Arquitectura de Sistemas Paralelos

Práctica 4

Programación de GPUs
CUDA

Michael Alexander Fajardo Malacatus
Gloria del Valle Cano
michael.fajardo@estudiante.uam.es & gloria.valle@estudiante.uam.es

Grado en Ingeniería Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid
24-04-2020

Índice general

Índice de figuras

Introducción

En esta práctica utilizaremos CUDA, una arquitectura de software y hardware que permite a GPUs ejecutar programas escritos en diferentes lenguajes, un programa híbrido que ejecuta código secuencial (CPU) y en paralelo (GPU). Aprovecharemos los recursos que nos ofrece para investigar su potencial y comparar el rendimiento en problemas que tratamos anteriormente en otras arquitecturas.

Ejercicios

1.2. Convolución de señales unidimensionales

La convolución la hemos realizado tomando dos matrices de entrada, `X` como matriz de entrada y `K` como matriz de núcleo convolucional. Estas dos generan una matriz `Y` como resultado de la operación indicada en el enunciado. Tomamos un tamaño `M` para `K`, siendo este menor que `N`, tamaño de `X`.
La implementación que hemos realizado ha sido con memoria compartida dentro de cada bloque, intentando que sea más rápida de leer. Para ello, creamos la matriz de memoria compartida (`x_s`) con `__shared__` y la actualizamos con los elementos de entrada para cada bloque. Para guardar el instante actual del bloque actual y del posterior utilizamos dos variables, `actual` y `siguiente`. Estas variables han sido relevantes para poder verificar si un elemento de `X` se encuentra dentro del bloque actual para poder recuperar la información de la memoria compartida. Lo único malo que hemos visto es que la memoria compartida sólo está a nivel de bloque, por lo que si los elementos que se encuentran en el principio/final de los bloques tienen vecinos en otros bloques, esos elementos no se pueden explotar tanto y tal vez habría que utilizar un enfoque global.
Tras comprobar los resultados, podemos determinar una serie de situaciones idóneas para utilizar esta tecnología:

- Para algoritmos con orden de ejecución cuadrático.
- Existe poca independencia de datos o secciones críticas.
- Hay gran carga de cálculo por thread.

Sabiendo que es verdad que no para todos los problemas se ha de usar esta tecnología, en este caso ha sido muy interesante dado que la versión serie tiene *O(MxN)*, requiriendo dos bucles, además de que la convolución sirve para muchas aplicaciones, como por ejemplo, en redes neuronales convolucionales. Con este planteamiento hemos conseguido un tiempo de ejecución de **0.105 ms**. Sin embargo, en este caso tan sencillo, no es necesario utilizar esta tecnología, pero útil para practicar para futuros problemas mayores.

Adjuntamos la versión serie, `convolucion_secuencial.c`, que tiene un tiempo de ejecución con $N = 10$ y $M = 5$ de 0.00100 ms en la misma máquina.

1.3. Estimar el valor de Pi a partir de números aleatorios

En este ejercicio hemos vuelto a aproximar pi por Monte Carlo que explicamos previamente en anteriores prácticas. Aprovechando los recursos de CUDA, esta vez hemos utilizado 200 hilos por bloque y una elección de `1e8` muestras, obteniendo un tiempo de ejecución de **0.122 ms**, tiempo mucho mejor que la versión serie que adjuntamos, `monte_carlo.c`, que tarda **11.835 ms** en la misma máquina.

1.4. Multiplicación de Matrices (NxN)

Para la realización de este ejercicio se ha realizado sin pedir el tamaño de la matriz por terminal, por comodidad se ha definido una macro para ajustar dicho tamaño. Para el ajuste de las variables **blocksPerGrid y threadsPerBlock**, se ha hecho en caso de ser el tamaño de la matriz menor que 1024, que se cree que es el número máximo de threads por bloque, el suelo de la raíz cuadrada del tamaño de la matriz más 1, esto es para coger el techo del posible número que haya dado como resultado.En caso contrario se utiliza 32.

Se utiliza **dim3** para inicializar el número de bloques y el número de threads por bloque. Al utilizar más de una dimensión, los valores se separan por comas. Quedando el número de bloques por grid, como el número de bloques en la dimensión x por el número de bloques de la dimensión y. Esto se aplica también para los threads.

