



Instituto Tecnológico de Costa Rica

Escuela de Ingeniería en Computadores

Arquitectura de computadores II

Taller CUDA

Axel Cordero Martínez | 2019052017

Kevin Guzmán Navarro | 2017238886

Jose Ignacio Calderón Díaz | 2019031750

Erick Josue Barrantes Cerdas | 2017125821

Luis Alonso Barboza Artavia

II Semestre 2024

## Investigación:

### 1. ¿Qué es CUDA?

CUDA (Compute Unified Device Architecture) es una plataforma de computación paralela y un modelo de programación desarrollado por NVIDIA. Permite a los desarrolladores utilizar la GPU (Unidad de Procesamiento Gráfico) para realizar tareas de cómputo general (GPGPU), lo que significa que las GPUs, tradicionalmente utilizadas para renderizado gráfico, pueden ser usadas para realizar cálculos matemáticos y científicos intensivos en paralelo. CUDA facilita la programación en lenguajes como C, C++, y Python, extendiéndolos con un conjunto de instrucciones específicas para aprovechar el poder de procesamiento masivo de las GPUs.

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

Un *kernel* en CUDA es una función que se ejecuta en la GPU. En un programa CUDA, el código que se ejecuta en la CPU (host) y el que se ejecuta en la GPU (dispositivo) se distingue claramente. Los *kernels* son funciones que se ejecutan en paralelo en múltiples hilos sobre la GPU. Se definen utilizando la palabra clave `__global__` y se lanzan desde el código del host con una sintaxis especial que indica cuántos bloques y cuántos hilos por bloque se usarán.

Ejemplo de definición de un kernel:

```
1. __global__ void sumaVectorial(int *a, int *b, int *c, int n) {  
2.   int i = blockIdx.x * blockDim.x + threadIdx.x;  
3.   if (i < n)  
4.     c[i] = a[i] + b[i];  
5. }
```

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

En CUDA, el trabajo a procesar se divide en bloques de hilos (*threads*), y estos bloques se organizan en una cuadrícula (*grid*). Cada hilo dentro de un bloque tiene un identificador único, y cada bloque dentro de la cuadrícula también tiene un identificador único. Los bloques de hilos pueden ser unidimensionales, bidimensionales o tridimensionales, y lo mismo se aplica a los hilos dentro de un bloque.

La asignación de hilos y bloques se realiza al lanzar un kernel, donde el programador especifica la cantidad de bloques y la cantidad de hilos por bloque. La sintaxis para lanzar un kernel se ve así:

```
1. sumaVectorial<<<numBlocks, threadsPerBlock>>>>(a, b, c, n);
```

Aquí, numBlocks es el número de bloques que se utilizarán, y threadsPerBlock es el número de hilos por bloque. CUDA se encarga de distribuir estos hilos y bloques en los multiprocesadores de la GPU.

#### 4. Plataforma Jetson Nano: Arquitectura de hardware

La Jetson Nano es una plataforma de computación de alto rendimiento desarrollada por NVIDIA, diseñada para aplicaciones de inteligencia artificial (IA) y cómputo embebido. Está basada en la arquitectura ARM y utiliza una GPU con la arquitectura NVIDIA Maxwell.

- **GPU:** La Jetson Nano cuenta con una GPU NVIDIA Maxwell de 128 núcleos CUDA, lo que la hace capaz de realizar tareas intensivas en paralelo, como redes neuronales y procesamiento de imágenes.
- **CPU:** Integra una CPU ARM Cortex-A57 de cuatro núcleos a 1.43 GHz, que maneja tareas generales del sistema operativo y las aplicaciones.
- **Memoria:** La placa incluye 4 GB de memoria LPDDR4, lo que proporciona ancho de banda suficiente para manejar aplicaciones de IA.
- **Almacenamiento:** Soporta almacenamiento a través de microSD y dispone de interfaces para expansión como PCIe, I2C, y UART.
- **Conectividad:** Ofrece puertos USB 3.0, Ethernet, y GPIO, entre otros, facilitando la conexión con diversos periféricos y sensores.

