FaMAF
Facultad de Matemática, Astronomía y Física
Universidad Nacional de Córdoba

GPGPU
Computing Group

NVIDIA

Simulación del modelo de Potts de *q* estados en GPUs

Juan P. de Francesco[1], Ezequiel E. Ferrero[1,2,3], Nicolás Wolovick[1], Sergio A. Cannas[1,2]

[1] *Facultad de Matemática, Astronomía y Física; Universidad Nacional de Córdoba, Ciudad Universitaria, 5000 Córdoba, Argentina*

[2] *Instituto de Física Enrique Gaviola (IFEG-CONICET), Ciudad Universitaria, 5000 Córdoba, Argentina*

[3] *ferrero@famaf.unc.edu.ar*

Diseñadas y desarrolladas para la industria de los video juegos, las placas de video o **GPU**s (por ***G**raphics **P**rocessing **U**nits*), poseen un poder de cálculo que en la actualidad excede olgadamente al de una CPU. Esta capacidad resulta de la relativa simplicidad de la arquitectura de la GPU, comparada con la CPU, combinada por un gran número de unidades de procesamiento en paralelo en un mismo chip.

En los últimos años ha cobrado notoriedad el uso de placas de video en diversas aplicaciones científicas. Para aprovechar el potencial de las GPUs para cómputos en general es necesario programar el problema teniendo en cuenta el paralelismo inherente de las placas y la estructura jerárquica de accesos a memoria. **nVidia**, uno de los principales fabricantes de GPUs, ha desarrollado **CUDA** (siglas de ***C**ompute **U**nified **D**evice **A**rchitecture*): un compilador y un conjunto de herramietas de desarrollo que permiten a los programadores usar una variación del lenguaje de programación C para codificar algoritmos en GPUs de nVidia.

Presentamos una implementación en CUDA de simulaciones de Monte Carlo del modelo ferromagnético de Potts de *q* estados bidimensional. Utilizamos como estrategia de paralelización la técnica del "tablero de ajedrez". Validamos el código reobteniendo curvas de equilibrio conocidas y ciclos de histéresis asociados a la transición de 1er orden para $q>4$. A grandes razgos obtenemos una performance mayor a **150x** en el tiempo de ejecución de los programas. Discutimos en particular un problema abierto [1] que esta nueva potencia de cálculo nos permite atacar en tiempos razonables.

Arquitectura GPU Nvidia

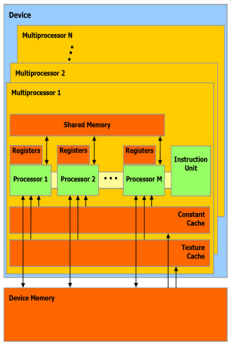
• Tarjeta gráfica (**GPU**) compuesta por un arreglo escalable de **multiprocesadores**.

•Cada multiprocesador contiene: **8 cores** de procesadores escalares, una unidad de instrucción multithread, **memoria compartida** y un core de precisión doble.

•Cada multiprocesador contiene un **registros** de **32bits** por procesador y todos los procesadores en un mismo multiprocesador comparten la shared memory.

•También existen disponibles **memorias caché** de dos tipos: **constante** y de **textura**.

•La memoria global de la GPU es compartida por todos los multiprocesadores. Es aproximadamente 10 veces más rápida que la memoria RAM de una máquina actual.



Device = GPU
Host = CPU

La GPU destina más transistores al procesamiento de datos

Programación con CUDA

•Programación en código **C** (o **Fortran**) standard usando extensiones **CUDA**.

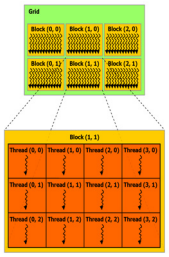
• Cuando un programa C (o Fortran) con extensiones CUDA que corriendo en el **host** (la CPU) invoca un **kernel** GPU (sinónimo de una función GPU), un largo número de copias de este kernel, "**threads**" o "**hilos**", son distribuidas en los multiprocesadores del **device** (la placa), en donde se ejecutan.

• Una **grilla** de kernels se subdivide en **bloques** y cada bloque en varios hilos. Esta capa de abstracción facilita la programación.

•Una **SIMT** (single-instruction multiple thread unit) se encarga de manejar y ejecutar un gran número de threads en los cores disponibles.

•La SIMT crea, planifica y ejecuta los threads en grupos de 32.

•Un grupo de 32 threads forma un warp. Threads de un mismo warp corren en el mismo multiprocesador.



"UFO" (nuestra máquina):
Intel DX38 motherboard con Intel Core 2 Duo E8400, 2GB DDR3 1333MHz, con dos Nvidia GeForce GTX280 (240 cores y 1GB de memoria global, cada una).

