Investigación sobre CUDA

Ignacio Grané Rojas *Escuela de Ingeniería en Computadores*
Tecnológico de Costa Rica
Cartago, Costa Rica
ignaciograne@estudiantec.cr
Carlos Andrés Mata Calderón *Escuela de Ingeniería en Computadores*
Tecnológico de Costa Rica
Cartago, Costa Rica
carlos.andres12001@estudiantec.cr
Luis Felipe Vargas Jimenez *Escuela de Ingeniería en Computadores*
Tecnológico de Costa Rica
Cartago, Costa Rica
felipevargas13@estudiantec.cr
Harold Jose Espinoza Matarrita *Escuela de Ingeniería en Computadores*
Tecnológico de Costa Rica
Cartago, Costa Rica
hrldspnz@estudiantec.cr
Jonathan Daniel Gonzalez Sanchez *Escuela de Ingeniería en Computadores*
Tecnológico de Costa Rica
Cartago, Costa Rica
jongs@estudiantec.cr

***Resumen*—Este informe aborda la tecnología CUDA (Compute Unified Device Architecture) de NVIDIA, que permite el desarrollo de aplicaciones de alto rendimiento utilizando unidades de procesamiento gráfico (GPU). Se analiza el concepto de CUDA, la definición y uso de kernels, la asignación de hilos y bloques, y la arquitectura de la plataforma Jetson Nano. Además, se discuten los pasos para compilar un código CUDA y se realiza un análisis de un kernel básico para operaciones en vectores. Este estudio es esencial para entender el potencial del paralelismo en la computación moderna.**

I. Introducción

CUDA, o Compute Unified Device Architecture, es una plataforma de computación paralela y un modelo de programación creado por NVIDIA que permite a los desarrolladores utilizar la potencia de las GPUs (unidades de procesamiento gráfico) para tareas de computación general (GPGPU). Este informe tiene como objetivo responder a una serie de preguntas clave sobre CUDA, incluyendo su definición, la manera en que se manejan los hilos y bloques, detalles sobre la compilación de códigos CUDA, y la arquitectura de la plataforma Jetson Nano.

II. Desarrollo

¿Qué es CUDA?

CUDA es una arquitectura de computación paralela y un entorno de desarrollo proporcionado por NVIDIA que permite el uso eficiente de GPUs para la ejecución de cálculos en paralelo a gran escala. En términos técnicos, CUDA ofrece una abstracción de hardware que permite a los desarrolladores escribir código en C, C++, o Fortran, que luego es compilado y ejecutado en las GPUs. Este modelo permite ejecutar miles de hilos concurrentes, distribuyendo automáticamente las tareas a través de los múltiples núcleos de la GPU, optimizando así el rendimiento en aplicaciones intensivas en cómputo. CUDA también incluye bibliotecas especializadas para tareas como álgebra lineal, transformadas de Fourier y procesamiento de imágenes, lo que facilita el desarrollo de aplicaciones científicas y de ingeniería de alto rendimiento [1], [2].

¿Qué es un kernel en CUDA y cómo se define?

Un kernel en CUDA es una función especial que se ejecuta de manera masivamente paralela en la GPU, permitiendo el procesamiento simultáneo de miles de hilos. Desde un punto de vista arquitectónico, un kernel se define utilizando la palabra clave `__global__` y es invocado desde el host (CPU) pero ejecutado en el dispositivo (GPU). Cada invocación del kernel lanza una cantidad masiva de hilos organizados en bloques, que a su vez están organizados en una cuadrícula (grid). Estos hilos ejecutan la misma instrucción en diferentes datos, aprovechando el modelo SIMD (Single Instruction, Multiple Data) de la GPU. La definición de un kernel incluye la especificación del número de hilos y bloques, optimizando la ejecución en función de la arquitectura subyacente de la GPU [1], [3].

¿Cómo se maneja el trabajo a procesar en CUDA? ¿Cómo se asignan los hilos y bloques?

En CUDA, el trabajo se descompone en pequeños subprocesos que se ejecutan en paralelo. La arquitectura de CUDA

utiliza un modelo jerárquico donde los hilos son organizados en bloques, y estos bloques se organizan en una cuadrícula (grid). Cada hilo tiene un identificador único dentro de su bloque y cada bloque tiene un identificador dentro de la cuadrícula. La asignación de trabajo en CUDA se realiza a través de un modelo de programación explícito donde el desarrollador define la cantidad de hilos por bloque y bloques por cuadrícula al invocar un kernel. El tamaño y la organización de hilos y bloques se eligen para maximizar el uso de los recursos de la GPU, como la memoria compartida y los registros, minimizando la latencia de acceso a la memoria global. Este modelo permite aprovechar al máximo el paralelismo masivo inherente a las GPUs, escalando eficientemente las aplicaciones con un gran número de operaciones concurrentes [3], [4].

