Optimización de cómputo paralelo científico en un entorno híbrido *Multicore - MultiGPU*

Gustavo Wolfmann
Lab. de Computación - FCEFyN - Univ. Nac. de Córdoba
Av. Velez Sarsfield 1611, Córdoba, Argentina, gwolfmann@gmail.com, (0351) 4334409

Abstract—*Es creciente el número de algoritmos de cómputo científico que corren con mayor eficiencia en las placas procesadoras de video (GPGPU - General Programming Graphics Processors Units) que en los propios procesadores multicore de nuestros días. Sin embargo, la capacidad de memoria de estos dispositivos es cerca de un orden de magnitud inferior a las computadoras equipadas con multiprocesadores simétricos (SMP - Symmetric Multiprocessors). Por ello, para problemas cuyos datos requieren más memoria que la disponible en una placa GPGPU, los datos deben procesarse parcialmente en la placa, y además, planificarse el avance global del procesamiento. Frente a la existencia de múltiples cores y uno o más GPGPU(s) en el SMP, se investiga el uso optimizado de ambos recursos de procesamiento en forma conjunta, enfocándonos en problemas de cómputo científico cuyo requerimiento de memoria esté en el rango de la memoria del dispositivo gráfico y la del computador. Este problema es típico del procesamiento paralelo, por cuanto una adecuada división de datos y tareas es fundamental para obtener un resultado acorde a los recursos utilizados. En este trabajo se presentan lineas de investigación al respecto, como así también, algunos resultados preliminares.*

***Palabras Clave**: Procesamiento Paralelo - Cómputo Científico - Procesamiento híbrido multicore/GPGPU.*

1. Contexto

La investigación está respaldada por los proyectos de investigación ”Diseño e implementación de algoritmos y *hardware* optimizados para sistemas de computación paralelos en ingeniería”, proyecto presentado ante la SECYT de la UNC para el período 2012-2013 y por el proyecto “Aceleración de Algoritmos en Procesadores Multicore, GPGPU y FPGA“, subvencionado por el MINCyT de la Prov. de Córdoba, convocatoria PID 2010, de 2 años de duración.

2. Introducción

La capacidad de procesamiento de las GPGPU’s en áreas de cómputo científico tiene una tendencia creciente, superior a las CPU’s *multicore*, en términos de FLOP’s teóricos alcanzables[Fog11], [Enc]. Existen, sin embargo, dos factores que limitan el procesamiento de las GPGPU’s. El primero referido a la dependencia de funcionamiento con una computadora, ya que necesariamente se conectan físicamente al *motherboard* de esta, y es un programa ejecutado en la CPU principal, quien llama a rutinas de cómputo que corren en la placa. El segundo factor limitante es la memoria de datos que disponen las GPGPU’s, cerca de un orden de magnitud inferior a la memoria de la computadora que las alojan.

En problemas cuyo volumen de datos a procesar es inferior a la memoria de datos de la GPGPU, esta es candidata a realizar todo el procesamiento, siempre que existan o se programen las rutinas pertinentes optimizadas. Una situación diferente se plantea si el volumen de datos es mayor, y/o si se utilizan conjuntamente más de una placa GPGPU conectada al *motherboard*, y/o se dispone de un procesador *multicore*, el cual se desea utilizar en forma simultanea con la(s) GPGPU(s). En estos casos, usar la totalidad de la capacidad de cómputo disponible, deriva en un problema de procesamiento paralelo, siendo necesario particionar datos y tareas de cómputo y programar un algoritmo que controle la asignación de datos y tareas a cada procesador y el avance global del procesamiento.

Delimitamos la investigación a aquellos programas a ejecutarse en computadoras que dispongan unidades de procesamiento *multicore* y de una o más placas GPGPU, cuyo volumen de datos sea mayor a la memoria de una GPGPU, por cuanto es necesario distribuir procesamiento. En cuanto a las rutinas a utilizar, se usarán bibliotecas de álgebra lineal ya optimizadas para ambos tipos de procesadores. No forma parte de nuestros objetivos la programación que maximice el uso de cada tipo de procesador, sino al contrario, utilizando las bibliotecas ya disponibles, optimizar el uso de estas con problemas cuyo tamaño justifique la utilización conjunta de ambos recursos de procesamiento.

