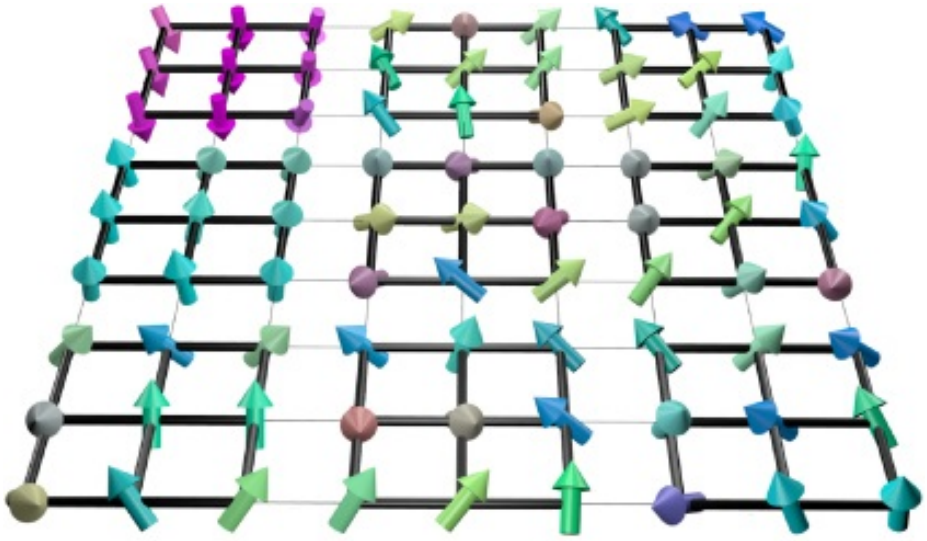
Simulación de Monte Carlo paralela para sistemas ferromagnéticos en un modelo de Heisenberg incluyendo interacciones dipolares

Trabajo de titulación

Universidad de Santiago de Chile
Trabajo de Titulación
Profesor guía: Fernando Rannou
Alumno: Sebastián Vizcay

sebastian.vizcay@usach.cl



Paralelización de simulación de Monte Carlo

Tabla de contenidos

- Introducción.
- Marco teórico.
- Arquitecturas y modelos de computación paralela.
- Trabajo realizado.
- Experimentos y resultados.
- Conclusiones.

Introducción

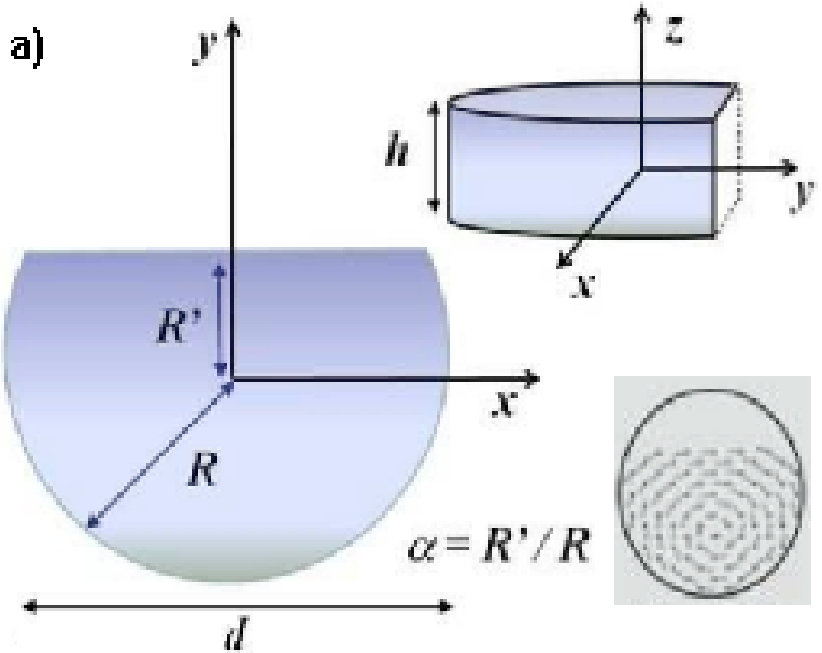
Introducción

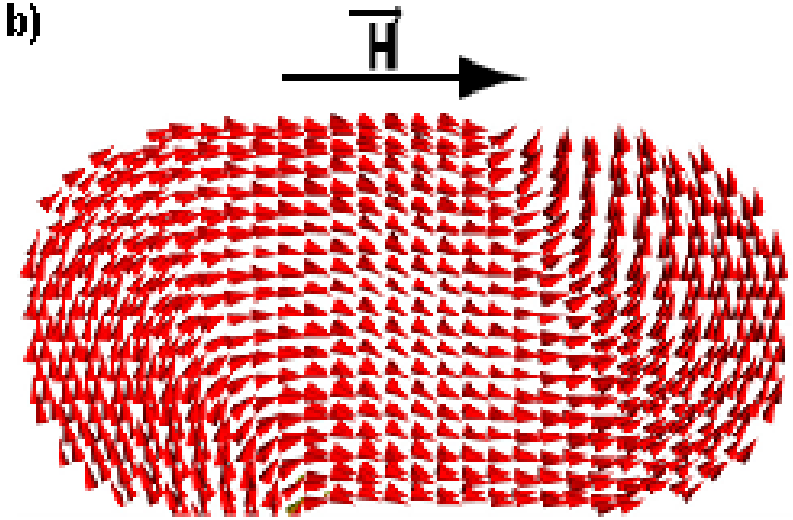
Antecedentes y motivación

- Avances en la ciencia han permitido la sintetización de nanopartículas (otorgándole diversas formas y tamaños).
- Propiedades físicas de nanoestructuras difieren de las macroscópias, las cuales son bien conocidas.
- Estructuras ferromagnéticas a escala nanométrica presentan propiedades tales como: magneto resistencia gigante, efecto Hall gigante, entre otras.
- Interés en el estudio de estas propiedades emergentes.
- Propiedades se encuentran determinadas por la geometría, tamaño y material.
- Búsqueda de nuevas geometrías con aplicaciones interesantes o con fines teóricos.

Introducción

Geometrías convencionales

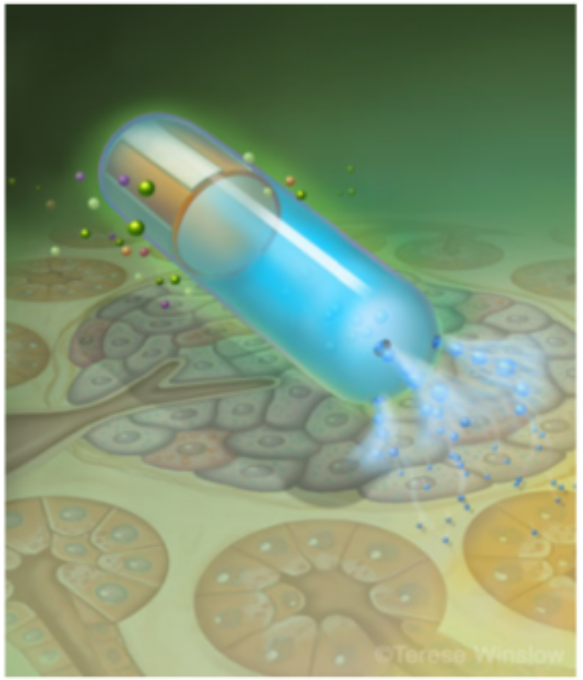




Introducción

Antecedentes y motivación

- Investigadores del Depto de Física de la Universidad de Santiago en conjunto con CEDENNA realizan estudio sistemático de estas estructuras a través de simulaciones de Monte Carlo.
- Aplicaciones van desde almacenamiento de información de alta densidad hasta portadores para entrega de drogras.



Introducción

Problemas

- Método de Monte Carlo es el método clásico en la Física Estadística para evaluar observables.
- Poder de cómputo actual no hace factible simular geometrías con más de 10^9 momentos magnéticos.
- Se utiliza el método *Fast Monte Carlo* para simular sistemas escalados.
- Tiempo de ejecución todavía es demasiado (orden de dos o tres semanas para 3.000 momentos magnéticos).
- Si se acelera el proceso, se podría estudiar sistemas más grandes o simplemente reducir los tiempos de simulación para proceder a analizar si la geometría es útil o no.