#### 5. ¿Cómo se compila un código CUDA?

La compilación de un código CUDA se realiza utilizando el compilador nvcc, que es parte del toolkit de CUDA proporcionado por NVIDIA. nvcc compila el código CUDA y lo enlaza con las bibliotecas necesarias para que el programa pueda ejecutarse en la GPU.

Ejemplo básico de compilación:

```
1. nvcc -o sumaVectorial sumaVectorial.cu
```

Aquí, sumaVectorial.cu es el archivo fuente que contiene el código CUDA, y -o sumaVectorial especifica el nombre del archivo ejecutable resultante. Una vez compilado, el ejecutable se puede ejecutar en una máquina con GPU compatible con CUDA.

## Análisis:

### 1. Analice el código vecadd.cu.

El código está diseñado para sumar vectores en la CPU y la GPU, utilizando CUDA para paralelizar las tareas del GPU. Los vectores a,b,c y c2 se almacenan en la memoria del CPU y luego se asignan memorias a los vectores a\_d, b\_d, c\_d en la GPU. La suma de los vectores se hace por medio de bloques e hilos en CUDA. Se realiza la suma de los vectores tanto en el CPU como en el GPU, lo que permite comparar el tiempo de ejecución y se espera que la GPU sea más rápida cuando el tamaño del vector es grande. Se utiliza un sincronizador de hilos para asegurar que las operaciones del kernel en la GPU han terminado antes de continuar con las siguientes.

2. Analice el código fuente del kernel vecadd.cu. A partir del análisis del código, determine ¿Qué operación se realiza con los vectores de entrada? ¿Cómo se identifica cada elemento a ser procesado en paralelo y de qué forma se realiza el procesamiento paralelo?

Es la parte que se ejecuta en paralelo en el GPU, se usa en vecAdd en la siguiente operación.

$C[i] = A[i] + B[i];$

Cada elemento se identifica mediante el índice global i, y se calcula utilizando los índices del bloque "blockIdx.x" y del hilo "threadIdx.x"

$\text{int } i = \text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x};$

### 3. Realice la compilación del código vecadd.cu.

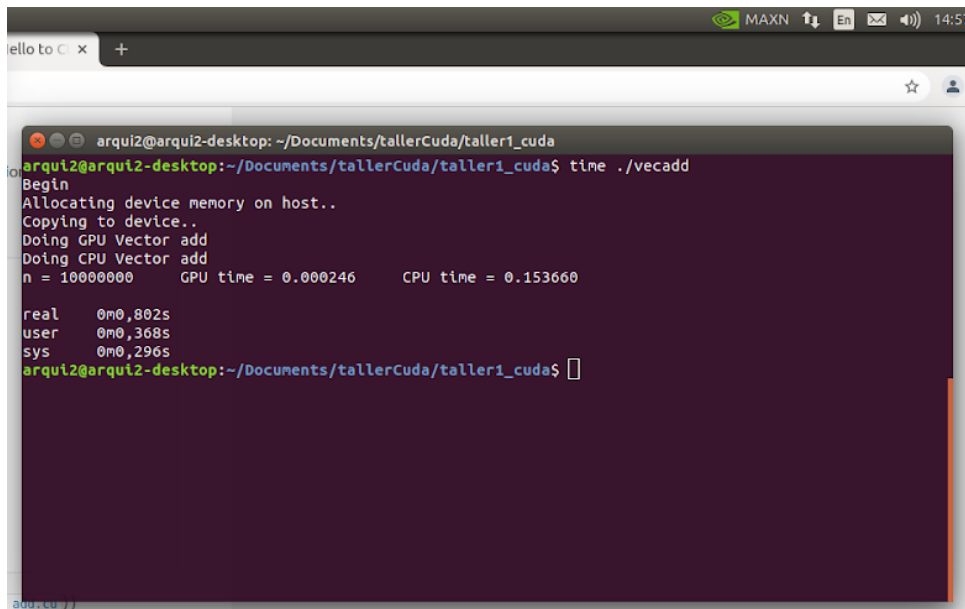
```
arquí2@arquí2-desktop: ~/Documents/tallerCuda/taller1_cuda
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
Unpacking objects: 100% (3/3), done.
arquí2@arquí2-desktop:~/Documents/tallerCuda$ cd taller1_cuda/
arquí2@arquí2-desktop:~/Documents/tallerCuda/taller1_cuda$ git status
On branch main
Your branch is up to date with 'origin/main'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        vecadd.cu

nothing added to commit but untracked files present (use "git add" to track)
arquí2@arquí2-desktop:~/Documents/tallerCuda/taller1_cuda$ nvcc vecadd.cu -o vecadd
nvcc fatal   : Unknown option '-o'
arquí2@arquí2-desktop:~/Documents/tallerCuda/taller1_cuda$ nvcc vecadd.cu -o vecadd
vecadd.cu: In function 'int main(int, char**)':
vecadd.cu:74:23: warning: 'cudaError_t cudaThreadSynchronize()' is deprecated [-Wdeprecated-declarations]
    cudaThreadSynchronize();
                        ^
/usr/local/cuda/bin/../targets/aarch64-linux/include/cuda_runtime_api.h:957:46: note: declared here
extern __CUDA_DEPRECATED __host__ cudaError_t CUDARTAPI cudaThreadSynchronize(void);
```

