Uso del conocimiento de la arquitectura Fermi para mejorar el rendimiento en aplicaciones CUDA

Yuri Torres, Arturo González-Escribano y Diego R. Llanos [1]

Resumen— **Las unidades de procesamiento gráfico (GPUs) actualmente están jugando un papel muy importante como aceleradores para cómputo de propósito general. La implementación de códigos paralelos de alto rendimiento en GPUs es una tarea recomendada para programadores experimentados, debido al alto grado de dificultad de explotar eficientemente el uso de los sus recursos. La elección del tamaño y la forma de los bloques de hilos son decisiones importantes ya que tienen un impacto muy significativo sobre el rendimiento de las aplicaciones. La arquitectura Fermi de NVIDIA introduce nuevos criterios a la hora de seleccionar los tamaños y la geometría de los bloques de hilos. En este artículo mostramos un estudio de dichos criterios, así como una guía general para seleccionar un bloque de hilos apropiado para diferentes tipos de aplicaciones.**

Palabras clave— **CUDA, Fermi, auto-tuning, GPGPU.**

I. Introducción

LAS unidades de procesamiento gráfico (GPUs) han conseguido ser una importante plataforma computacional en muchos campos científicos tales como computación masiva de datos o computación biomédica. Entre las principales ventajas de estos dispositivos destaca el bajo coste operacional, la facilidad y sencillez ofrecida por los entornos de programación y el alto potencial de rendimiento.

La programación basada en el modelo GPGPU (computación GPU de propósito general) ha sido simplificada al introducir modelos de programación como CUDA [1]. Sin embargo, explotar eficientemente estos dispositivos implica utilizar un elevado conocimiento de la arquitectura para aplicar adecuadamente técnicas de optimización de código.

Fermi [2] [3] [4] es la última arquitectura desarrollada por la compañía NVIDIA para dispositivos GPUs. Comparada con versiones anteriores, Fermi introduce varias mejoras que influyen significativamente en las estrategias básicas de optimización existentes en las optimizaciones de códigos para GPU. Por ejemplo, explotar la coalescencia, maximizar la ocupación de los multiprocesadores (SMs), la precarga de datos y el unrolling. También introduce la posibilidad de configurar su nueva memoria caché L1.

La elección del tamaño y la geometría del bloque de hilos es una de las decisiones más importantes a la hora de implementar cualquier código de alto rendimiento en dispositivos GPU. Actualmente, el tamaño y la geometría de los bloques son típicamente seleccionados mediante prueba y error sin conocimiento previo de los posibles efectos que puedan causar sobre la arquitectura.

En este artículo se presenta un estudio práctico de la arquitectura Fermi orientado a la elección del tamaño y la geometría de los bloques de hilos. Se describe un nuevo enfoque para determinar el mejor bloque de hilos y la mejor configuración de la memoria caché L1 para diferentes tipos de aplicaciones.

Los resultados de estudio experimental muestran la utilidad del enfoque y como el conocimiento de la arquitectura hardware de estos dispositivos se puede explotar para utilizar más eficiente sus recursos.

El resto del artículo está organizado de la siguiente forma: en la sección 2 se comentan los detalles introducidos por la arquitectura Fermi comparándolos con las versiones anteriores. En la sección 3 se discute cómo la arquitectura afecta significativamente a la hora de seleccionar el tamaño y la geometría del bloque de hilos. En la sección 4 se describe el diseño de experimentos con el objetivo de verificar nuestras hipótesis. En la sección 5 se detalla el banco de pruebas utilizado. En la sección 6 se muestran los resultados obtenidos mientras que en la sección 7 se refleja el trabajo relacionado. Finalmente, la sección 8 contiene las conclusiones de este artículo.

II. Arquitectura NVIDIA Fermi

Fermi es la última generación de arquitecturas CUDA [2] [3]. Esta nueva arquitectura incluye operaciones de doble precisión, soporte para la corrección de errores, un incremento en la rapidez de los cambios de contexto y de las operaciones atómicas, y una jerarquía transparente de memorias caché L1/L2 con la posibilidad de desactivar la L1 o ampliar su tamaño a costa de reducir la memoria compartida explícitamente manejable por el usuario. Esta sección discute los detalles de la arquitectura Fermi más significativos para nuestro estudio.