Introducción

Solución y objetivo

- Se propone el diseño e implementación de un programa paralelo equivalente al secuencial actual que reduzca los tiempos de ejecución.

Objetivos específicos

- Determinar método de paralelismo (GPU/CPU).
- Implementar software.
- Validar resultados (que sean correctos).
- Proponer experimentos para evaluar rendimiento computacional.
- Realizar las pruebas de ejecución.

Marco teórico

Marco teórico

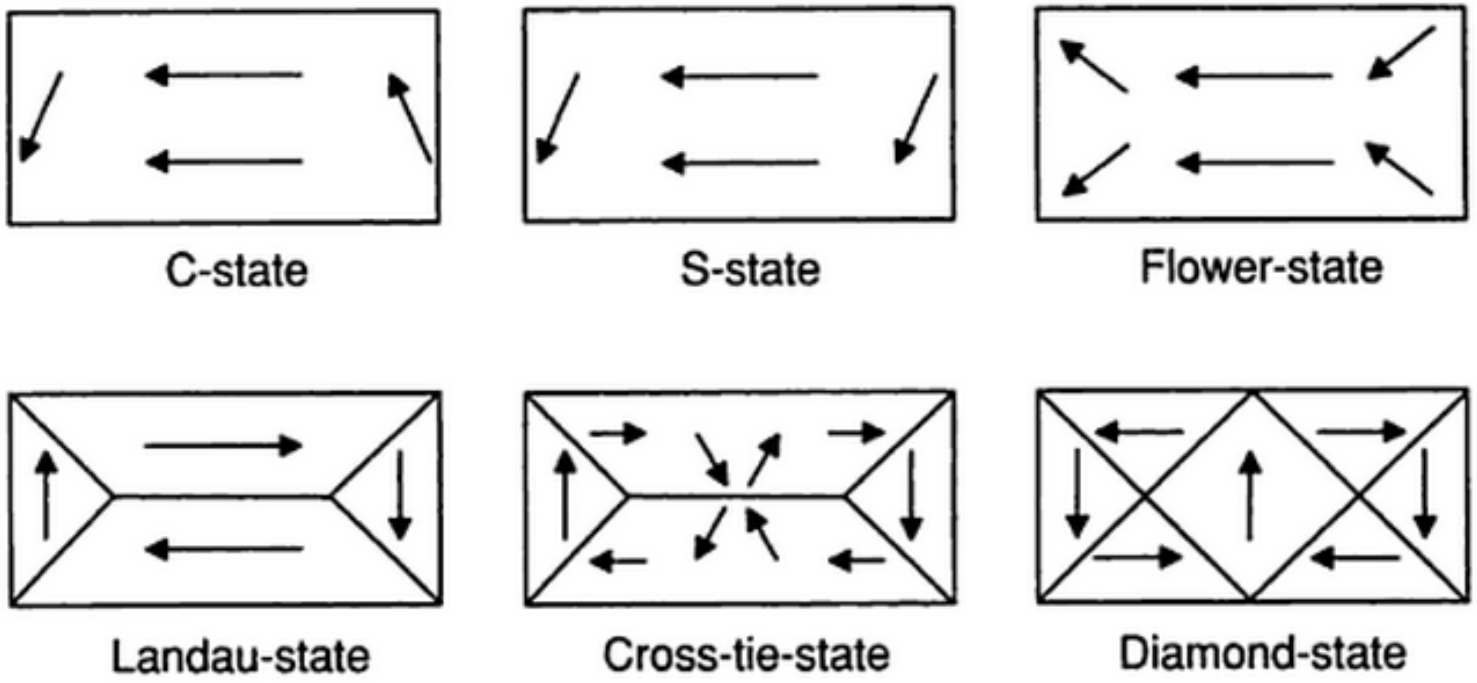
Modelos de espines

- Momento magnético: vector compuesto del momento angular del átomo y de su momento angular intrínseco o espín.
- Materiales pueden ser clasificados acorde a como sus espines reaccionan a un campo magnético externo.
- Materiales ferromagnéticos quedan permanentemente magnetizados al ser expuestos.

Marco teórico

Modelos de espines - Estados fundamentales

- Modificación de las estructuras ferromagnéticas, como respuesta a un campo externo, genera diferentes estados.

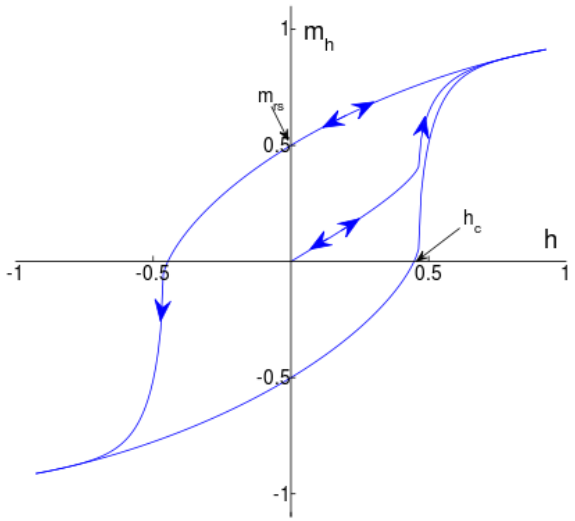


Marco teórico

Modelos de espines - Histéresis (modo de reversión)

- Forma en que el sistema recorre diferentes valores de campo desde la saturación en una dirección opuesta.

- Coercitividad: intensidad necesaria para eliminar magnetización.
- Magnetización de remanencia.



Marco teórico

Modelos de espines

- Uso de modelos de grilla de espines (*lattice spin models*).
- Modelos son descritos a través de un Hamiltoniano (función que determina la cantidad de energía del sistema).
- En este trabajo se utiliza el modelo de Heisenberg.

Marco teórico

Modelos de espines - Energías

- Energía Dipolar.
- Energía de Intercambio (vecinos).
- Energía de Zeeman (campo externo y espín).
- Energía de anisotropía (dirección preferencial).

$$H = \frac{1}{2}\sum_{i \neq j}[E_{ij}^{dip} - E_{ij}^{ex}] + \sum_{i} E_i^k + \sum_{i} E_i^z \quad (1)$$

Marco teórico

Simulaciones de Monte Carlo

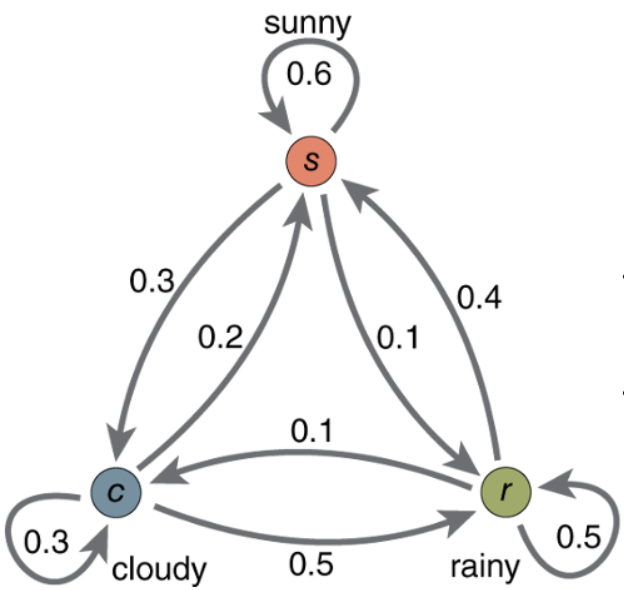
- Simulaciones de los modelos anteriores son realizadas a través del método de Monte Carlo.
- Técnica matemática computarizada que hace uso de números aleatorios para simular procesos estocásticos.
- Trayectoria estocástica en el espacio de fase.

$$X_{MC} = \frac{1}{M} \sum_{i=1}^{M} f(x_i) \tag{2}$$

Marco teórico

Markov Chain Monte Carlo

- Uso de cadenas de Markov para samplear estados con mayor probabilidad.
- En base a si se cumplen ciertas propiedades, la cadena converge a una distribución estacionaria.



weather tomorrow

weather today	s	c	r
s	0.6 (1,1)	0.3 (1,2)	0.1 (1,3)
c	0.2 (2,1)	0.3 (2,2)	0.5 (2,3)
r	0.4 (3,1)	0.1 (3,2)	0.5 (3,3)

Marco teórico

Algoritmo de Metrópolis

- Es un caso particular de *Markov Chain* Monte Carlo en donde se descompone la probabilidad de transición en una probabilidad de selección un estado y luego aceptar tal transación.
- Lo anterior permite realizar una simplificación en donde se elimina la constante de normalización (la cual no puede ser calculada debido a la infinidad de estados posibles).

