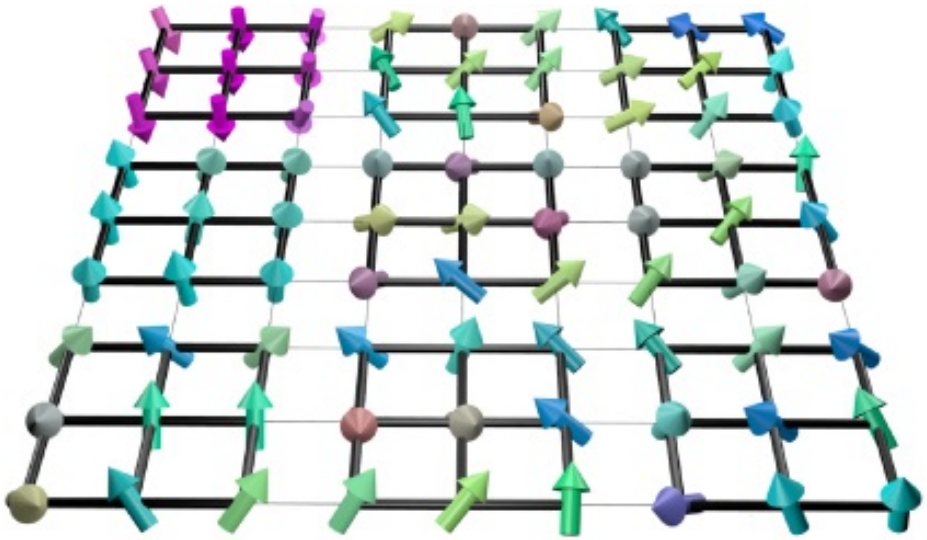
UdeSantiago de Chile

# 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.


# Arquitecturas y modelos de computación paralela

## Programación en GPU

- asdf


# Trabajo realizado

# Trabajo realizado

**Algoritmo 4.1:** Simulación de Monte Carlo

**Entrada:** Parámetros de la simulación: $nrSeeds$, $nrHysteresisPoints$, $nrMCS$, $nrSpins$, Lista de magnetización de los espines $magnetization$.

**Salida:** Lista de magnetización de los espines actualizada $magnetization$.

```
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

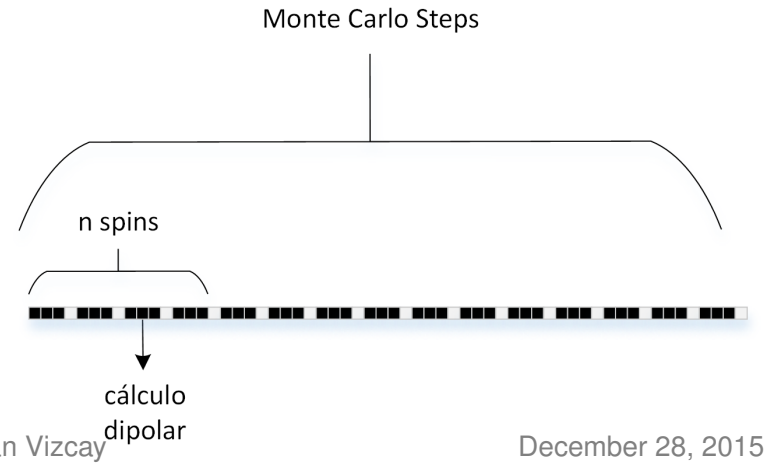
## Análisis de código y detección de zonas paralelizables

- Speedup alcanzable queda limitado por la ley de Amdahl.
- Primer bucle, encargado de construir una curva de histéresis por cada semilla, es trivialmente paralelizable.
- Segundo y tercer bucle (valor de campo y número de MCS) son secuenciales.
- Cuarto bucle, el encargado de tomar muestras al azar, es secuencial si de desea cumplir la condición de *balance detallado*.
- Paralelización es solo aplicable al cálculo de la energía realizado en el bucle más interno.

# 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.

$$d_f = 3(\hat{n}_{ij} \cdot \vec{m}_j)$$
$$B_d = \frac{d_f \hat{n}_{ij} - \vec{m}_j}{r_{ij}^3} \quad (1)$$

# 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).
- Tampoco existen divergencias en el kernel más allá del *if* inicial que verifica si se trata de un id válido.

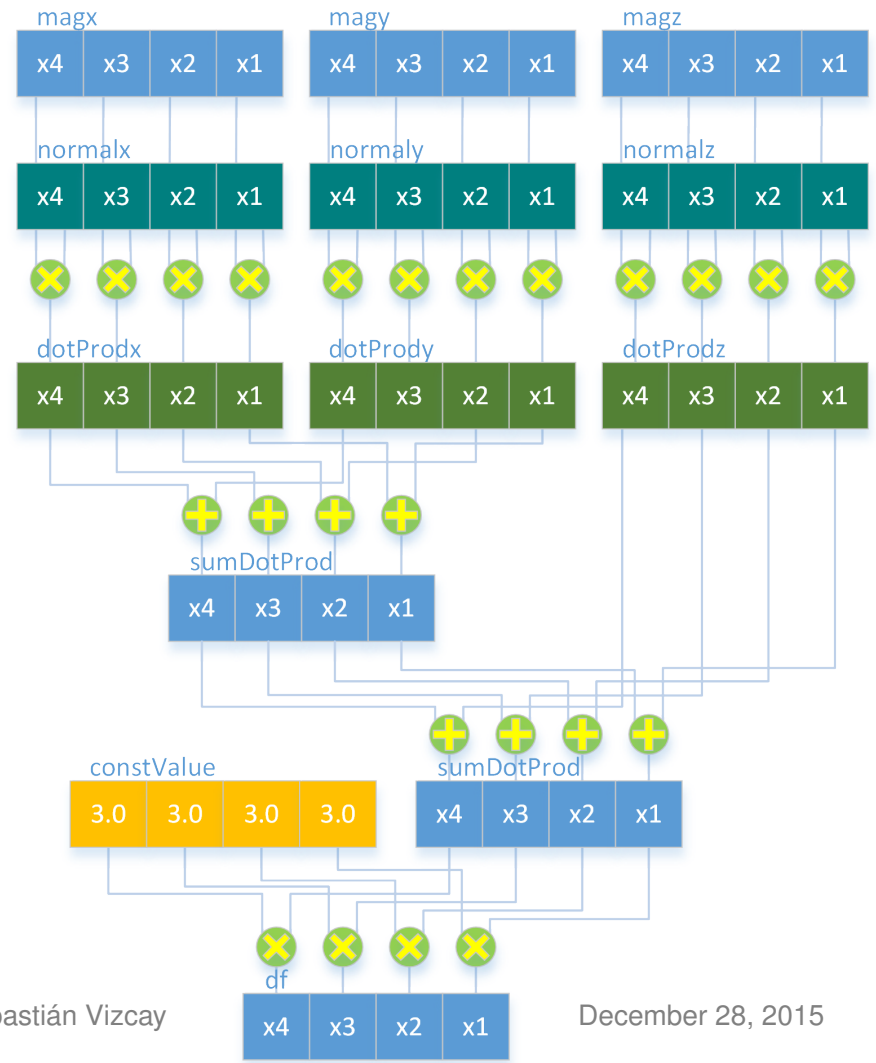
# Trabajo realizado