A. Memorias caché L1/L2

En arquitecturas pre-Fermi cada SM tenía 16 KB de memoria compartida manejable explícitamente por el usuario, sin presencia de memorias caché. Fermi introduce dos memorias caché transparentes (L1 y L2). La memoria L1 es configurable por el usuario. Por defecto tiene 16 KB de caché L1 y 48 KB de memoria compartida, pero se puede configurar antes de lanzar cada kernel para tener 48 KB de caché L1 y 16 KB de memoria compartida. La memoria caché L2 es de tamaño fijo con 768 KB de capacidad. Hasta la llegada de Fermi el tamaño de los segmentos

[1]Dpto. de Informática, Univ. de Valladolid, e-mail: {yuri.torres|arturo|diego}@infor.uva.es

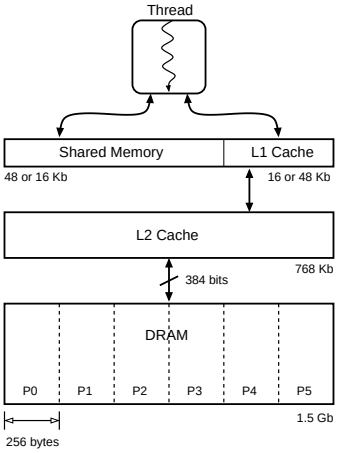

Fig. 1. Jerarquía de memoria Fermi (NVidia GTX-480).

de transacción de memoria eran de 32, 64 y 128 Bytes. Sin embargo, en Fermi son de 128 ó 32 Bytes, dependiendo de si la caché L1 está activa.

B. Bloque de hilos, Warps y SMs

Fermi dobla el número de registros por SM (32 KB). El número de procesadores (SPs) por SM ha sido multiplicado por cuatro (32 SPs). En Fermi el número máximo de hilos por bloque es 1024, mientras que el máximo número de hilos concurrentes en un SM es 1536.

C. Conflictos en memoria global

Actualmente en Fermi la memoria global de estos dispositivos está distribuida en 5 ó 6 bancos con sus correspondientes controladores independientes. Un problema frecuente a la hora de implementar códigos paralelos sobre arquitecturas pre-Fermi era el partition camping [5]. Este problema surge cuando hilos concurrentes activos en cualquier SM solicitan segmentos de memoria diferentes pertenecientes al mismo banco de memoria global. Ello implica una serialización de dichas transacciones. Con la introducción de las cachés L1 y L2 en Fermi, este problema se alivió cuando los mismos segmentos de transacción se solicitan una y otra vez.

III. selección del tamaño y la geometría del bloque de hilos

En esta sección discutimos cómo los detalles de la arquitectura Fermi afectan a las decisiones de los programadores respecto al tamaño y geometría de los bloques de hilos, así como a la configuración de la memoria caché L1.

A. Tamaño de bloque de hilos

A.1 Maximizar ocupación

En CUDA el factor ocupación de un SM es el ratio entre el número de warps presentes en el mismo y el número máximo de warps soportados por un SM (48 warps en Fermi). Maximizar la ocupación es importante para ocultar las latencias en los accesos a memoria global.

Para obtener máxima ocupación es necesario seleccionar un tamaño de bloque de hilos apropiado. La arquitectura Fermi soporta 1024 hilos por bloque, 1536 hilos por SM y un máximo de 8 bloques residiendo simultáneamente en un SM. Por lo tanto, el tamaño de bloques de hilos debe ser inferior o igual a 1024, a la vez divisor entero de 1536 y el máximo número de hilos del SM dividido entre el tamaño del bloque no puede exceder de 8. Por ello concluimos que los únicos tamaños de bloques de hilos que maximizan la ocupación son 192, 256, 384, 512 y 768.

Finalmente, cuando se va a lanzar un kernel que necesita menos de 192 hilos por el número de SMs en el dispositivo, puede ser recomendable escoger tamaños de bloques más pequeños, que no consigan maximizar la ocupación. De esta forma, se aumenta el número de bloques y se distribuyen mejor el conjunto de hilos entre los SMs, aumentando el grado de paralelismo global.

A.2 La coalescencia y la carga de accesos a memoria

La coalescencia es una técnica de implementación para conseguir que hilos consecutivos de un warp, cuando solicitan memoria global, pidan direcciones lógicas contiguas. Esta técnica permite minimizar el número de segmentos de transacción solicitados a la memoria global. La coalescencia es especialmente importante en aquellos códigos donde existe un alto ratio de accesos a memoria.