Algoritmo 2.1: Algoritmo de Metropolis

1: **for** $k < nrIteraciones$ **do**
2: elegir un espín random s_i
3: considerar s_i^v = nueva orientacion random
4: calcular $P = \min\{1, e^{-\beta(E_v - E_\mu)}\}$
5: generar numero random r uniforme entre [0,1)
6: **if** $r < P$ **then**
7: aceptar nueva configuracion s_i^v
8: **else**
9: mantener configuracion s_i^μ
10: **end if**
11: **end for**

Marco teórico

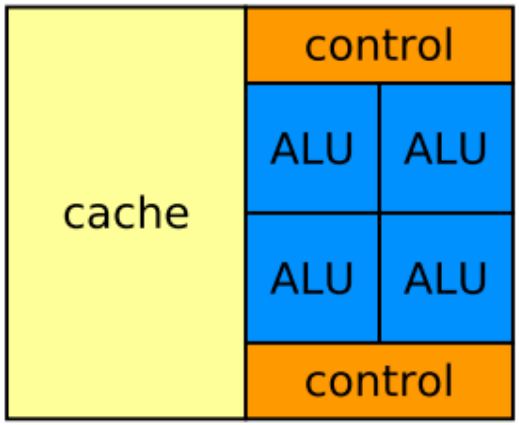
Técnicas para acelerar Monte Carlo

- Métodos de clustering: Swendsen-Wang, Wolff (utilizados en sistemas con temperatura cercana a la temperatura crítica).
- Parallel tempering (utilizado en sistemas en criticalidad y en modelo *Spin Glass*).
- Descomposición en tablero de ajedrez (no considera dipolar ni respeta el balance detallado).
- Se opta por mantener el método de actualización secuencial (*single spin flip*).
- Se optimiza el cálculo del Hamiltoniano, calculando el delta de energía solamente en base al espín seleccionado de forma aleatoria.

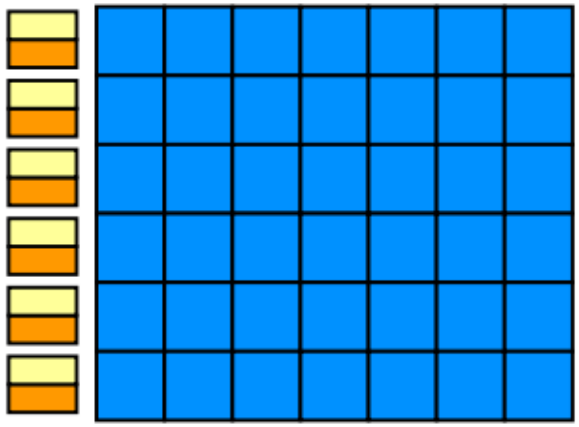
Arquitecturas y modelos de computación paralela

Programación en GPU - Intro

- Historia de la GPU (ley de Moore). Aumento del número de unidades de procesamiento.
- Alternativa económica.
- Computación en CPU (orientados a ocultar la latencia) vs GPU (orientados al *throughput*).



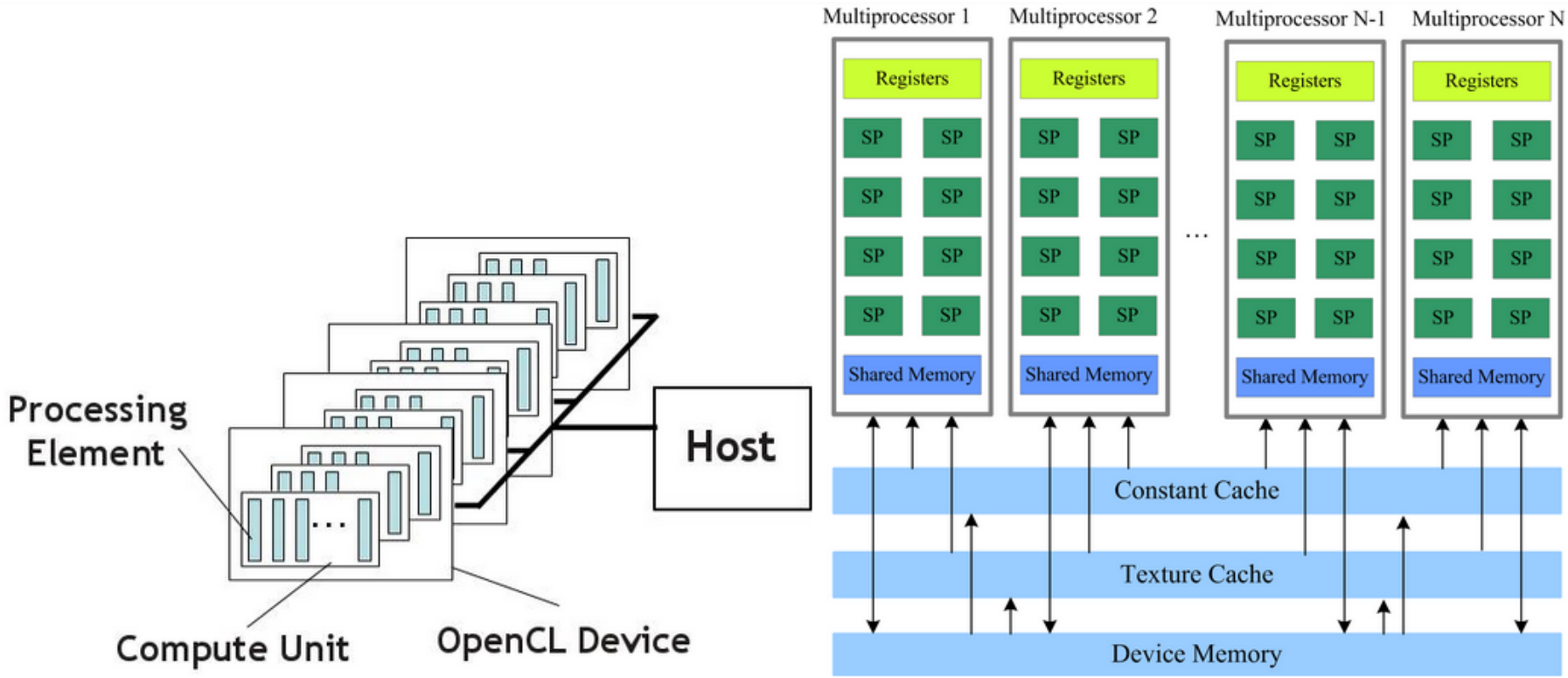
CPU



GPU

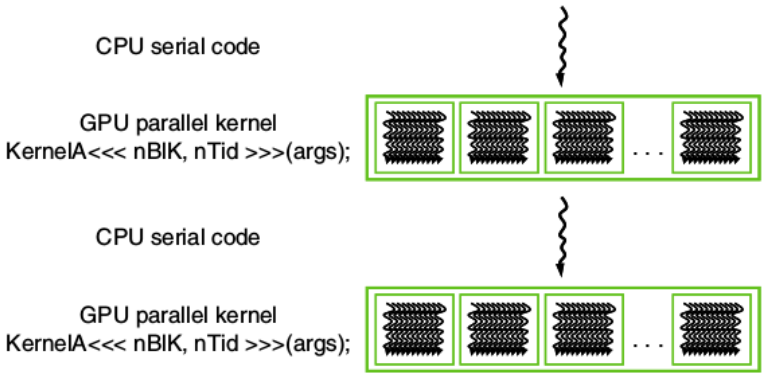
Arquitecturas y modelos de computación paralela

