Congr. Int. Ing. Electrón. Mem. Electro 2012, vol.34, pp. 1-6 Chihuahua, Chih. México
http://depi.itchihuahua.edu.mx/display/memorias_electro/MemoriaElectro2012.zip
ISSN 1405-2172

COMPARACIÓN DEL DESEMPEÑO ENTRE CUDA Y THRUST EN PROCESOS ESTOCÁSTICOS

Acuña-Galván Israel, Álvarez-Cedillo Jesús Antonio, Herrera Lozada Juan Carlos
Centro de Innovación y Desarrollo Tecnológico en Cómputo, Instituto Politécnico Nacional
Cómputo Científico
Unidad Profesional Adolfo López Mateos, Av. Juan de Dios Bátiz s/n casi esq. Miguel Othón de Mendizábal.
Colonia Nueva Industrial Vallejo, C.P. 07700, México D.F.
Tel. 5729 6000, Ext. 52528.
iacunag1200@alumno.ipn.mx, jaalvarez@ipn.mx, jlozada@ipn.mx

RESUMEN

En la nueva modalidad de hacer súper cómputo ya no se utilizan grandes sistemas centralizados, ni grandes cantidades de computadoras agrupadas, todo se centra en la tarjetas de vídeo, estas unidades de procesamiento gráfico, GPU ("Graphics processing unit") por sus siglas en inglés, son una opción accesible para realizar súper cómputo económico para cualquier centro de investigación, la programación de estos dispositivos es especializada y es necesario aprender un lenguaje específico, CUDA o una API de esta llamada THRUST, estas herramientas son nuevas y la documentación sobre estas es escasa. En este trabajo se hace una comparación de la eficiencia de estas dos plataformas aplicada a la simulación de procesos estocásticos, obteniéndose mejores resultados con THRUST. También se analiza el impacto sobre el desempeño al utilizar diferentes tipos de datos, en este caso los datos tipo primitivos son mas eficientes.

1. INTRODUCCIÓN

La tecnología GPGPU (general purpose graphics processing units) se ha estado aplicando en varias áreas del cómputo, desde el cómputo científico, el cómputo en la nube, visualización, juegos, por mencionar algunas, incluso está re-definiendo la forma de asimilar el cómputo paralelo [1].

Un problema se puede resolver en una computadora si es posible crear un algoritmo que implemente la solución del problema, el encontrar la solución del problema dependerá de la complejidad computacional del algoritmo, algunos algoritmos tienen una alta complejidad computacional, lo cual demanda mucho tiempo de procesamiento, por lo tanto no son viables de implementar. Una forma de disminuir la complejidad computacional de un algoritmo es por medio de la paralelización, es decir, hacer varios pasos a la vez.

Muchos problemas se pueden resolver computacionalmente por medio de procesos estocásticos, por ejemplo en el área de electrónica, de economía, física, entre otras, los procesos estocásticos se basan en el estudio y modelización de sistemas que evolucionan a lo largo del tiempo, o del espacio, de acuerdo a unas leyes no determinísticas, esto es, de carácter aleatorio [2]. Una herramienta de este tipo muy utilizada en la investigación es el método Monte Carlo. En éste se generan números aleatorios con cierta distribución, si se usa una computadora se tendrán que generar números pseudoaleatorios por medio de algoritmos aritméticos [3] posteriormente estos valores son tratados con base al problema que se quiere resolver, pero por lo general las operaciones sobre un dato son independientes de las operaciones sobre los demás y al final se obtienen una serie de promedios, por tal motivo el método de Monte Carlo es susceptible de ser paralelizado [4] y por lo tanto de ser implementado en una GPU (*Graphics processing unit*).

El método de Monte Carlo ha sido paralelizado e implementado en varias arquitecturas paralelas, el hacerlo en una gpu tiene las siguientes ventajas; un costo relativamente bajo, de fácil acceso en el mercado, nulo mantenimiento, pueden ser instaladas en una computadora de escritorio o una portátil, bajo consumo de energía en comparación con clusters convencionales [5].

