Optimización Sistólica sobre GPUs

Pablo Vidal[1], Francisco Luna[2], and Enrique Alba[2]

[1] Universidad Nacional de La Patagonía Austral, Caleta Olivia, Sta. Cruz, Argentina
pjvidal@uaco.unpa.edu.ar
[2] E.T.S.I. Informática, Universidad de Málaga, Málaga, España
{flv,eat}@lcc.uma.es

Resumen En este trabajo se estudia una nueva aproximación algorítmica basada en un modelo de procesamiento paralelo conocido como computación sistólica. Este algoritmo se ejecuta sobre una Unidad de Procesamiento Gráfico empleando CUDA (Compute Unified Device Architecture). El algoritmo se compone de una matriz de celdas, donde cada una de ellas trabaja sobre una solución, realizando operaciones que la modifican para luego moverla a la celda contigua y repetir el proceso. Hemos evaluado el comportamiento del algoritmo sobre diferentes instancias del problema de la mochila multidimensional. La evaluación experimental sobre las instancias seleccionadas demuestra la eficiencia y competitividad de nuestra aproximación.

Keywords: Computación Sistólica, CUDA, Optimización, GPU

1. Introducción

El interés en la computación paralela ha ido creciendo progresivamente desde la primera aparición de los ordenadores. La necesidad de resolver problemas de forma cada vez más rápida ha llevado a los investigadores a crear nuevo hardware y, así mismo, a desarrollar nuevas herramientas de software para afrontar las demandas de ejecución en diversos ámbitos tales como la física, simulación, bioinformática, comunicaciones y otros campos de investigación y negocios [8].

Uno de los enfoques paralelos implementados en la literatura ha sido la Computación Sistólica. La Computación Sistólica fue desarrollada en la Universidad de Carnegie-Mellon por Kung y Leiserson en 1979 [10]. La idea básica se centra en la creación de un vector conformado por diferentes procesadores, y donde cada uno de ellos realiza operaciones simples de cómputo. La información se mueve a continuación al procesador contiguo para repetir el proceso, generando así un continuo flujo de información en toda la estructura. Sin embargo, esta arquitectura presentó diferentes dificultades en el pasado para la construcción de ordenadores sistólicos y, sobre todo, para programar algoritmos de alto nivel sobre una arquitectura de bajo nivel. No obstante, hoy en día, los avances tecnológicos han hecho emerger nuevas plataformas de cómputo tales como las Unidades de Procesamiento Gráfico (GPUs, por sus siglas en inglés), las cuales están especialmente diseñadas para trabajar con paralelismo masivo y proporcionan un entorno donde es posible desarrollar diferentes modelos algorítmicos.

Recientemente, se ha propuesto explotar la capacidad computacional disponible en las tarjetas gráficas de las computadoras personales, a fin de solucionar problemas de propósito general [12]. Desde entonces los investigadores han considerado a las GPUs como un área de investigación prometedora.

En este trabajo, presentamos una nueva aproximación llamada *Systolic Neighborhood Search* (SNS, por sus siglas en inglés), la cual utiliza conceptos de la computación sistólica implementados sobre la arquitectura de la GPU. Por consiguiente, nuestra primera contribución es considerar la GPU como una plataforma adecuada para llevar a cabo lo que denominamos *optimización sistólica*. Este tipo de modelo ha demostrado ser eficaz en otros problemas de optimización como se muestra en [2]. El modelo se basa en una estructura de malla donde cada componente tiene asignada una solución en particular. Cada celda realiza perturbaciones simples sobre la solución y después se desplaza a la celda contigua para repetir el proceso. Al realizar perturbaciones simples y mover soluciones entre celdas se busca un procesamiento rápido y un continuo descubrimiento en el espacio de búsqueda, intentando tener un impacto directo sobre la calidad de las soluciones. Utilizamos el problema de la mochila multidimensional para evaluar el comportamiento numérico y también, comparamos los resultados con los obtenidos por una implementación de la búsqueda aleatoria en CPU y GPU.

El trabajo está estructurado de la siguiente manera. En la sección 2 se explica detalladamente la idea de de computación sistólica y el trabajo relacionado. En la sección 3 se desarrolla el concepto de GPU y de la plataforma CUDA. En la sección 4 se introduce el nuevo algoritmo desarrollado y se detallan aspectos del mismo. En la sección 5 se presenta el problema y las instancias seleccionadas, luego se describe los experimentos realizados y posteriormente se analizan los resultados obtenidos. Por último, la sección 6 incluye conclusiones y se sugieren futuras líneas de investigación.