## 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.
- Se utiliza la extensión AVX que permite operar en registros de 256 bits.
- Además de ofrecer vectorización, se evita además accesos a memoria al procesar la información directamente en los registros del procesador.
- Se implementan dos programas, uno que utiliza precisión simple y otro que utiliza precisión doble.
- 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 |

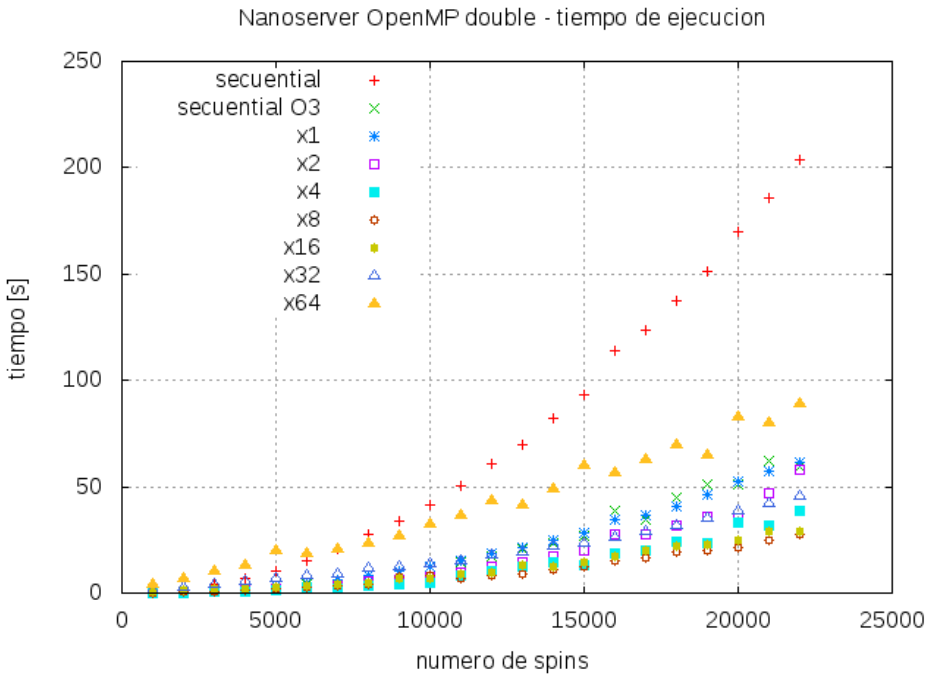
# Experimentos

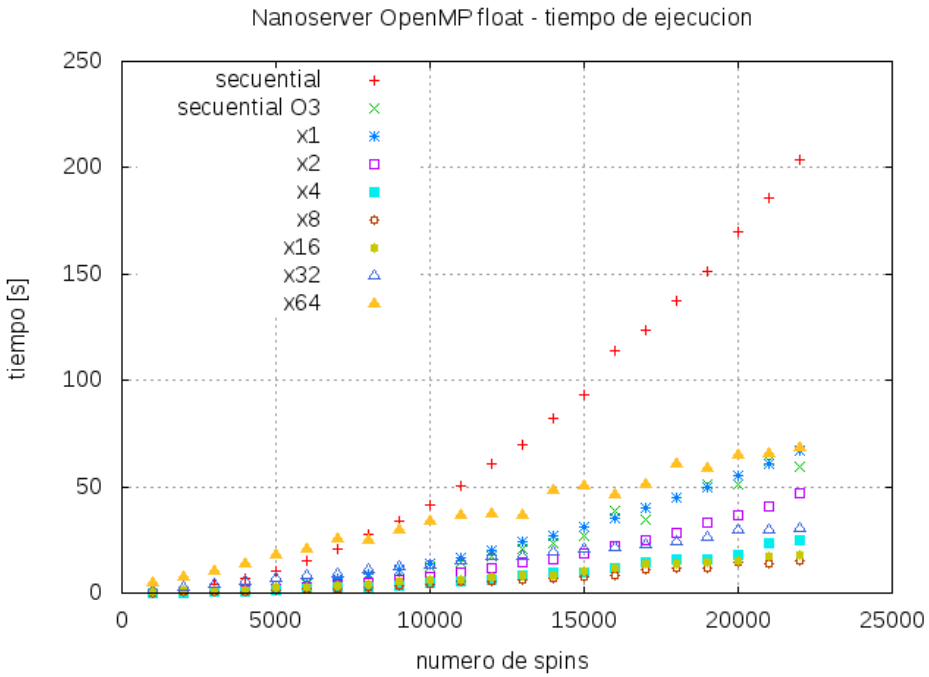
## Entorno de Pruebas: Tarjeta gráfica

| | 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

