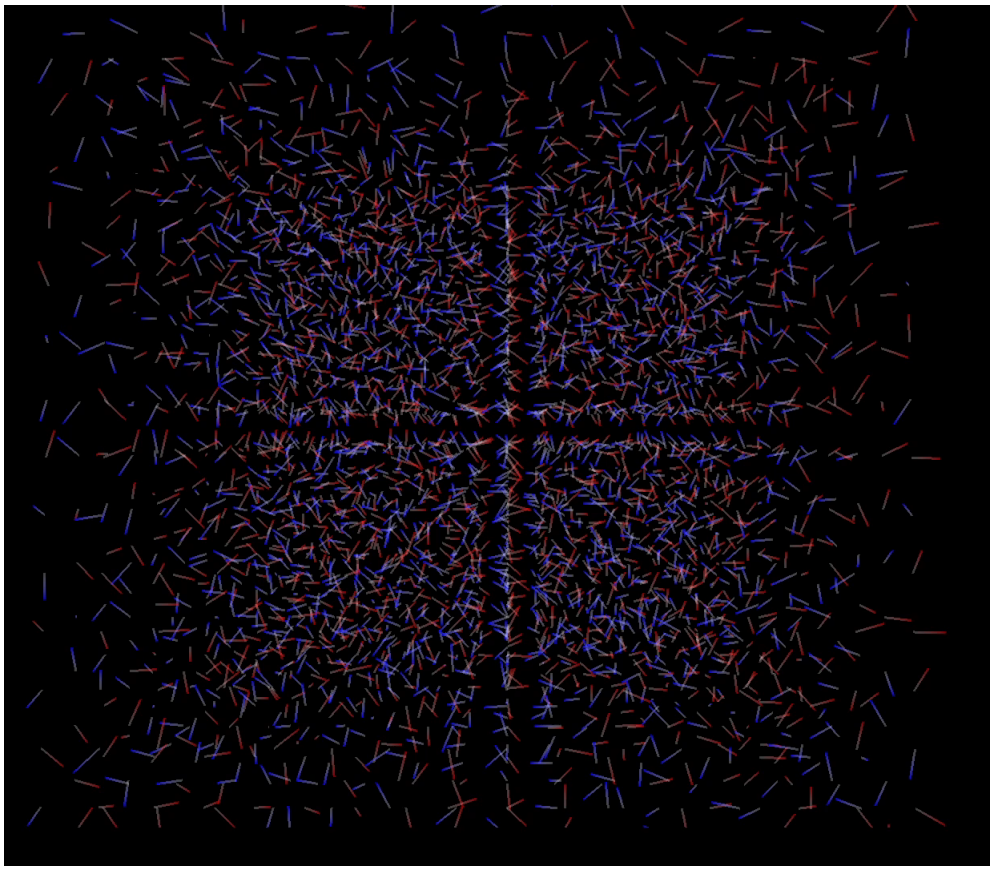


Figura 4.4: Visualización de red cubica de spines.

5 Mediciones y Comparaciones

En este Capítulo se contará sobre las diferentes mediciones que se usaron y como se desempeñan en estas cada una de las implementaciones.

Todas las mediciones fueron hechas sin la representación gráfica, ya que esta implica movimiento extra de datos y usar la GPU para graficar. Además todas las mediciones fueron sacadas de las últimas versiones de cada implementación.

Todas las pruebas se hicieron en tres sistemas. Uno es el servidor JupiterAce que cuenta con:

- CPU x2: Intel Xeon de 14 núcleos (28 hilos) ("Intel Xeon E5-2680v4", 2020)
- GPU: NVIDIA RTX 2080 Ti ("NVIDIA RTX 2080 Ti", 2020)

del cual se tomaron mediciones en la GPU.

El segundo es el servidor zx81 que cuenta con:

- CPU x2: Intel Xeon de 14 núcleos (28 hilos)("Intel Xeon E5-2680v4", 2020)
- GPU: NVIDIA GTX 1070 Ti ("NVIDIA GTX 1070 Ti", 2020)

del cual solo se tomaron mediciones en CPU.