Una técnica común para facilitar la programación de la coalescencia es utilizar estructuras de datos alineadas con el número de hilos por warp y el tamaño de los segmentos de transacción. Para reducir las probabilidades de la aparición de partition camping, es también aconsejable usar estructuras de datos que estén perfectamente alineadas con el número de controladores de memoria global del dispositivo GPU. Por ejemplo, facilita la programación el escoger arrays de enteros o floats cuya última dimensión es múltiplo de 32 y a la vez del número de bancos.

Para kernels de CUDA donde prevalezca el patrón de acceso coalescente y un alto ratio de accesos a memoria (uno o más accesos a memoria por operación aritmética), consideramos que utilizando bloques de 192 hilos se obtendrán los mejores resultados por dos motivos: (1) se maximiza la ocupación, (2) se consigue el mayor número de bloques concurrentes por SM (8). De esta forma, cuando los warps de un bloque van terminando este se desaloja lo antes posible por tener un menor número de warps. Por tanto, la cantidad de warps que están activos en el SM se mantiene en el máximo (48) la mayor cantidad de tiempo posible dando más oportunidades a la ocultación de las latencias.

Kernels de programas como la multiplicación de matrices, a pesar de su coalescencia, presentan características diferentes. Tienen mucha carga computacional por hilo y reutilizan datos en segmentos de memoria requeridos previamente. En este caso se espera que aumentar el tamaño de bloque hasta 768 hilos (mayor tamaño de bloque que maximiza la ocupa-

ción) mejore el rendimiento ya que se explota mejor la reutilización reduciendo el número de segmentos de transacción solicitados.

A.3 Accesos no coalescentes

En códigos cuyo patrón de acceso a memoria global no es coalescente el número de segmentos de transacción que solicita un warp se ve incrementado significativamente, incrementando a su vez la probabilidad de conflictos en los bancos de memoria global (partition camping).

Para reducir el cuello de botella provocado por la cantidad de segmentos de transacción solicitados a memoria global, proponemos dos estrategias básicas: (1) desconectar el uso de la memoria caché L1, ya que de esta forma, el tamaño de los segmentos se reduce a una cuarta parte y (2) disminuir el tamaño de bloque reduciendo la ocupación para reducir el número de segmentos solicitados por SM en un periodo de tiempo dado.

B. La forma de los bloques

Para un tamaño de bloque de hilos dado existen diferentes posibles geometrías, con el mismo ratio de ocupación. Para la misma ocupación cada posible geometría tiene un impacto diferente sobre la coalescencia, los conflictos en los bancos de memoria y los cuellos de botella en los accesos a la memoria global.

En programas cuya codificación explota eficientemente la coalescencia, los mejores resultados se obtendrán con bloques de hilos cuyo número de columnas sea múltiplo de 32 (tamaño del warp). Las peticiones de memoria global de cada warp coinciden con segmentos de memoria completos, minimizando la cantidad de segmentos a transferir.

IV. Diseño de experimentos

En esta sección introducimos el diseño de los experimentos realizados para verificar las hipótesis previamente presentadas. Para ello, programamos una serie de benchmarks en CUDA y los ejecutamos con diferentes tamos de bloque.

Los benchmarks discutidos en esta sección incluyen: una reducción de vectores, suma de matrices, una modificación de suma de matrices para simular un aumento de carga, multiplicación de matrices y copias de datos con accesos dispersos y aleatorios sobre arrays bidimensionales.

Para todos nuestros benchmarks consideramos bloques de diferentes tamaños, hasta 1024 hilos (el máximo en permitido en Fermi). Escogemos un número de elementos por dimensión para la geometría del bloque que es potencia de dos, o potencia de 2 multiplicada por 3. Así conseguimos todas las combinaciones de geometrías que maximizan la ocupación y otros tamaños de bloque más pequeños.

Para el problema unidimensional de reducción de vectores se utiliza 4096×1023 elementos. Considerando que cada hilo trabaja sobre dos elementos, esto permite bloques de 32 hilos o más sin superar el número máximo de bloques por dimensión soportados por Fermi (65535).

