Paralelización con CUDA de un código de dinámica molecular

Eneko Mateo, Alexander Mendiburu, José Miguel-Alonso
Intelligent Systems Group
Computer Science and Engineering School
The University of the Basque Country UPV/EHU
San Sebastian, Spain 20018
enekomateo@gmail.com,{amendiburu, j.miguel}@ehu.es

Resumen

En este artículo hacemos un resumen del primer contacto que hemos tenido con la computación GPGPU. Se ha portado parte de un programa Fortran a *CUDA* de *NVIDIA*. Para ello se ha realizado un *profiling* de la aplicación serie para detectar las partes con mayor coste computacional, que serán el objetivo a paralelizar. Se describen las transformaciones que se han realizado al código, que incluyen algunas derivadas de la necesidad de hacer interoperar dos lenguajes de programación diferentes. Por último, se incluye una comparación del rendimiento entre la versión secuencial y la paralelizada con *CUDA*.

1. Introducción

El paralelismo cada vez es más accesible. Hoy en día no es necesario acudir a un centro HPC (*High-performance computing* o computación de alto rendimiento) para poder ver varias aplicaciones corriendo concurrentemente, puesto que casi cualquier procesador contiene dos o más núcleos capaces de ejecutar varias aplicaciones a la vez.

Hasta hace poco lo habitual para la computación en paralelo era unir varios procesadores o montarse clústeres (grupos de ordenadores comunicados entre sí), pero ¿es el procesador el único elemento del ordenador que realiza cálculos? Obviamente no. Las tarjetas gráficas realizan gran cantidad de cálculos al tratar las imágenes. A la utilización de estos dispositivos para realizar computación de propósito general se la denomina GPGPU (*general-purpose computing on graphics procesing units*).

Para entender un poco mejor por qué se pueden realizar estos cálculos, veamos a grandes rasgos la evolución tanto de las tarjetas gráficas como de los microprocesadores. Las tarjetas gráficas (GPUs) tomaron una dirección muy diferente a la de los procesadores (CPUs). Mientras que los procesadores tomaron el camino de ser elementos de computación general, las tarjetas gráficas se especializaron en el tratamiento de imágenes digitales. Esto provocó que ambas arquitecturas se fueran distanciando y, mientras las CPUs se hacían con más memoria cache para almacenar y manejar diferentes datos con mayor facilidad, las GPUs se hacían con más unidades de cálculo (más sencillas que las integradas en las CPUs), puesto que su principal trabajo era realizar las mismas tareas múltiples veces sobre diferentes datos.

Por las cualidades que ofrecen las GPUs, ya sea su gran nivel de paralelismo o su reducido coste en comparación con un superordenador, nos ha parecido interesante realizar un acercamiento a esta nueva tecnología para averiguar cuál es su rendimiento, así como hacia dónde podría evolucionar.

En este artículo describimos las razones que nos llevaron al uso de *CUDA* (Sección 2), el programa Fortran a acelerar (Sección 3), las transformaciones realizadas al código original (Sección 4), incluyendo algunas medidas de rendimiento. Finalizamos con las conclusiones en la Sección 5.

2. Elección de CUDA

Aunque a estas alturas ya es de sobra conocido, *CUDA* son las siglas de (Compute Unified Device Architecture) que hace referencia tanto a la arquitectura utilizada en tarjetas *NVIDIA* como a un conjunto de herramientas de desarrollo creadas por *NVIDIA* que permiten a los programadores usar una variación del lenguaje de programación

C/C++ para codificar algoritmos en GPUs de *NVIDIA*.

CUDA no es el único modo de crear aplicaciones de propósito general sobre GPUs, pero sí es uno de los primeros; debido a eso hay disponibles múltiples recursos (libros, artículos, páginas web y foros) que ofrecen abundante información [1][2]. La competencia más directa en estos momentos es *OpenCL* (Open Computing Lenguage o Lenguaje de Computación Abierto) [3], que es un estándar abierto y puede ejecutarse tanto en GPUs como CPUs. Otra ventaja de este estándar emergente es que puede funcionar tanto en tarjetas *NVIDIA* como *AMD/ATI*. En este proyecto se decidió apostar por *CUDA*, dada la madurez de las herramientas y la disponibilidad de hardware de *NVIDIA*. Más adelante nos plantearemos la posibilidad de portar los códigos a *OpenCL*, tarea que se prevé fácil dada la similitud entre ambos entornos.