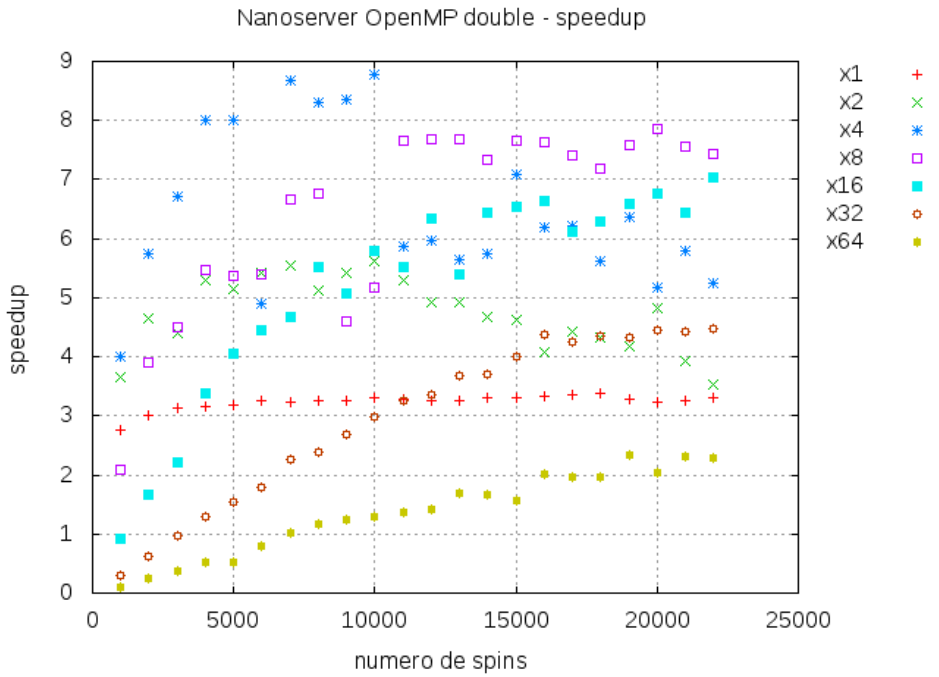
## Tiempos de ejecución: Nanoserver OpenMP

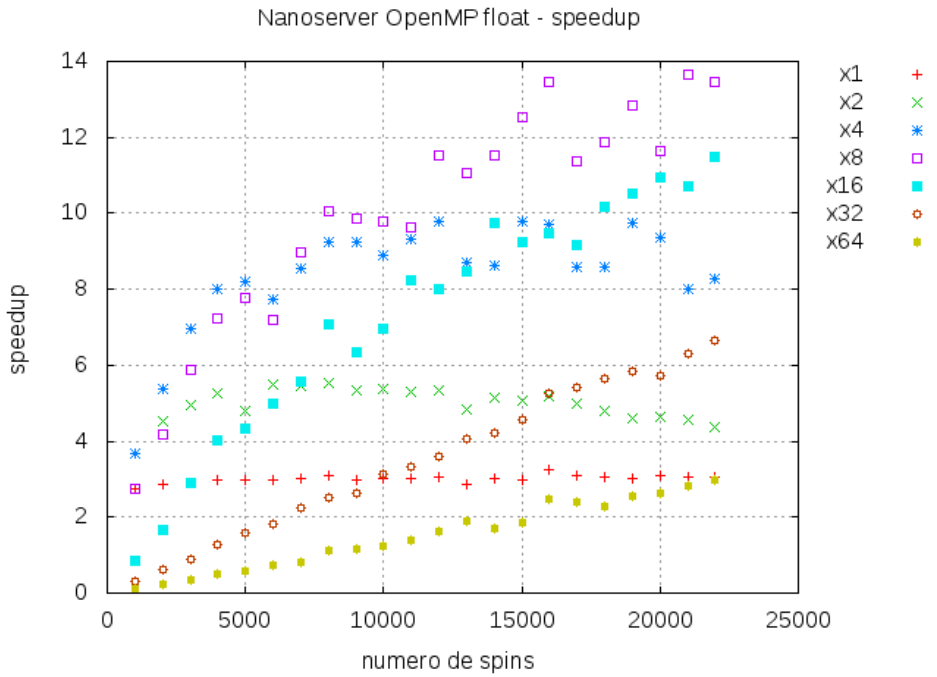


# Experimentos

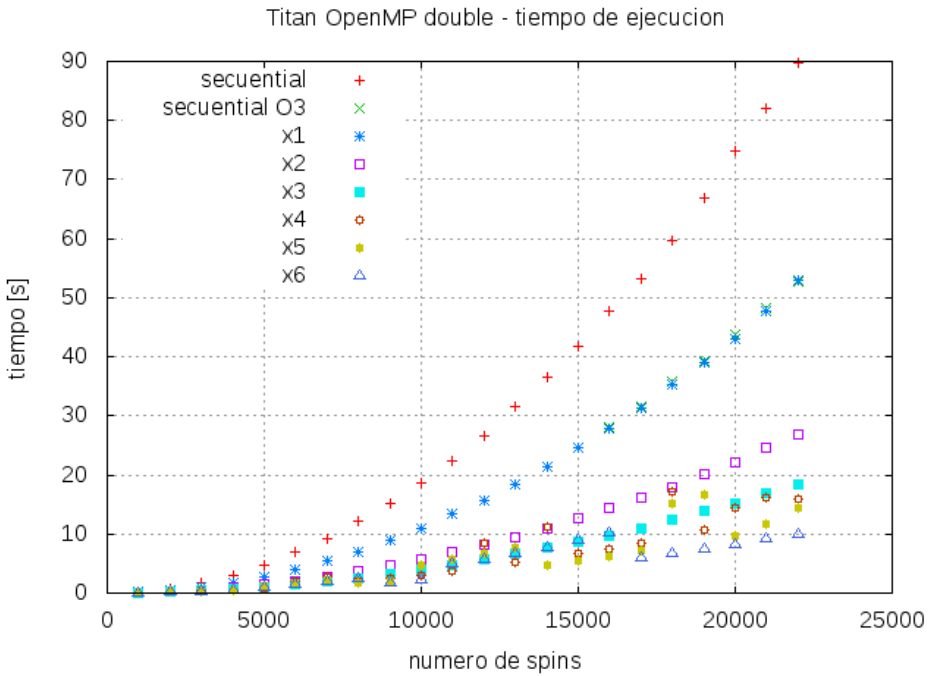
## Speedup: Nanoserver OpenMP

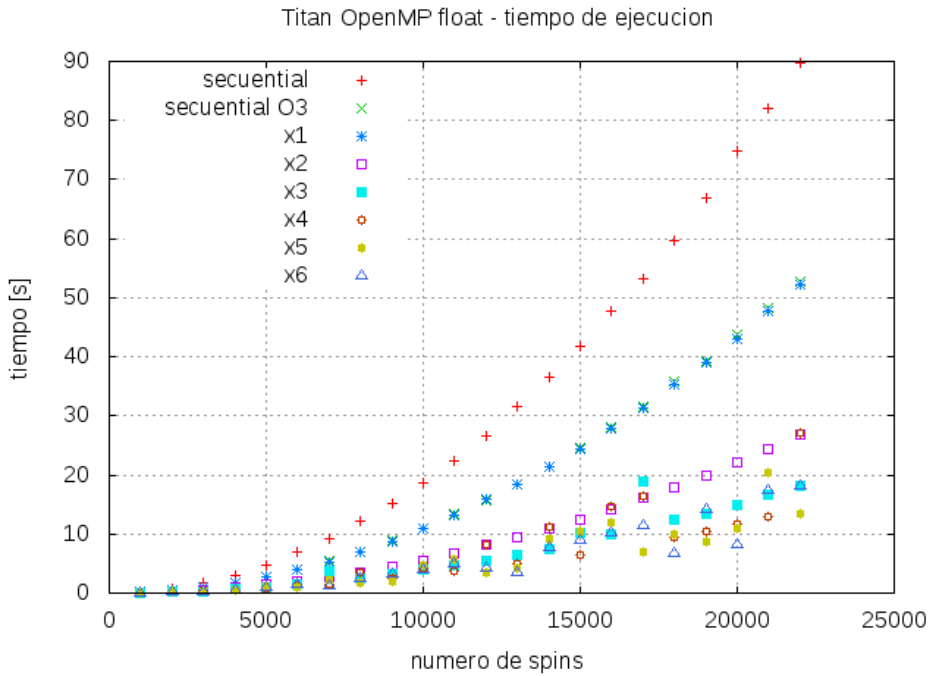


# Experimentos

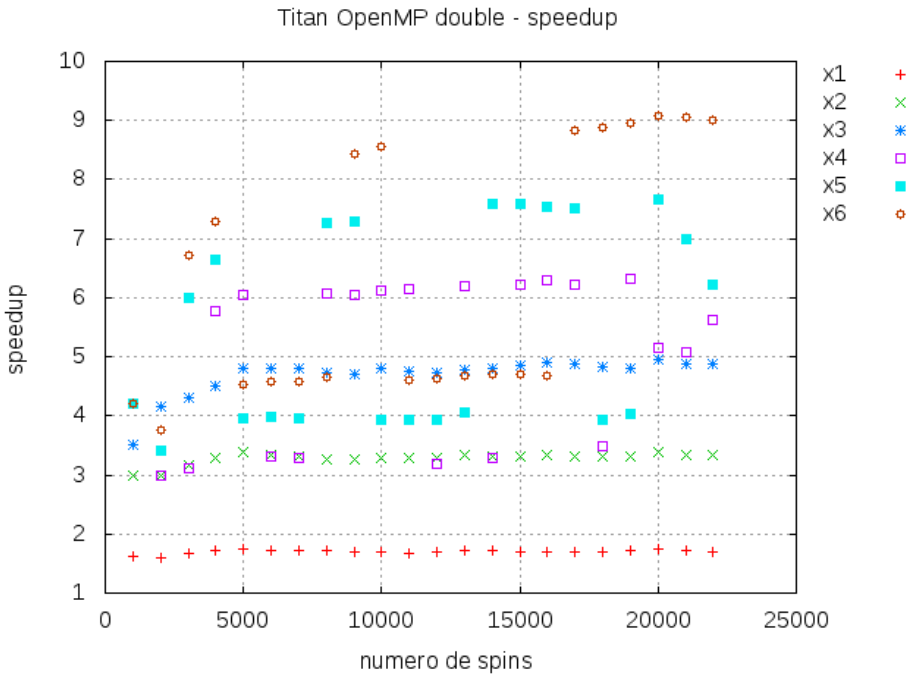