Para el resto de benchmarks usamos matrices cuadradas de 6144 elementos por dimensión. Este tamaño es suficientemente pequeño como para alojar 3 matrices en la memoria global del dispositivo. Además, las cardinalidades de cada dimensión son múltiplo del número de bancos de memoria global de nuestro dispositivo, así como de las dimensiones de las geometrías probadas.

Los experimentos han sido ejecutados sobre la tarjeta Nvidia GeForce GTX 480. El sistema es un Intel(R) Core(TM) i7 CPU 960 3.20GHz de 64 bits con 6 GB de memoria principal. El sistema operativo usado es UBUNTU desktop 10.10 (64 bits). El driver de CUDA utilizado es el correspondiente a la versión 3.0 del toolkit.

V. Benchmarks

En esta sección describimos los benchmarks clasificados por tipo de aplicación.

A. Accesos coalescentes

El primer benchmark es una *reducción de vector* basado en uno de los ejemplos del SDK de CUDA. Ha sido necesario modificarlo para poder variar el tamaño del bloque. El kernel de este código aplica una reducción de dos elementos por cada hilo, escogidos de forma coalescente en cada warp. El kernel se lanza varias veces con la mitad de bloques cada vez hasta terminar la reducción.

El segundo es un código trivial de *suma de matrices* donde cada hilo está asociado al cálculo de una posición de la matriz resultado. Esto implica 3 accesos a memoria por cada hilo (dos de lectura y uno de escritura). Implica un patrón de acceso a memoria completamente coalescente sin reutilización de ningún elemento. En este caso, cada warp necesita únicamente un segmento de transacción de 128 bytes para las 32 peticiones realizadas por los hilos de un warp en cada acceso.

Se ha escogido también un código trivial de *multiplicación de matrices* [1] donde cada hilo calcula el producto escalar de una fila de la primera matriz y una columna de la segunda. A pesar de ser un algoritmo sencillo presenta un patrón de acceso diferente en cada una de las tres matrices utilizadas. La combinación de los diferentes patrones produce un efecto complejo de analizar.

B. Accesos no coalescentes

Se ha construido un benchmark sintético, denominado *accesos dispersos regulares* diseñado con el objetivo de crear un patrón de acceso disperso sobre la memoria global del dispositivo, de tal forma que cada hilo pida un segmento de transacción diferente. Cada hilo pide un elemento entero que se encuentra a 32 posiciones de memoria global de distancia del anterior. Una vez accedido al elemento deseado se incrementa su valor y se almacena en la posición

siguiente, que está en el mismo segmento de transacción.

También se utiliza una variante del benchmark anterior denominado *accesos aleatorios*. Cada hilo calcula un par de valores aleatorios, los cuales sirven para determinar la posición a la que se accede en una matriz. Al igual que el benchmark anterior el espacio de hilos tiene tantos elementos como la matriz. Cada hilo realiza alrededor de 20 operaciones aritméticas para el cálculo de los índices aleatorios.

VI. Resultados experimentales

En esta sección presentamos los resultados obtenidos por nuestros benchmarks, estudiando su relación con los detalles de la arquitectura Fermi comentados en secciones previas.

A. Acceso coalescente

A.1 Kernels pequeños sin reutilización de datos

En la tabla I se muestran los tiempos de ejecución de la reducción de vectores. Por el tamaño de datos de entrada escogido, el número de hilos por bloque tiene que ser como mínimo de 32. La implementación de la reducción presenta un patrón coalescente, donde no existe reutilización de datos entre hilos y cada hilo apenas tiene carga computacional, únicamente para calcular las posiciones de trabajo en el vector. La tabla I confirma el resultado general predicho en la sección III: el mejor tamaño de bloque es el menor que maximiza la ocupación, es decir 192.

La tabla II muestra los tiempos de ejecución para la suma de matrices con las diferentes geometrías consideradas. Los resultados correspondientes a bloques de hilos que maximizan la ocupación de los SMs están resaltados en negrita. Los resultados son similares al caso anterior, obteniendo el mejor tamaño de bloque en 192 hilos. El tiempo de ejecución crece rápidamente al reducir el número de columnas por debajo de 32 y no aprovecharse el efecto de coalescencia. No se reflejan en la tabla bloques con menos de 8 columnas por problemas de espacio. Siguen la tendencia esperada.

Los experimentos realizados desactivando la caché L1 no muestran un impacto significativo sobre los resultados. Se reduce la cantidad de datos por segmento a cambio de aumentar proporcionalmente el número de éstos.