2. Computación Sistólica

La computación sistólica es un modelo real que fusiona los conceptos de *pipelining*, paralelismo y conexión entre elementos. Un algoritmo sistólico utiliza un vector de elementos (celdas) interconectados entre sí llamado vector sistólico. El trabajo de cada celda es el mismo y se centra en realizar operaciones muy simples, por ejemplo suma o multiplicación. Luego de cada operación, la información se traslada a la celda contigua para volver a realizar otra operación simple. La Figura 1 muestra la entrada de los datos d y e en cada celda o, en cada una de ellas se realiza la misma operación sobre los datos y luego se traslada la información a otra celda según un flujo predefinido síncrono. La idea es que el vector sistólico mantenga un flujo constante de la información y que la misma se vaya transformando hasta producir una salida deseable.

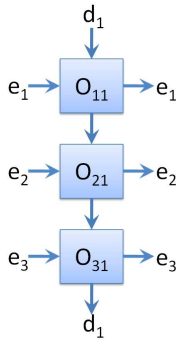


Figura 1: Vector Sistólico

En la literatura se han registrado apenas algunas implementaciones en el campo de optimización [4] [11] sin mayor notoriedad ni continuación del trabajo debido a los inconvenientes de programar en hardware.

3. Unidades de Procesamiento Grafico

Las GPUs son consideradas comúnmente como un dispositivo, usado como coprocesador gráfico de la CPU. Con este tipo de arquitectura se busca aliviar la carga computacional de la CPU. Los modelos actuales de GPU suelen tener una gran cantidad de procesadores, los cuales están optimizados para ejecutar una instrucción simple sobre cada elemento de un extenso conjunto de elementos (SIMD, por sus siglas en inglés).

Para poder utilizar adecuadamente la capacidad de cómputo de la GPU, NVIDIA ha desarrollado una herramienta de programación llamada CUDA [1]. CUDA permite a los programadores implementar funciones llamadas *kernels*. Un *kernel* contiene la porción de código que sera ejecutada en la GPU. Esta función es invocada desde el host y se despliega en la GPU. CUDA nos da la posibilidad de implementar los kernels usando lenguaje C estándar más algunas extensiones de NVIDIA. Además, nos permite organizar el paralelismo en tres niveles: rejilla, bloque e hilo. Cada vez que se invoca un kernel, se crea una rejilla de bloques, los cuales a su vez agrupan múltiples hilos (ver Figura 2).

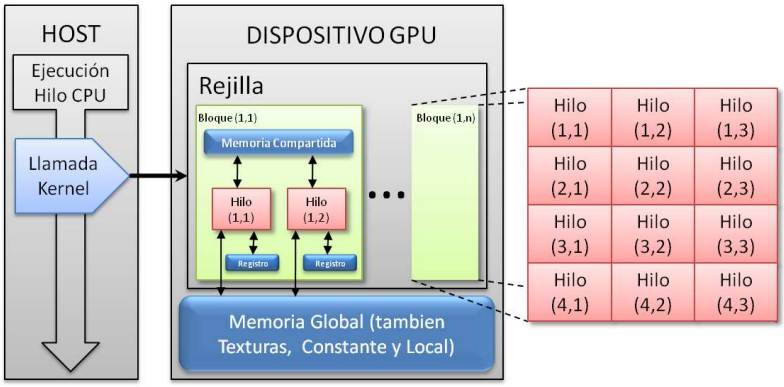


Figura 2: Arquitectura de la GPU

Durante la ejecución del kernel, cada hilo tiene acceso a diferentes tipos de memoria dentro de la GPU, según una jerarquía predefinida por NVIDIA. Esta jerarquía abarca registros, memoria local y compartida, luego continua con la memoria global, constante y texturas (siguiendo un planteamiento abajo-arriba).

Los registros proveen lectura/escritura para cada hilo, mientras que la memoria compartida es utilizada por aquellos hilos dentro de un mismo bloque. La memoria global es la más grande dentro del dispositivo GPU, sin embargo, existen espacios reservados en ella para otros tipos de memoria tales como la local, constante o para texturas. La memoria compartida y los registros son las más rápidos, pero su tamaño está limitado por estar dentro del circuito integrado de la GPU. Por otra parte, la memoria localizada en la tarjeta del dispositivo (local, global, constante y texturas) es grande pero presenta mayor latencia en los accesos, en comparación con las dos anteriores.