## Tiempos de ejecución: Titan OpenMP

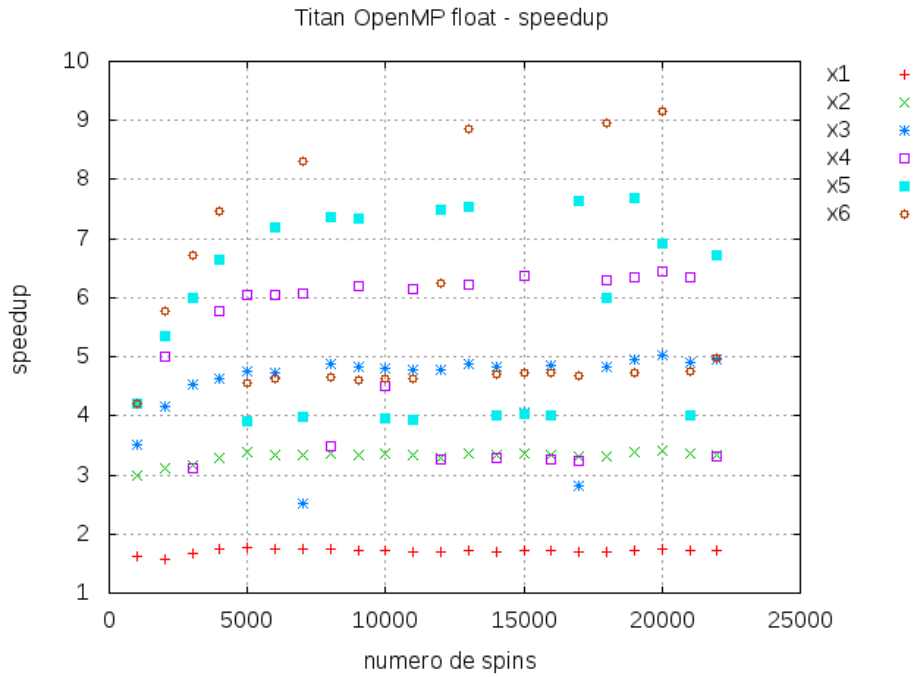


# Experimentos

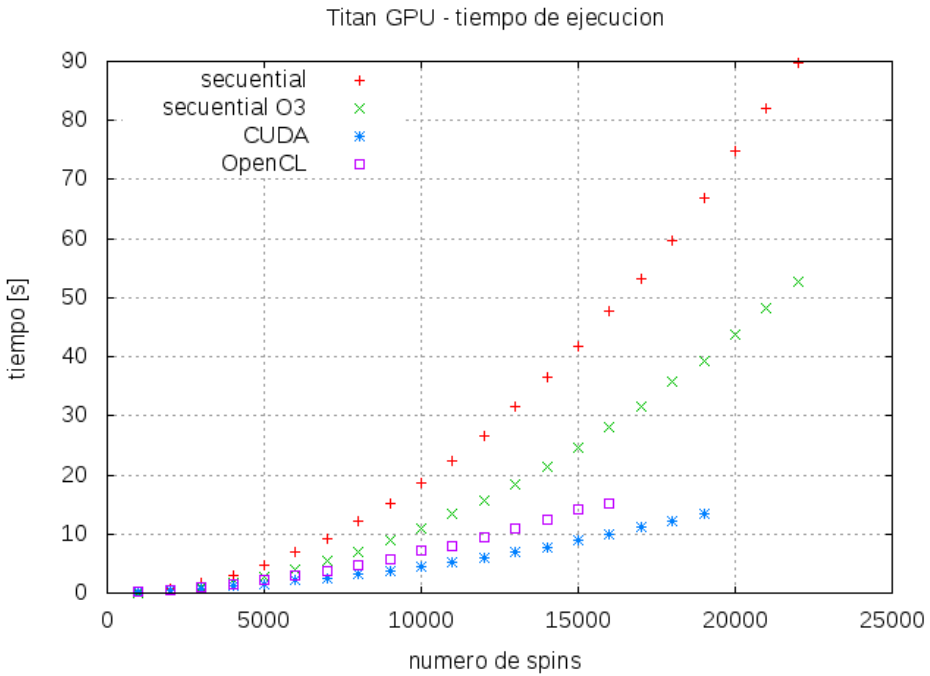
## Speedup: Titan OpenMP

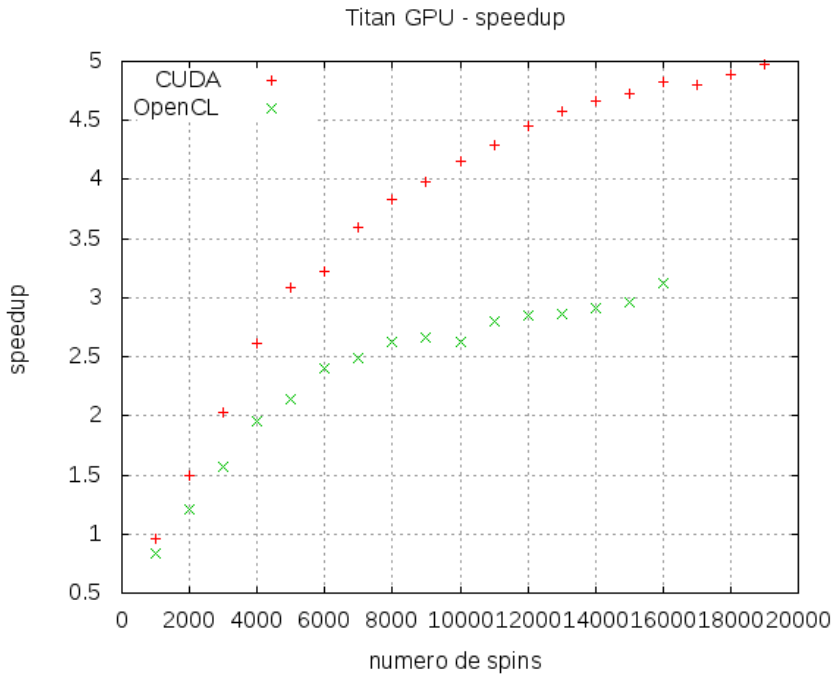


# Experimentos

## Tiempos de ejecución y speedup: CUDA y OpenCL



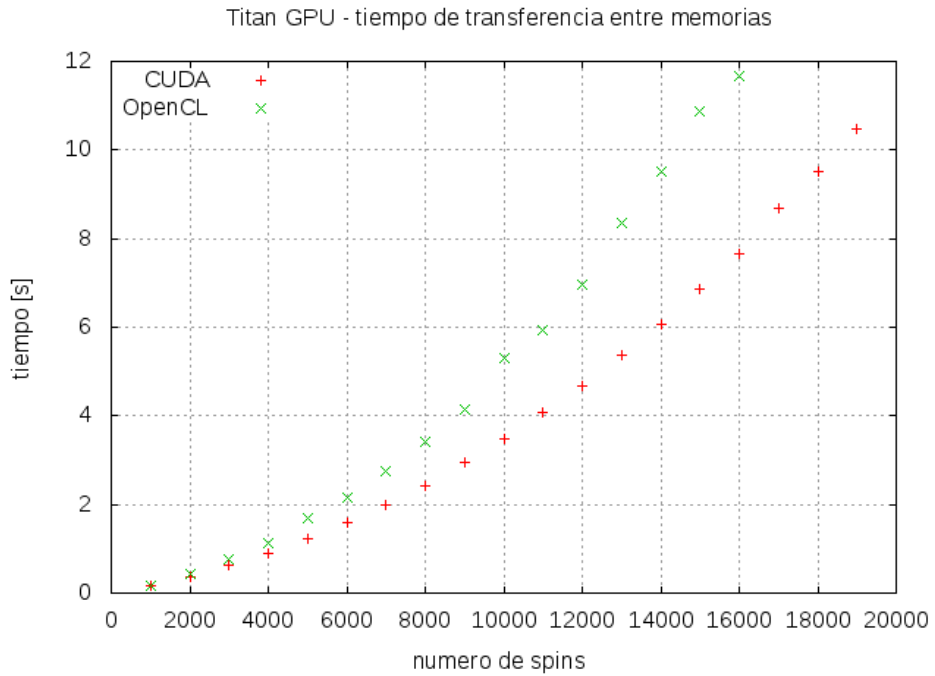
# Experimentos