Y el tercer sistema cuenta con:

- CPU: AMD Ryzen 5 - 6 núcleos (12 hilos) ("AMD Ryzen 5 - 2600X", 2020)
- GPU: NVIDIA GTX 1660 Ti ("NVIDIA GTX 1660 Ti", 2020)

del cual se tomaron mediciones tanto en GPU como CPU. Todas estas fueron

CPU/GPU	Fecha de Lanza-miento	#Núcleos-#Hilos	Pico Teórico GFLOPS float32	Pico Teórico Ancho de Banda (BW)
2080 Ti	Q3'18	4352	13450	616GB/s
1660 Ti	Q1'19	1536	5437	288GB/s
1070 Ti	Q4'17	2432	8186	256GB/s
E5-2680v4	Q1'16	14 - 28	850	63 GB/s
Ryzen5 2600x	Q2'18	6 - 12	316	43 GB/s

Para asegurar la corrección de la simulación se tuvieron en cuenta varias métricas que fueron comparadas tanto con pruebas teóricas como con las mismas métricas pero de la simulación original en Fortran cuya corrección ya estaba verificada.

Entre estas se tienen en cuenta:

- Cambios aceptados: se verificaba que para altas temperaturas se aceptaran entre un 50 % y un 80 %, dependiendo de los parámetros usado, y a medida que baja la temperatura, el sistema se estabiliza y se aceptan cada vez menos cambios llegando a 0.01 % o menos.

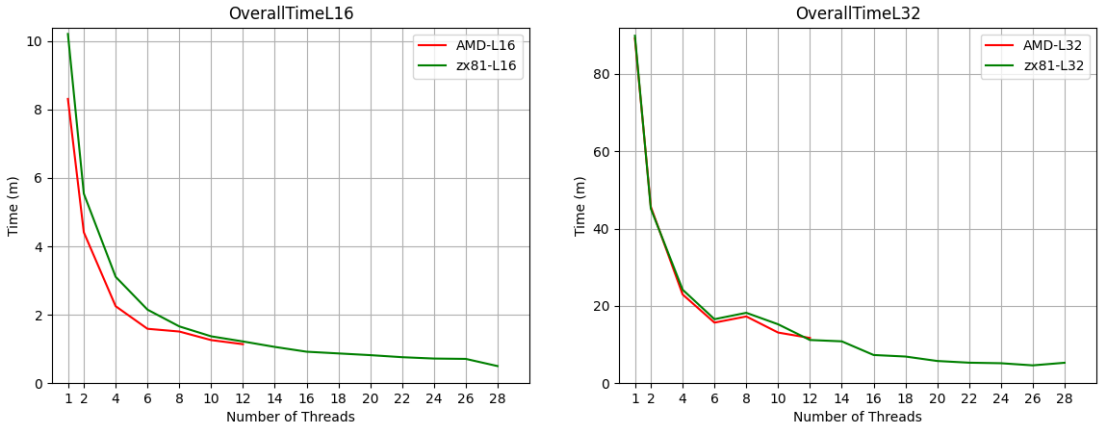
- Magnetización: usando Montecarlo se hace un promedio de la magnetización en cada subred en función de la temperatura. Al finalizar la simulación se grafican las mediciones de magnetización que se tomaron, contra las temperaturas correspondientes, y se observa que la magnetización incremente tendiendo a 1. La temperatura a la cual comienza el incremento depende de los parámetros que se usen para la simulación.

- Energía: con los cálculos que se hacen en la función de energía se pueden verificar que los 3 tipos de interacciones estén correctamente calculados.

En cuanto a las mediciones de performance, por un lado se mide el tiempo total de la simulación. Esto toma en cuenta las inicializaciones,

los llamados a Update y los llamados tanto a calculate como energy. Siendo estas últimas 3 funciones las que mas impactan en performance.

Por otro lado, se mide cuanto se tarda en actualizar cada vector de spin, esto es, los 3 valores *(x,y,z)*, que definen el vector de cada spin. Para esto se mide cuanto tarda la función update en calcular los nuevos valores para las 3 matrices, y se lo normaliza dividiendo por N. Para un valor dado de temperatura se hacen $t\_max \times 2$ updates. Las mediciones de estos updates son promediadas, y al final de la simulación se hace un promedio de estos promedios. De esta forma se obtiene un estimativo de cuanto tarda en actualizarse cada spin.



(a) Matriz entra en caché.

(b) Matriz excede la caché.