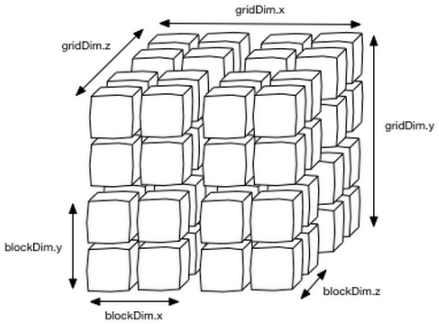
Programación en GPU - Modelo del dispositivo

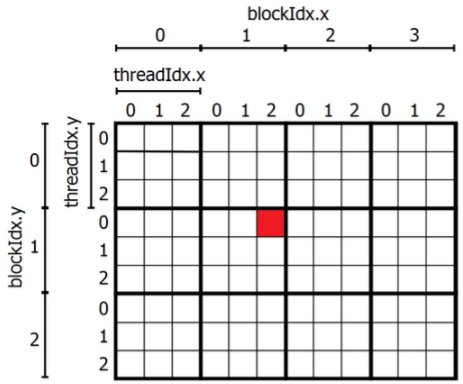


Arquitecturas y modelos de computación paralela

Programación en GPU - Modelo de ejecución

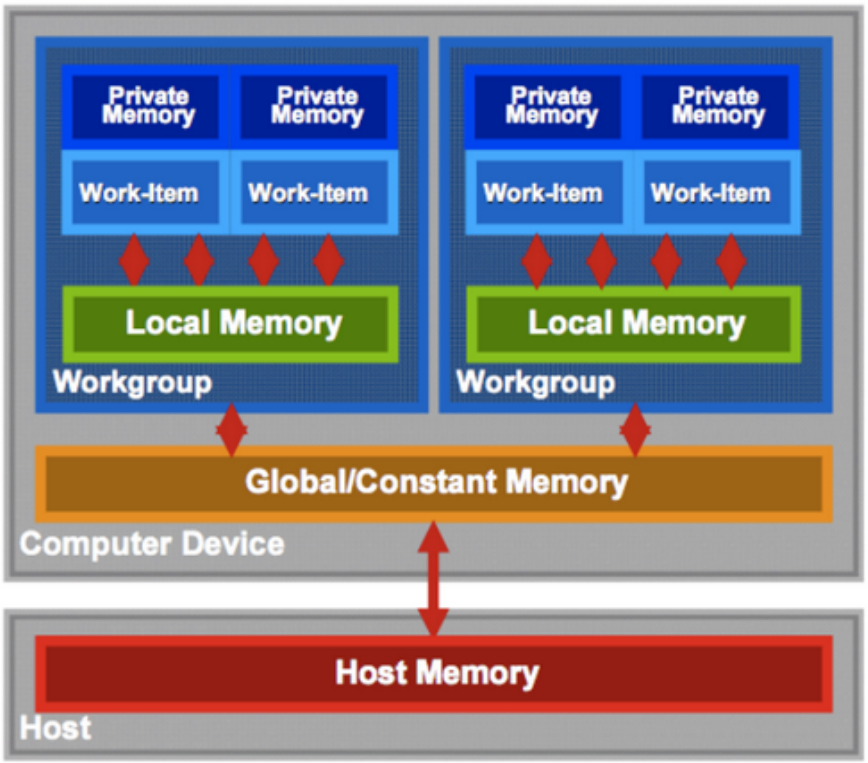


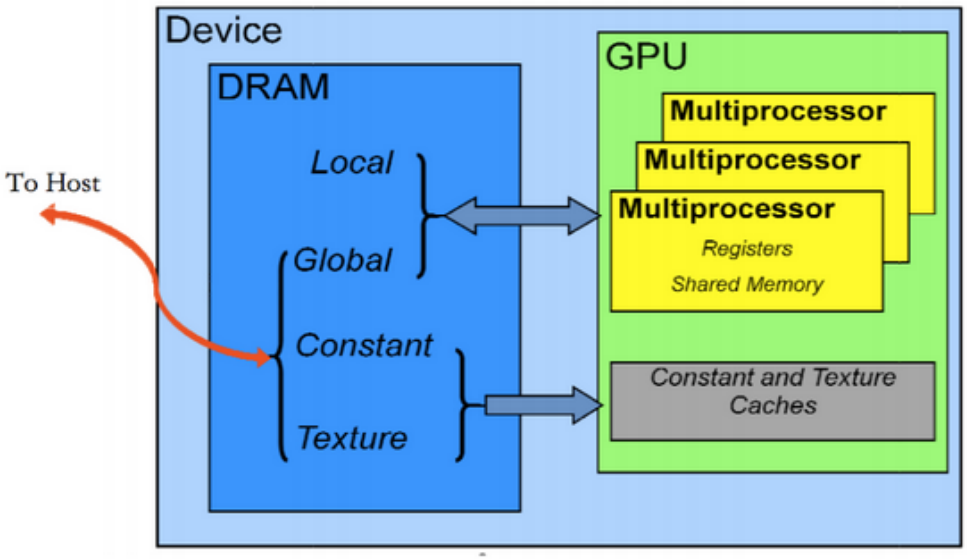




Arquitecturas y modelos de computación paralela

Programación en GPU - Modelo de memoria





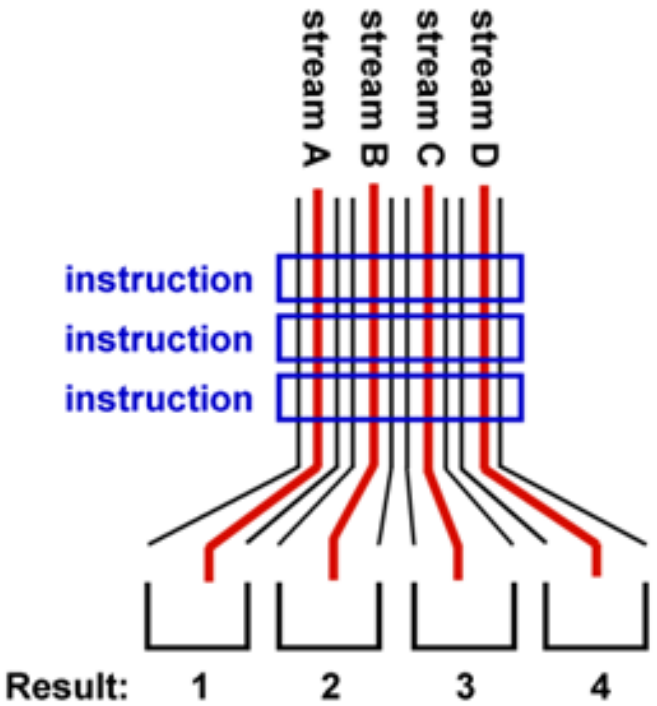
Programación en GPU - Consejos generales

- Kernels grandes, pocos accesos y gran cantidad de operaciones (ocultar latencia).
- Evitar colisiones en los accessos (uso de IDs).
- Bloques deben ser de tamaño múltiplo de *warp-wavefront*.
- Mantener a la GPU ocupada (*oversubscribe*).
- Evaluar uso de memoria compartida y memoria constante.

Arquitecturas y modelos de computación paralela

SIMD y extensiones SSE

- Operaciones vectoriales haciendo uso de registros con mayor capacidad.