Las GPU de NVIDIA se han diseñado para ser programados usando CUDA (Compute Unified Device Architecture), que es un conjunto de instrucciones que permiten por medio de bibliotecas ser accedidas por lenguajes de alto nivel tales como C/C++ o FORTRAN.

ISSN 1405-2172

CUDA permite a los desarrolladores especificar en forma detallada como serán descompuestos y ejecutados los cálculos en el GPU, por lo tanto facilita la implementación eficiente de algoritmos, una desventaja de CUDA es que requiere del conocimiento a nivel de hardware de la arquitectura [6]. THRUST es una biblioteca de algoritmos optimizados utilizada para programar con CUDA esta tiene la ventaja de ser similar a la STL (standard template library) de C, motivo por el cual hace la programación de GPU's concisa, legible y eficiente además que es totalmente compatible con CUDA [7].

En este trabajo se realizan dos aproximaciones por el método de Monte Carlo, se implementan tres versiones de cada uno, una secuencial, otra con CUDA y la última con THRUST, se analiza cual implementación es la más veloz y se evalúa que tipo de datos es mas conveniente usar en base al cálculo del speedup.

2. DESARROLLO

2.1. Metodología

Se realizaron seis de programas para aproximar el valor de π por medio del método de Monte Carlo, estos programas se dividieron en dos casos, en el caso *a* se utilizan arreglos de tipo flotante para realizar los cálculos, mientras que en el caso *b* se utilizaron arreglos de una estructura que representa un par ordenado, para cada caso se implementaron tres versiones, una secuencial, una con CUDA y una con THRUST, en las versiones paralelas los números aleatorios fueron generados en el GPU.

El método utilizado para aproximar el valor de π se basa en obtener la probabilidad de que *n* cantidad de puntos aleatorios generados dentro de un cuadrado estén también dentro de un circulo de radio *r* inscrito en el cuadrado, el algoritmo es el siguiente.

1. Sea **P** un vector de *n* pares ordenados aleatorios con una distribución uniforme generados en el intervalo [-1, 1], tal que **P** = $[P_0, ..., P_i, ..., P_n]$ donde $0 \leq i \leq n$.
2. Sea **D** un vector de *n* elementos, tal que **D** = $[D_0, ... D_i, ..., D_n]$ donde $0 \leq i \leq n$.
3. Para cada P_i calcular la distancia D_i con respecto al origen.
4. Contar los elementos *c* de **D,** tales que $D_i \leq r$.
5. $\pi \leftarrow 4 \cdot c/n$

Posteriormente se aproximo el volumen de una esfera de radio r por el método de Monte Carlo, la cantidad de programas y el enfoque que se les dio fue el mismo que se siguió al aproximar el valor de π, el algoritmo utilizado es el que se muestra a continuación.

1. Sea **P** un vector de *n* pares ordenados aleatorios con una distribución uniforme generados en el intervalo [-1, 1], tal que **P** = $[P_0, ..., P_i, ..., P_n]$ donde $0 \leq i \leq n$.
2. Sea **D** un vector de *n* elementos, tal que **D** = $[D_0, ... D_i, ..., D_n]$ donde $0 \leq i \leq n$.
3. Para cada P_i calcular la distancia D_i con respecto al origen.
4. Contar los elementos *c* de **D,** tales que $D_i \leq r$.
5. $f \leftarrow 0$
6. Para cada D_i si este es menor o igual a *r*, $f \leftarrow f + D_i$.
7. Volumen = $2 \cdot \pi \cdot r^2 \cdot f / n$.

Cada archivo fue ejecutado 10 veces y se promediaron los tiempos que duró cada ejecución. Posteriormente se comparó la eficiencia con respecto a las versiones secuenciales calculando el speedup. Esto fue relizado tanto para la aproximación de π como para el volumen de la esfera.

Para la generación de números aleatorios se ocupo la función rand( ), la función curandGenerateUniform de libreria curand y la función thrust::random::uniform_real_distribution para las implementaciones secuenciales, con CUDA y con THRUST respectivamente, todas estas generan pseudonúmeros aleatorios con una distribucion uniforme.