El proyecto focaliza en cómo particionar datos y tareas y sincronizar eficientemente, para que la capacidad conjunta de procesamiento sea optimamente utilizada. Los problemas a estudiar son algoritmos clásicos de Álgebra Lineal: factorización y multiplicación de matrices. Dado que no existe una forma única de dividir datos a procesar en paralelo, y teniendo en cuenta que cada tipo de procesador dispone de una capacidad de procesamiento diferente, surgen varias alternativas en la forma de distribuir los datos en cada unidad de procesamiento.

Desde el punto de vista del modelo de programación paralela, estamos frente a un caso del modelo de memoria compartida. La programación paralela con hilos (*threads*)

es lo usualmente utilizado para este tipo de *hardware*, utilizando OpenMP para nuestros *tests*.

Una posible división de datos es, que el total de datos sea dividido en partes de igual tamaño, provocando que cada tipo de procesador tarde distinto tiempo en completar igual tarea, debiendo los más rápidos esperar a los más lentos. Este hecho genera un desbalance de carga indeseable.

Para solucionar el problema del desbalance, consideramos dos alternativas:

1. Mantener la división de datos en partes iguales, pero los procesadores más rápidos no sincronizan con los más lentos para la siguiente tarea, sino que comiencen con esta sin esperar, es decir, un avance desigual de procesamiento. Esto genera una mejor utilización de la capacidad de procesamiento, pero implica un mayor costo de coordinación global, ya que cada procesador avanza a su propio ritmo.
2. Hacer que cada procesador trabaje sobre un bloque de datos que demande igual tiempo de procesamiento para cada tipo de procesador, es decir, una partición de datos en bloques diferentes, los de mayor tamaño destinadas a los procesadores más veloces y los de menor tamaño a los procesadores más lentos.

La forma en que los datos sean divididos impacta en las tareas que se deban realizar para alcanzar el procesamiento total. Para la multiplicación de matrices, donde es casi nula la dependencia de datos, la división de datos solo está limitada por la regla de las dimensiones de los bloques para que la multiplicación pueda realizarse. Para los algoritmos de factorización, la fuerte dependencia de datos es un limitante serio para el avance desigual de tareas.

Los grafos de dependencia de datos son usuales para la planificación de tareas y permiten analizar y estudiar el funcionamiento del procesamiento paralelo. Esta herramienta de modelado de procesamiento paralelo es utilizada en el proyecto PLASMA de la Universidad de Tennessee [oTICL] para el desarrollo de una biblioteca de rutinas para procesadores *multicore*. En particular, en el reporte técnico LAWN 191 (*LAPACK Working* Note 191) [BLKD07] se presenta un *scheduler* de tareas llamado *Graph driven asynchronous execution*, donde a partir del grafo de dependencia de tareas, se construye el algoritmo que define el orden de ejecución de las mismas. El reporte trabaja solo con procesadores *multicore* y con bloques de datos de igual tamaño. El reporte LAWN 213 [KLDB09] extiende el 191, con distintas opciones de *scheduling*, usando también los grafos de dependencia.

Para la división desigual de bloques, orientada al avance sincróno de procesamiento, la existencia de distintas tareas impacta en la determinación del tamaño del bloque. Las GPGPU's no son CPU's más rápidas, sino que tienen una arquitectura diferente, por lo que si bien, en general son más veloces para cómputo numérico, no tienen la misma eficiencia relativa para distintas rutinas. La división de datos óptima sería específica por rutina, lo cual es impracticable. Así, definir un único tamaño de bloque para cada tipo de procesador provoca un rendimiento subóptimo del mismo.

El reporte LAWN 250 [STD11] propone la división desigual de bloques. Se determinan bloques de datos de dos tamaños, el menor para la CPU y el mayor, para la GPGPU. Utiliza un modelo estático de asignación de bloques entre *cores* de CPU y GPGPU's. El modelo de ejecución se asemeja al modelo *Master/worker*, y es complejo y con una alta carga de sincronización.