4. Systolic Neighborhood Search

El objetivo de esta sección es presentar nuestra propuesta algorítmica, llamado *Systolic Neighborhood Search* (SNS). La idea esencial trabaja con una malla toroidal donde las soluciones se encuentran localizadas en cada celda. En cada una de las celdas se realiza una operación simple de perturbación sobre la solución y el resultado se mueve a la celda contigua para realizar el mismo proceso. Las soluciones se mueven a través de toda la fila sufriendo múltiples cambios en cada celda con el objetivo de mejorar su calidad. El potencial del algoritmo se basa en la exploración del espacio de búsqueda de manera eficiente y estructurada mediante la ejecución de continuas modificaciones en las soluciones de manera sistemática.

El pseudocódigo del SNS se detalla en el Algoritmo 1. El SNS comienza definiendo el tamaño de la malla de acuerdo al tamaño del problema a resolver. El SNS genera aleatoriamente cada solución y posteriormente evalúa cada una de ellas. A continuación, se realizan diversas operaciones en cada celda buscando mejorar la solución residente actual. El SNS trabaja con base a tres componentes principales: una topología definida, la computación en celdas y el flujo de información. Cada uno de ellos es explicado a continuación. Asimismo, estas operaciones se encuentran en el pseudocódigo presentado en el Algoritmo 1.

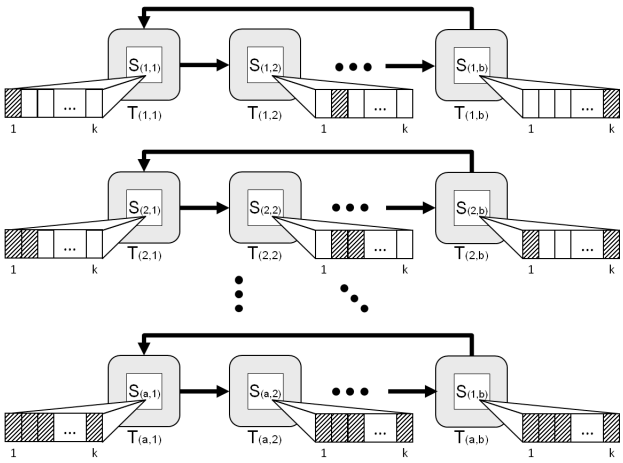


Figura 3: Systolic Neighborhood Search

Topología El SNS utiliza una malla toroidal para ubicar las soluciones. La Figura 3 muestra la disposición de las celdas y soluciones. La malla está definida por un tamaño $a \times b$. Las dimensiones se definen con base al número n de elementos dado por un problema que se va a resolver. De esta manera, a y b son configuradas de acuerdo al valor de n (tamaño del problema).

Cómputo en Celdas Una vez definida la malla, es necesario explicar la configuración de las operaciones realizadas en cada una de las celdas. Cada solución

Algorithm 1 Pseudocódigo del SNS canonico

```
1: Definir variables a y b dependiente del tamaño del problema
2: Definir malla M de tamaño a × b
3: para todo sol_i ∈ M hacer en paralelo
4:    sol_i ←generarSolucion(sol_i);
5:    sol_i ←evaluarSolucion(sol_i);
6: fin for
7: mientras (no se cumpla criterio de parada) hacer
8:    para todo sol_i ∈ M hacer en paralelo
9:       Calcular la posición (x, y) para cada sol_i
10:      sol'_i ← sol_i
11:      para init = 0 a (x − 1) hacer
12:         sol'_i ← aplicar operaciones de cambio sobre la posición (y+init) en el vector de valores
13:      fin para
14:      sol'_i ← parcial_eval(sol'_i)
15:      si sol'_i MEJOR QUE sol_i entonces
16:         sol_i ← sol'_i
17:      fin si
18:      esperar la terminación de todas las operaciones en todas las celdas (entre lineas 8-17)
19:      calcular sigPosicion = (y + 1) mód n
20:      sol_(x,sigPosicion) = sol_(x,y)
21:   fin for
22: fin mientras
23: getBest(M).
```

sol_i esta conformada por un vector solución, que contiene la especificación de cada solución. En cada celda se ejecuta una operación simple de perturbación sobre un cierto grupo de componentes en el vector solución.

