



Universidad Politécnica
de Madrid

**Escuela Técnica Superior de
Ingenieros Informáticos**



Grado en Grado en Ingeniería Informática

Trabajo Fin de Grado

Paralelización de algoritmos irregulares en arquitecturas GPU

Autor: Luis Arijá González
Tutor: Julio Mariño Carballo
Cotutor: Ignacio Ballesteros González

Madrid, Enero - 2024

Este Trabajo Fin de Grado se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Grado

Grado en Grado en Ingeniería Informática

Título: Paralelización de algoritmos irregulares en arquitecturas GPU

Enero - 2024

Autor: Luis Arija González

Tutor: Julio Mariño Carballo

DLSIIS

Escuela Técnica Superior de Ingenieros Informáticos

Universidad Politécnica de Madrid

Cotutor: Ignacio Ballesteros González

DLSIIS

Escuela Técnica Superior de Ingenieros Informáticos

Universidad Politécnica de Madrid

Tabla de contenidos

1. Introducción	1
1.1. Descripción General	1
1.2. Problema de los N Cuerpos	1
1.3. Solución Propuesta	1
2. Preliminares	3
2.1. CPU y GPU	3
2.2. CUDA	4
2.3. Kernels en CUDA	5
2.4. Jerarquía de Bloques en CUDA	6
2.5. Compartición de Memoria en CUDA	7
Bibliografía	9

1. *Introducción*

1.1. **Descripción General**

En el ámbito de la computación de alto rendimiento, la paralelización de algoritmos irregulares en arquitecturas GPU ha emergido como un campo de investigación crucial para abordar problemas computacionales complejos. Este Trabajo de Fin de Grado se centra en la aplicación de CUDA (Arquitectura Unificada de Dispositivos de Cómputo), una plataforma de computación paralela desarrollada por NVIDIA. Específicamente, se abordará el desafío computacional conocido como el problema de los N cuerpos.

1.2. **Problema de los N Cuerpos**

El problema de los N cuerpos es un desafío computacional que surge en diversas disciplinas, desde la simulación astronómica hasta la dinámica molecular. En esencia, implica predecir y analizar las trayectorias y posiciones de N objetos en un sistema, teniendo en cuenta las interacciones gravitacionales entre ellos.

La complejidad inherente a este problema radica en la dependencia no lineal de cada partícula respecto a todas las demás, lo que resulta en un conjunto de ecuaciones de movimiento altamente interconectadas.

1.3. **Solución Propuesta**

La solución exacta para el problema de los N cuerpos implica resolver estas ecuaciones para cada objeto en cada instante de tiempo, una tarea que se vuelve computacionalmente prohibitiva a medida que el número de objetos aumenta. Debido a ello, la paralelización de este problema se convierte en una estrategia esencial para reducir el costo temporal de la resolución de sistemas con un gran número de objetos.

El propósito de este Trabajo de Fin de Grado es explorar y desarrollar estrategias efectivas de paralelización utilizando GPU's para abordar el problema de los N cuerpos. Se investigarán y compararán enfoques de paralelización para algoritmos irregulares, considerando la naturaleza altamente interconectada de las interacciones gravitatorias en este contexto. Además, se evaluará el rendimiento

Capítulo 1. Introducción

de las implementaciones paralelas en función de diversos parámetros, como el número de partículas y la arquitectura de la GPU.

2. Preliminares

2.1. CPU y GPU

La CPU (Unidad Central de Procesamiento) es el componente principal de todo computador el cual se encarga de procesar las señales electromagnéticas y hacer posible la computación de las mismas.

Está compuesto por una pequeña cantidad de núcleos de procesamiento de gran capacidad computacional, lo cual lo vuelve un potente motor de ejecución adecuado para las cargas de trabajo en las que el rendimiento por núcleo es importante.

La GPU (Unidad de Procesamiento Gráfico) es un procesador compuesto de una gran cantidad de núcleos de menor capacidad computacional pero especializados para la rápida resolución de cálculos, lo cual la vuelve perfecta para la paralelización de cálculos algorítmicos complejos y extensos.

Debido a la naturaleza del lenguaje de programación utilizado, la CPU y la GPU estarán en comunicación para transferir información y comandos a ejecutar.