3. Investigación y Desarrollo

La estrategia definida para la programación paralela en un SMP con uno o más GPGPU's, es la siguiente: supongamos que se disponen de c *cores* en el equipo *multicore* y de g GPGPU's, donde vale $c \geq g$. Se dispone de c hilos, uno por cada *core*, de los cuales g se destinan al control de cada GPGPU y los $c-g$ restantes se destinan a procesamiento en CPU. En la práctica, el algoritmo paralelo maneja solo $g+1$ hilos, usando g para control de cada GPGPU y el restante para la invocación de la rutina de BLAS necesaria. Dado que se usan bibliotecas optimizadas para equipos *multicore*, el llamado a la rutina lleva un parámetro de valor $c-g$, destinado a utilizar ese número de hilos para cómputo.

Se investigará tanto en la avance desigual de tareas como en la división desigual de datos. Para la multiplicación de matrices, se buscará determinar la división desigual de datos que permita minimizar el costo de sincronización del procesamiento conjunto GPGPU y CPU.

Para el caso de factorización de matrices, algoritmos de Cholesky, LU y QR, se investigará tanto para la división desigual de datos, como el avance desigual de tareas. A priori, ambas estrategias presentan problemas. La división desigual de datos impone restricciones en cuanto al tamaño de los bloques a dividir los datos, ya que en estos algoritmos, existe un *update* de datos con resultados parciales que utiliza suma y multiplicaciones de matrices, por cuanto la dimensión de los bloques deben respetar la regla de la dimensiones de ambas operaciones.

La estrategia de avance desigual tiene el problema de la coordinación del avance de las tareas. Para ello se experimentará modelando los algoritmos con de Redes de Petri Coloreadas (*Coloured Petri Nets* - CPN)[JK09]. Una Red de Petri es generalmente definida como un grafo dirigido bipartito, cuyos nodos representan lugares o transiciones (nodos *places* y nodos *transitions*) y sus arcos conectan un nodo de un tipo con un nodo de otro tipo, por lo que un arco que tiene como origen a un nodo *place* tendrá obligatoriamente como destino un nodo *transition* o viceversa [Dia09]. Existen *tokens* que representan hechos o eventos y están contenidos solo en nodos de tipo *place*. En general, los *places* representan estados y las *transition* representan acciones [Jen93].

Los *tokens* pasan de un *place* a otro cuando una *transition* se dispara. Cuando esto sucede, se genera un movimiento de *tokens*, ya que se eliminan *tokens* de los *places* que tiene como origen y se inyectan *tokens* sobre los *places* que tiene como destino. Esta modelo se ajusta a la ejecución de un algoritmo paralelo: los estados, o *places*, representan el avance del procesamiento y las *transitions* el cómputo. La concurrencia es inherente al modelo, ya que nada impide que varias *transitions* puedan dispararse en simultáneamente, si hay suficientes *tokens* en los respectivos *places* de entrada.

Las Redes de Petri Coloreadas son una variedad de las Redes de Petri donde los *tokens* toman identidad mediante su asociación a un *"color"*, permitiendo representar distintos hechos [JKW07]. Esto es apropiado para nuestro estudio, ya que se identificará cada bloque de datos con un *"color"*.

Se experimentará el uso de las CPN no solo para modelar el procesamiento paralelo, sino también como *scheduler* de tareas con avance desigual. Los *places* serán usados para representar el avance del procesamiento, y las *transitions*, la rutina a ejecutar. Los bloques de datos estarán representados por *tokens* de distinto "color". De esta forma, cuando los *places* tengan disponibles los *tokens* necesarios para la ejecución de una rutina, esta será lanzada en algún procesador inactivo bajo una determinada política. Este *scheduling* tiene la ventaja de prescindir de la necesidad de explicitar los puntos de sincronización en el algoritmo paralelo y por lo tanto, cada tarea de procesamiento es lanzada con el solo requisito de que los datos y un procesador este disponible.

Nuestra hipótesis es que utilizando las CPN como modelo y *scheduler* del algoritmo paralelo, los procesadores quedarán inactivos solo cuando no se disponga de datos para procesar, y no por esperar sincronización con otros procesadores, lo cual generará una mejora en la *performance*. En términos de Redes de Petri, estarán inactivos cuando los *tokens* necesarios aun no estén disponibles en los *places* que habilitan una *transition*. La correctitud del resultado de la ejecución del algoritmo depende del correcto modelado en la Red de Petri del algoritmo.