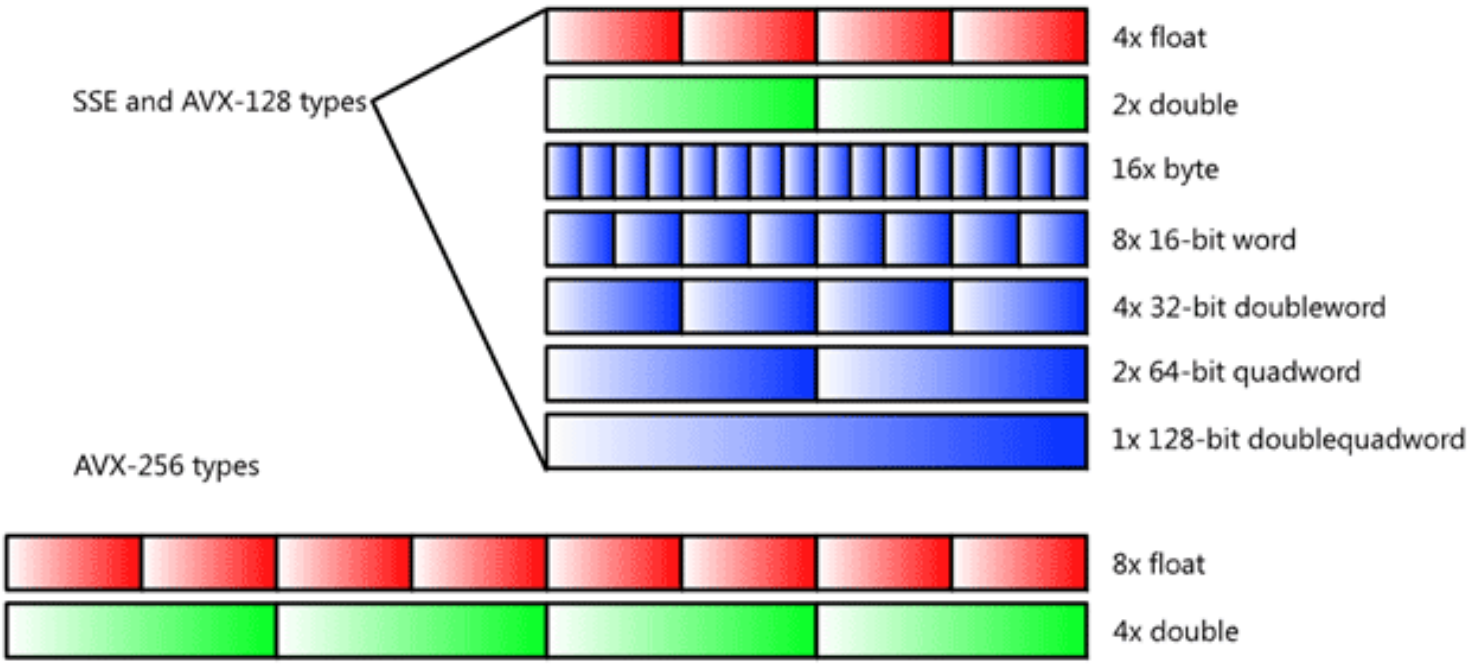
Arquitecturas y modelos de computación paralela

SIMD y extensiones SSE - Evolución de la tecnología

- MMX.
- SSE (*Streaming SIMD Extensions*).
- SSE2.
- SSE3.
- SSSE3.
- SSE4
- AVX.
- AVX512.

Arquitecturas y modelos de computación paralela

SIMD y extensiones SSE - Empaquetamiento



Trabajo realizado

Trabajo realizado

Algoritmo 4.1: Simulación de Monte Carlo

Entrada: Parámetros de la simulación: $\boldsymbol{nrSeeds}$, $\boldsymbol{nrHysteresisPoints}$, $\boldsymbol{nrMCS}$, $\boldsymbol{nrSpins}$, Lista de magnetización de los espines $\boldsymbol{magnetization}$.

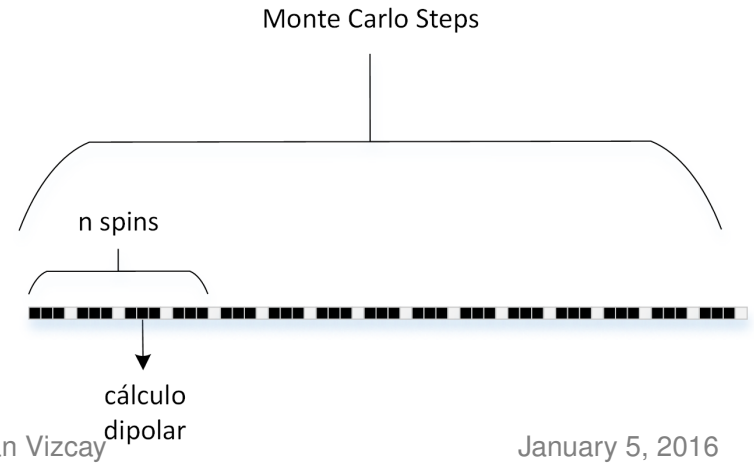
Salida: Lista de magnetización de los espines actualizada $\boldsymbol{magnetization}$.

```
1:  for seed = 0 → nrSeeds do
2:    for hs = 0 → nrHysteresisPoints do
3:      for mcs = 0 → nrMCS do
4:        for spin = 0 → nrSpins do
5:          randomSpin = random();
6:          currentEnergy = calculateEnergy();
7:          previousMagnetization = magnetization[randomSpin];
8:          magnetization[randomSpin] = random();
9:          deltaEnergy = calculateDeltaEnergy();
10:         coin = random();
11:         if deltaEnergy ≤ coin then
12:           magnetization[randomSpin] = previousMagnetization;
13:         end if
14:       end for
15:     end for
16:   end for
17: end for
```

Trabajo realizado

Análisis de código y detección de zonas paralelizables

- El cálculo de la interacción dipolar corresponde a otro bucle anidado, ya que debe calcularse la interacción entre cada uno de los espines.
- De las mediciones realizadas, el simulador tarda un 99% del tiempo total en calcular la interacción dipolar.
- Tal cálculo no es en sí computacionalmente costoso, pero éste debe ser realizado un gran número de veces.



Trabajo realizado

Paralelización a través de OpenMP

- Primer intento: *straightforward parallelism*.
- Basado en filosofía de OpenMP (paralelismo incremental).
- Creación y destrucción de hebras resulta ser ineficiente.

Trabajo realizado

Primer intento con OpenMP

```
1   for (int j = 0; j < nr_deltaH; j++) {
2     for (int k = 0; k < MCS; k++) {
3       for (int l = 0; l < nr_spins; l++) {
4         // seleccionar spin al azar
5
6         // calcular dipolar
7         #pragma omp parallel for
8         for (int i = 0; i < nr_spins; i++) {
9           // codigo dipolar
10        }
11
12        // suma valores parciales dipolar
13
14        // calcular resto de las interacciones
15        // y decidir si mantener o actualizar la configuracion
16      }
17    }
18  }
```

Trabajo realizado

Paralelización a través de OpenMP

- Segundo intento: mantener hebras activas.
- Por defecto, todo el código es ejecutado por todas las hebras.
- Se debe definir ahora zonas de códigos que serán ejecutadas por tan solo una hebra y agregar mecanismos de sincronización.

Trabajo realizado

Segundo intento con OpenMP

```
1  #pragma omp parallel
2  {
3  for (int j = 0; j < nr_deltaH; j++) {
4    for (int k = 0; k < MCS; k++) {
5      for (int l = 0; l < nr_spins; l++) {
6        #pragma omp single
7        // seleccionar spin al azar
8
9        // calcular dipolar
10       calculateDipolar();
11
12       #pragma omp critical
13       // suma valores parciales dipolar
14
15       #pragma omp barrier
16
17       #pragma omp single
18       // calcular resto de las interacciones
19       // y decidir si mantener o actualizar la configuracion
20     }
21   }
22 }
23 }   // end pragma omp parallel
```

Trabajo realizado

Paralelización en GPU a través de CUDA y OpenCL

- No se logra implementar la misma estrategia de mantener las hebras activas ya que el resultado debe ser comunicado de vuelta al host y no existen mecanismos de sincronización entre hebras que perteneces a bloques distintos.
- Se debe transferir los valores de magnetización por cada ejecución del kernel.
- Para compensar el costo de comunicación, la cantidad de espines debe ser lo suficientemente grande como para que el cálculo de la interacción dipolar valga la pena ser calculado utilizando miles de hebras.