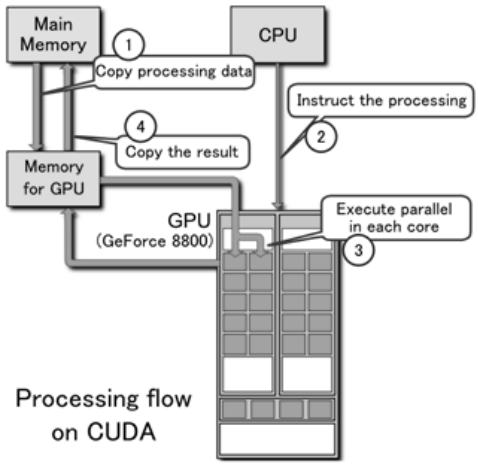

Figura 1. Flujo de ejecución *CUDA*[1]

Como se puede apreciar en la figura 1, el flujo de ejecución de CUDA es bastante sencillo. En primer lugar se copian los datos desde la memoria principal a la memoria de la GPU (1). Luego el procesador dispara la puesta en marcha de la ejecución en paralelo (2). La tarjeta gráfica se encarga de la ejecución en paralelo (3). Y para terminar se copian los resultados de la memoria de la GPU a la memoria principal (4).

3. El programa a paralelizar

Nuestro objetivo principal era acelerar, usando *CUDA*, un código desarrollado en Fortran por el Dr. Andreas Heidenreich [4] que denominamos **plasma**.

3.1. Funcionamiento del programa plasma

Estamos ante un programa que realiza simulaciones de dinámica molecular. Estas simulaciones se realizan para analizar la ionización multielectrón extrema y la dinámica de electrones durante attosegundos (trillonésima parte de un segundo, 1 as = 10^{-18} s) o femtosegundos (milbillonésima parte de un segundo, 1 fs = 10^{-15} s) en grupos elementales y moleculares impulsados por campos láser ultra-intensos (picos de intensidad IM=1015–1016 W cm^{-2}), ultra-rápidos (ancho de pulso temporal τ = 10–100 fs) y cercanos al espectro infrarrojo (frecuencia fotón 0.35 fs-1). Para ser más claros, dado el alto grado de tecnicismos aparecidos en las líneas precedentes, las simulaciones realizan cálculos de cómo afecta la excitación mediante un haz láser, con las características recientemente mencionadas, a un grupo de átomos de xenón y sus correspondientes electrones. Los cálculos de las simulaciones sirven para averiguar si la energía transmitida por dicha excitación es lo suficientemente potente como para crear electrones independientes. También se tienen en cuenta estos electrones a la hora de realizar los cálculos, si colisionan con otros electrones y debido a esa colisión se liberan de sus órbitas generando más electrones independientes.

El programa está escrito en Fortran 77 y consta de dos partes, la primera parte se encuentra en el archivo **mdset1_2.f** que se encarga de realizar las inicializaciones pertinentes en el archivo de salida **init.mdrst**. Una vez compilada la parte correspondiente a **mdset**, hay que indicarle cuales serán los parámetros iniciales que se encuentran en el archivo **set.inp**. Modificando el radio que aparece en este archivo conseguiremos aumentar o disminuir el número de iones a tratar. Esta parte sólo se ejecutará para

[1] http://es.wikipedia.org/wiki/Archivo:CUDA_processing_flow_(En).PNG

crear el estado inicial, por lo tanto paralelizar esta sección no supondría un ahorro de tiempo sustancial. La segunda parte se encuentra en el archivo **plasma1_11a.f** y se encarga de realizar todos los cálculos necesarios en la simulación. Esta parte también tiene un archivo, **plasma.inp**, donde se guardan los parámetros que le indicarán como tiene que ser la simulación.

```
C Definir parámetros atómicos
    CALL PARAMS
C Entrada del control de la ejecución DM
    CALL INPUT
C Leer desde el archivo de reanudación
    CALL INDATA
C Inicializar algunos archivos
    CALL FILINI
C Calcular la trayectoria
    CALL PROPAG
C Análisis al finalizar la simulación
    CALL TERM
```

Algoritmo 1. Estructura del programa **plasma**.