4. Resultados preliminares

Se realizaron experimentos con el algoritmo de multiplicación de matrices y el de factorización de Cholesky. Ambos utilizando un computador equipado con un procesador AMD-Phenom II de 6 *cores*, 3.2 Ghz de velocidad, 8 GB de RAM y dos placas GPU NVIDIA GTX 470, con 448 *stream processors* y 1.2 GB de memoria en cada una de ellas.

Los experimentos fueron realizados utilizando el compilador icc 11.1 de Intel y las implementaciones de BLAS y LAPACK, MKL de Intel para CPU y CUBLAS 4.0 de NVIDIA para las GPGPU's.

Los *tests* se realizaron utilizando 1 o 2 GPGPU's. En cada caso, del total de 6 *cores* disponibles, se utilizaron 1 o 2 *cores* para el control de cada GPGPU y los restantes 5 o 4, para cómputo, respectivamente.

4.1. Multiplicación de Matrices

El algoritmo implementado resuelve el producto de $C = A \times B$, donde A, B y C son matrices cuadradas, de rango n, el cual se elige suficientemente grande para que el requerimiento de memoria de las matrices exceda los 1.2 GB de memoria que dispone cada placa GPGPU, pero sea inferior a los 8 GB de la memoria de la CPU.

La división de datos se realizó bajo el criterio de la división desigual. Para evitar el problema de dimensiones de bloques incompatibles para realizar la multiplicación, se optó por dividir en bloques de igual tamaño, asignándole varios de estos bloques a las GPGPU's, de forma tal que el bloque de mayor tamaño propio de las GPGPU's es un factor del bloque asignado a cada *core* de la CPU.

Con el objetivo de reducir el tráfico de datos entre la memoria principal y la memoria de cada GPGPU, el cómputo realizado en cada una de estas, es el producto de toda la matriz A por bandas columna de B. Si suponemos que el número de bandas es nb, las bandas columna de B son de dimensión n x (n/nb). Esto define que el producto de A por una banda columna B_i, de por resultado un bloque de la matriz resultante C_i, de rango n x (n/nb).

Para asignar tareas se definió que cada *core* destinado a cómputo multiplique A por un bloque columna de B. Para las GPGPU, se definió un factor $f > 1$, tal que la tarea de cada GPGPU es la multiplicación de A por f bloques columna de B, obteniendo por resultado f bloques de C.

Este esquema de división de datos presenta un obstáculo para rangos de matrices superiores a $n = 16000$, *single presicion*, debido a que toda la matriz A más las bandas de B y de C no quepan en la memoria de cada GPGPU. Con el fin de eludir este limitante, se dividió A en un número na de bandas filas, de forma tal que cada GPGPU tenga una parte

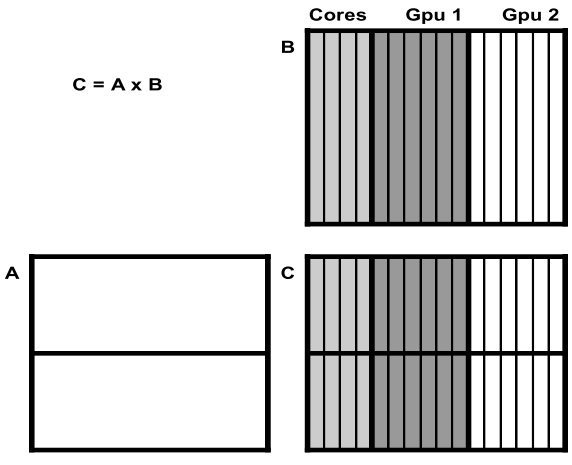

Fig. 1

DIVISIÓN DE DATOS, CON VALORES $nb = 16$, $na = 2$ Y $f = 6$. PUEDE VERSE QUE PARTE DE B ES DESTINADA A CADA GPU Y A LOS *cores*.

de A en lugar de su totalidad, lo que implica una importante reducción en la memoria ocupada.