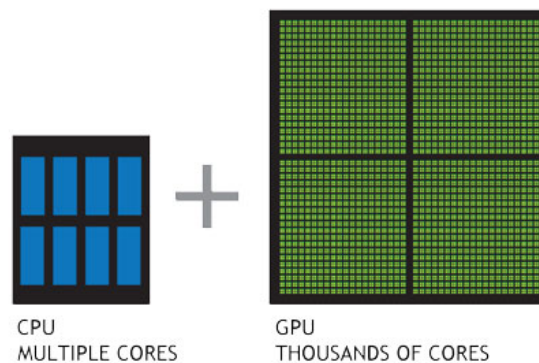


Figura 2.1: Representación de Núcleos de CPU y GPU

2.2. CUDA

CUDA, o Arquitectura Unificada de Dispositivos de Cómputo, es una potente plataforma de computación paralela desarrollada por NVIDIA y lanzada por primera vez el 27 de junio del 2007.

La programación en CUDA se realiza mediante un lenguaje basado en extensiones de C, lo que facilita a los desarrolladores la creación de funciones llamadas "kernels". Estos kernels son esenciales ya que se ejecutan en paralelo dentro de la GPU, permitiendo realizar cálculos intensivos de manera eficiente al paralelizarlos.

2.3. Kernels en CUDA

En CUDA, un kernel es una función especial que se ejecuta en paralelo dentro de la GPU. A diferencia de las funciones clásicas -orientadas a resolver todo el problema con una única invocación- las funciones kernel están diseñadas de tal manera que se pueden invocar múltiples instancias concurrentes de un kernel con una única llamada con el objetivo de que cada instancia resuelva una parte del problema.

Cada instancia sería ejecutada por un hilo individual en la GPU, con múltiples hilos trabajando en paralelo. Es dentro de la propia llamada al kernel donde se especifica cuántas instancias se invocarán de dicho kernel y en que formato será instanciado.

Esto se hace mediante una convención que sigue el formato:

->NombreFunción «NBloques, NHilos» (Arg1, ..., ArgN);

En este formato, NBloques y NHilos son enteros positivos que representan la cantidad de bloques de hilos que se invocarán para la función y la cantidad de hilos en cada bloque, respectivamente. Este enfoque paralelo proporciona una manera eficiente de aprovechar el poder de procesamiento masivo de las GPU para tareas que requieren un alto rendimiento computacional.

A la hora de definir el kernel este se marca con la palabra clave 'global' para indicar que son funciones que se ejecutarán en la GPU. Además, los parámetros del kernel suelen incluir punteros a datos en la memoria del dispositivo (GPU).

```
33  // ...
37  __global__ void AddMatrix(int* out, int* a, int* b) {
38      int nThread = threadIdx.x;
39      int nBlock = blockIdx.x;
40      int blockDim = blockDim.x;
41      int id = nBlock * blockDim + nThread;
42      out[id] = a[id] + b[id];
43  }
```

Figura 2.2: Ejemplo de definición de Kernel

2.4. Jerarquía de Bloques en CUDA

Siempre que se invoca un kernel en CUDA se ha de declarar de cuantos bloques está compuesto el grid y de cuantos hilos está compuesto cada bloque. Tras ello se ejecutará el kernel un número de veces igual a la multiplicación del número de bloques por el número de hilos por bloque.

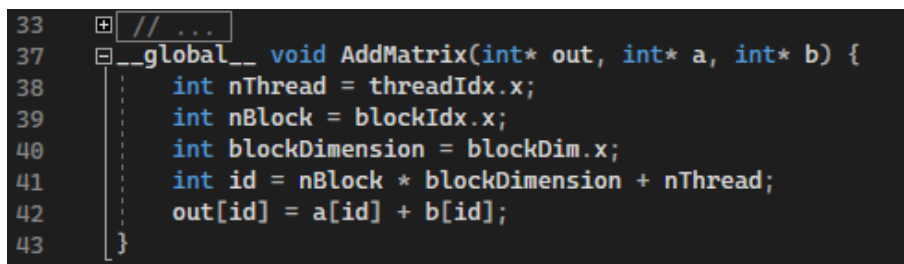
Esa distinción entre bloques e hilos nace del hecho de que los núcleos de GPU no pueden soportar un número infinito de procesos concurrentes. Cada bloque es en verdad un núcleo, y cada núcleo tiene un límite máximo de 1024 hilos concurrentes.