## 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% del total de tiempo de simulación.
- Computación realizada en el kernel no justifica el alto costo de comunicación.

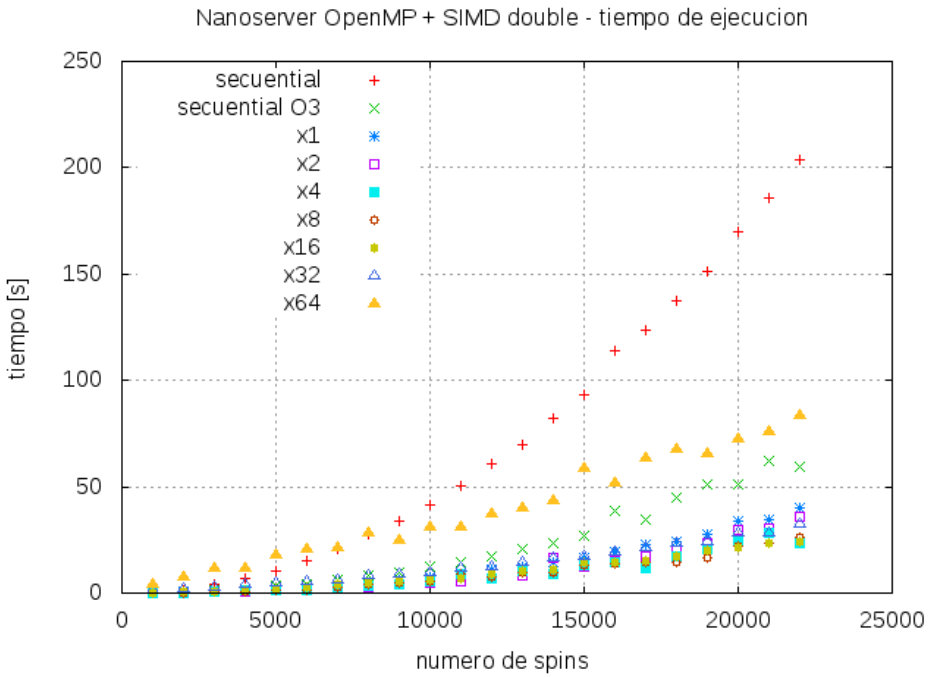
# Experimentos

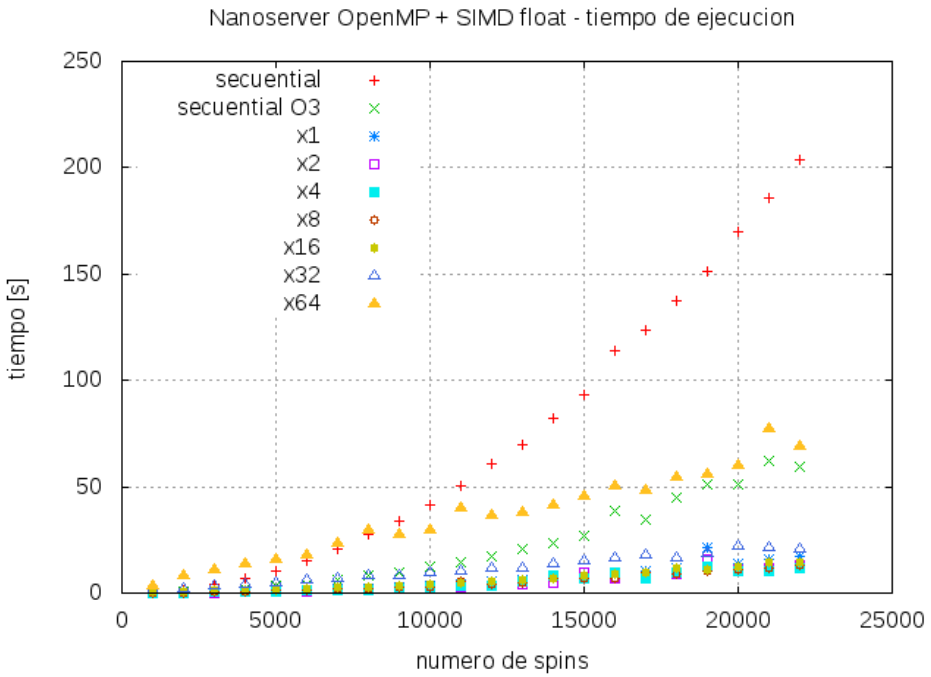