Un ejemplo de esta división de datos y tareas, es definir $n = 16000$, $nb = 16$, $f = 6$ y $na = 2$, usando 2 GPGPU's. Así, los datos de B están divididos en 16 bloques columna. La división desigual se implementa asignando 6 bloques a cada GPGPU y los restantes 4 bloques, a los 4 *cores* destinados a cómputo de la CPU, como se grafica en Fig.(1).

size (n)	cpu	gpu	f	nb	na	mem cpu	tmpo cpu (seg)	tmpo gpu (seg)	flops (G)
16800	6	0	1	1	1	3144	72.3	0.0	131.1
16800	1	1	1	40	1	3144	0.0	20.0	473.0
16800	6	1	15	20	20	3144	22.1	14.5	429.1
16800	6	1	25	30	4	3144	14.7	14.3	645.2
16800	6	2	8	20	4	3144	21.9	7.6	433.0
16800	6	2	46	96	2	3144	9.0	8.6	1053.0
21400	6	0	1	1	1	5004	149.0	0.0	131.5
21400	6	1	5	10	4	5004	90.8	17.5	215.9
21400	6	1	25	30	4	5004	31.5	30.4	622.2
21400	6	2	2	8	4	5004	117.0	10.7	167.5
21400	6	2	23	50	4	5004	19.3	19.3	1015.5
24000	6	0	1	1	1	6468	209.9	0.0	131.7
24000	6	1	15	20	4	6468	65.6	37.0	421.5
24000	6	1	25	30	8	6468	42.4	42.1	652.1
24000	6	2	8	20	8	6468	92.9	66.0	277.0
24000	6	2	23	50	4	6468	25.9	25.8	1067.5

Table 1

TIEMPOS PARA DIFERENTES COMBINACIONES DE CPU'S, GPU'S, RANGO DE MATRICES Y PARTICIONES DE DATOS, *single presicion*. LA ÚLTIMA COLUMNA TIENE EL CÁLCULO DE LOS FLOPS OBTENIDOS.

Resultados preliminares de estos experimentos se muestran en la Tabla 1. Allí son presentados valores para distintas combinaciones de rango de matriz, de GPGPU's usados y de esquemas de división de datos. También se presentan los *flops* obtenidos como el resultado del total de operaciones realizadas bajo la fórmula $flops = (2 * n^3 - n^2)/tiempo$ y expresado en términos de Giga-flops. Solo algunos resultados seleccionados son incluidos, entre ellos el mejor valor de *flops* obtenido para los recursos utilizados, y otros donde el resultado no alcanza la misma eficiencia, a pesar de utilizar iguales recursos, pero con distinta división de datos.

Una conclusión preliminar es que el máximo rendimiento se obtiene cuando se igualan los tiempos insumidos por cada tipo de procesador. Sin embargo, determinar cual es la división de datos que permite obtener dichos tiempos, es netamente experimental y dependiente del número de *cores* de CPU y de GPGPU's destinados a cómputo, de la potencia de cómputo relativa entre ambos tipos de procesadores, y de lo optimizada que esté programada la rutina a ejecutar. El siguiente paso consiste en modelizar el *scheduling* con Redes de Petri para mejorar el balance de carga.

4.2. Factorización de Cholesky

La factorización o descomposición de Cholesky es un método que resuelve un problema clásico del álgebra lineal, donde una matriz cuadrada simétrica y definida positiva A, de rango n, es factorizada como $A = L * L^T$, siendo L una matriz triangular.

Los valores de L se obtienen siguiendo las fórmulas:

$$l_{ij} = \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} * l_{jk} \right) / l_{jj} \quad 1 \leq j < i \leq n \quad (1)$$

$$l_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1} l_{ik}^2} \quad 1 \leq i \leq n \quad (2)$$

Las ecuaciones (1) y (2) imponen una fuerte dependencia de datos, ya que para calcular los elementos de una fila i, primero deben computarse los valores l_{ij} de dicha fila, con $j < i$ y luego el elemento de la diagonal principal l_{ii}. A su vez, para computar cada l_{ij} deben primero estar calculados todos los valores de la fila i y de la fila j, hasta la columna $j - 1$ y luego realizar la división por l_{jj}.

