Metodología de la Programación Paralela
Curso 2023/24

Práctica 4:

Introducción a la Programación de GPUs con CUDA

En esta práctica se presentan los conceptos básicos necesarios para analizar y comprender cómo se lleva a cabo la paralelización de aplicaciones en GPU mediante el modelo de programación definido por CUDA, poniendo de manifiesto las ventajas e inconvenientes que conlleva esta tarea. Inicialmente, se analizarán algunos ejemplos que ilustran el uso de los tipos de datos, las funciones de librería y las directivas para la programación con CUDA. A continuación, se propondrán algunos ejercicios sencillos con el fin de asimilar los diferentes conceptos estudiados.

Esta práctica representa el 10% de la calificación de la asignatura.

La fecha de entrega de la práctica será el **15 de diciembre a las 23:55**. Entregas posteriores no se tendrán en cuenta. La entrega se realizará a través de una tarea habilitada en el Aula Virtual, adjuntando el código fuente modificado para resolver las diferentes cuestiones y un documento en formato pdf que describa cómo se han resuelto, incluyendo las partes del código modificadas en su caso. Asimismo, el documento ha de incluir una sección en la que se indique cómo ha sido la coordinación, el reparto del trabajo entre los miembros del grupo y el tiempo dedicado.

Para realizar la práctica se utilizará la GPU de los equipos del Laboratorio 2.1: Geforce GT 1030. Opcionalmente, los alumnos podrán usar cualquier otra GPU de NVIDIA presente en sus portátiles, especificando el modelo utilizado. Se puede descargar CUDA desde la página oficial de NVIDIA: https://developer.nvidia.com/cuda-downloads.
Guía de programación: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

A continuación, se muestra una breve descripción de los ejemplos de código que se proporcionan, que deberán ser analizados antes de resolver cada una de las cuestiones propuestas:

1. *deviceQuery*: proporciona información de la GPU presente en el sistema.
2. *bandwidthTest*: microbenchmark para evaluar el rendimiento de las transferencias de datos entre la memoria del host (CPU) y la memoria del dispositivo (GPU).
3. *cudaTemplate*: proporciona un *kernel* que crea bloques de threads que se sincronizan entre sí.
4. *vectorAdd*: suma dos vectores.

NOTA: en los ejemplos indicados no se comprueba que las dimensiones del grid y de los bloques de threads no superan las dimensiones máximas permitidas.

Se facilitan los ficheros *Makefile* y *run.sh* dentro de la carpeta de cada código de ejemplo para compilar y realizar las ejecuciones de cada uno de ellos. Los *Makefile* incluyen las variables y configuración mínima necesarias para compilar en los equipos del Lab. 2.1, con el sistema operativo Ubuntu 22.04 y CUDA 12.1. El directorio *Common* incluye algunas funciones necesarias.

Además, la carpeta *Anexos* contiene dos versiones extendidas de los *Makefile* que permiten la configuración automática y específica para diferentes sistemas operativos (Linux o Mac OS X) y arquitecturas, con distintos argumentos que se pueden modificar para compilar y ejecutar correctamente en el sistema concreto utilizado.

Cuestión 1. Ejecuta el programa *deviceQuery* y obtén la siguiente información para la GPU:

- Número de Streaming Multiprocessors (SMP).
- Número de Streaming Processors (SP).
- Total de CUDA Cores.
- Número Máximo de Threads por SMP.
- Número Máximo de Threads por Bloque.
- Cantidad de Memoria Global.
- Registros Disponibles por Bloque.
- Cantidad de Memoria Compartida por Bloque.
- Dimensiones Máximas del Grid.
- Dimensiones Máximas del Bloque de Threads.

Cuestión 2. Utilizando el programa *bandwidthTest*:

a) Representa en una gráfica el ancho de banda de las transferencias entre el host y el dispositivo, dentro del dispositivo, y entre el dispositivo y el host, con bloques de memoria desde 1 KB hasta 64 MB (en incrementos de 512 KB) cuando se usa "*pageable memory*".
b) Repite el experimento haciendo uso de "*pinned memory*" y explica las diferencias obtenidas.

Cuestión 3. Ejecuta el programa *cudaTemplate* con diferentes tamaños de grid y bloques de threads.

a) ¿Qué hace el código del kernel?
b) Analiza el uso que se hace de la memoria compartida en el código del kernel y verifica si el tamaño indicado es correcto. Si no lo fuese, explica cómo hay que modificar el código.
c) ¿Podría eliminarse la primera llamada a `__syncthreads()` en el kernel? ¿Y la segunda?
d) Modifica el código para que el kernel utilice bloques de threads tridimensionales, es decir, para que tenga en cuenta la coordenada z.

Cuestión 4. Ejecuta el programa *vectorAdd* con diferentes tamaños de vector y threads por bloque.

a) Anota los tiempos de ejecución obtenidos y analiza la influencia que tiene variar el número de threads por bloque para cada tamaño de problema.
b) Analiza el código del kernel y explica cómo se produce la suma de los vectores.
c) Analiza el código que calcula el número de bloques del grid. ¿Qué sucede si el número total de elementos del vector no es múltiplo del tamaño del bloque de threads?
d) Explica cómo puede afectar al rendimiento que el bloque de threads no sea potencia de 2. Para ello, realiza una breve búsqueda bibliográfica sobre el concepto de *warp*, la unidad de planificación de hilos de CUDA.