A.2 Reutilización intensiva de datos

En la tabla III se muestran los resultados para la multiplicación de matrices. Con menos de 32 columnas de hilos, el tiempo de ejecución aumenta rápidamente ya que el patrón de acceso ya no es coalescente en la primera matriz (se omiten dichos resultados en la tabla). Debido a la reutilización de los datos de las dos matrices de entrada, bloques más grandes tienen más posibilidades de reutilizar las mismas líneas de caché. Los mejores resultados se obtienen con 768 hilos, los bloques más grandes que maximizan la ocupación.

Respecto a la geometría, para bloques de hilos con el mismo tamaño los mejores resultados se obtienen reduciendo el número de filas aumentando correspondientemente el número de columnas. Esta tendencia se mantiene hasta geometrías de una fila donde los tiempos vuelven a incrementarse debido a que en este algoritmo domina la pérdida completa la reutilización de los datos de la segunda matriz de entrada.

B. Accesos no coalescentes

La tabla IV muestra los tiempos de ejecución para este benchmark. Es un kernel sin un patrón de memoria coalescente y con muy poca carga de computacional por hilo. Como se discutió en la sección III-A.3 este tipo de códigos funcionan mejor por debajo de máxima ocupación. Esto es debido a que: (1) las latencias producidas por la gran cantidad de segmentos solicitados no se pueden ocultar con el máximo número de warps por SM, es decir, 48 y (2) reduciendo el número de warps por SM se decrementa el número total de segmentos solicitados simultáneamente, aliviando el cuello de botella que serializa los accesos a memoria global.

Por último en la tabla V se observan los resultados obtenidos para el código con accesos aleatorios. Este algoritmo utiliza demasiados recursos por SM, por lo que bloques de 1024 hilos llevan a una ocupación nula, marcados con asteriscos en la tabla. Este programa tiene una carga media en cuanto a operaciones aritméticas por acceso a memoria se refiere. Las latencias de los accesos se balancean con el tiempo de cómputo en otros warps. Por tanto, los mejores resultados vuelven a obtenerse con los bloques más pequeños de máxima ocupación (192 hilos). Al no haber coalescencia, los resultados dependen exclusivamente del tamaño de bloque, no de su geometría.

La desactivación de la caché L1 reduce el tamaño de los segmentos de transacción a una cuarta parte. Por tanto, en patrones dispersos o aleatorios donde apenas exista reutilización de segmentos, un menor tamaño de estos disminuye el tráfico de datos entre las distintas memorias y alivia los cuellos de botella. En el caso de accesos dispersos regulares se consigue una mejora de rendimiento de entre 20-40 % para diferentes tamaños de bloques. Para los accesos aleatorios, la mejora no es significativa ya que la alta carga computacional en los warps se solapa con los tiempos de trasferencia.

VII. Trabajo relacionado

La estrategia más común a la hora de implementar un código CUDA es seleccionar bloques de hilos cuyos tamaños consigan maximizar la ocupación de los SMs. El objetivo de maximizar la ocupación es reducir las latencias en los accesos a memoria global [1]. Este trabajo sigue la tendencia habitual de seleccionar bloques cuadrados cuyas cardinalidades son potencias de dos con el objetivo de facilitar la implementación de los códigos. El impacto de geometrías no cuadradas, así como las cardinalidades de las dimensiones correspondientes a potencias de tres,

	Columns										
Rows	32	48	64	96	128	192	256	384	512	768	1 024
1	1,1087	0,9048	0,7893	0,6852	0,6582	0,6268	0,6358	0,6312	0,6330	0,6635	0,7885

TABLA I

Reducción de vector. Execution times (ms.)