Estrategia principal de paralelización (checkerboard)

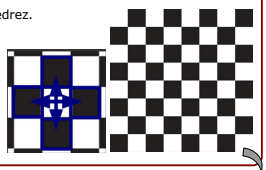
•Nuestro problema es 2D y con interecciones sólo a primeros vecinos (ver cuadro Modelo de Potts)

•La paralelización es trivial identificando a la matriz como un tablero de ajedrez.

•Todas los spins en casilleros blancos (negros) pueden ser actualizados simultáneamente (en paralelo) sin violar la secuencia temporal de la dinámica de Monte Carlo (i.e., sin violar balance detallado).

•Un paso de Monte Carlo consiste entonces en actualizar todas las blancas en paralelo e inmediatamente después todas las negras en paralelo.

•Los resultados son idénticos a los obtenidos en el Monte Carlo usual.



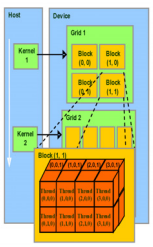
Características del código

•Asociamos cada spin a un hilo de ejecución diferente:
spin <--> thread

•Utilizamos un RNG independiente para cada thread: contamos con **512x512 RNG independientes** disponibles (pares buenos de semillas).

•Subdividimos el sistema LxL en "frames" de 512x512 que mandamos al device secuencialmente.

•Para el cálculo de cantidades físicas tales como energía y magnetización utilizamos primero un cálculo local en paralelo y luego sumas recursivas en paralelo (ambas sumas tipo mariposa), acelerando el programa.




```
void update(const float temp, Matrix S, float* energia, int* m) {
  /* Calculo array de probabilidades */
  double tmp_p[9];
  tmp_p[0] = exp(-4*temp);
  tmp_p[1] = exp(-3*temp);
  tmp_p[2] = exp(-2*temp);
  tmp_p[3] = exp(-1*temp);
  tmp_p[4] = 1;
  tmp_p[5] = exp(1*temp);
  tmp_p[6] = exp(2*temp);
  tmp_p[7] = exp(3*temp);
  tmp_p[8] = exp(4*temp);
  /* Copio el arreglo de probabilidades a la memoria constante */
  CUDA_SAFE_CALL(cudaMemcpyToSymbol(p, tmp_p, sizeof(tmp_p)));
  //necesito nFramesxnFrames de 512x512 cada uno
  //int nFrames = L >> 9; //en realidad lo "harcodeo" en los for's
  /* 512 en vez de L (ya que largo cuadrados de 512x512) */
  dim3 dimBlock(16, 16);
  dim3 dimGrid(512 * dimBlock.x / (512/2 + dimBlock.x), 512 * dimBlock.y / (512/2 + dimBlock.y));
  /* Actualizo los colores de una mitad de los bloques de las negras */
  updateCUDA<<<dimGrid, dimBlock>>>(temp, S, WHITES, S.blacks);
  CUDA_SAFE_CALL(cudaThreadSynchronize());
  /* Actualizo los restantes (la recorrida los valores de las blancas */
  updateCUDA<<<dimGrid, dimBlock>>>(temp, S, BLACKS, S.whites);
  CUDA_SAFE_CALL(cudaThreadSynchronize());
}
```

Pinta del código

Llamada a kernel

•Llegado el momento el programa corriendo en el host hace el llamdo a un kernel que se ejecuta en el device.

•Esta función se identifica por poseer las marcas "<<<" ">>>", conteniendo a la estructura de grillas y bloques de threads elegida para la ejecución del kernel.

•El transpaso de información (memoria) entre el host y el device se hace explícitamente, con una directiva CUDA, sólo cuando es necesario. Ésta es la operación más lenta en términos relativos.

El modelo ferromagnético de Potts y el problema de Binder

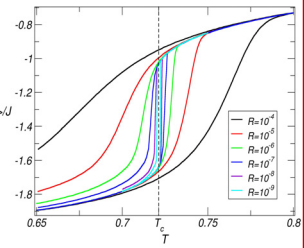
Modelo de Potts de q estados

$$\mathcal{H}_P = -J\sum_{\langle i,j\rangle}\delta(\sigma_i,\sigma_j) \qquad \sigma_i = 1,2,\ldots,q$$

temperatura de transición (exacta) [2]

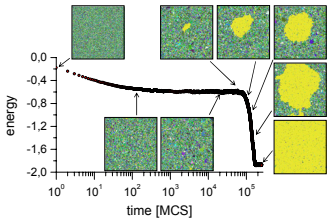
$$k_B T_c/J = 1/\ln\left(1+\sqrt{q}\right)$$

La transición es de 2do orden para q=2,3,4. Para ***q>4*** el sistema tiene una transición de **1er orden** acompañada *presumiblemente* por metaestabilidad e histéresis asociada [3].



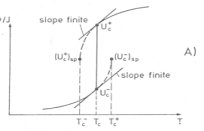
Ciclos de histéresis con T para q=9, N=2048² y distintas tasas de enfriamiento/calentamiento R.