A la hora de desarrollar código en CUDA, uno ha de saber utilizar el concepto de bloque e hilo a su favor. Uno no suma dos vectores con 1000 valores en su interior uno a uno, sino que invoca un kernel con 1000 instancias para que cada instancia sume un único valor.

Y es esa fragmentización del trabajo de la cual parte CUDA, esta complicándose con el correcto uso de los bloques e hilos, y la correcta preparación del entorno y kernel para la correcta devolución de resultados.

Eso se debe a que solo hay un dato que difiere entre todas las instancias del kernel: El id del propio kernel, el cual solo se sabe una vez iniciado la instancia del kernel. Este id depende de tres valores: El bloque al que pertenece el hilo (id del bloque), el hilo al que pertenece dentro de ese bloque (id del hilo), y la dimensión del bloque misma.

Esos tres datos ayudan a crear un id único para cada instancia del kernel, y tiene un rango de [0, (Número Total de Hilos - 1)].



```
33 // ...
37 __global__ void AddMatrix(int* out, int* a, int* b) {
38     int nThread = threadIdx.x;
39     int nBlock = blockIdx.x;
40     int blockDim = blockDim.x;
41     int id = nBlock * blockDim + nThread;
42     out[id] = a[id] + b[id];
43 }
```

Figura 2.3: Ejemplo de Kernel que suma dos vectores

Trabaja con los mismos datos, el código es el mismo, pero tiene que dar un resultado propio en una posición concreta del vector de resultados.

Eso fuerza al usuario a codificar los kernels de tal manera que dependan de gran manera de los distintos id's del kernel. No solo definen donde va a resultar el valor final, sino que valores se van a utilizar para obtenerlo.

2.5. Compartición de Memoria en CUDA

Los kernels trabajan dentro de la propia GPU, por ello todo dato que utilicen ha de encontrarse dentro de su memoria. Dado que en CUDA la información no se comparte de manera pasiva entre CPU y GPU, es necesario invocar ciertas funciones para pasar conjuntos de datos de una pieza a la otra antes y después de invocar un kernel.

Las funciones siguen cierto orden:

- Primero se prepara un espacio en la GPU del tamaño del dato que se quiera transferir con una versión de CUDA de Malloc, llamada `cudaMalloc`;
- Tras ello se copia el dato en cuestión dentro de ese espacio de la GPU con `cudaMemcpy`;
- Una vez se tienen los datos en la GPU, se invoca el kernel;
- Obtenido el resultado, se envía de vuelta a la CPU con `cudaMemcpy`;
- Se libera el espacio en la GPU con `cudaFree`;

```
//Hacer espacio en la GPU
cudaMalloc((void**)&d_v1, sizeof(int) * size1);
cudaMalloc((void**)&d_v2, sizeof(int) * size1);
cudaMalloc((void**)&d_vOut, sizeof(int) * size1);

//Pasar información al GPU
cudaMemcpy(d_v1, vector1, sizeof(int) * size1, cudaMemcpyHostToDevice);
cudaMemcpy(d_v2, vector2, sizeof(int) * size1, cudaMemcpyHostToDevice);

//Invocar funcion de suma
AddVector << <1, size1>> > (d_vOut, d_v1, d_v2);

//Pasar información resultante de vuelta
cudaMemcpy(vOut, d_vOut, sizeof(int) * size1, cudaMemcpyDeviceToHost);

//Liberar el espacio usado en la GPU
cudaFree(d_v1);
cudaFree(d_v2);
cudaFree(d_vOut);
```

Figura 2.4: Ejemplo de Código que envuelve a un kernel de suma de vectores

Dentro de los propios hilos del kernel hay compartición de datos en cuanto a los argumentos debido a que se suelen pasar punteros. Más allá de eso, cada hilo genera información única a la cual ningún otro hilo tiene acceso a no ser que se declare lo contrario.

Bibliografia