	Columns														
Rows	8	12	16	24	32	48	64	96	128	192	256	384	512	768	1 024
128	9.01														
96	**6.35**														
64	**5.49**	**5.48**	5.78												
48	**5.24**	5.60	**4.00**												
32	**4.86**	**4.80**	**3.36**	**3.88**	4.23										
24	**4.70**	4.68	**3.28**	4.17	**3.07**										
16	5.05	**4.43**	**3.23**	**3.45**	**3.04**	**3.38**	4.14								
12	5.42	4.58	**3.28**	3.55	**3.02**	3.43	**3.05**								
8	6.05	4.97	3.42	**3.41**	**2.96**	**3.05**	**2.95**	**3.17**	4.45						
6	7.18	5.52	3.87	3.53	**2.95**	3.06	**2.94**	3.46	**3.19**						
4	9.70	6.73	5.06	3.92	3.11	**3.10**	**2.90**	**3.00**	**3.05**	**3.19**	4.44				
3	11.94	8.48	6.16	4.64	3.53	3.12	**2.89**	2.96	**2.95**	3.35	**3.18**				
2	16.43	11.43	8.55	6.12	4.54	3.68	3.08	**2.93**	**2.93**	**2.95**	**2.96**	**2.99**	3.95		
1	29.97	20.24	15.16	10.51	7.74	5.73	4.40	3.49	3.08	**2.89**	**2.89**	**2.91**	**2.94**	**3.01**	3.94

TABLA II

Suma de matrices. Execution times (ms.)

	Columns										
Rows	32	48	64	96	128	192	256	384	512	768	1 024
32	6441										
24	**5842**										
16	**5218**	**6094**	6579								
12	**5121**	6478	**5979**								
8	**4982**	**5862**	**5265**	**5479**	6470						
6	**4860**	5775	**5293**	5940	**5457**						
4	6177	**4746**	**4855**	**4898**	**4915**	**4743**	6066				
3	7960	5918	**4653**	4928	**4649**	5421	**4520**				
2	11890	8121	6103	**4415**	**4339**	**4325**	**4450**	**4288**	6172		
1	23730	16086	12073	8399	6967	**5855**	**5866**	**5909**	**6120**	**5951**	7223

TABLA III

multiplicación de matrices. Execution times (ms.)

	Columns														
Rows	8	12	16	24	32	48	64	96	128	192	256	384	512	768	1 024
128	12.8														
96	**12.6**														
64	**12.5**	**12.6**	12.6												
48	**12.4**	12.4	**12.5**												
32	**12.2**	**12.3**	**12.3**	**12.3**	12.3										
24	**12.1**	12.1	**12.1**	12.1	**12.1**										
16	11.8	**11.9**	**11.9**	**12.0**	**12.0**	**12.0**	11.9								
12	11.6	11.7	**11.8**	11.8	**11.8**	11.8	**11.8**								
8	11.2	11.4	11.5	**11.7**	**11.7**	**11.7**	**11.7**	**11.7**	11.6						
6	10.8	11.1	11.2	11.5	**11.6**	11.6	**11.6**	11.5	**11.6**						
4	11.5	11.5	11.5	11.1	11.3	**11.5**	**11.5**	**11.5**	**11.5**	**11.5**	11.2				
3	10.2	10.4	10.5	10.9	11.0	11.3	**11.4**	11.4	**11.4**	11.3	**11.4**				
2	10.2	9.9	10.0	10.5	10.6	11.0	11.1	**11.3**	**11.4**	**11.3**	**11.3**	**11.3**	10.9		
1	12.8	11.4	10.0	9.7	9.9	10.4	10.5	10.8	11.0	**11.3**	**11.3**	**11.3**	**11.2**	**11.2**	11.1

TABLA IV

Accesos regularmente dispersos. Execution times (ms.)

Rows	Columns 32	48	64	96	128	192	256	384	512	768	1 024
32	*										
24	**347.27**										
16	**388.41**	**345.63**	*								
12	**334.81**	367.85	**347.79**								
8	**330.47**	**332.90**	**388.28**	**347.56**	*						
6	**324.86**	325.46	**334.77**	369.78	**347.23**						
4	325.50	**347.18**	**330.47**	**334.99**	**388.41**	**347.18**	*				
3	328.07	357.41	**324.82**	327.35	**334.75**	369.59	**347.30**				
2	370.40	326.11	325.48	**324.88**	**330.43**	**334.72**	**388.42**	**347.17**	*		
1	634.43	490.32	370.41	328.04	325.47	**324.82**	**330.43**	**334.80**	**388.49**	**347.45**	*

TABLA V
ACCESOS ALEATORIOS. EXECUTION TIMES (MS.)

no están suficientemente estudiadas.

Wynters, en [6], muestra una implementación trivial de la multiplicación de matrices donde se prueban varios tamaños de bloques. Este es otro ejemplo donde no se consideran las geometrías rectangulares. Además, este trabajo está basado en arquitecturas pre-Fermi.