## Tiempos de transferencia: CUDA y OpenCL



# Experimentos

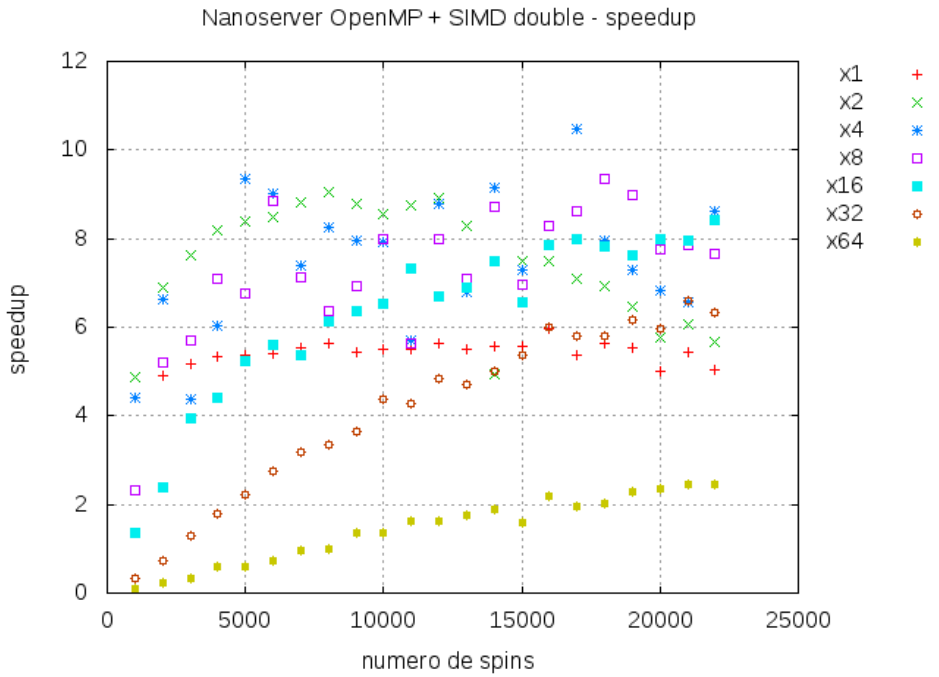
## Tiempos de ejecución: Nanoserver OpenMP + SIMD

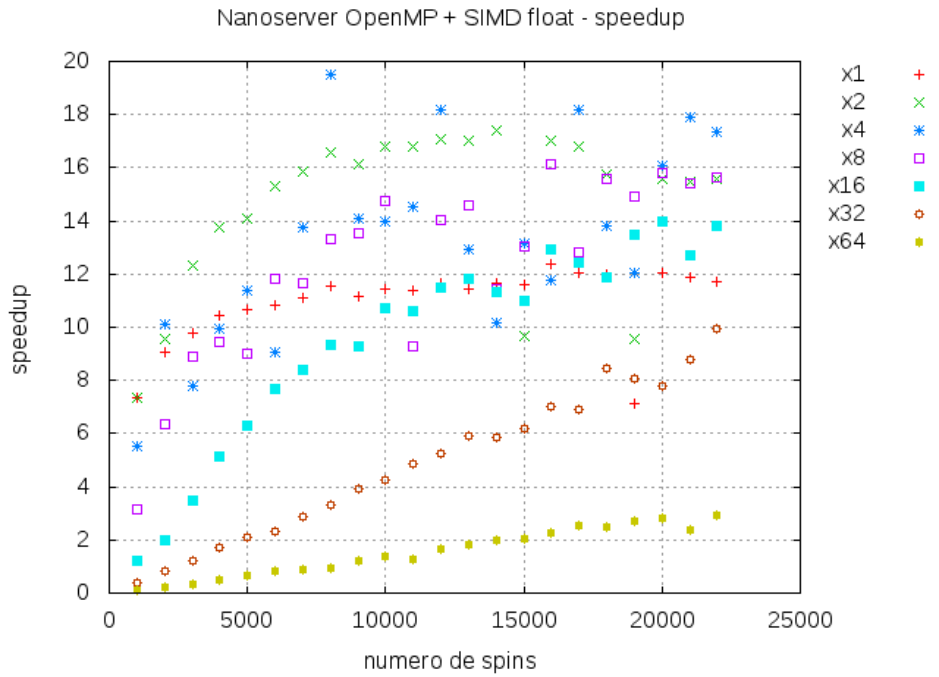


# Experimentos

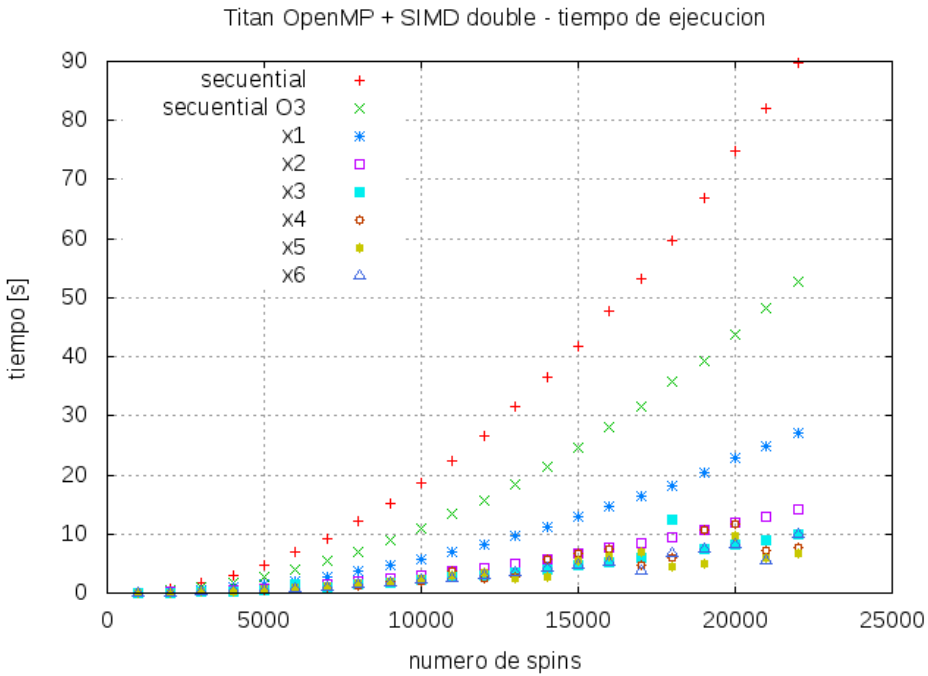