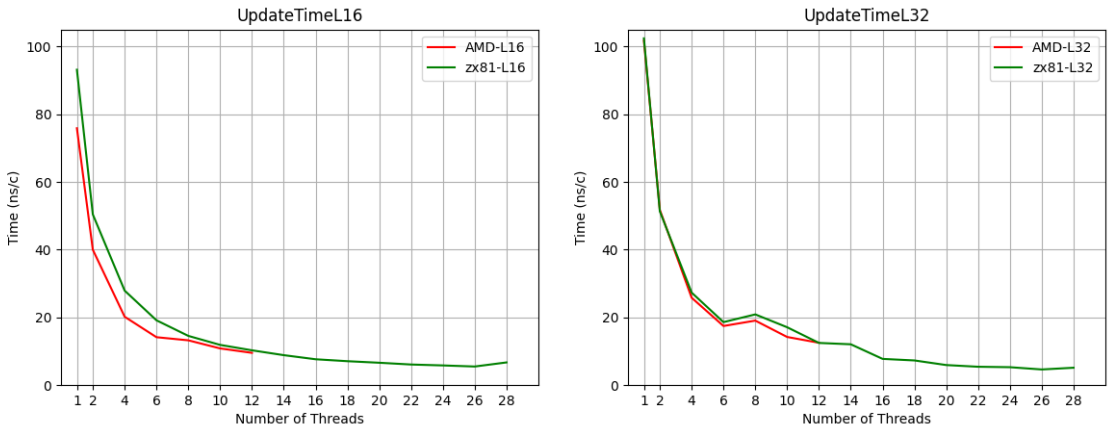
Figura 5.1: Tiempo total (en minutos) con matriz de tamaño fijo aumentando cantidad de hilos.

Las primeras comparaciones que veremos compara el desempeño en el Ryzen5 contra el Xeon. Las Fig (5.1) son en base a mediciones en CPU, con matriz de tamaño fijo. En (a) tenemos matriz con $L = 16 \times 2$ y por lo tanto $N = 32768$; en (b) con $L = 32 \times 2$ y por lo tanto $N = 262144$. Como se esperaba, podemos ver que al agregar unidades de procesamiento los tiempos disminuyen.

Ambas figuras muestran mediciones de tiempo total de la simulación, pero como se puede ver hay una magnitud de diferencia entre (a) y (b). En (a), para 1 solo hilo toma 10,2 y 8,3 minutos respectivamente, mientras que en (b), también para 1 solo hilo toma poco más de 89 minutos en ambos CPUs. Similarmente cuando se usan los 12 o 28 hilos (el máximo de hilos de cada CPU) podemos ver que (a) llega hasta 1.14 y 0.5 minutos, mientras que en (b) llega hasta 11.69 y 5.28 minutos.

Por otro lado si observamos el tiempo de actualización de cada celda en las mismas corridas, mostrados en Fig (5.2), podemos ver que

la diferencia no es tan amplia como en los tiempos totales de simulación. Esta medición se muestra en nanosegundos/celda, esto quiere decir cuantos nanosegundos toma cargar los valores viejos, realizar los cálculos necesarios y escribir los nuevos valores a memoria.



(a) matriz entra en caché.

(b) Matriz excede caché.

Figura 5.2: Tiempo de actualización (en nanosegundos/celda) con matriz de tamaño fijo aumentando cantidad de hilos.

En estos casos, a diferencia de Fig (5.1), los resultados obtenidos con un tamaño de matriz no están tan alejados de los resultados del otro tamaño. Para 1 hilo con L16 los promedios alcanzan 75.8 ns/c y 93.11 ns/c, mientras que para el máximo de hilos alcanza 9.54 ns/c y 6.72 ns/c. Para L32, los promedios en 1 hilo alcanzan 101 ns/c y 102 ns/c, mientras que para el máximo hilos alcanza 15.52 ns/c y 5.15 ns/c.

Aunque en Fig. (5.1) parece que ambas figuras son muy similares, en Fig. (5.3) podemos ver la diferencia para cada CPU entre ambos tamaños. Notese que con mas de 20 hilos se consigue incluso una mejora en los tiempos de actualización.