En esas pruebas se utilizó un GPU GeForce 310M de Nvidia con 512 MB de memoria, la mayor cantidad de puntos aleatorios alcanzada fue de diez millones.

3. RESULTADOS

Al analizar los datos correspondientes a la aproximación de π (Figura 1) se observa que cuando los datos son menores a diez mil la implementación secuencial es mas rápida que cualquiera de las implementaciones paralelas, pero cuando los datos son mayores a diez millones la versión en THRUST es la más rápida siguiéndole la de CUDA, también se puede apreciar que en las

ISSN 1405-2172

versiones paralelas al usar arreglos de estructuras (datos compuestos) se obtienen mejores velocidades por debajo de los diez mil datos, pero cuando los datos son mayores a diez millones los mejores tiempos se obtuvieron al utilizar datos tipo flotantes (datos primitivos).

Este comportamiento se repite al analizar los datos correspondientes a la aproximación del volumen de la esfera (Figura 2), cuando los datos son mayores a los diez millones los mejores tiempos se obtuvieron al manejar datos tipo flotante. Tanto para la aproximación de π como para la aproximación del volumen de la esfera se observa que el comportamiento de los tiempos de ejecución de los programas implementados con CUDA y con THRUST tienen cierto grado de paralelismo, siendo los de THRUST más rápidos, esto significa que las implementaciones en CUDA utilizdas en este trabajo son susceptibles de ser optimizadas mientras que los algoritmos que utiliza la API de THRUST son algoritmos optimizados [8].

Al analizar el desempeño de las versiones de CUDA y THRUST con respecto a su versión secuencial, con excepción de la implementación en CUDA que aproxima el volumen de la esfera, se observa que tanto para la aproximación de pi (Figura 3) como para el volumen de la esfera (Figura 4) los mejores valores de speedup se obtienen al utilizar datos de tipo flotante en vez de estructuras, y debido a que los algoritmos que utiliza THRUST son optimizados, las implementaciones con THRUST son mejores que las utilizan solamente CUDA. Comportamientos similares, donde las implementaciones con THRUST requieren menos tiempo que las implementadas con CUDA han sido reportados en otros trabajos [9] [10].