El proceso se inicia realizando una copia llamada sol'_i de la solución original sol_i. Los cambios que se producen sobre sol'_i son definidos a partir de los parámetros (x, y) (es el índice para cada celda). El parámetro x define el número de valores a cambiar en cada solución, de esta forma, a medida que el número de la fila va aumentando, también lo hace el número de elementos que cambiar dentro de cada solución. El parámetro y indica a partir de qué posición se producirán los cambios en el vector solución. Por ejemplo, si tenemos una solución $sol_{(3,2)}$, se modificarán tres valores a partir de la posición 2. De ahí, la importancia de tener una malla con dimensiones iguales al tamaño del problema, que busque evaluar todos los grupos posibles de cambios sobre el vector solución, partiendo de cada posición factible dentro del mismo.

Una vez realizados los cambios, procedemos a re-evaluar sol'_i, usando una función de evaluación parcial para no consumir demasiado tiempo de cómputo realizando una re-evaluación completa de toda la solución. De esta manera, solo se evalúan aquellas partes de la soluciones que han sufrido modificaciones. Sí sol'_i es mejor que sol_i, se remplaza sol_i con la copia, de otra forma, la solución antigua queda sin sufrir ningún cambio.

Flujo de Información El SNS mueve cada solución a la siguiente celda de manera síncrona al mismo tiempo. De esta forma, al tener una malla con n columnas se pueden perturbar todos los grupos posibles de componentes dentro de las soluciones.

5. Experimentación

En esta sección presentamos y analizamos los resultados obtenidos por el SNS al resolver instancias del Problema de la Mochila Multidimensional. Para ello procederemos en tres etapas, abordando el estudio de la eficiencia numérica, seguido de un análisis del consumo de tiempo ejecución de los algoritmos examinados, para finalizar con una discusión del comportamiento del modelo propuesto.

Hemos organizado esta sección en tres subsecciones. La primera formaliza el problema estudiado y las instancias seleccionadas. En la segunda expondremos los parámetros utilizados para la ejecución de los algoritmos, mientras que en la tercera estudiaremos los resultados obtenidos por los diferentes algoritmos en cada instancia.

5.1. Problema de la Mochila Multidimensional (MKP)

El MKP es un problema de optimización combinatoria que ha recibido amplia atención de la comunidad investigadora, ya que el planteamiento matemático de muchos problemas prácticos de la vida cotidiana lleva al modelo del MKP, por ejemplo, [6] [7] [13] .

$$\text{maximizar } \sum_{j=1}^{n} p_j x_j \tag{1}$$

$$\text{sujeto a } \sum_{j=1}^{n} w_{ij} x_j \leq C_i,\ i \in I,\ I=\{1,...,m\} \tag{2}$$

$$x_j \epsilon\ \{0,1\},\ j \in J,\ J=\{1,...,n\} \tag{3}$$

Este problema está formado por un conjunto n de elementos y un conjunto m de mochilas. El parámetro p_j indica el beneficio del elemento j, w_{ij} indica el peso del elemento j respecto a la mochila i y C_i es la capacidad de la mochila i. Se busca encontrar un vector $x = (x_1, x_2, ..., x_j)$, que maximice la ecuación 1, y al mismo tiempo cumpla las restricciones de la ecuación 2.

Aquellas soluciones que no cumplen las restricciones de la ecuación 2 se convierten en soluciones no factibles. Nosotros hemos utilizado una función presentada por Gottlieb [9] para penalizar las soluciones no factibles:

$$penalizacion_x = \frac{p_{max} + 1}{w_{min}}.max\{CV_{(x,i)} |\ i\ \epsilon\ I\} \tag{4}$$

donde $CV_{(x,i)} = max(0, \sum_{j \epsilon J} w_{ij} * x_j - C_i)$ e indica la cantidad de restricciones violadas para una restricción $i \in I$, $w_{min} = min\{w_{ij} | i \in I, j \in J\}$ indica el mínimo peso que se puede tener y finalmente el máximo beneficio es representado por $p_{max} = max\{p_j | j \in J\}$. $(p_{max} + 1)/w_{min}$ tiene como finalidad correlacionar la función objetivo sobre la base de los beneficios con la demanda de recursos que superan las limitaciones de las mochilas. Esta función se basa en una estimación pesimista de los beneficios que se perderían sí los elementos fueran retirados de la mochila con el fin de obtener soluciones factibles. Para más detalles, ver [9].