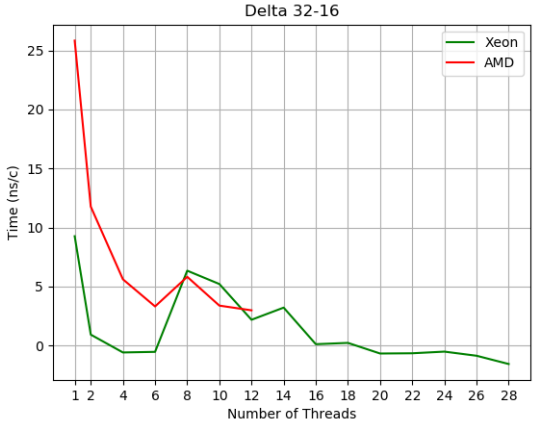
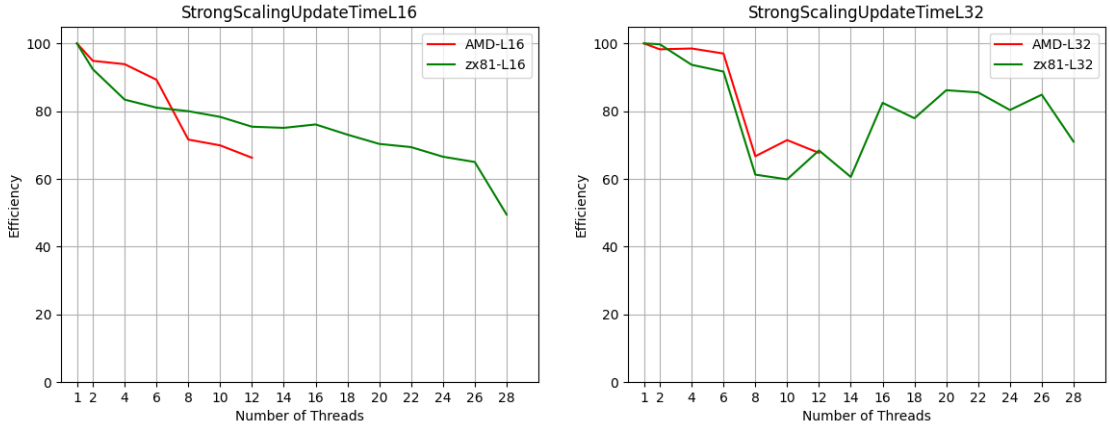


Figura 5.3: Delta entre matriz en cache vs excede

Otra métrica a tener en cuenta es la *eficiencia de utilización*. Esta refleja el aprovechamiento de las unidades de ejecución a medida que estas se incrementan, donde 100 refleja n núcleos, n veces mejora. Esto lo podemos ver en Fig. (5.4), que muestra como evoluciona el tiempo de actualización en relación al incremento en cantidad de hilos en matrices con $L = 16 * 2$ y $L = 32 * 2$. En ambas figuras podemos ver una fuerte caída de eficiencia a los 28 hilos, esto se debe al overhead de la comunicación y sincronización.

Como bien se puede observar en Fig. (5.4b) superados los 14 hilos se obtiene una muy buena eficiencia superando el 80 %, mostrando que para obtener una mejor utilización de los recursos de memoria, hacen falta grandes cantidades de datos. Además a partir de los 14 hilos se activa el ancho de banda del segundo chip del Xeon, esto impulsa la subida y mantiene una buena eficiencia hasta los 26 hilos.



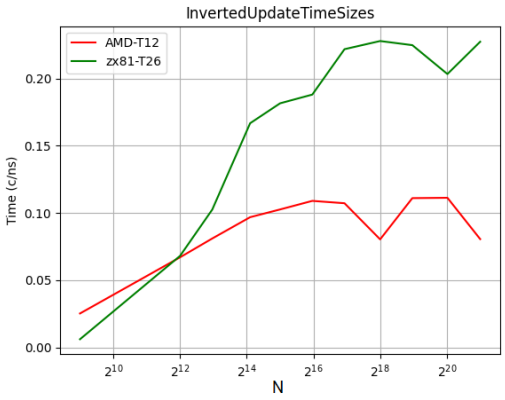
(a) matriz entra en caché. (b) Matriz excede caché.

Figura 5.4: Escalado Fuerte.

