

Programación de Arquitecturas Emergentes [G4012452] [2024/2025]

Lab 2 - Programación básica y Programación de algoritmos paralelos con CUDA

Resumen

El objetivo de esta práctica es entender la arquitectura de GPUs de NVIDIA y el manejo básico de la API, y aplicar las metodologías de programación paralela vistas en clase utilizando CUDA.

1. Lab 2.1 - Compilar y ejecutar programas

Al conectar al CESGA, tenemos cargado el módulo de CUDA. Para compilar un código escrito en lenguaje C con la API de CUDA usando el compilador nvcc:

```
1 nvcc 00_dev_query.cu -o 00_dev_query -allow-unsupported-compiler
```

Las últimas versiones de CUDA no soportan un compilador GNU anterior a la versión 9. La opción `'-allow-unsupported-compiler'` es necesaria para compilar código en el CESGA, al tener instalada una versión del compilador `gcc > 9`. El envío de trabajos sigue siendo mediante un sistema de colas, por ejemplo:

```
1 #!/bin/bash
2 #SBATCH -N 1                # 1 nodo en total
3 #SBATCH -c 32               # 1 core por tarea
4 #SBATCH -t 00:02:00         # Run time (hh:mm:ss) - 2 min
5 #SBATCH --mem-per-cpu=100M  # Memory per core demandes
6 #SBATCH --gres=gpu:a100:1   # Request 1 GPU of 2 available on an average A100 node
7 # Programa a ejecutar
8 ./00_dev_query
```

📖 Durante el desarrollo usa un nodo de forma interactiva: `compute --gpu`.

Revisa la sección *Using sbatch and GPUs y AI nodes (GPU nodes)* de la documentación técnica del CESGA. Los enlaces a dichas secciones están disponibles en el campus virtual.

2. Lab 2.1 - Bloque 1: Programación básica con CUDA

En el campus virtual tienes un archivo `lab2.1-code.zip` con el material para trabajar en este bloque.

1. Compila y ejecuta el código `00_dev_query.cu`.

📖 ¿Cuál es el número máximo de hilos que puede ejecutar la GPU de forma simultánea?.

2. Completa el código del archivo `01_cudaMemcpy.cu` para copiar datos entre CPU y GPU.

3. Compila y ejecuta el archivo *02_cudaErrorchk.cu*, y comprueba la salida. Añade la función `checkCUDAError()` para comprobar errores y corrige el código.
4. La función para comprobar errores del apartado anterior lo hace sobre llamadas a la API de CUDA en tiempo de ejecución.
 - Compila y ejecuta el archivo *03_out_of_bounds.cu*, y comprueba la salida.
 - Comprueba los errores desde la terminal con `cuda-memcheck 03_out_of_bounds` y corrige el código.
5. Comprueba que la memoria unificada es accesible desde cualquier procesador del sistema.
 - Modifica el archivo *04_unified_memory.cu* para copiar datos entre host y device.
6. Implementa en un kernel la suma de 2 vectores de números en simple precisión de tamaño $n=262144$.
 - Inicializa los vectores en el host (CPU) de forma secuencial.
 - Reserva la memoria en GPU y copia los datos de forma explícita como en el ejercicio 1.
7. *Thrust* es una librería basada en la STL (Standard Template Library) de C++, permite implementar aplicaciones paralelas con un alto rendimiento con mínimo esfuerzo y es totalmente interoperable con CUDA C. Estudia el código *06_thrust_vec_add.cu* y modifícalo para comprobar la interoperabilidad. Prueba lo siguiente:
 - Cambia la llamada al kernel `thrust::transform` y añade el kernel del apartado 4.
 - Cambia la copia de datos implícita a la GPU en las asignaciones de `d_A` y `d_B` por copias de datos explícitas usando `cudaMemcpy()`.
8. Implementa en un kernel la suma de 2 vectores usando directivas del compilador `openacc`.