Las instancias seleccionadas están incluidas dentro de las propuestas por Chu y Beasly en la OR-Library [5]. Hemos seleccionado 3 instancias llamadas WEISH18 $= \{n = 105, m = 2\}$, 10.250.0 $= \{n = 250, m = 10\}$, 10.500.0 $= \{n = 500, m = 10\}$. Hemos utilizado trabajos anteriores existentes en la literatura para comparar con los mejores óptimos conocidos [15] [16], estos valores se muestran en la Tabla 1a, en la fila *Optimo*.

5.2. Parametrización

Con el fin de hacer una comparación justa entre todos los algoritmos hemos utilizado un número dinámico de evaluaciones, para garantizar una exploración similar en el espacio de búsqueda en cada algoritmo. Para ello, se han tomado como base trabajos previos sobre MKP [5] [14] [3]. Dado el valor resultante de $evals = n * 5000$, el número de evaluaciones es:

$$maxEvals = \begin{cases} n * 5000, si(evals >= 400000) \& (evals <= 3000000) \\ 400000, si(evals < 400000) \\ 3000000, si(evals > 3000000) \end{cases} \quad (5)$$

La búsqueda aleatoria (RS, por sus siglas en inglés) en CPU (RS_{CPU}) utilizada para comparar es la version canónica existente en la literatura y se usa para asegurar que el SNS tiene un comportamiento inteligente no trivial. Con respecto a la versión en GPU (RS_{GPU}), esta funciona con una población de tamaño n. Básicamente, cada solución es administrada por un hilo de la GPU. Cada hilo aplica en paralelo el proceso de una RS canónico sobre cada solución. Cuando la condición de parada es alcanzada, el algoritmo selecciona la mejor solución vista en la vida del algoritmo.

Se han realizado 30 ejecuciones independientes para cada algoritmo en cada instancia. Para obtener resultados estadísticamentes significativos en primer lugar, la prueba de Kolmogorov-Smirnov se lleva a cabo con el fin de comprobar si los valores de los resultados siguen una distribución normal (Gaussiana) o no. Si la distribución es Normal, entonces se aplica el test de Levene para comprobar la homogeneidad de las varianzas. Sí las muestras tienen varianzas iguales (test de Levene positivo), una prueba de ANOVA se lleva a cabo, de lo contrario una prueba de Welch se lleva a cabo. Para distribuciones no Normales, la prueba no paramétrica de Kruskal-Wallis se utiliza para comparar las medianas de los algoritmos. En este trabajo siempre tenemos en cuenta un nivel de confianza del 95 %. Ésto significa que podemos garantizar que las diferencias de los algoritmos de comparación son significativas o no con una probabilidad del 95 %.

Las pruebas se han ejecutado sobre un PC con un procesador Intel (R) i7 CPU 920, con 4096 MB de RAM. El sistema operativo Ubuntu Lucid 10.04.2. Para el caso de la GPU, hemos usado una NVIDIA GeForce GTX 285 con 1024 MB de DRAM. Hemos utilizado CUDA versión 3.1 y el controlador para la tarjeta gráfica es la versión 257.21.

5.3. Resultados Experimentales

A continuación abordaremos los resultados de fitness y tiempo obtenidos para cada algoritmo. Los resultados de fitness se resumen en las Tablas 1a, donde cada fila indica el fitness promedio de las 30 ejecuciones independientes obtenido respecto de cada algoritmo en cada instancia. Con letra **negrita** y resaltado en gris aparecen los mejores valores obtenidos. La ultima fila indica si existe significancia estadistica (con el signo +) o no (signo −).

Tabla 1: Resultados de fitness y tiempo de los algoritmos para cada instancia

(a) Resultados de Fitness

Algoritmo	Instancias		
	WEING7	10.250.0	10.500.0
RS_{CPU}	933266,800	31590,000	85620,000
RS_{GPU}	931947,967	36030,000	86166,000
SNS_{CPU}	1086461,333	40577,750	88173,200
SNS_{GPU}	1089370,833	41696,667	89833,333
Optimo	1095445,000	58097,000	119215,000
	+	+	+

(b) Resultados de Tiempo

Algoritmo	Instancias		
	$WEING7$	10,250,0	10,500,0
RS_{CPU}	0,668	9,307	35,542
RS_{GPU}	1,406	10,098	20,181
SNS_{CPU}	0,684	9,727	42,490
SNS_{GPU}	0,080	1,082	4,339