Arquitectura de la plataforma Jetson Nano

La Jetson Nano es una plataforma de desarrollo de bajo costo creada por NVIDIA, diseñada para aplicaciones de inteligencia artificial y computación en el borde. A nivel de hardware, la Jetson Nano integra un CPU ARM Cortex-A57 de cuatro núcleos y una GPU NVIDIA Maxwell con 128 núcleos CUDA, lo que permite la ejecución de aplicaciones CUDA en un entorno compacto y de bajo consumo. La arquitectura de la Jetson Nano soporta hasta 472 GFLOPS de rendimiento, lo que la hace ideal para aplicaciones en tiempo real como visión por computadora, robótica, y procesamiento de señales. La combinación de CPU y GPU en un solo módulo permite a los desarrolladores implementar soluciones que requieren tanto procesamiento secuencial como paralelo, optimizando el rendimiento en aplicaciones que demandan alta capacidad de cómputo en entornos restringidos en recursos [3], [5].

Compilación de un código CUDA

Para compilar un código CUDA, se utiliza el compilador `nvcc` (NVIDIA CUDA Compiler), que es parte del CUDA Toolkit. El proceso de compilación implica la traducción del código escrito en C o C++ que incluye funciones CUDA (kernels) en un binario ejecutable que puede ser ejecutado en la GPU. El comando básico es:

```
nvcc NOMBRE.cu -o NOMBREoutput
```

donde `NOMBREarchivo.cu` es el archivo fuente que contiene el código CUDA. Durante la compilación, `nvcc` realiza varias fases de optimización, incluyendo la generación de código intermedio (PTX) y la posterior compilación hacia código máquina específico para la arquitectura de la GPU de destino. Este proceso asegura que el código esté altamente optimizado para la ejecución en paralelo, maximizando el uso de los recursos disponibles en la GPU [3], [6].

III. Análisis

Operación realizada con los vectores de entrada

A grandes rasgos, la aplicación ejecuta una **suma elemento a elemento** entre dos vectores de números enteros. Es decir, para cada posición i, se calcula la siguiente operación:

$$c[i] = a[i] + b[i]$$

Esta suma se lleva a cabo tanto en la GPU, utilizando el kernel `vecAdd`, como en la CPU, mediante la función `vecAdd_h`. El objetivo es combinar cada elemento del vector `a` con su correspondiente en el vector `b`, almacenando el resultado en los vectores `c` (para la GPU) y `c2` (para la CPU).

Identificación y procesamiento paralelo de cada elemento

En cuanto a la identificación de cada elemento a ser procesado en paralelo, la aplicación hace uso de las variables de índice proporcionadas por CUDA. Cada hilo en la GPU calcula su índice global i utilizando la siguiente fórmula:

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
```

Donde:

- `blockIdx.x`: Índice del bloque actual en la dimensión `x`.
- `blockDim.x`: Número de hilos por bloque en la dimensión `x`.
- `threadIdx.x`: Índice del hilo actual dentro de su bloque en la dimensión `x`.

De esta manera, cada hilo obtiene una posición única que corresponde a un elemento específico en los vectores de entrada y salida. El procesamiento paralelo se realiza siguiendo estos pasos:

1. **Asignación de Hilos**: Se lanzan múltiples bloques (`block_no`), cada uno con un número determinado de hilos (`block_size`).
2. **Cálculo del Índice Global**: Cada hilo calcula su índice global i, que identifica su posición en los vectores.
3. **Procesamiento Independiente**: Cada hilo ejecuta la operación `C[i] = A[i] + B[i];` de forma independiente, procesando su elemento correspondiente.

Este enfoque permite aprovechar la arquitectura masivamente paralela de las GPUs, donde miles de hilos pueden ejecutarse simultáneamente. Así, se logra acelerar significativamente el cálculo en comparación con una ejecución secuencial en la CPU.

Funcionalidad final de la aplicación

La aplicación lleva a cabo los siguientes pasos:

1. **Inicialización y Llenado de Vectores**:
   - Se crean y se llenan dos vectores de entrada, `a` y `b`, con valores enteros secuenciales.
2. **Ejecución de la Suma en la GPU**:
   - **Asignación de Memoria**: Se reserva memoria en la GPU para los vectores `a_d`, `b_d` y `c_d`.
   - **Transferencia de Datos**: Los vectores `a` y `b` se copian desde la memoria del host a la memoria de la GPU.
   - **Lanzamiento del Kernel**: Se ejecuta el kernel `vecAdd` para realizar la suma en paralelo.
   - **Sincronización**: Se espera a que todos los hilos finalicen su ejecución.

- **Recuperación de Resultados**: El vector resultado `c_d` se copia de vuelta a la memoria del host en `c`.

3. **Ejecución de la Suma en la CPU**:
   - Se ejecuta la función `vecAdd_h` para realizar la suma de forma secuencial, almacenando el resultado en `c2`.
4. **Medición y Comparación de Tiempos**:
   - Se miden los tiempos de ejecución tanto en la GPU como en la CPU.
   - Se imprimen los resultados para comparar el rendimiento.
5. **Liberación de Memoria**:
   - Se libera la memoria asignada en la GPU para evitar fugas de memoria.

La aplicación tiene como propósito **demostrar y comparar el rendimiento** entre la ejecución paralela en GPU y la ejecución secuencial en CPU para una operación de suma vectorial sobre un gran conjunto de datos. Esto permite ilustrar las ventajas del cómputo paralelo utilizando GPUs en tareas que requieren un alto desempeño computacional.

Cambie la cantidad de hilos por bloque y el tamaño del vector. Compare el desempeño antes al menos 5 casos diferentes.

Tabla I
RESULTADOS DE LA EJECUCIÓN DE PRUEBAS DE SUMA VECTORIAL EN GPU Y CPU

Num	Bloque	Vector	Tiempo GPU (ms)	Tiempo CPU (ms)
1	32	1,024	21.463	0.094
2	32	1,048,576	2.975	5.440
3	128	262,144	0.980	1.402
4	512	262,144	0.947	2.749
5	512	1,048,576	2.823	5.945

1. **Prueba 1:** Pequeño número de hilos por bloque con tamaño de vector pequeño.
2. **Prueba 2:** Pequeño número de hilos por bloque con tamaño de vector grande.
3. **Prueba 3:** Número mediano de hilos por bloque con tamaño de vector mediano.
4. **Prueba 4:** Gran número de hilos por bloque con tamaño de vector mediano.
5. **Prueba 5:** Gran número de hilos por bloque con tamaño de vector grande.

Noteemos que al aumentar los hilos por bloque, los tiempos en la GPU mejoran, mostrando la eficacia de la paralelización masiva, especialmente con vectores grandes. No obstante, los tiempos en la CPU aumentan con el tamaño del vector, reflejando sus limitaciones para tareas grandes.

Además, entre las pruebas 3 y 4, el pequeño cambio en tiempos de GPU sugiere que añadir más hilos a partir de cierto punto ofrece beneficios marginales. Así, Ousterhout nos recordaría la importancia de optimizar el uso de recursos para maximizar la eficiencia sin desperdiciar capacidad computacional.

IV. EJERCICIO PRÁCTICO

Para los ejercicios prácticos se hizo uso de una tarjeta de video Nvidia RTX 3050 Mobile para el caso de los ejemplos en CUDA, y de un Raspberr Pi para los ejemplos de Neo ARM. Para la carga de imágenes a bytes en los ejemplos CUDA, se utilizó la librería proporcionada por el usuario lukas783 [7]

Figura 1. Serie de imágenes de ejemplo.

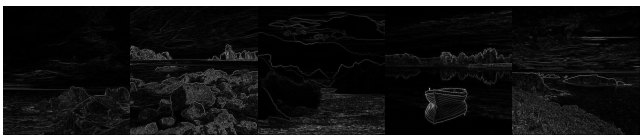

Figura 2. Serie de resultados.

Tabla II
COMPARACIÓN DE RENDIMIENTO CON Y SIN PARALELISMO Y EN ARM NEON

Método	**Tiempo (ms)**				
	Img1	**Img2**	**Img3**	**Img4**	**Img5**
Sin paralelismo	3.442	3.407	3.425	3.336	3.336
Con paralelismo	16.260	0.165	0.157	0.153	0.209
Con Neon ARM	1.285	1.610	3.729	3.586	0.419

V. CONCLUSIONES

CUDA es una herramienta poderosa para desarrollar aplicaciones que requieren un alto rendimiento computacional. La capacidad de las GPUs para procesar tareas en paralelo permite una mejora significativa en el rendimiento de aplicaciones científicas, gráficas, y de inteligencia artificial. La comprensión de los conceptos básicos como kernels, hilos, y bloques es esencial para aprovechar al máximo esta tecnología.

REFERENCIAS

[1] D. B. Kirk y W.-m. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2017.

[2] J. Nickolls, I. Buck, M. Garland y K. Skadron, «Scalable Parallel Programming with CUDA,» *ACM Queue*, vol. 6, n.º 2, págs. 40-53, 2008.

[3] NVIDIA Corporation, *CUDA C Programming Guide*, 2023. dirección: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[4] S. Cook, *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Newnes, 2013.

[5] M. Jung, Y. Kwon, K. Park y J. Lee, «Real-time Object Detection System with Jetson Nano,» en *IEEE International Conference on Consumer Electronics (ICCE)*, 2019, págs. 1-2.

[6] H. Cheng, *CUDA for Engineers: An Introduction to High-Performance Parallel Computing*. Addison-Wesley, 2014.

[7] A. Name(s), *Sobel Edge Detector*, Requires a Nvidia CUDA capable graphics card and the Nvidia GPU Computing Toolkit. Compiling: nvcc -Wno-deprecated-gpu-targets -O3 -o edge sobelFilter.cu lodepng.cpp -std=c++11 -Xcompiler -fopenmp. Usage: edge [filename.png]. Description: This program runs a sobel filter using three different methods: a basic CPU single thread, parallelized with OpenMP, and through a NVIDIA GPU. It compares performance across these methods, typically finding that GPU execution is the fastest., oct. de 2017. dirección: https://github.com/lukas783/CUDA-Sobel-Filter.