En [7] y [8] se muestran implementaciones de varios problemas variando una serie de parámetros significativos tales como el tamaño de bloque, el tamaño de los datos de entrada, la precarga de datos y la carga de trabajo por hilo entre otros. La información del rendimiento obtenido en estas pruebas se analiza para reducir el espacio de búsqueda y ayudar al programador a seleccionar la configuración óptima. De nuevo, este trabajo sólo considera arquitecturas pre-Fermi.

Sobre Fermi, en [9], muestra cómo la jerarquía de memoria caché introducida por esta arquitectura mejora significativamente la localidad de los datos aumentando el rendimiento global de las aplicaciones. Sin embargo, este trabajo no estudia como se relacionan los efectos producidos por las memorias caché con los diferentes tamaños y geometrías de los bloques de hilos.

VIII. Conclusiones

Hoy en día desarrollar códigos que consigan explotar eficientemente las capacidades de los dispositivos GPU sigue siendo un trabajo muy complicado ya que es necesario conocer estrechamente la arquitectura de cada dispositivo.

Una de las decisiones más importantes a la hora de desarrollar códigos para estos dispositivos es la elección apropiada de los parámetros globales de configuración. Dichos parámetros incluyen el tamaño y la geometría del bloque, así como configuración la memoria caché L1. Estas elecciones presenta un impacto significativo sobre el rendimiento global de las aplicaciones, explicables a partir de las características de la arquitecta del dispositivo GPU.

La arquitectura Fermi introduce nuevas características hardware que impiden reutilizar el conocimiento adquirido con las versiones anteriores. Este trabajo presenta un estudio y evaluación inicial de la arquitectura Fermi para ayudar a determinar la configuración óptima.

La elección de los parámetros globales están estrechamente relacionados con la implementación de cada problema. En este artículo mostramos que un estudio combinado del conocimiento de la arquitectura GPU y las características propias de la implementación del código puede significativamente ayudar en la elección del tamaño y la geometría del bloque de hilos, así como en la configuración de la memoria caché L1.

Agradecimientos

Esta investigación está parcialmente financiada por el Ministerio de Industria (CENIT MARTA, CENIT OASIS, CENIT OCEANLIDER), Ministerio de Ciencia y Tecnología (CAPAP-H3 network, TIN2010-12011-E) y el proyecto HPC-EUROPA2 (Nº:228398).

Referencias

[1] David B. Kirk and Wen-mei W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, Morgan Kaufmann, Feb. 2010.

[2] NVIDIA, "Whitepaper: NVIDIA's next generation CUDA compute architecture: Fermi," 2010, http://www.nvidia.com/object/fermi_architecture.html, Last visit: Nov, 2010.

[3] NVIDIA, "Fermi Architecture Home Page," Last visit: August 2, 2010, http://www.nvidia.com/object/fermi_architecture.html.

[4] NVIDIA, "Nvidia cuda programming guide 3.0 fermi," 2010.

[5] Paulius Micikevicius Greg Ruetsch, "Nvidia optimizing matrix transpose in cuda," http://developer.download.nvidia.com/compute/cuda/3_0/sdk/website/CUDA/website/C/src/transposeNew/doc/MatrixTranspose.pdf, June 2010, Last visit: Dec 2, 2010.

[6] Erik Wynters, "Parallel processing on nvidia graphics processing units using cuda," *J. Comput. Small Coll.*, vol. 26, pp. 58–66, January 2011.

[7] Shane Ryoo, Christopher I. Rodrigues, Sam S. Stone, John A. Stratton, Sain-Zee Ueng, Sara S. Baghsorkhi, and Wen-Mei W. Hwu, "Program optimization carving for GPU computing," *Journal of Parallel and Distributed Computing*, vol. 68, no. 10, pp. 1389–1401, Oct. 2008.

[8] Sara S. Baghsorkhi, Matthieu Delahaye, Sanjay J. Patel, William D. Gropp, and Wen-mei W. Hwu, "An adaptive performance modeling tool for gpu architectures," *SIGPLAN Not.*, vol. 45, pp. 105–114, January 2010.

[9] Changyou Zhang Xiang Cui, Yifeng Chen and Hong Mei, "Auto-tuning dense matrix multiplication for GPGPU with cache," in *Proc. ICPADS'2010*, Shanghai, China, Dec. 2010, pp. 237–242.