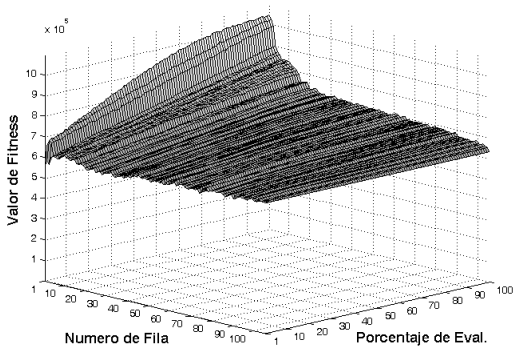
El primer resultado observable en la Tabla 1a es que ninguno de los algoritmos ejecutados alcanza el mejor óptimo conocido. Entre los algoritmos, el SNS_{GPU} destaca por tener valores de fitness muy cercanos al óptimo con respecto a los otros algoritmos. Como segundo resultado vemos que el SNS_{CPU} también obtiene valores apenas por debajo del SNS_{GPU}. Las versiones del RS han quedado muy por debajo de los valores de fitness de las dos versiones SNS. Ésto nos indica que el esquema del algoritmo SNS obtiene ventaja en la exploración del espacio de búsqueda a través de los cambios sistemáticos realizados en cada celda y el movimiento continuo de información a través de la malla. Los resultados además indican que la estructura del SNS puede realizar una exploración inteligente del espacio de búsqueda respecto de la búsqueda aleatoria.

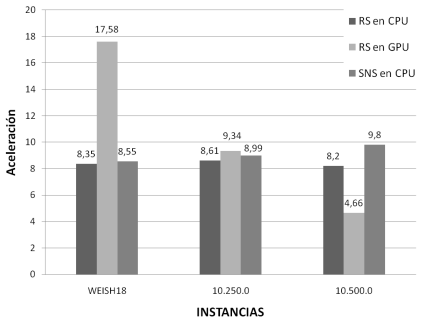
La Figura 2a muestra la evolución del promedio de los valores fitness por fila del SNS_{GPU}. La figura indica en color gris las peores soluciones y a medida que las soluciones mejoran el color se va aclarando hasta indicar las mejores soluciones en blanco. El SNS_{GPU} presenta una rápida evolución en la calidad de las soluciones en las primeras filas (1 a 10 aproximadamente). Los cambios de grupos pequeños de elementos en las soluciones durante la evolución del algoritmo produce un incremento promedio en la calidad de las soluciones. La modificación de pequeños grupos evita también el caer en óptimos locales o en caso de caer, el poder salir explorando zonas contiguas en el vector solución. Por otro lado, se observa que los cambios producidos en el resto de las filas no tiene un impacto directo sobre la calidad de la mejor solución e inclusive no se aprecia una mejora importante en la calidad media de estas filas.

La Tabla 1b presenta el promedio de los tiempos obtenidos (expresados en segundos) durante las 30 ejecuciones independientes. Con letra **negrita** y resaltado en gris aparecen los mejores valores valores obtenidos de tiempo para las instancias ejecutadas. Los resultados de esta tabla indican claramente que el

Tabla 2: Resultados de fitness y tiempo de los algoritmos para cada instancia

(a) *ResultadosdeFitness* (b) *ResultadosdeTiempo*





menor tiempo de ejecución es obtenido por el SNS en GPU. Asimismo, obtiene una diferencia significativa respecto a los otros algoritmos, en especial de las versiones del RS y SNS en CPU a medida que aumenta el número de elementos en las instancias.

La Figura 2b indica la aceleración del SNS_{GPU} con respecto al resto de algoritmos. Esta medida es una métrica muy usada para comparar la aceleración entre versiones en CPU y GPU. La métrica divide el tiempo transcurrido al ejecutar un algoritmo respecto a la versión SNS_{GPU}. De esta forma, un valor sobre 1.0 significa un mayor eficiencia de tiempo del SNS_{GPU} respecto del otro algoritmo. En general la aceleración del SNS_{GPU} con respecto a los otros algoritmos se encuentra entre 8 y 17.5 veces, demostrando la eficiencia del modelo y cómo éste maximiza la eficiencia de la arquitectura GPU. En particular, vemos que el RS_{GPU} reduce la diferencia respecto del SNS_{GPU} a medida que el número de elementos aumenta.