4. Realice la ejecución de la aplicación vecadd. ¿Qué hace finalmente la aplicación?



```
arquí2@arquí2-desktop: ~/Documents/tallerCuda/taller1_cuda
arquí2@arquí2-desktop:~/Documents/tallerCuda/taller1_cuda$ time ./vecadd
Begin
Allocating device memory on host..
Copying to device..
Doing GPU Vector add
Doing CPU Vector add
n = 10000000    GPU time = 0.000246    CPU time = 0.153660

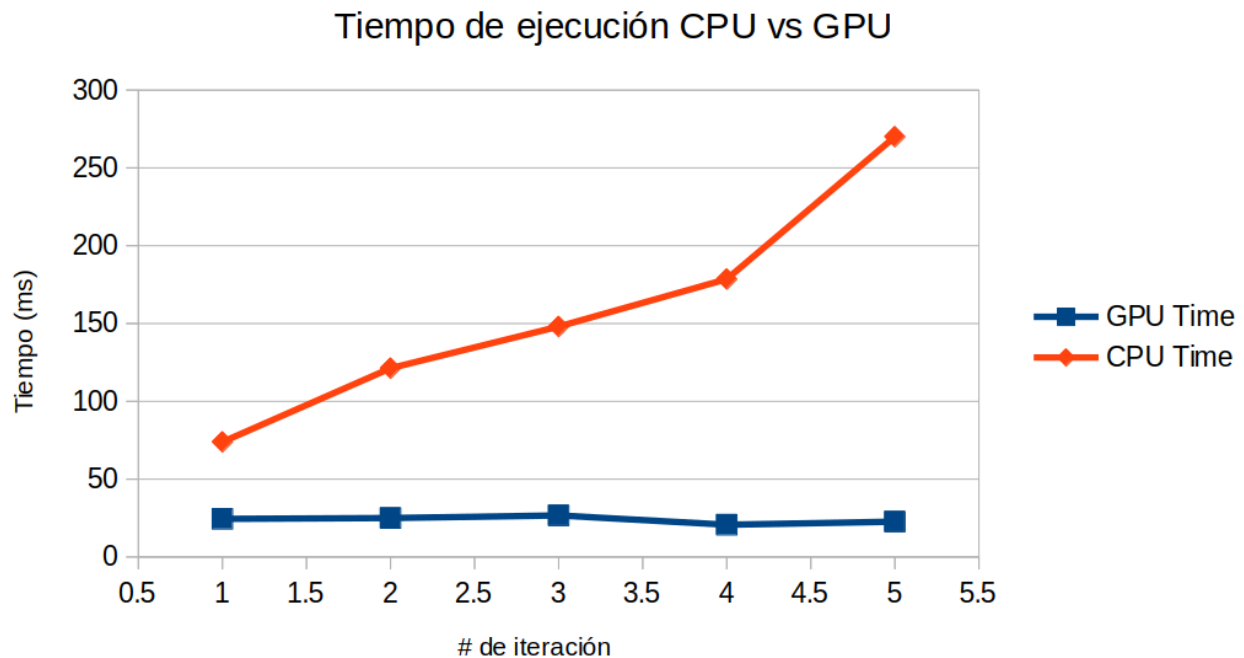
real    0m0.802s
user    0m0.368s
sys     0m0.296s
arquí2@arquí2-desktop:~/Documents/tallerCuda/taller1_cuda$
```