## Speedup: Nanoserver OpenMP + SIMD

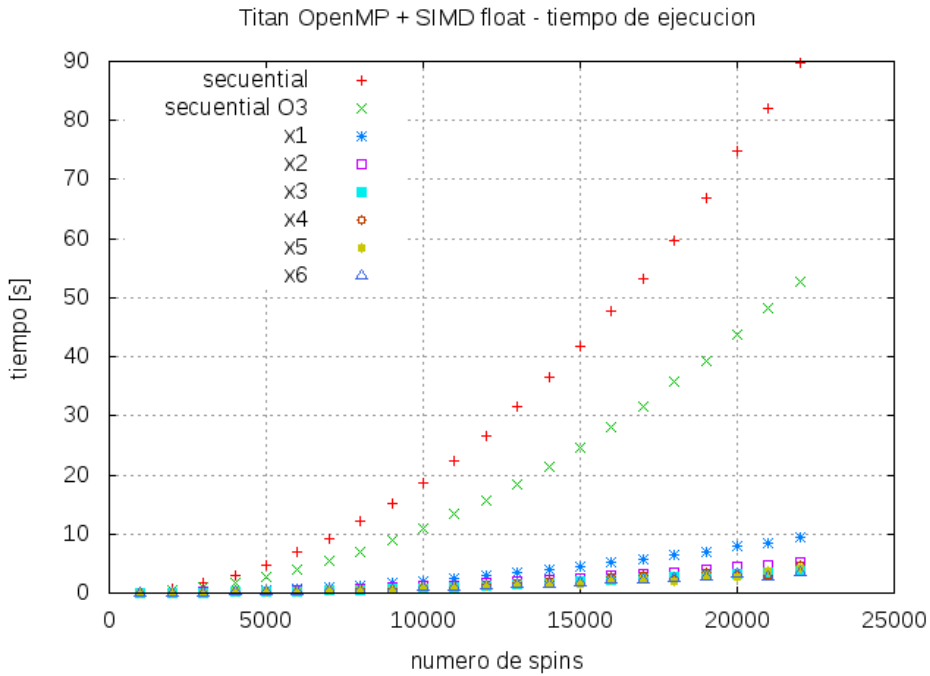


# Experimentos

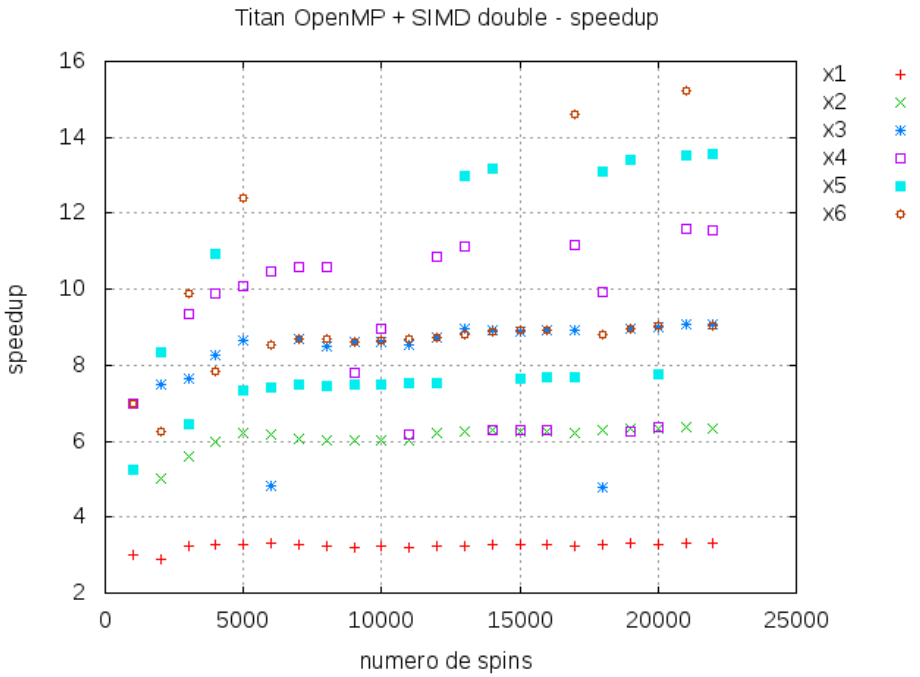
## Tiempos de ejecución: Titan OpenMP + SIMD

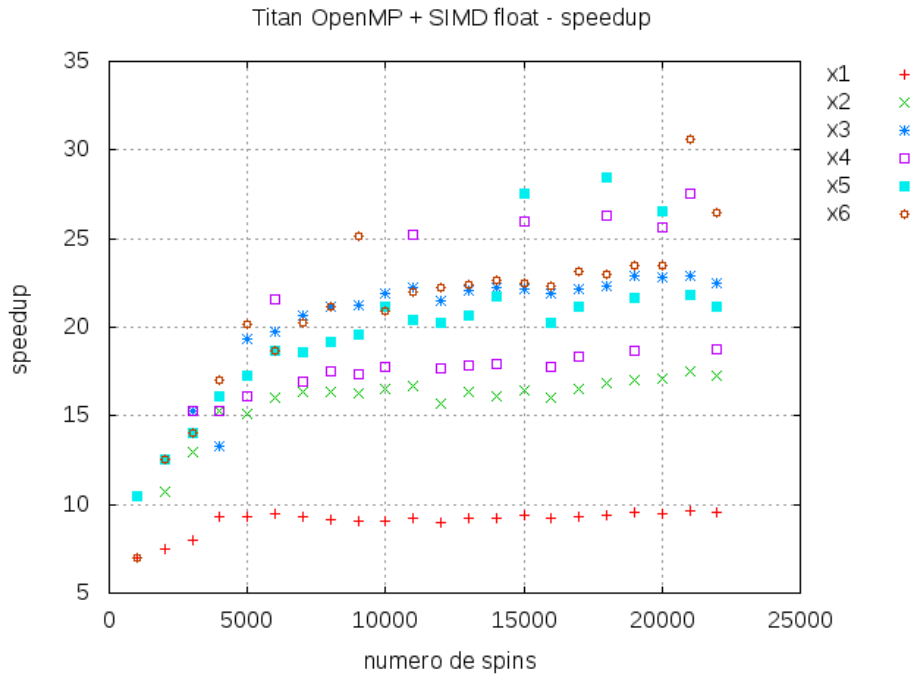


# Experimentos

## 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