6. Conclusiones y Trabajos Futuros

En este trabajo hemos estudiado una aproximación algorítmica especialmente pensada para el hardware de una GPU, utilizando un nuevo diseño con base en el modelo sistólico. Las principales características del SNS son la simplicidad en las operaciones realizadas en cada celda y el continuo flujo de información a través de la malla toroidal.

Se han realizado experimentos sobre el SNS utilizando diversas instancias del MKP, obteniendo soluciones de alta calidad y tiempos de ejecución muy rápidos, sobre todo en la versión SNS_{GPU}.

La naturaleza inherentemente paralela del SNS aprovecha al máximo los recursos de la GPU en este tipo de problema, aunque es necesario estudiar sí este comportamiento se mantiene para otros problemas.

Como futuro trabajo se estudiarán nuevas versiones del SNS, variando tanto el tamaño de la malla como las operaciones que se realizan en cada celda.

Agradecimientos

Los autores agradecen el apoyo del Ministerio español de Ciencia e Innovación Europea FEDER bajo el contrato TIN2008-06491-C04-01 (M* project) así como del TIN2011-28194 (RoadMe project), y del CICE, Junta de Andalucía según el contrato P07-TIC-03044 (DIRICOM project). También a la Universidad de la Patagonia Austral, por el continuo apoyo recibido.

Referencias

1. *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, 2007.
2. Enrique Alba and Pablo Vidal. Systolic optimization on gpu platforms. In *EUROCAST (1)*, pages 375–383, 2011.
3. Wilbaut C. and Said Hanafi. New convergent heuristics for 0–1 mixed integer programming. *European Journal of Operational Research*, pages 62 – 74, 2009.
4. Heming Chan and Pinaki Mazumder. A systolic architecture for high speed hypergraph partitioning using a genetic algorithm. In Xin Yao, editor, *Progress in Evolutionary Computation*, volume 956 of *Lecture Notes in Computer Science*, pages 109–126. Springer Berlin / Heidelberg, 1995.
5. P.C. Chu and J.E. Beasley. A genetic algorithm for the multidimensional knapsack problem. *Journal of Heuristics*, 4:63–86, 1998.
6. B. Gavish et al. Allocation of data bases and processors in a distributed computting system. *Management of Distributed Data Processing*, page 215–231, 1982.
7. P. C. Gilmore and R. E. Gomory. The theory and computation of knapsack functions. *Operations Research*, 14(6):1045–1074, 1966.
8. Ananth Grama, George Karypis, Vipin Kumar, and Anshul Gupta. *Introduction to Parallel Computing (2nd Edition)*. Addison Wesley, 2003.
9. Gottlieb J. On the feasibility problem of penalty-based evolutionary algorithms for knapsack problems. In *Applications of Evolutionary Computing*, Lecture Notes in Computer Science, pages 50–59. 2001.
10. H. T. Kung. Let's design algorithms for vlsi systems. In *Proc. Conf. Very Large Scale Integration: Architecture, Design, Fabrication*, pages 65–90, Jan. 1979.
11. G.M. Megson and I.M. Bland. Synthesis of a systolic array genetic algorithm. In *Parallel Processing Symposium, 1998. IPPS/SPDP 1998. Proceedings of the First Merged International ... and Symposium on Parallel and Distributed Processing 1998*, pages 316 –320, mar-3 apr 1998.
12. John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, pages 80–113, 2007.
13. Wei Shih. A branch and bound method for the multiconstraint zero one knapsack problem. *J. Operational Research Society*, 30(4):369–378, 1979.
14. Michel Vasquez and Yannick Vimont. Improved results on the 0-1 multidimensional knapsack problem. *European Journal of Operational Research*, 165:70–81, 2005.
15. Xiaoxia Zhang, Zhe Liu, and Qiuying Bai. A new hybrid algorithm for the multidimensional knapsack problem. In *Bio-Inspired Computing and Applications*, Lecture Notes in Computer Science, pages 191–198. 2012.
16. Qian Zhou and Wenjian Luo. A novel multi-population genetic algorithm for multiple-choice multidimensional knapsack problems. In Zhihua Cai, Chengyu Hu, Zhuo Kang, and Yong Liu, editors, *Advances in Computation and Intelligence*, volume 6382 of *Lecture Notes in Computer Science*, pages 148–157. Springer Berlin / Heidelberg, 2010.