El algoritmo de *test* es desarrollado dentro del contexto de realizar los cómputos sobre bloques de matrices y utilizar rutinas de álgebra lineal previamente desarrolladas. Supongamos que la matriz a factorizar es dividida en $q \times q$ bloques cuadrados. El cómputo de los bloques de la diagonal principal se realiza usando las rutinas `xpotrf` y `xsyrk`, y los restantes bloques con `xgemm` y `xtsrm`, donde la $x = \{d|s\}$, según se use *double* o *single presicion*. El algoritmo es expuesto en la siguiente tabla y el avance es graficado en Fig.(2), para el supuesto que $q = 5$.

1	$i = 1$
2	mientras ($i \leq q$)
3	calcular el bloque diagonal l_{ii} invocando `xsyrk` y luego `xpotrf`
4	calcular los bloques l_{ji}, $j > i$ invocando a `xgemm` y luego `xtsrm`
5	$i = i + 1$
6	volver a 2

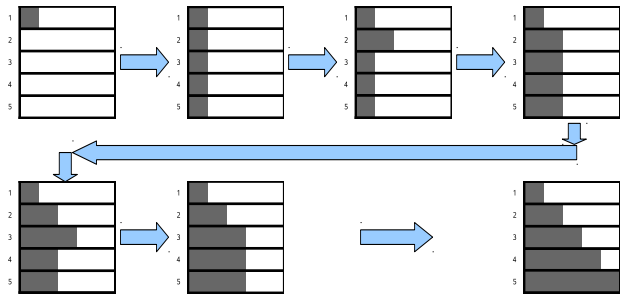

Fig. 2

AVANCE DE TAREAS PARA EL ALGORITMO DE CHOLESKY PARA $q = 5$

La división de datos y distribución de las tareas es más compleja que para multiplicación de matrices. Ha de tenerse en cuenta, que de las 4 rutinas necesarias, 3 están definidas

en BLAS, y una, `xpotrf`, está definida en LAPACK, y que la biblioteca CUBLAS solo implementa BLAS, y no LAPACK, por lo que `xpotrf` debe ejecutarse en CPU [1].

El reporte LAWN 223 [LTN+09] trata específicamente sobre la ejecución del algoritmo en cuestión en un entorno híbrido *multicore - multigpu*. Plantea una división de datos homogénea (bloques cuadrados de igual tamaño), asignando tareas a las GPGPU's, salvo la factorización de los bloques de la diagonal principal, es decir, la llamada a `xpotrf`, la cual es siempre ejecutada en CPU. La dependencia de datos es respetada siguiendo un grafo de dependencia, el cual es definido específicamente para el algoritmo y el número de bloques en que sea dividida la matriz.

Uno de los desafíos que el reporte señala es determinar el número de bloques a dividir la matriz para lograr un balance de carga. No hay referencia a como es armado el grafo de dependencias, lo cual de por si solo es complejo, ya que depende del algoritmo y del número de bloques. Además, no plantea la posibilidad de que la CPU realice alguna de las tareas que por defecto están asignadas a las GPGPU's para lograr un mejor balance de carga.

La utilización de una Red de Petri para guiar la ejecución del programa está orientada a eludir dichos inconvenientes, adecuándose al esquema de avance desigual de tareas. La definición de la Red puede hacerse con un meta-lenguaje, XML por ejemplo, lo cual facilita el uso del *scheduler* para distintos algoritmos. Si bien XML también puede utilizarse para definir un grafo de dependencias, el cambio del número de bloques en que se divida la matriz afecta el número de componentes del grafo, no así para la Red de Petri, facilitando la determinación del tamaño de bloque óptimo.

Otra ventaja del uso de una Red de Petri es la flexibilidad que la misma brinda en lo referente a la asignación de tareas. La ejecución siguiendo un grafo de dependencia es relativamente estática y predeterminada. Utilizar una Red de Petri facilita la asignación dinámica de tareas.