En el Algoritmo 1 se puede apreciar la parte principal de **plasma**, que simplemente es una secuencia de llamadas a subrutinas. Como se puede leer en los comentarios, la subrutina **PARAMS** realiza la tarea de definir parámetros atómicos y moleculares. Seguidamente aparece una función llamada **INPUT**, que lee de la entrada estándar el nombre del archivo donde se encuentran los parámetros de la ejecución. También crea el archivo de salida **plasma.txt**, que contiene un resumen de la ejecución. En ese resumen se pueden encontrar todo tipo de datos relacionados con la simulación, entre ellos la cantidad de átomos y electrones, la anchura de pulso, intensidad y periodo del láser, distribución de las cargas por valores energéticos, media de energías, así como un pequeño análisis de rendimiento. Ese archivo se va rellenando según avanza la ejecución, y lo actualizan varias subrutinas, ya que algunos datos sólo se pueden escribir al final de la ejecución del programa principal, por ejemplo el tiempo que ha transcurrido durante la ejecución. **INDATA** coge el archivo que se le ha indicado desde la entrada estándar y lo va leyendo para ir creando y actualizando los parámetros con los valores que se utilizarán en los cálculos. Después de que **INDATA** haya actualizado los datos, se ejecuta **FILINI**, que se encarga de crear algunos archivos e inicializarlos. Crea los archivos **bsi.txt**, **elimp.txt** y **outer.txt**, los cuales tendrán información importante sobre las colisiones ocurridas. Entre los archivos que inicializa también se encuentra **plasma.txt**. **PROPAG** es el centro de la aplicación, porque se encarga de expandir el sistema, es decir, es el que se encarga de la simulación. La parte principal de **PROPAG** es un bucle que se repite las veces que se indica en el archivo **plasma.inp**. Una vez terminadas todas las iteraciones de **PROPAG**, entra en ejecución la subrutina **TERM** y realiza el análisis final de la trayectoria. Ese análisis queda plasmado en el final del archivo **plasma.txt**.

3.2. Análisis del código

Para averiguar que partes son las que tienen mayor carga computacional dentro de la sección correspondiente a la simulación, se ha utilizado la herramienta de profiling *VTune* de *Intel* [5]. En la Figura 2 se puede apreciar parte del resultado del análisis realizado. Como se puede comprobar, la mayor parte del tiempo es consumido por la librería matemática implementada con SSE2, realizando cálculos de potencias. Después de esta librería, se puede apreciar que **bsi** y **forcel** son las subrutinas que más tiempo consumen, cada una con más o menos un 21% del tiempo total de ejecución. Su aceleración, por tanto, sería la que mayores mejoras de rendimiento generarían. Procede un análisis más profundo de las mismas para estudiar si son aptas para la paralelización. La siguiente subrutina que más tiempo ocupa es **elimp** que representa casi un 9% del tiempo total. Esta sería un posible objetivo para futuras mejoras de la paralelización. El resto de funciones representan cada una menos del 1% de ejecución por lo que su paralelización no supondría una mejora sustancial.

Name	CPU sam	INST_ sampl	Clock per...	CPU_CL %	INST_RE %
__libm_sse2_pow	1,804	3,354	0,538	44,60%	58,76%
bsi_	865	794	1,089	21,38%	13,91%
forcel_.	858	714	1,202	21,21%	12,51%
elimp_.	358	651	0,550	8,85%	11,41%
__svml_pow2.L	58	91	0,637	1,43%	1,59%
correc_.	40	36	1,111	0,99%	0,63%
propag_	15	25	0,600	0,37%	0,44%
forcat_.	15	18	0,833	0,37%	0,32%

Figura 2. Análisis de **plasma** con VTune.