Para q>4 quencheando de $T\sim\infty$ a $T^*(q)<T<T_c(q)$ vemos:

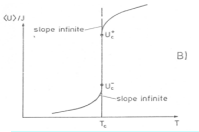


Realización individual para una muestra de L=200, luego de un quench a T=0.99Tc(q=24)

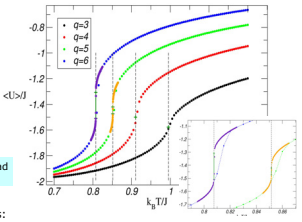
En 1981 Binder [1] estudia (con Monte Carlo) propiedades de equilibrio para q=3,4,5,6 y propone dos escenarios posibles para la transición de 1° orden (q>4) (en el límite termodinámico):



transición 1° orden con metaestabilidad pendiente finita $d<U>/dT$ a T_c



transición 1° orden sin metaestabilidad pendiente infinita $d<U>/dT$ a T_c



cada uno de estos escenarios serían consistentes con los comportamientos:

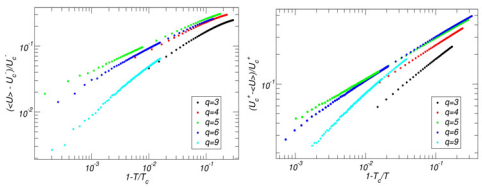
$$\langle U\rangle_{T<T_c} = (U_c^-)_{sp} - A^-(1-T/T_c^+)^{1-a_-} \qquad \langle U\rangle_{T<T_c} = U_c^- - A^-(1-T/T_c)^{1-a_-},$$

$$\langle U\rangle_{T>T_c} = (U_c^+)_{sp} + A^+(1-T_c^-/T)^{1-a_+} \qquad \langle U\rangle_{T>T_c} = U_c^+ + A^+(1-T_c/T)^{1-a_+},$$

Energía por spin vs. T para q=3,4,5,6, (símil fig.3 de [1]). En el inset se muestran un zoom para notar la dependencia con el tiempo de equilibración (t_{max}) en las cercanías de T_c: t_{max} = 2x10³ (azul y verde), t_{max} = 2x10⁵ (violeta y naranja). N=1024² .

Para determinar si se cumple A) o B) la propuesta es corroborar uno de estos dos comportamientos. Dado que U_c^+ y U_c^- son conocidos exactamente [2,4], podemos medir las diferencias de energía $\langle U\rangle - U_c^-)/U_c^-$ y $\langle U\rangle - U_c^+)/U_c^+$ y representarlas en un gráfico log-log. Si el escenario correcto es A) deberíamos observar un crossover a una pendiente (exponente) 1 al acercarnos lo suficiente a T_c.

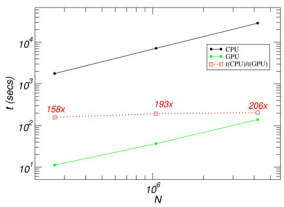
Para el escenario A) las estimaciones de $\Delta T/T_c$ son del orden de 10^{-3} para $q=6$ y aún menores para $q=5$. Debido a fenómeno de redondeo por tamaño finito, para distinguir en estos casos necesitamos sistemas de tamaño mayor a $10^3 x 10^3$ [1].



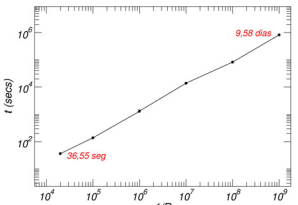
Sumado a esto, hemos detectado que además de tamaños grandes necesitamos tiempos largos de equilibración y buena estadística.

Creemos que el uso de las GPU nos permite ahora alcanzar el nivel de presición necesario para atacar esta pregunta en tiempos razonables. Este es el trabajo en curso.....

Performance



Tiempos de reloj característicos para un caso testigo (ciclo completo de histéresis para una muestra con q=9 entre T=0.65 y 0.8 a una tasa R=10⁻⁵) y distintos tamaños de sistema. En rojo se muestra la performance del código CUDA.



Pruebas del alcance de las medidas posibles en tiempos razonables utilizando las GPU. El punto correspondiente a R=10⁻⁹ hubiese demandado más de 5 años en CPU.

La performance de nuestro algoritmo ~200x resulta bastante superior a la obtenida recientemente para el modelo de Ising en GPUs (~60x) [5].

Referencias:

[1] K. Binder, *J. Stat. Phys.* **24**, 69 (1981).
[2] T. Kihara, Y. Midzuno, T. Shizume, *J. Phys. Soc. Japan.* **9**, 681 (1954).
[3] E.S. Loscar, E.E. Ferrero, T.S. Grigera, S.A. Cannas, *J. Chem. Phys.* **131**, 024120 (2009).
[4] R.J Baxter, *J. Phys. C* **6**, L445 (1973).
[5] T. Preis, P. Virnau, W. Paul, J.J. Schneider, Journal of Computational Physics 228, 4468 (2009)