Trabajo realizado

Paralelización en GPU a través de CUDA y OpenCL

- Tamaño del sistema está limitado por el tamaño de la memoria global.
- Cantidad de accesos a memoria deben ser idealmente menores a la cantidad de operaciones realizadas en el kernel.

$$d_f = 3(\hat{n}_{ij} \cdot \vec{m}_j)$$
$$B_d = \frac{d_f \hat{n}_{ij} - \vec{m}_j}{r_{ij}^3} \quad (3)$$

Trabajo realizado

Función kernel

```
1  kernel_dipolar(float *magnetization, float *nx, float *ny, float *nz,
2          float *cube, int site, int nrAtoms, float *output) {
3      // index es el id de la hebra
4      if (index >= nrAtoms)                    // 1 comparacion
5          return;
6
7      int inputIndexX = 0 * nrAtoms + index;   // 2 op aritmeticas
8      int inputIndexY = 1 * nrAtoms + index;   // 2 op aritmeticas
9      int inputIndexZ = 2 * nrAtoms + index;   // 2 op aritmeticas
10
11     float mujX = magnetization[inputIndexX];
12     float mujY = magnetization[inputIndexY];
13     float mujZ = magnetization[inputIndexZ];
14
15     int indexJSite = site * nrAtoms + index;     // 2 op aritmeticas
16
17     float distanceX = nx[indexJSite];
18     float distanceY = ny[indexJSite];
19     float distanceZ = nz[indexJSite];
20
21     ...
```

Trabajo realizado

Función kernel (continuación)

```
1   ...
2       float cubeDistance = cube[indexJSite];
3
4       // 6 operaciones aritmeticas
5       float df = 3 * (mujX*distanceX + mujY*distanceY + mujZ*distanceZ);
6
7       output[inputIndexX] = (df * distanceX - mujX) / cubeDistance; // 3 op aritimeticas
8       output[inputIndexY] = (df * distanceY - mujY) / cubeDistance; // 3 op aritimeticas
9       output[inputIndexZ] = (df * distanceZ - mujZ) / cubeDistance; // 3 op aritimeticas
10  }
```

Trabajo realizado

Paralelización en GPU a través de CUDA y OpenCL

- Kernel anterior tiene un ratio CGMA (*Compute to Global Memory Access*) de dos, es decir, por cada dos accesos a memoria se realizan dos operaciones.
- Dada la natureleza del problema, tampoco resulta factible utilizar otros tipos de memoria como la memoria compartida, constante, etc.
- Información se encuentra almacenada de forma contigua con el fin de que hebras contiguas accedan a posiciones contiguas (se evitan colisiones).

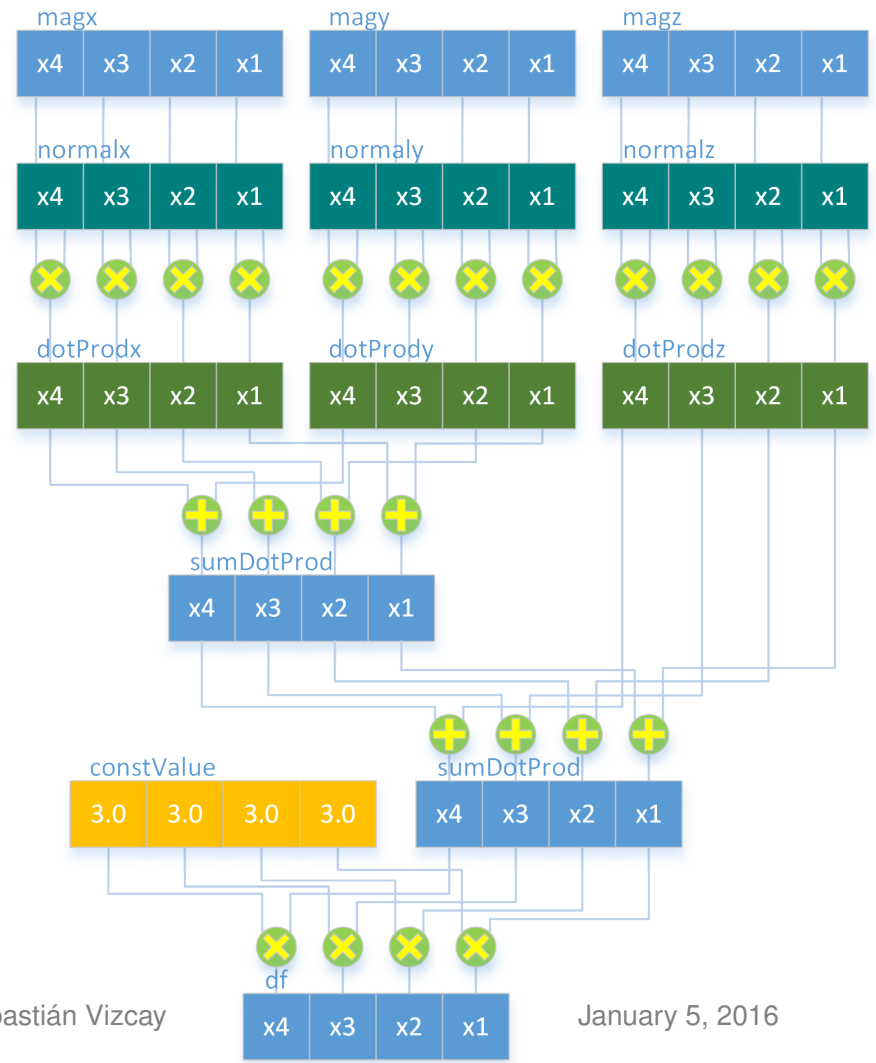
Trabajo realizado

Paralelización a través de OpenMP e instrucciones SIMD

- Además de dividir la carga de trabajos entre hebras, se realiza procesamiento vectorial por cada una de ellas.
- Además de ofrecer vectorización, se evita además accesos a memoria al procesar la información directamente en los registros del procesador.
- Información debe ser empaquetada de forma correcta para realizar transferencias óptimas a los registros SIMD y poder operar así de forma vectorial.

Trabajo realizado

Cálculo del valor *df*



Trabajo realizado

Paralelización a través de OpenMP e instrucciones SIMD

- Operaciones SIMD para el programa de precisión simple difieren de las del programa de precisión doble.
- Se agrega un *overhead* al realizar las operaciones finales del cálculo de la interacción dipolar, al retornar los valores de este vector de forma contigua.

Experimentos y resultados

Experimentos

Mediciones

- Mediciones fueron realizadas con `clock` para el programa secuencial (tiempo de procesador utilizado en cantidad de *ticks* de reloj) y `omp_get_wtime` para los programas paralelos (tiempo de reloj de muralla).

Conjunto de pruebas

- Programa original secuencial (*double*) con y sin -O3.
- Programa paralelizado con OpenMP (*float* y *double*).
- Programas paralelizados en GPU utilizando CUDA y OpenCL.
- Programa paralelizado con OpenMP + SIMD (*float* y *double*).