4. Transformación

Cómo se acaba de ver, las funciones o subrutinas candidatas a paralelizar son: **bsi** y **forcel**. Se ha optado por empezar con **forcel**, puesto que su funcionamiento es algo más sencillo de entender que el de **bsi**, aunque las secciones a paralelizar realicen un trabajo parecido en ambas subrutinas.

4.1. Funcionamiento de forcel

El objetivo de la función **forcel** es calcular las magnitudes de las fuerzas que inciden sobre los electrones independientes. En el Algoritmo 2 se puede apreciar un pseudocódigo explicando cual es el funcionamiento de la subrutina **forcel**. El primer bucle realiza la inicialización de la matriz **fel** que guardará los valores de las fuerzas en los ejes X, Y y Z, por lo que se podría concluir que **fel** es la unión de tres vectores (de hecho así será el tratamiento en la parte *CUDA*) cuya longitud es el número de electrones. El siguiente bucle realizará los cálculos de las fuerzas que inciden sobre cada electrón; para ello primero inicializa unas variables que servirán de base para realizar los cálculos necesarios en los siguientes bucles. El primero de estos bucles realiza los cálculos de las fuerzas que realizan los **I**-1 electrones anteriores al electrón **I** sobre este último. El segundo bucle interior realiza una tarea parecida, pero esta vez calcula las fuerzas que efectúan todos los átomos de xenón sobre el electrón **I**. Después de estos bucles se realiza la actualización de la matriz **fel** y vuelve a empezar el ciclo. El último bucle hace las correcciones pertinentes al campo electromagnético sobre las fuerzas almacenadas en **fel**.

```
do I=1 hasta nelecs
     inicializar_fel
end do
do I=1 hasta nelecs
     inicializar_variables
     do J=1 hasta I
            fuerzas_elec_elec
     end do
     do J=1 hasta natoms
            fuerzas_atom_elec
     end do
     actualizar_fel
end do
do I=1 hasta nelecs
     fuerzas_electromagnéticas
end do
```

Algoritmo 2. Pseudocódigo de **forcel**.

4.2. Código *CUDA* para forcel

Analizando la estructura del programa, con un bucle más externo y otros internos, decidimos afrontar la paralelización de los bucles interiores, los que se encargan de los cálculos de las fuerzas electrón-electrón y átomo-electrón. Cada uno de esos bucles ha sido transformado en un kernel, es decir, lo que en un principio era una forma de tratar los elementos de forma secuencial, se ha convertido en pequeños programas que se ejecutarán de forma paralela en la GPU.

El primer bucle se ha transformado en el kernel que aparece en el Algoritmo 3. La transformación es bastante intuitiva, debido a que la mayoría de acciones son cálculos matemáticos sin dependencias entre diferentes iteraciones del bucle. El bucle se realiza **I**-1 veces siendo **I** el numero de la posición del electrón dentro del vector.

Un cambio importante en el código *CUDA* es que los elementos **fi1**, **fi2** y **fi3** pasan de ser números reales, *floats* en este caso, a ser vectores, donde se guardará el valor calculado, a los que luego se les aplica una reducción para obtener el total de las fuerzas electrón-electrón que inciden sobre el electrón **I**. En la solución *CUDA*, para simplificar las cosas, como ya se ha comentado, vectores como **fel** ya no son multidimensionales sino que serán divididos en sus correspondientes vectores cómo: **fel1**, **fel2** y **fel3**. La forma de dividirlos se verá más adelante cuando se explique la funcionalidad del **wrapper** (código adicional de

preparación del entorno CUDA para poder ejecutar los kernels).