La aplicación realiza suma de matrices de manera escalar y vectorial, con un numero n de iteraciones. Se obtiene la comparativa entre tiempos de ejecución de la implementación escalar y vectorial.

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

Se realizo el análisis para los siguientes escenarios:

| Iteración | Hilos por bloque | Tamaño del vector |
|-----------|------------------|-------------------|
| 1         | 100              | 5000000           |
| 2         | 400              | 8000000           |
| 3         | 800              | 10000000          |
| 4         | 1000             | 12000000          |
| 5         | 100              | 18000000          |



En el grafico anterior se evidencia una mejora considerable en los tiempos de ejecución en la implementación paralelizada mediante CUDA.

## Ejercicios prácticos

1. Realice un programa que calcule el resultado de la multiplicación de dos matrices de 4x4, utilizando paralelismo con CUDA.

Para esta implementación se realiza un kernel de multiplicación para 2 matrices de la siguiente manera

```
1. __global__ void matrix_mul(int *a, int *b, int *c, int n) {  
2. int i = blockIdx.x * blockDim.x + threadIdx.x;  
3. int j = blockIdx.y * blockDim.y + threadIdx.y;  
4. int sum = 0;  
5. if (i < n && j < n) {  
6. for (int k = 0; k < n; k++)  
7. sum += a[i * n + k] * b[k * n + j];  
8. c[i * n + j] = sum;  
9. }  
10. }
```

Seguidamente, se mide el tiempo de ejecución de la multiplicación de dos matrices de 4x4. Se obtiene el resultado de la multiplicación y el tiempo de ejecución de manera paralelizada.

```

arqui2@arqui2-desktop:~/Documents/tallerCuda/taller1_cuda$ nvcc multiplicacionMat
t.cu -o multiplicacionMat
arqui2@arqui2-desktop:~/Documents/tallerCuda/taller1_cuda$ ./multiplicacionMat
0 14 28 42
0 20 40 60
0 26 52 78
0 32 64 96
N = 4 GPU time = 0.000060
arqui2@arqui2-desktop:~/Documents/tallerCuda/tal
ler1_cuda$

```

2. Implemente el filtro de imágenes Edge Detection de forma serial (sin paralelismo), utilizando CUDA, así como la extensión Neon ARM. Debe comparar las tres implementaciones con 5 imágenes diferentes. Se debe medir tiempos de ejecución.

### Primera Implementación en Python

Este código implementa la detección de bordes en una imagen utilizando el filtro de Sobel de manera serial. Primero, carga una imagen en escala de grises y define dos máscaras de Sobel, una para detectar cambios de intensidad en la dirección horizontal (sobel\_x) y otra en la dirección vertical (sobel\_y). El filtro de Sobel es un operador que calcula aproximaciones de las derivadas en estas direcciones, resaltando los bordes donde hay cambios bruscos de intensidad en la imagen. Luego, se aplica una convolución entre la imagen y estas máscaras para obtener los gradientes en ambas direcciones. La magnitud del gradiente se calcula combinando estos gradientes con la fórmula de la raíz cuadrada, lo que da como resultado la intensidad del borde en cada píxel. Posteriormente, se normalizan los valores de la magnitud para que estén en el rango 0-255, facilitando su visualización. Finalmente, se mide el tiempo de ejecución del proceso y se muestra la imagen resultante con los bordes resaltados. Todo el código se ejecuta de manera secuencial, sin paralelismo.