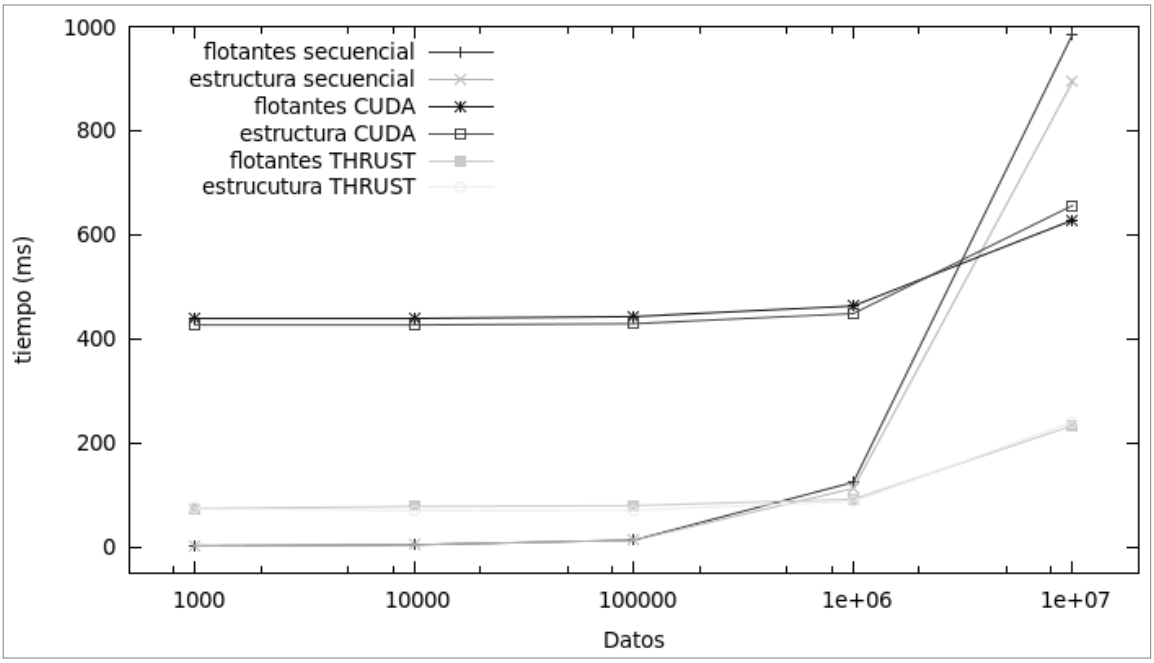

Figura 1. Comparación de los tiempos que duran diferentes implementaciones, con diferentes tipos de datos al aproximar π

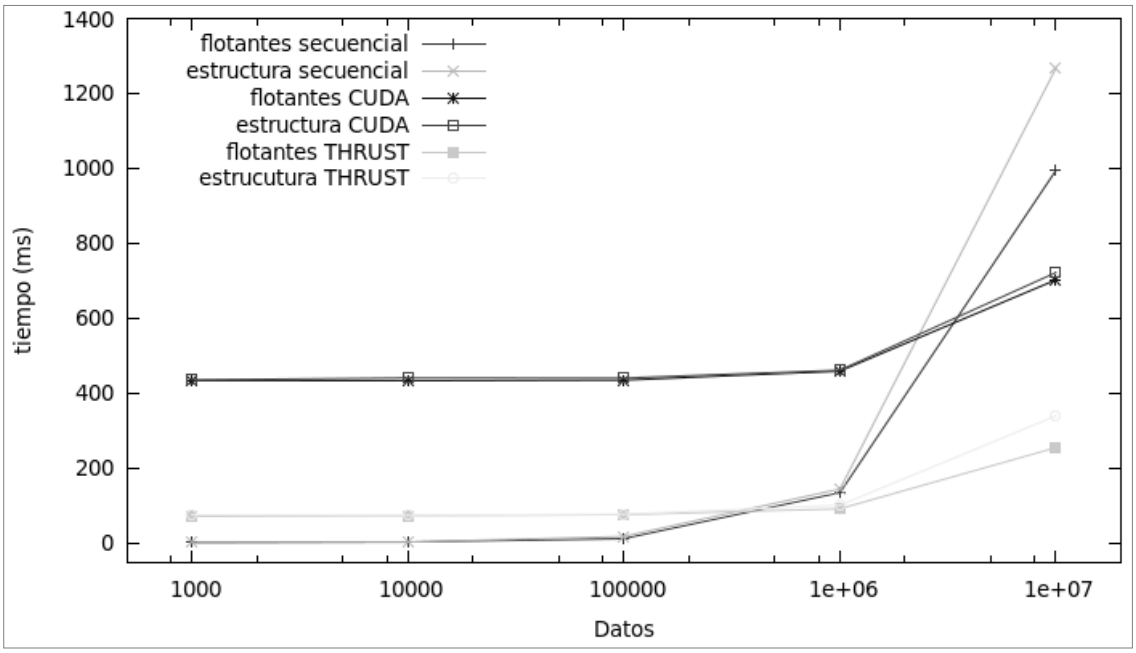

Figura 2. Comparación de los tiempos que duran diferentes implementaciones, con diferentes tipos de datos al aproximar el volumen de una esfera por el método de Monte Carlo