Los siguientes ejercicios forman parte de la evaluación del Bloque 1: Programación básica con CUDA

9. Inicializa una matriz con números secuenciales de la forma `fila*N+columna` de tamaño $M \times N = 1\text{GiB}$ repartiendo el trabajo entre bloques de hilos:
 - El código del kernel debe ser el mismo para cualquier tamaño de bloque. Prueba con tamaños de bloque diferentes, por ejemplo, bloques del *tamaño de un warp*, y bloques cuadrados del *máximo tamaño* que te permita la arquitectura.
 - ¿Para qué tamaños de bloque alcanzas la máxima ocupancia?.
10. Implementa un kernel para invertir las posiciones de los valores de un array `x` de tamaño 8192.
 - Inicializa el array de la forma `x[i] = i` en GPU en un kernel diferente.
 - Usa un array de entrada `x[]` y un array de salida `y[]`.
 - Al invertir las posiciones, el elemento de la posición `x[0]`, pasará a la posición `y[8191]`, el elemento en la posición `x[1]`, a la posición `y[8190]`, ...
 - Comprueba en el host (CPU) que el resultado es correcto al terminar los cálculos en la GPU.

3. Lab 2.2 - Bloque 2: Programación de algoritmos paralelos con CUDA


1. Implementa el **algoritmo de DAXPY** usando usando memoria unificada.
2. Implementa el **algoritmo del histograma** usando memoria global y memoria compartida (dos versiones).
3. Implementa el **algoritmo de convolución** usando memoria global y la librería **NVIDIA 2D Image and Signal Processing Performance Primitives (NPP)**. Tienes información adicional en el Anexo I.

4. Especificaciones

1. Usa un nodo interactivo durante el desarrollo y calcula el tiempo de ejecución (**wall time**) como una media de 10 ejecuciones usando un **nodo A100**¹ cuando el programa esté preparado y libre de errores.
2. Compila los programas con optimizaciones **-O2** del compilador y calcula el speedup respecto a la mejor versión secuencial. Ver *Hall of Fame* disponible en el campus virtual.
3. Separa en el tiempo de ejecución todo el overhead relacionado con gestión de memoria en CPU y GPU (malloc e inicialización) que necesites para resolver el problema de forma paralela.
4. Explora diferentes tamaños de bloque incluso cuando la ocupancia sea la misma.
5. Comprueba que las implementaciones paralelas son correctas comparando los resultados con la versión secuencial.

5. Formato y fecha de entrega

1. Sube al campus virtual un archivo comprimido **lab2.1.zip** sólo con el código del ejercicio 9 y 10 del Lab 2.1. Antes de subirlo, confirma con el profesor que cumples los requisitos.
2. Sube al campus virtual en otro archivo comprimido **lab2.2.zip** los ejercicios del Lab 2.2 junto con un informe breve y detallado en formato pdf.
 - a) El código no puede tener comentarios.
 - b) El informe debe presentar los resultados de forma clara y resumida mediante gráficas que muestren la aceleración global incluyendo transferencia de datos inicial, y sólo de los kernels.
 - c) El informe no tiene que explicar el código implementado pero si cuestiones relacionadas con la arquitectura, como configuración de bloques e hilos, límites de la capacidad computacional, discusión de los resultados, observaciones, conclusiones propias y toma de decisiones.
 - d) Los datos numéricos deben guardarse por si fuera necesario una consulta por parte del profesor.

 **Fecha límite de entrega:** 24 de marzo de 2025, a las 9:00 horas.

¹Repasar **sección GPU Nodes** en la guía de utilización del FinisTerra3 (Minimum cpus requested should be 32 per demanded a100 gpu).

6. Evaluación

1. La evaluación se hará en base a las implementaciones paralelas, uso correcto de los recursos en la GPU, y a la calidad y defensa del informe.
2. Esta práctica tiene un **peso de 2.5 puntos en la nota final** y se evalúa sobre 10 puntos (5 puntos código, 5 puntos informe).

ANEXO I

Para emplear la librería NPP, es necesario incluir `nppi.h` y compilar usando las opciones:

- `-lnppc -lnppif`

,La documentación sobre la convolución en CUDA con NPP está accesible en;

- https://docs.nvidia.com/cuda/npp/image_filtering_functions.html#image-convolution