Experimentos

Entorno de Pruebas: CPU

	Nanoserver	Titan
Modelo	AMD Opteron 6282 SE	Intel Core i7-4960X
Nr. cores	64 (4 x 16)	6
Frecuencia	2.6 GHz	3.6 GHz

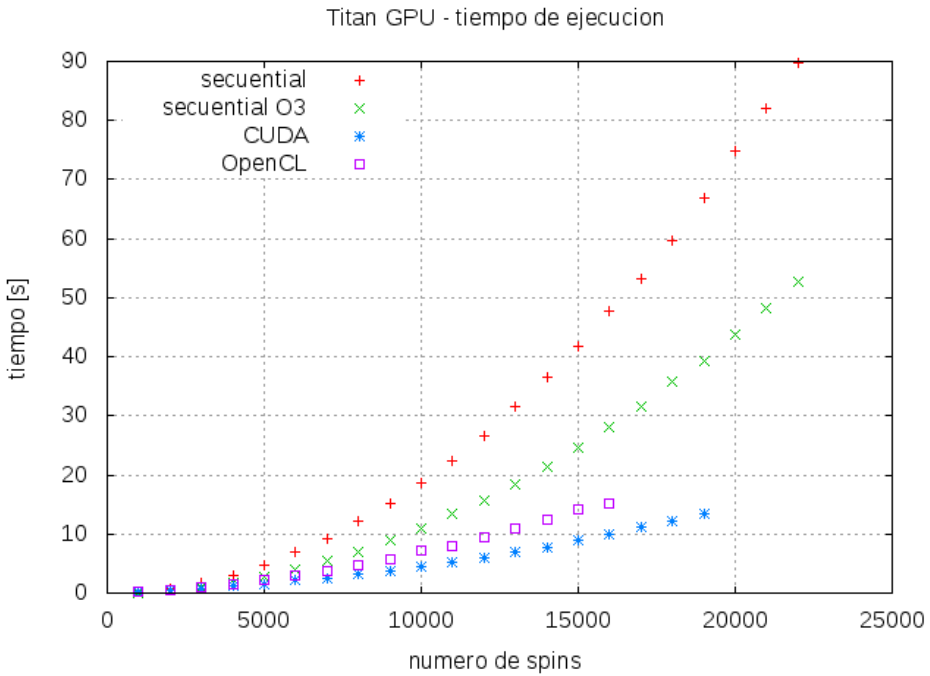
Experimentos

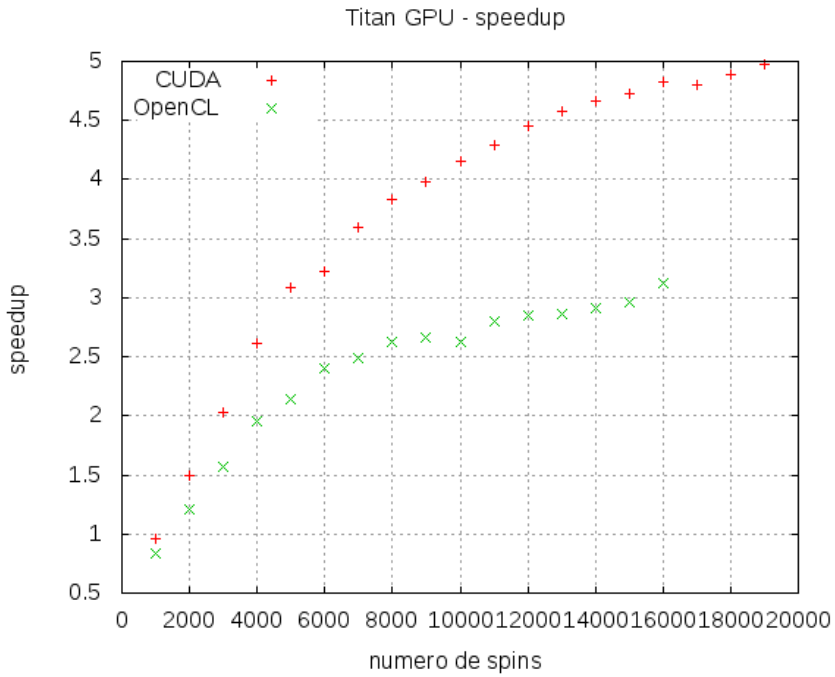
Entorno de Pruebas: Tarjeta gráfica

	Nanoserver	Titan
Modelo	Nvidia Tesla c2075	Nvidia Titan Black
RAM	6 GB	6 GB
Bus de memoria	384 bit	384 bit
Bandwidth	144 GB/s	336 GB/s
Conexión	PCIe 2.0x12	PCIe 3.0x16
Frecuencia GPU	1.150 MHz	889 MHz
Nr. shaders	448 (14 SM x 32 SP)	2.880 (15 SM x 192 SP)

Experimentos

Tiempos de ejecución y speedup: CUDA y OpenCL





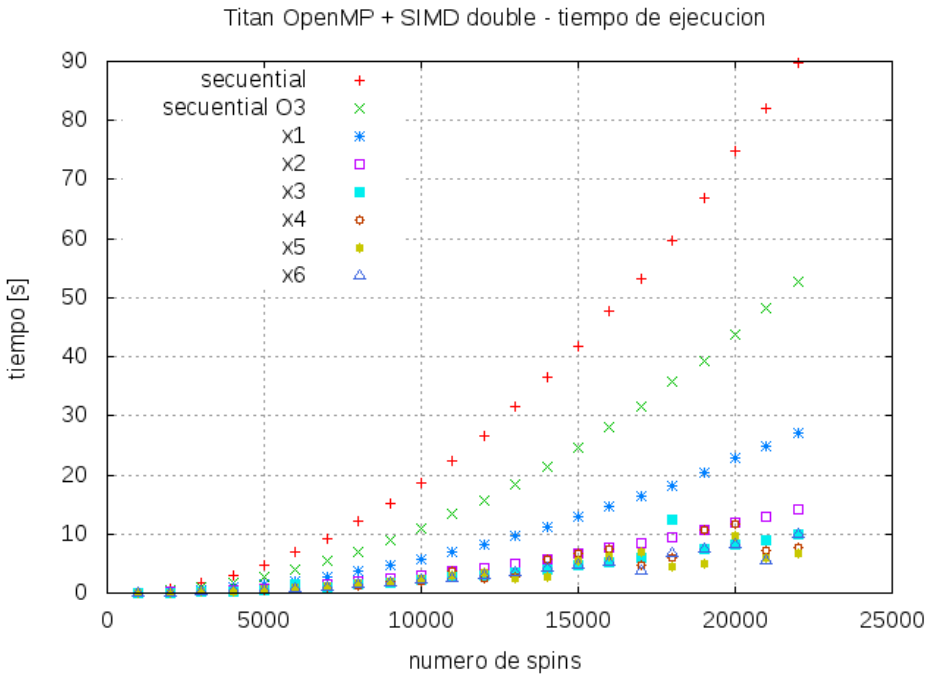
Experimentos

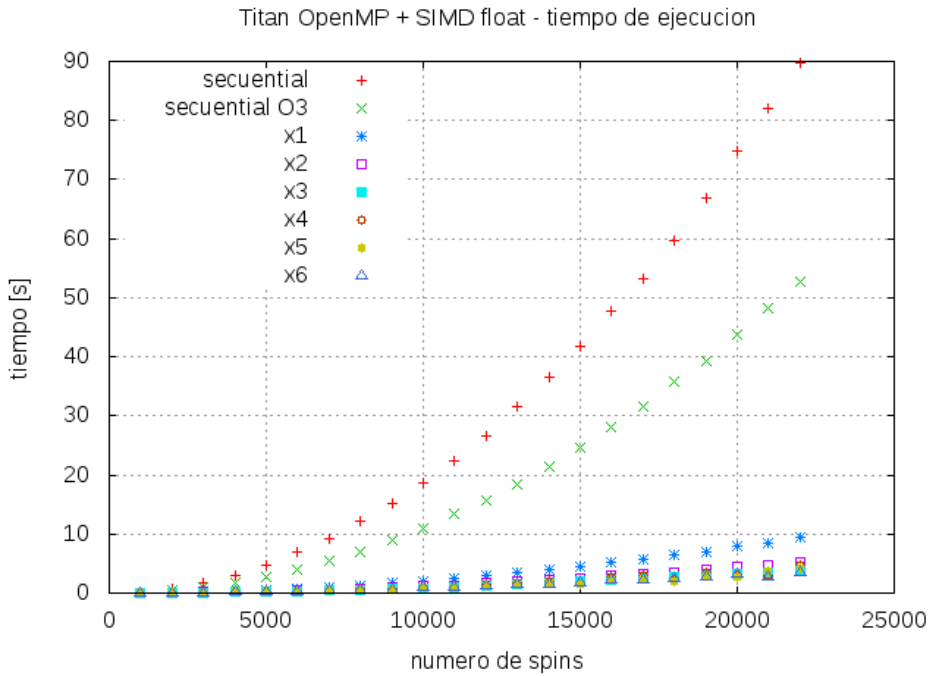
Comentarios: CUDA y OpenCL

- Configuración utiliza el máximo número de hebras por bloques.
- A partir de 3.000 espines, el tiempo de ejecución en GPU es menor al O3.
- Speedup creciente dada la pendiente de la curva de speedup.
- A mayor tamaño de problema, mayor es la transferencia CPU-GPU y mayor es la cantidad de veces a realizar tal comunicación.
- Tiempos de transferencia corresponden en promedio a un 77% (CUDA) y un 74% (OpenCL) del total de tiempo de simulación.
- Computación realizada en el kernel no justifica el alto costo de comunicación.