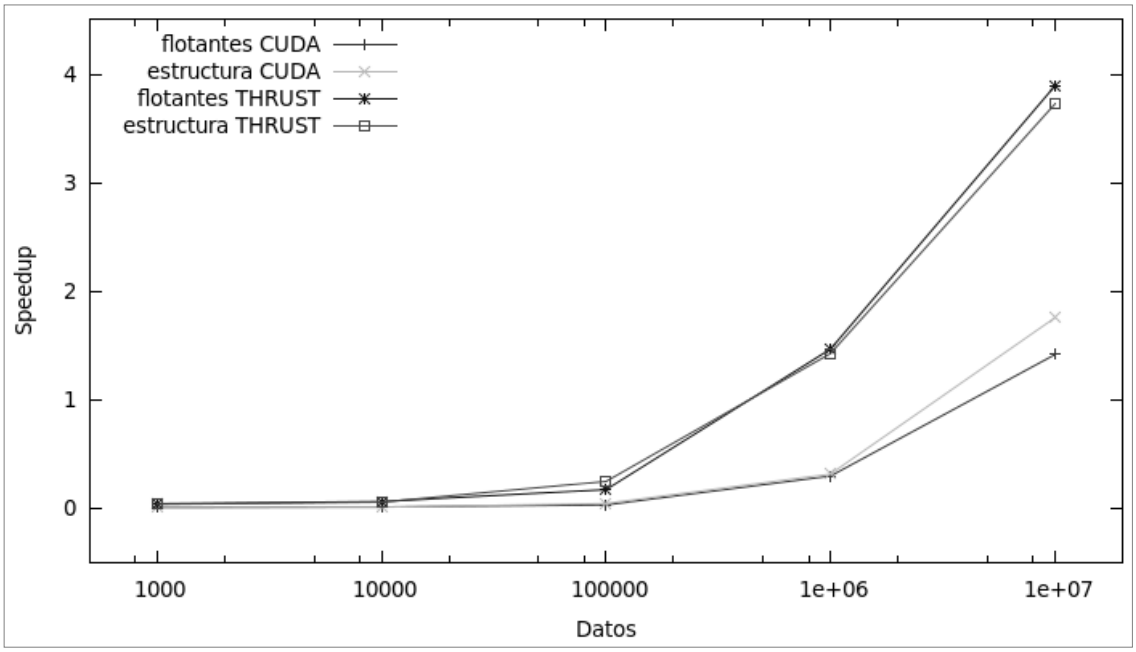

Figura 3. Comparación del speedup para CUDA y THRUST con diferentes tipos de datos al aproximar π por el método de Monte Carlo