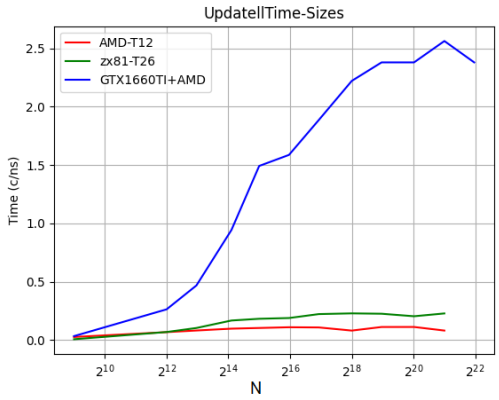
Aunque el tener 2 chips habilite una mejora por el incremento en el ancho de banda esto podría tener como consecuencia una mala ubicación de los hilos (7 en el socket 0 y 7 en el socket 1). Cuando la matriz excede la cache podemos observar una fuerte caída de la eficiencia en AMD por el HyperThreading.

Dado que en las diferentes implementaciones estos tiempos tienden a 0, para apreciar apropiadamente los saltos de performance, se los invierte. En las figuras con los valores invertidos veremos como evolucionan las métricas a medida que se incrementa N, en eje X. En el eje Y, ya no se mostrará ns/c, sino al inverso, c/ns. Esto representa cuantas celdas se actualizan en 1 nanosegundo. Para estas mediciones se fijó la cantidad de hilos que corre cada CPU[1], donde mejor performance presentaba, y se fue aumentando el tamaño de la matriz.

En Fig. (5.5a) podemos ver como en matrices con N mayor a 2048 (esto se considera una matriz pequeña), los 26 hilos del Xeon superan ampliamente a los 12 hilos del Ryzen 5, por aproximadamente el doble. Con las mejores mediciones de CPU estando en N superior a 512000. Siendo 8.99 ns/c para el Ryzen5 y 4.39 ns/c para el Xeon.



(a) Tiempos de actualización invertidos en CPU.



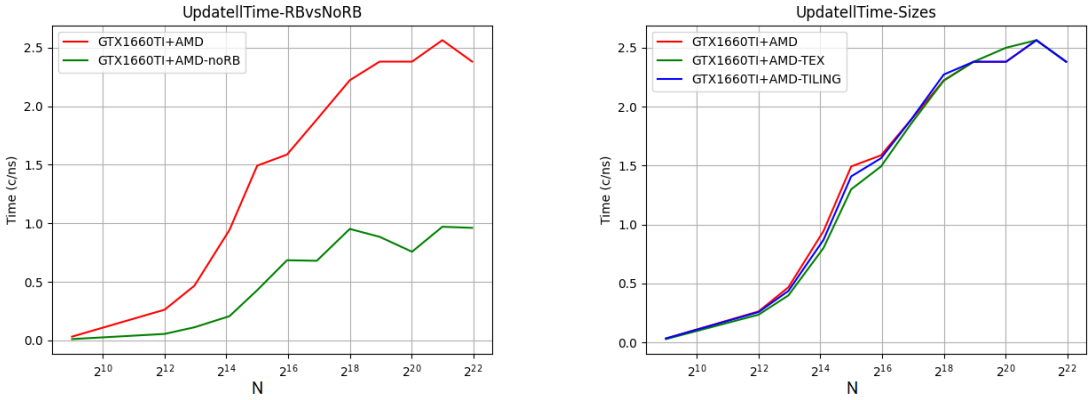
(b) CPU vs GPU.

En Fig. (5.5b) tenemos la primera comparación entre CPU y GPU. Muestra una gran mejora en performance pasando de los las casi 0.25 c/ns a más de 2.5 c/ns en el mejor caso, esto es 0.44 ns/c para el Ryzen5 con la GTX 1660 Ti, una mejora de casi 10 veces. Además, gracias a las capacidades de la GPU se pudo probar en tamaños mas grandes, llegando a matrices de hasta 4 millones de elementos ($L = 80 * 2$).

[1]Para zx81 se uso 26 y no 28, porque en algunas simulaciones los resultados fueron inestables debido a comunicación y sincronización.

Estos valores son, en principio, mas bajos que los resultados obtenidos en M. Bernaschi, 2011, siendo los mejores resultados 3.5 ns/c en un Intel X5680 y 0.63 ns/c en una GTX 480 . Aunque si uno compara el hardware usado en sus pruebas, podría esperar mejores resultados considerando los saltos que ha dado la tecnología en los últimos años. Pero dado que su investigación es en base a un modelo distinto, con una distribución distinta, en su programa necesitan menos cálculos y cargar menos datos desde memoria para actualizar un espin. Sufriendo mucho los limites del ancho de banda.