Experimentos

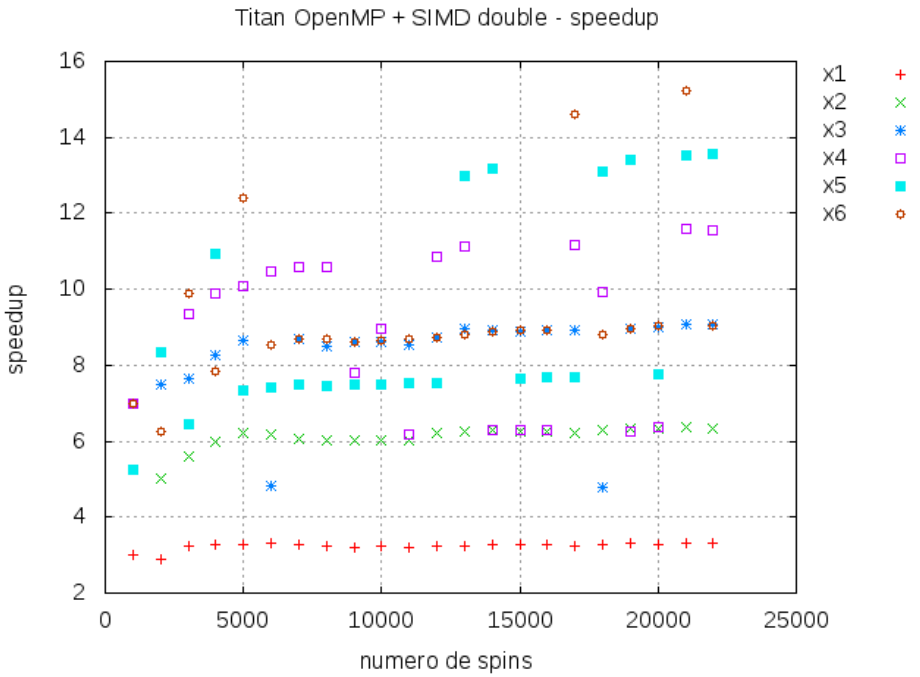
Tiempos de ejecución: Titan OpenMP + SIMD

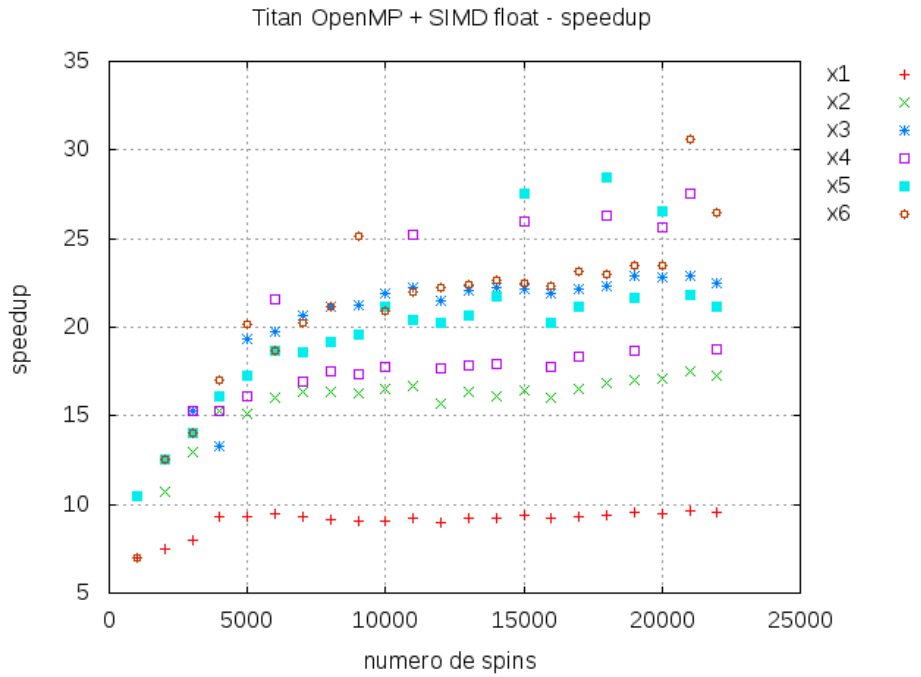




Experimentos

Speedup: Titan OpenMP + SIMD





Experimentos

Comentarios: Mejores speedups obtenidos

Programa	Nanoserver	Titan
OpenMP double	8.78 (4)	9.01 (6)
OpenMP float	13.67 (8)	9.14 (6)
CUDA	sin pruebas	4.98 (19 bloques x 1024 hebras)
OpenCL	sin pruebas	3.13 (16 bloques x 1024 hebras)
OpenMP + SIMD double	10.47 (4)	15.21 (6)
OpenMP + SIMD float	19.5 (4)	30.65 (6)

Experimentos

Comentarios adicionales

- Speedups obtenidos en el Titan son por lo general mejores que los obtenidos en el Nanoserver.
- Pendiente de las curvas de speedups son mayores en el Nanoserver, por lo que se podría seguir explotando la mayor cantidad de cores que posee el Nanoserver al seguir incrementando la cantidad de espines.
- Speedups obtenidos en GPU son considerablemente más bajos que los obtenidos con los otros métodos de paralelización.

Experimentos

Comentarios adicionales

- Paralelización en GPU también presenta el problema de contar con un menor tamaño de memoria.
- Computación en GPU requiere obviamente hardware adicional.
- tecnología SIMD resulta prometedora. Próxima generación de procesadores contarán con AVX512.
- De las pruebas realizadas, nunca se obtuvo un speedup lineal perfecto (siempre hay un costo por paralelización).

Experimentos

Validación

- Uso del comando diff para buscar diferencias entre archivos.
- Programas que utilizan `doubles` producen exactamente los mismos resultados.
- Programas que utilizan `floats` producen resultados que difieren tan solo a partir de la quinta o sexta posición decimal.
- Valores originales corresponden a valores obtenidos a través de simulaciones.

Conclusiones

Conclusiones

- Se investigó sobre técnicas de paralelismo en GPU y CPU y su aplicabilidad en el modelo de Heisenberg.
- Primera paralelización utilizando *straightforward parallelism* con OpenMP no cumplió las expectativas.
- Se ofreció un segundo método de paralelización en OpenMP, el cual busca mantener las hebras activas. Este método presentó mejores resultados pero la estrategia utilizada no pudo ser portada a GPU.
- Dentro de las pruebas realizadas, se alcanzó un speedup de 30 utilizando OpenMP + SIMD, al simular 21.000 espines utilizando 6 hebras.
- Resultados obtenidos en GPU fueron considerablemente menores de lo esperado. Esto se debe tanto a la inclusión del término dipolar, la mayor cantidad de memoria necesitada en el modelo de Heisenberg frente a otros modelos, y a la arquitecturas de las GPUs.

Conclusiones

- En GPU, resultados obtenidos utilizando CUDA son levemente mejores que los obtenidos con OpenCL.
- En cuanto a analisis de escalabilidad, se determinó que la solución OpenMP + SIMD es escalable y se espera alcanzar cifras mayores de speedups al simular instancias con mayor número de espines.
- Se determinó la ventaja de utilizar números flotantes de precisión simple y se verificó que los resultados obtenidos en estas simulaciones están en conformidad con los que se obtienen al utilizar *doubles*.
- Se plantea como trabajo a futuro implementar el método de Monte Carlo en alguna FPGA, lo cual sería lo más cercano a construir hardware específico para tal simulación.

Consultas