El kernel se lanza en bloques, cada uno con su colección de hilos. Para que cada hilo procese sólo el elemento que le corresponde se ha procedido de la siguiente manera: cada hilo tiene un identificador que va desde 0 a **N**-1 (donde **N** es el número de hilos que tiene el bloque); pasa lo mismo con los identificadores de bloque, que van desde 0 a **M**-1 (donde **M** es el número de bloques que se lanzan al iniciar la ejecución). Para que cada hilo sólo ejecute un elemento del vector, se ha multiplicado el identificador de bloque por la cantidad de hilos que tiene cada bloque y se le ha sumado el identificador de hilo, tal como se indica en la Ecuación 1.

$$Idx=blockIdx.x \times blockDim.x+threadIdx.x \quad (1)$$

Como lo que se lanzan son **M** bloques de **N** hilos se puede dar la situación que haya más hilos que elementos a procesar. Para que esto no tenga efectos en el resultado se ha añadido el control **idx**<**I**; de esta forma solo se procesan **I**-1 elementos, que son los que se procesan en las veces que se repite el bucle original.

```
const float POW32=-3./2.;
if (idx<N) {
  xij=xi-rel1[idx];
  yij=yi-rel2[idx];
  zij=zi-rel3[idx];
  rij2=xij*xij+yij*yij+zij*zij+R02;
  ffac=pow(rij2,POW32);

 //Fuerzas en electrón I causadas
 //por electrón J
  fxi=xij*ffac;
  fyi=yij*ffac;
  fzi=zij*ffac;
  fi1[idx]=fxi;
  fi2[idx]=fyi;
  fi3[idx]=fzi;
//Fuerzas en electrón J causadas
//por electrón I
  fel1[idx]-=fxi;
  fel2[idx]-=fyi;
  fel3[idx]-=fzi;
}
```

Algoritmo 3. Cuerpo del kernel correspondiente al primer bucle de **forcel**.

Con el segundo bucle, el que calcula las fuerzas átomo-electrón, se ha realizado un kernel muy similar, puesto que solo varían unos pocos cálculos. Al igual que en el kernel anterior, algunas variables pasarán a ser vectores a los que luego se les aplicará una reducción.

La reducción que se ha utilizado es básicamente igual a la que proporcionan en el SDK de *NVIDIA*. La única diferencia es que se realiza la reducción de tres vectores a la vez y se utiliza una función atómica para la suma de floats [2].

Hasta ahora las transformaciones se han centrado en la adaptación de ciertas variables para que encajen en una ejecución paralela, pero todavía queda preparar el entorno para la ejecución de ese código. Para ello se ha utilizado un **wrapper**, al cual se le llama desde la parte Fortran. Este **wrapper** es el encargado de acceder a los common blocks de Fortran y posibilitar tanto su copia cómo su reestructuración para que encajen bien en la memoria de la GPU.

El funcionamiento del **wrapper** es muy sencillo y, como se acaba de comentar, una de sus funciones es adaptar los parámetros de Fortran para su procesamiento en la GPU. Las funciones que realiza son las siguientes: primero, la definición de variables locales; después viene la reserva de la memoria en la GPU, junto con la copia y adaptación de las variables provenientes del código Fortran; a continuación se realiza el lanzamiento de los kernels, indicando el número de hilos (que, como ya se ha comentado, se calcula a partir del número de bloques y la cantidad de hilos que contiene cada bloque). Una vez terminados todos los kernels, se realiza la recopilación de resultados, copiándolos nuevamente en la memoria principal. Para finalizar, el **wrapper** se encarga de liberar los recursos de la GPU. En el Algoritmo 4 aparece esquematizado el **wrapper** elaborado.

```
extern "C" void wrapper_(clase
*variables)
   {definición_variables_locales
    reserva_memoria_GPU
    copia_variables_Fortran
    lanzamiento_kernels
    recopilar_y_copiar_resultados
    liberar_recursos_GPU
   }
```

Algoritmo 4. Esquema del **wrapper**.

Dado que *CUDA* es una extensión de C/C++, para su convivencia con Fortran se han seguido las mismas convenciones y técnicas que para Fortran con C [6]. Por ejemplo, como el **wrapper** será invocado desde el código Fortran, hay que utilizar la palabra reservada **extern** junto con el carácter "_" al final del nombre de la función, para indicarle al compilador que será invocado desde otro lugar ajeno al archivo donde está definida esta función. Otro ejemplo, ya comentado, es que algunas matrices (como **fel** y **rel**) han sido transformadas en tres vectores. Esta necesidad está provocada porque en Fortran el almacenamiento en memoria se realiza por columnas y en C/C++ se hace por líneas, por eso hemos tenido que realizar la transformación que aparece en el Algoritmo 5.