### Segunda Implementación en CUDA

Carga una imagen en escala de grises usando la biblioteca `stb_image` y asigna memoria tanto en la CPU (host) como en la GPU (device) para la imagen de entrada y salida. El kernel CUDA (`sobelFilterKernel`) se ejecuta de manera paralela utilizando bloques e hilos definidos por `blockDim` y `gridDim`, donde cada hilo procesa un píxel de la imagen, calculando los gradientes en las direcciones X e Y y generando la imagen con los bordes detectados. El tiempo de ejecución del procesamiento en la GPU se mide con `clock()` y se imprime. Al final, la imagen procesada se copia de vuelta a la CPU, se guarda en un archivo PNG, y se libera la memoria asignada tanto en la CPU como en la GPU.

### Tercera Implementación en Neon ARM

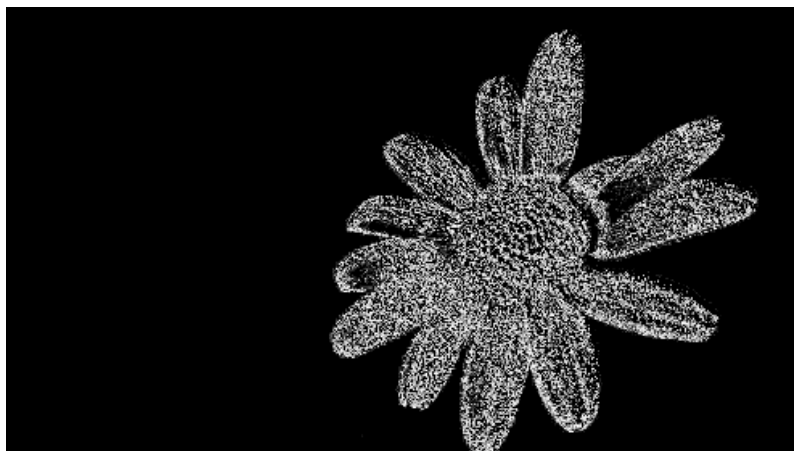
Se hace el código para la implementación Neon ARM. Se carga la imagen en escala de grises usando `stb_image` y define dos máscaras sobel. En la función `sobelFilterNeon` se hace el procesamiento de cada píxel usando instrucciones NEON, que primero realiza operaciones vectoriales en paralelo. Por cada píxel se cargan los valores de los pixeles adyacentes y se interpretan como valores con signo(`int8x8_t`), luego se aplican las máscaras mediante operaciones vectorizadas de multiplicación y acumulación(`vmlal_s8`). Después se calcula la magnitud del gradiente sumando los valores absolutos de la gradiente en X y Y. Finalmente se toma el tiempo de ejecución, se guarda la imagen y se libera la memoria asignada para las imágenes de entrada y salida.

### Comparación de los tiempos de ejecución

| Implementación | Imagen 1 | Imagen 2 | Imagen 3 | Imagen 4 | Imagen 5 |
|----------------|----------|----------|----------|----------|----------|
| Python         | 1,156    | 0,078    | 0,173    | 0,262    | 0,334    |
| CUDA           | 0,081    | 0,066    | 0,066    | 0,066    | 0,074    |
| Neon ARM       | 3,498    | 0,206    | 0,495    | 0,742    | 1,071    |

### Comparación de calidad en imagen

#### Resultado de filtro en código de Python





Resultado de filtro con la implementación en CUDA



Resultado de filtro con la implementación en ARM NEON