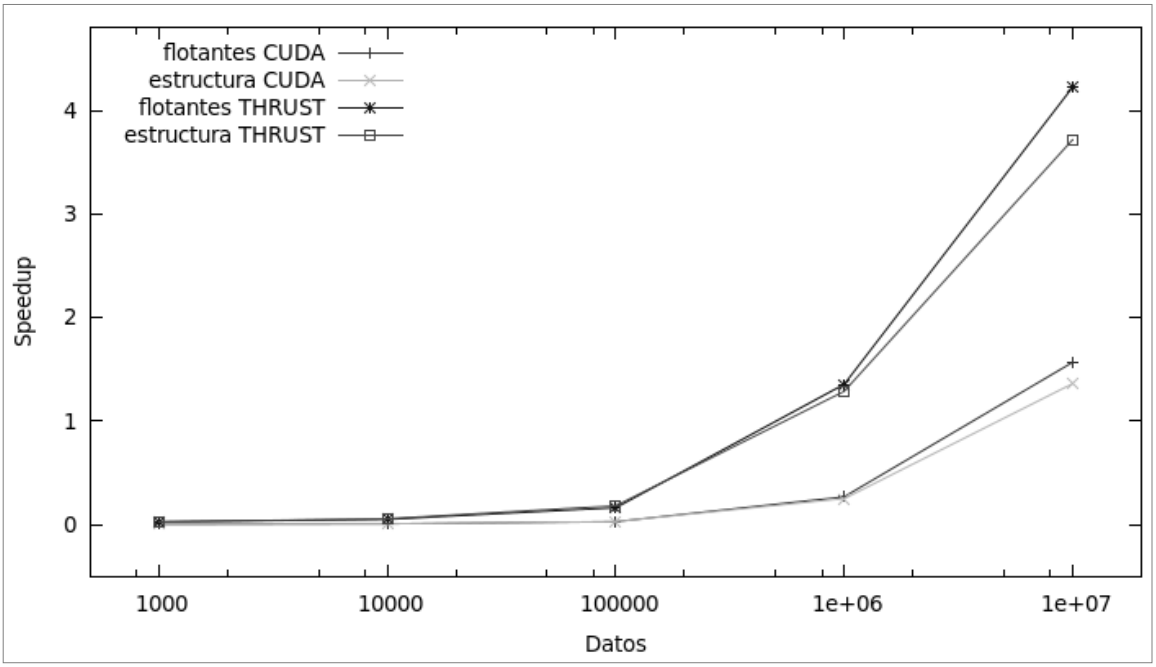

Figura 4. Comparación del speedup para CUDA y THRUST con diferentes tipos de datos al aproximar el volumen de una esfera por por el método de Monte Carlo

4. CONCLUSIONES

La paralelización de algoritmos estocásticos llevada a cabo sobre un GPU se puede implementar tanto con CUDA como con TRHUST, sin embargo, como se mencionó antes, el uso de CUDA implica tener conocimientos de la arquitectura a nivel de hardware, por ejemplo se debe distribuir el trabajo entre los procesadores del gpu por medio de las asignación de threads, bloques y mallas, por otro lado al usar THRUST la programación no requiere de estos conocimientos, además la programación con este es más eficiente, esto no implica que no sea posible alcanzar estos niveles de eficiencia con CUDA, ya que finalmente THRUST es una biblioteca basada en CUDA. Cabe señalar que el código resultante al utilizar THRUST es menor que el se obtiene al utilizar CUDA, además su interfaz es similar a la de la STL (standard template library) lo cual hace sencilla su implementación, y se puede combinar con CUDA en un mismo programa.

Con respecto al tipo de dato, las mejores implementaciones se lograron al utilizar datos de tipo flotante en lugar de estructuras, generalizando con base en este trabajo es mejor implementar un algoritmo con base en datos primitivos en vez de datos compuestos.

5. REFERENCIAS

[1] R. Faber, "Cuda Application Design and Development", Elsevier, 2011.

[2] D. Xiu, "Numerical Methods for Stochastic Computations: A Spectral Method Approach", Princeton University Press, 2010.

[3] R. Y. Rubinstein, "Simulation and the Monte Carlo Method", John Willey & Sons. 1981.

[4] J. S. Rosenthal, "Parallel computing and Monte Carlo algorithms", Far East Journal of Theoretical Statistics, vol. 4,pp 207–236 , 2000.

[5] A. Lee. C. Yau, M. B. Giles. A. Doucet, C. C. Holmes, (2009, July). "On the utility of graphics cards to perform massively parallel simulation of advanced Monte Carlo methods". [On line]. Available: http://arxiv.org/abs/0905.2441

[6] S. Che. J. Meng. J. W. Sheaffer. K. Skadron, (2008, June). "A Performance Study of General Purpose Applications on Graphics Processors". *Journal of Parallel and Distributed Computing*, [On line]. 68 (10), pp 1370–1380. Available: http://www.sciencedirect.com/science/article/pii/S0743731508000932

[7] Wen-Mei W Hwu, Gpu Computing Gems Jade Edition, Elsevier, 2011.

[8] A. Aaichmayr. (2011, March). "Implementation of Virtual Embryology using the Thrust library for CUDA". [Online]. Available: http://hgpu.org

[9] G. Ulman. (2010, April). "Bayesian Particle Filter Tracking with CUDA". [online]. Available: http://public-digital-library.googlecode.com/svn/trunk/ComputerVision/Real-timeVisualTrackerbyStreamProcessing.pdf

[10] P. Weimann. S. Wenger. M. Magnor. (2001, Jan.) "CUDA expression templates". *Communication Papers Proceedings*. [Online]. pp. 185–192. Available: http://graphics.tu-bs.de/publications/wiemann2011cuda/