Por otro lado en Fig. (5.6a) tenemos una comparación entre 2 corridas de GPU. Esta vez ambas corridas son en el mismo hardware, y lo que cambia es la implementación. En verde tenemos la implementación Red/Black unificado y en rojo División de matrices. En esta figura podemos apreciar como al separar las matrices y leer o escribir espacios de memoria continuas trae una considerable mejora en performance, alcanzando en el mejor caso mas de un 250 % de mejora.



(a) RB unificada vs RB dividida

(b) Unified Cache

Figura 5.6: Tiempos de actualización en GPU.

En Fig. (5.6b) tenemos otra comparación entre las diferentes implementaciones para GPU. Esta vez comparamos los diferentes tipos de memorias que podemos usar. Como se puede observar sus desempeños son muy similares. Esto se condice con lo planteado en la sección Unified Cache, mostrando que aunque sean diferentes tipos de memorias, *share* o *texturas*, tienen la misma velocidad que la L1 usada cuando tratamos directamente con GM.

Por ultimo podemos ver en Fig. (5.7) las mejores métricas de performance obtenidas. Estas fueron usando el poder de la RTX 2080 Ti. Como se puede observar en esta plataforma se llega a calcular mas de 5 c/ns (esto es 0.18 ns/c) con $N = 4194304$. Siendo esto poco mas del

doble de la performance obtenida con la 1660Ti.

Si consideramos que tiene casi el triple de unidades de ejecución (1536 cuda cores en la 1660Ti y 4352 en la 2080Ti), nos da un indicio de que el limitante en este problema no es el poder de cálculo. Pero si observamos el pico de ancho de banda en cada placa, 288 GB/s en la 1660 y 616 GB/s en la 2080, vemos que estos valores tiene una relación proporcional a la diferencia en las curvas.

Otra cosa que podemos notar es que ambas placas tienen una performance bastante similar por abajo de los 32768 elementos. Esto es por que la 2080Ti tiene una Cache L2 de 5.5 MB, mientras que la 1660Ti solo de 1.5 MB. Entonces al tener mas de 32768 elementos en la matriz, esta ya no cabe en la cache de la 1660Ti y hay que volver a buscar algunos datos en memoria. Por eso se nota el cambio en la curva luego de este tamaño. Por otro lado en la 2080Ti, esto sucede a partir de los 262,144 elementos, donde notamos el cambio en la curva correspondiente a esta GPU.