Al realizar el cálculo con un tamaño de matriz de 400. Se obtiene que tarda tan sólo 0.000021 s, quedando un rendimiento de 38.000.

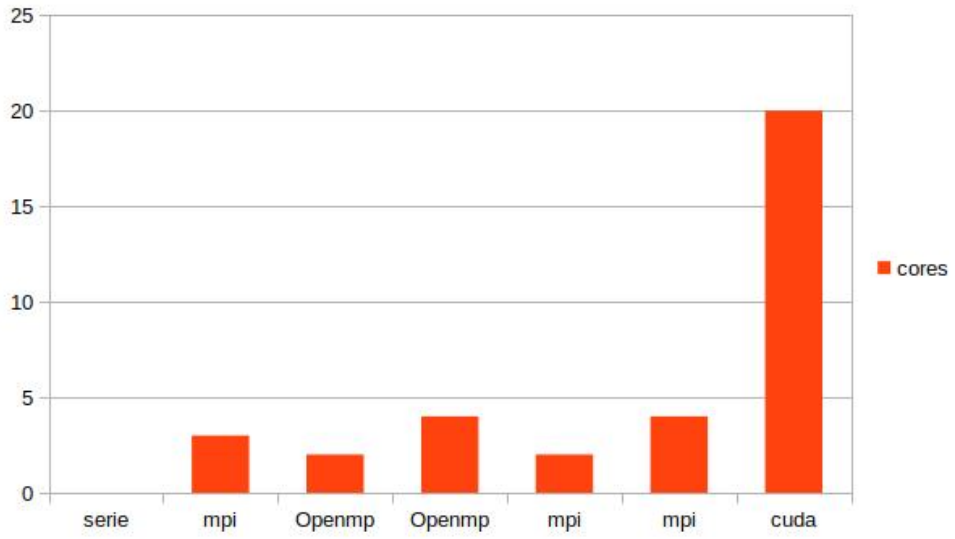

Figura 1: Unidades procesamiento (Multiplicación de matrices)

En la figura superior se puede que al utilizar una GPU se pueden hacer uso de un mayor número de unidades de cálculo, ya que para mpi depende del número de núcleos del equipo, que al igual que en open mp estas unidades de trabajo son inferiores a las de una GPU.

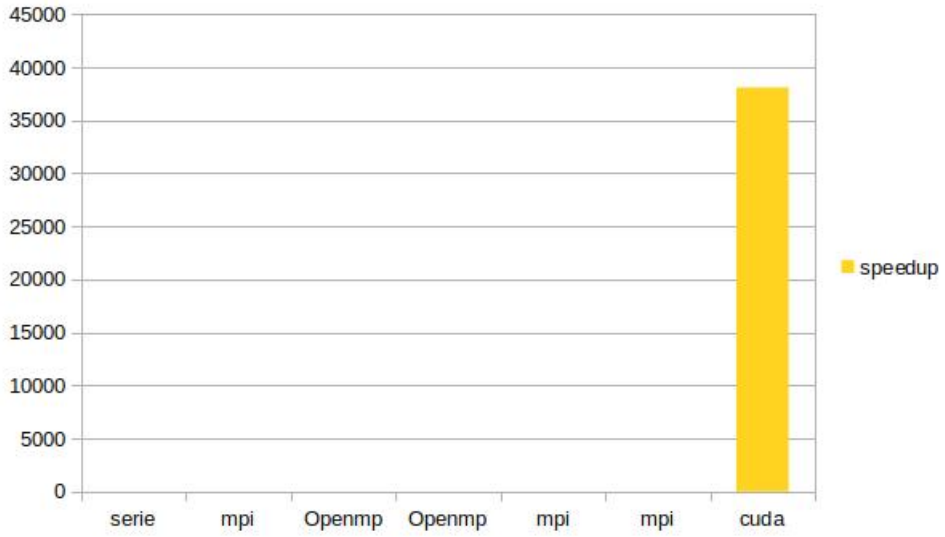

Figura 2: Speed Up (Multiplicación de matrices)

Como se puede observar el speed up que tiene una GPU es muchísimo mayor que cualquiera de las otras tecnologías. Ya que el speedup con `CUDA` da un valor de **38.000**.