En este algoritmo vemos la técnica que se utiliza para acceder, desde C, a los common blocks de Fortran: la definición de estructuras con la misma organización de esos common blocks. A continuación viene el código para hacer la reserva del espacio en la memoria de la GPU. Por último, usamos **cudaMemcpy2D** para hacer la copia a la memoria. En este caso no utilizamos **cudaMemcpy2D** para copiar toda la matriz, sino que la utilizamos para coger sólo los elementos que nos interesan. También hay que recordar que, dado que Fortran trata todos los parámetros por referencia, en el código *CUDA* hay que hacer lo mismo.

```
struct COORD {
    float rat[MAXAT][DIMS],
    rel[MAXEL][DIMS];
} coord;
…
float *rel1,*rel2,*rel3;
cudaMalloc( (void **)&rel1,
sizeof(float) * elem );
…
cudaMemcpy2D( rel1, sizeof(float),
&coord_.rel[0][0], sizeof(float)*3,
sizeof(float), elem,
cudaMemcpyHostToDevice);
```

Algoritmo 5. Importación de **REL**(3,**NELECS**) de Fortran a C, y transformación a **rel1**[**nelecs**].

Como se ha comentado, se decidió portar solo cierta parte del código, por lo que la mayor parte se reutiliza en Fortran, tal como está. Nuestro código *CUDA* se encuentra en un archivo separado, con extensión .cu. Por eso la compilación se ha tenido que realizar en dos fases: en la primera, mediante el compilador de *NVIDIA nvcc* se ha compilado el código *CUDA* a un archivo objeto; se ha tenido que indicar que la arquitectura donde se ejecutaría el código *CUDA* era sm_11, puesto que de lo contrario no se permitiría la ejecución de funciones atómicas. En la segunda fase se le ha indicado al compilador de Fortran cuál es el archivo objecto donde se encuentra nuestro **wrapper**, así como la ubicación de las librerías de *CUDA* para poder montar la aplicación completa.

4.3. Pruebas

Tanto el entorno de desarrollo como el de pruebas han consistido en un equipo de prestaciones modestas. Y, conviene indicar que el código secuencial ofrece un rendimiento muy elevado sobre este equipo.

Se trata de un portátil *Asus n50vn* con un procesador *Intel Core 2 Duo T9500* con 4 GB de memoria RAM y la tarjeta gráfica que integra es una *Nvidia 9650M GT* con 1 GB de memoria. El sistema operativo ha sido *Ubuntu 9.04* de 32 bits, por lo que solo eran paginables 3 GB de la memoria.

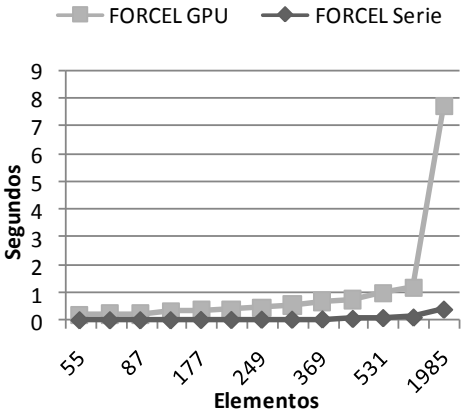

Figura 3. Tiempos de ejecución de Forcel.

En la Figura 3 aparece un gráfico con los tiempos de ejecución de la función **Forcel**, tanto en serie como en paralelo en la GPU. Por simplificar las cosas, los elementos que aparecen en el gráfico son el número de átomos y electrones independientes que se encuentran en la situación inicial, es decir que al comienzo de una ejecución hay 55 átomos de Xenón y 55 electrones independientes.

Los resultados, como se pueden ver, no son favorables a la versión *CUDA*. La versión secuencial es más rápida que la paralela, y aumentar el tamaño del problema no ayuda a vencer esta diferencia – de hecho, la aumenta.