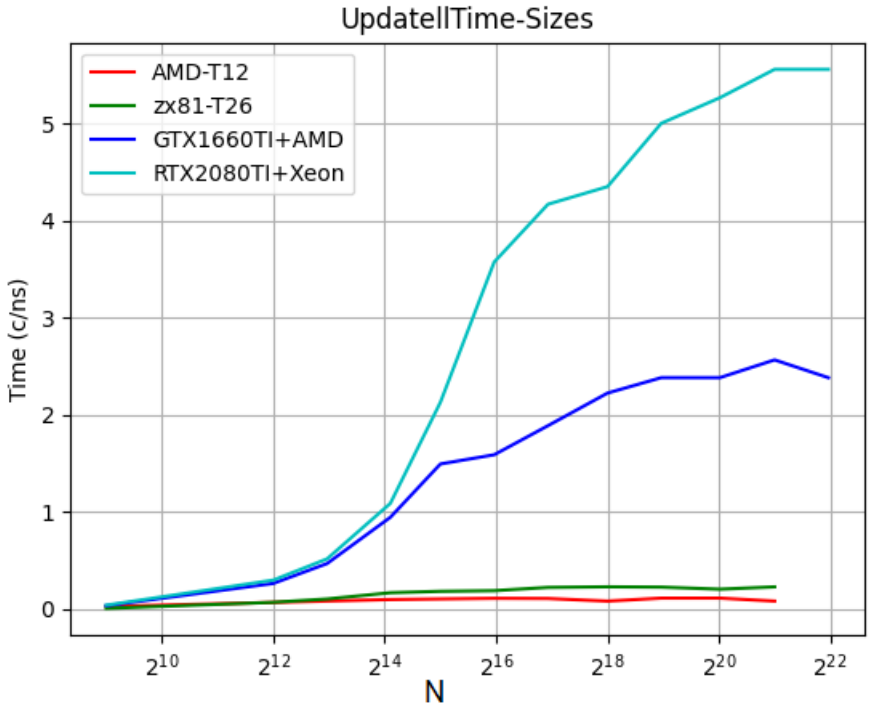


Figura 5.7: Tiempo de actualización en CPU[a] y GPU

[a]Las corridas con CPU se realizaron solo hasta $N = 2097152$ y no se probaron con el mayor tamaño por cuestiones de tiempo.

6 Conclusión y Trabajos Futuros

A lo largo de este trabajo se presentaron diferentes optimizaciones para una implementación del modelo de Heisenberg utilizando una simulación con el método de Monte Carlo. Para su optimización, se utilizó paralelismo en hilos a pequeña escala para CPUs y el paralelismo masivo presentado por las GPUs.

Para CPUs se uso OMP. Los `pragmas` más importantes fueron `parallel`, `for`, `reduce`, `collapse`, `share` y `private`. Para estos casos es mejor no utilizar `atomic`, ya que los hilos terminan perdiendo mucho tiempo en la sincronización para acceder a estas operaciones. En cambio se vió un mejor resultado al organizar los hilos para que escriban en diferentes partes de la memoria, así evitándose la necesidad de `atomic`, y luego utilizar un `reduce`. Además, darle más de una celda para calcular a cada hilo, minimizando llamados a funciones en común, aportó una notable mejoría.

En las GPUs se usó CUDA. Uno de los factores principales a tener en cuenta para sacarle el mayor provecho, es buscar la mejor combinación de tamaños de bloques, en este caso la mejor fue $16 \times 2 \times 1$ hilos por bloque. Es de suma importancia coordinar los hilos para que lean o escriban espacios de memoria contiguos, permitiendo hacer menos movimientos de memoria y asi aprovechar todo el ancho de la GDDR5.

Se realizaron optimizaciones que tuvieron un impacto positivo en ambas plataformas. Como bien se ve en Fig. (5.6a) el dividir las matrices en dos sub matrices y realizar las lecturas o escrituras siempre de la misma matriz (y no intercalando) fue una de las estrategias con el mayor impacto. Esto implicó dividir no solo las matrices *mx, my* y *mz*, sino también todas las matrices de constantes. Usar funciones que realicen conversión implícita o explícita a `double` cuando solo se usan `floats`, trae grandes deterioros en performance y es mejor buscar alternativas.

Del tiempo total de simulación, en las primeras versiones no optimizadas, el tiempo ocupado por la función Update era aproximadamente del 98 %. Esto se logró reducir hasta un 60 %, llevando el tiempo de actualización por celda de mas de 1500 ns/c a 0.18 ns/c.

Previo a las optimizaciones una corrida típica con $L = 20 \times 2$, o sea $N = 64000$, con $tmax = 100000$ y $tmic = 100$, o sea, actualizando $tmax \times 2$ veces mx, my y mz, y tomando mediciones de temperatura y energía cada 100 actualizaciones, tardaba entre 6 y 8 horas, en un i7 980 ("Intel Core i7 980", 2011) consumiendo aproximadamente 130W del CPU, sumándole aproximadamente 100Wh de las otras partes del sistema (placa madre, RAM, discos, etc), podriamos decir que el consumo energético por simulación es $230Wh \times 6 = 1380Wh$ tomando consumo pico sostenido. Tras las optimizaciones, con los mismos parámetros tarda poco mas de 13 minutos en la 2080 Ti. Durante esa simulación, la 2080 Ti consume aproximadamente 150Wh, sumándole aproximadamente 200Wh por las otras partes que necesita el sistema, entonces podríamos decir que el consumo energético de la simulación es $350 \times (13/60) = 75{,}83Wh$.

Con esta comparación podemos ver que gracias a estas optimizaciones se pueden lograr simulaciones precisas y con mayor numero de elementos, en un 3.6 % del tiempo y consumiendo un 4.5 % de la energía. Teniendo en cuenta que estos mediciones no deberían ser comparables ni en esfuerzo de optimización, ni en hardware, ni en plataformas, el objetivo principal de este trabajo es darle a un científico una herramienta que es 27x más rápida y 18x mas eficiente energéticamente, mejorando así su herramienta de trabajo y, esperamos, su forma de hacer ciencia.