Un primer test se realizó usando dos GPGPU's y dividiendo una matriz de rango 24000 en 6 bloques cuadrados, con datos *single precision*. La única restricción en cuanto a la ejecución es que la rutina `xpotrf` es ejecutada solamente en CPU por los motivos antes descriptos. La métrica de bondad utilizada tiene dos elementos. En primer lugar, los flops observados, usando $flops = (n^3/3 + n^2/2 + n/6)/tiempo$ [Wat04] y en segundo término, el porcentaje de tiempo que cada tipo de procesador (GPGPU's y *cores*) queda inactivo a la espera de datos, a los fines de determinar la tasa de uso de los procesadores. El resultado puede verse en Tabla 2.

Como conclusión de este primer test, se obtuvo un buen valor de flops, y una reducida inactividad de las GPU's. Se debe mejorar la participación de la CPU en el cómputo total ya que esta tiene una alta tasa de inactividad. De lograrse, impactará positivamente en la reducción del tiempo global y en el valor observado de flops.

size (n)	blqs	tmpo cpu (s)	tmpo gpu1 (s)	tmpo gpu2 (s)	tmpo máx (s)	cpu id-le (%)	gpu1 id-le (%)	gpu2 id-le (%)	flops (G)
24000	6	10.67	9.80	9.97	10.67	62.9	11.2	13.9	431.9

Table 2
TIEMPO PARA LA FACTORIZACIÓN DE CHOLESKY CON *scheduler* DE PETRI, 2 GPU'S.

5. Formación de Recursos Humanos

El proyecto de investigación se enmarca dentro del plan de Doctorado en Ciencias Informáticas en la Universidad Nacional de La Plata del autor. Cuatro alumnos de la carrera de Ingeniería en Computación de la UNC realizan sus tesinas de graduación en temas de este proyecto. Además, el grupo de investigadores está constituido por seis docentes del Departamento de Computación, FCEFyN, UNC.

References

[BLKD07] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. LAPACK Working Note 191, September 2007.

[Dia09] Michel Diaz. Chapter 1 - basic semantics. In Michel Diaz, editor, *Petri Nets: Fundamental Models, Verification and Applications*. Wiley-ISTE, 2009.

[Enc] Wikipedia Free Encyclopedia. Nvidia geforce 400 series. http://en.wikipedia.org/wiki/GeForce_400_Series.

[Fog11] Agner Fog. *The microarchitecture of Intel, AMD and VIA CPUs*. Copenhagen University College of Engineering, 2011.

[Jen93] Kurt Jensen. An introduction to the theoretical aspects of coloured petri nets. In *REX School/Symposium*, pages 230–272, 1993.

[JK09] Kurt Jensen and Lars Michael Kristensen. *Coloured Petri Nets - Modelling and Validation of Concurrent Systems*. Springer, 2009.

[JKW07] Kurt Jensen, Lars Michael Kristensen, and Lisa Wells. Coloured petri nets and cpn tools for modelling and validation of concurrent systems. *STTT*, 9(3-4):213–254, 2007.

[KLDB09] Jakub Kurzak, Hatem Ltaief, Jack Dongarra, and Rosa M. Badia. Scheduling linear algebra operations on multicore processors. LAPACK Working Note 213, February 2009.

[LTN+09] Hatem Ltaief, Stanimire Tomov, Rajib Nath, Peng Du, , and Jack Dongarra. A scalable high performant cholesky factorization for multicore with gpu accelerators. Technical Report 223, LAPACK Working Note, November 2009.

[oTICL] University of Tennessee Innovative Computing Laboratory. Plasma - parallel linear algebra for scalable multi-core architectures. http://icl.cs.utk.edu/plasma/index.html.

[STD11] Fengguang Song, Stanimire Tomov, and Jack Dongarra. Efficient support for matrix computations on heterogeneous multicore and multi-GPU architectures. LAPACK Working Note 250, June 2011. UT-CS-11-668.

[Wat04] D.S. Watkins. *Fundamentals of Matrix Computations*. Pure and Applied Mathematics: A Wiley Series of Texts, Monographs and Tracts. John Wiley & Sons, 2004.

[1] Existe una versión de LAPACK para placas NVIDIA desarrollada por tercera parte, incompleta para uso libre.