Las razones de este comportamiento hay que buscarlas en las copias de memoria entre CPU y GPU. Estos intercambios de información tienen un elevado coste, especialmente en la GPU de bajo coste usada en la experimentación. El trabajo de cómputo realizado por cada hilo de nuestro kernel es insuficiente para amortizar dicho coste.

Se podría haber reducido este coste usando GPUs más modernas con mayor ancho de banda con memoria, transmitiendo los datos usando DMA (sin intervención directa del procesador) o utilizando técnicas de *pipelining* (mientras se trabaja con un bloque de datos, ir transmitiendo otro bloque que se procesará después).

5. Conclusiones y líneas futuras

En este artículo se ha descrito nuestra primera experiencia portando a *CUDA* un código de dinámica de partículas escrito inicialmente en Fortran secuencial (**plasma**). Para ello, y como es habitual, se han identificado en primer lugar las partes del programa que requieren un mayor tiempo de ejecución, para posteriormente centrarnos en la adaptación de una de las subrutinas. Concretamente, la subrutina **forcel**.

El objetivo principal de esta aproximación era el de estudiar los pasos necesarios para adaptar un código secuencial a un entorno de computación basado en GPUs. Para ello, ha sido necesario por un lado el uso de "wrappers", que permitan llamar desde la CPU, a funciones que se ejecutarán en la GPU. Por otro lado, también ha sido necesario reescribir el código a paralelizar, creando unidades de ejecución (kernels), que puedan ser procesados por las GPUs de manera masivamente paralela.

Atendiendo a los resultados obtenidos, sería precipitado concluir que las GPUs no son aptas para la ejecución de un programa como el que presentamos en este trabajo. Es necesario recordar que, por un lado, sólo nos hemos centrado en una subrutina concreta (que sólo requiere el 21% del tiempo total de ejecución), y por otro lado el hardware disponible colocaba a la GPU en una posición muy desfavorable respecto a la CPU.

Por otro lado, la adaptación del código debe ser estudiada en detalle, de cara a mejorar su rendimiento. La propuesta actual requiere de un trasiego de datos importante entre la CPU y la GPU, lo que supone un cuello de botella respecto al tiempo de ejecución de las tareas. Además, el código ha sido adaptado a partir de la versión secuencial, hecho que muchas veces limita el potencial de un diseño nuevo y totalmente paralelo.

En resumen, a partir de la propuesta inicial presentada en este trabajo, nuestro objetivo es el de analizar en detalle el código desarrollado e investigar las posibles modificaciones que puedan llevarse a cabo para mejorar su rendimiento, así como la evaluación del mismo en GPUs más actuales, tales como la nueva línea Fermi, que presenta un nuevo diseño así como memorias caché de nivel 2. Finalmente, también sería

interesante reescribir el programa plasma usando la librería OpenCL.

Hoy por hoy, el futuro de la supercomputación está en el uso de miles de CPUs multi-núcleo acompañadas de coprocesadores especializados. Las GPUs están haciéndose un hueco importante como coprocesadores, frente a alternativas como el Cell o las FPGAs. Es de gran importancia, por tanto, conocer cómo explotar de forma eficiente estos potentes dispositivos.

Referencias

[1] Nvidia. CUDA zone. Available at: http://www.nvidia.es/object/cuda_home_new_es.html.

[2] Nvidia. Nvidia forum. Available at: http://forums.nvidia.com/index.php?

[3] Group K. OpenCL. Available at: http://www.khronos.org/opencl/.

[4] Heidenreich, A.; Last, I; Jortner J. Simulations of Extreme Ionization and Electron Dynamics in Ultraintense Laser – Cluster Interactions. *Israel Journal of Chemistry*. 2007;47(December 2006):89-98.

[5] Intel. Intel software development products. Available at: http://software.intel.com/en-us/intel-vtune/.

[6] Yolinux.com. Tutorial: Using C/C++ and Fortran together. Available at: http://www.yolinux.com/TUTORIALS/LinuxTutorialMixingFortranAndC.html.