6.1. Trabajos Futuros

Si alguien deseara profundizar sobre este trabajo algunas ideas o campos a explorar podrían ser los siguientes:

- Se puede profundizar sobre la visualización. Optimizando más todavía el manejo de datos se podrían visualizar matrices de mayor tamaño a mejores cuadros por segundo. También se podrían agregar diferentes "filtros", para ir viendo variaciones en temperatura o energía de cada espin en tiempo real.

- Se encontraron pocas investigaciones realizando implementaciones para TensorCores o similares, todas con la primera generación de estos. Dado que las nuevas generaciones de GPUs prometen mejoras en estos chips seria interesante una implementación para ver su desempeño en este tipo de problemas.

- Analizar la posibilidad de comprimir o unir ciertas matrices para minimizar la cantidad de datos que se necesitan cargar sin afectar a la funcionalidad ni precisión de los algoritmos.
- Considerar una implementación multi-GPU, sub-diviendo las matrices en franjas o slabs teniendo consideraciones especiales para los datos de los bordes.

Bibliografía

AMD Ryzen 5 - 2600X. (2020). https://www.amd.com/en/products/cpu/amd-ryzen-5-2600x

Amdahl, G. M. (1967). Validity of the single processor approach to achieving larg escale computing capabilities. *AFIPS Conference Proceedings*, *30*, 483-485.

Billoni, O. V. (2016). Magnetization reversal in mixed ferrite-chromite perovskites with non magnetic cation on the A-site. *Journal of Physics: Condensed Matter*, *28*(47).

CUB. (2020). https://nvlabs.github.io/cub/

CUB - Texture Iterator. (2020). https://nvlabs.github.io/cub/classcub_1_1_tex_obj_input_iterator.html

Harris, M. (2013). *Unified Memory in CUDA*. https://developer.nvidia.com/blog/unified-memory-in-cuda-6/

Igor Solovyev, N. H. & Terakura, K. (1996). Crucial Role of the Lattice Distortion in the Magnetism of LaMnO3. *Physical Review Letters*, *76*, 4825-.

Intel Core i7 980. (2011). https://ark.intel.com/content/www/es/es/ark/products/58664/intel-core-i7-980-processor-12m-cache-3-33-ghz-4-8-gt-s-intel-qpi.html

Intel Xeon E5-2680v4. (2020). https://ark.intel.com/content/www/us/en/ark/products/91754/intel-xeon-processor-e5-2680-v4-35m-cache-2-40-ghz.html

M. Bernaschi, L. P., G. Parisi. (2011). Benchmarking GPU and CPU codes for Heisenberg spin glass over-relaxation. *Computer Physics Communications*, *182*(6), 1265-1271.

N Dasari, A. S., P. Mandal & Vidhyadhiraja, N. S. (2012). Weak ferromagnetism and magnetization reversal in YFe1-xCrxO3. *Europhysics Letters*, *99*(1).

NVIDIA GTX 1070 Ti. (2020). https://www.nvidia.com/es-la/geforce/products/10series/geforce-gtx-1070-ti/

NVIDIA GTX 1660 Ti. (2020). https://www.nvidia.com/es-la/geforce/graphics-cards/gtx-1660-ti/

NVIDIA RTX 2080 Ti. (2020). https://www.nvidia.com/en-us/geforce/graphics-cards/rtx-2080-ti/

Open Graphics Library. (2020). https://www.opengl.org//

OpenGL MAthematics. (2020). https://glm.g-truc.net/0.9.9/index.html

OpenMP. (2020). https://www.openmp.org/

Random123: a Library of Counter-Based Random Number Generators. (2016). http://www.thesalmons.org/john/random123/releases/latest/docs/index.html

Tina's Random Number Generator Library. (2020). https://numbercrunch.de/trng/

Tobias Preis, W. P., Peter Virnau & Schneider, J. (2009). GPU accelerated Monte Carlo simulation of the 2D and 3D Ising model. *Journal of Computational Physics*, *228*(12), 4468-4477.

Turing Tunning Guide. (2020). https://docs.nvidia.com/cuda/turing-tuning-guide/index.html