

Instituto Tecnológico de Costa Rica

Área Académica de Ingeniería en Computadores

(Computer Engineering Academic Area)

Programa de Licenciatura en Ingeniería en Computadores

(Licentiate Degree Program in Computer Engineering)



Arquitectura de Computadores II

Taller 1: CUDA

Realizado por:

(Made by:)

Gerald Mora Mora

Christofer Azofeifa Ureña

Gabriel González Houdelath

Rubén Salas Ramírez

Kevin Rodriguez

Profesor:

(Professor)

Ronald Eduardo Garcia Fernandez

Fecha: Cartago, septiembre 12, 2024

(Date: Cartago, sept 12, 2024)

1. Investigación.

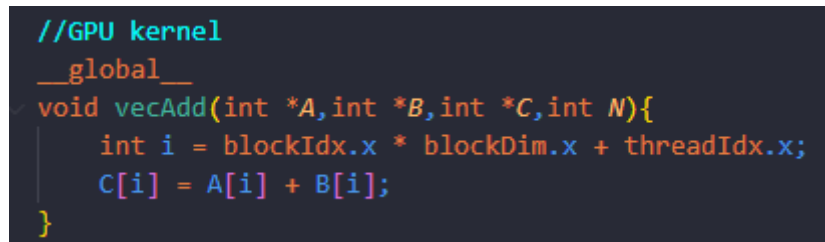
- **¿Qué es CUDA?**

CUDA es una plataforma variante de C y C++, creada por NVIDIA para la ejecución de programas en las GPU aprovechando las propiedades de paralelismo de las mismas, estos programas son lanzados por la CPU y se ejecutan en la GPU. CUDA tiene aplicaciones en el área de simulaciones científicas, procesamiento de imágenes, machine learning e Inteligencia Artificial.

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

Un Kernel en este contexto, es una función capaz de ejecutarse en una GPU, para definirla es necesario usar la anotación: `__global__`

Aca un ejemplo de cómo se define una función kernel:



```
//GPU kernel
__global__
void vecAdd(int *A,int *B,int *C,int N){
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    C[i] = A[i] + B[i];
}
```

Imagen 1. Bloque de código correspondiente a un kernel.

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

La ejecución de un kernel en CUDA se maneja de forma jerárquica de forma que los hilos se agrupan en bloques y los bloques en mallas, de esta forma se puede controlar la ejecución al conocer los índices de bloque o de hilo. Para la asignación se define el tamaño del bloque (# de hilos por bloque) y el tamaño de la malla (# de bloques por malla) estos argumentos se indican en el kernel de forma que cuando este se vaya a ejecutar lo hará con las dimensiones definidas según las necesidades, además CUDA permite obtener información sobre los identificadores de los bloques e hilos (threadIdx, blockIdx) o de las dimensiones de los bloques y mallas (blockDim, gridDim) para tener más control sobre la ejecución en cada hilo si es necesario.

- **Investigue sobre la plataforma Jetson Nano ¿cómo está compuesta la arquitectura de la plataforma a nivel de hardware?**

Es una pequeño, pero poderoso computador, creado por NVIDIA, enfocado en el desarrollo de aplicaciones embebidas, inteligencia artificial e IoT.

Su arquitectura está compuesta por:

GPU	NVIDIA Maxwell architecture with 128 NVIDIA CUDA® cores
CPU	Quad-core ARM Cortex-A57 MPCore processor
Memoria	4 GB 64-bit LPDDR4, 1600MHz 25.6 GB/s
Almacenamiento	16 GB eMMC 5.1
E/S	4x USB 3.0, USB 2.0 Micro-B, GPIO, I2C, I2S, SPI, UART, HDMI 2.0 and eDP 1.4, Gigabit Ethernet, M.2 Key E

- **¿Cómo se compila un código CUDA?**

La manera de compilar CUDA es muy similar a cuando se compila un código C/C++, ya que se usa una instrucción similar, solamente que en vez de utilizar el compilador gcc, se usa el compilador NVCC de NVIDIA, por lo que la instrucción de compilación sería la siguiente:

nvcc programa.cu -o programa

Para ejecutar el programa una vez compilado,, este se ejecuta con el siguiente comando:

./programa

2. Analisis de codigo fuente vecadd.cu

- **Analice el código vecadd.cu.**

El archivo vecadd.cu contiene el código fuente de una aplicación que se encarga de realizar la suma de dos vectores en paralelo, utiliza una función kernel para ejecutar la operación desde la GPU, también se encuentra un método convencional que realiza la misma función, pero, esta vez a nivel de CPU, para así reproducir los tiempos de ejecución y realizar la comparación del desempeño en ambos tipos de ejecución.

- **Analice el código fuente del kernel vecadd.cu. A partir del análisis del código, determine:**

- ¿Qué operación se realiza con los vectores de entrada?

Suma de los dos vectores A y B en paralelo, su resultado termina almacenado en el vector C.

- ¿Cómo se identifica cada elemento a ser procesado en paralelo y de qué forma se realiza el procesamiento paralelo?

Cada hilo tomará una instancia del código kernel y se encargará de procesar una posición específica de los vectores. El índice que se usa para recorrer

los vectores es calculado por medio de la instrucción: `int i = blockIdx.x * blockDim.x + threadIdx.x;`

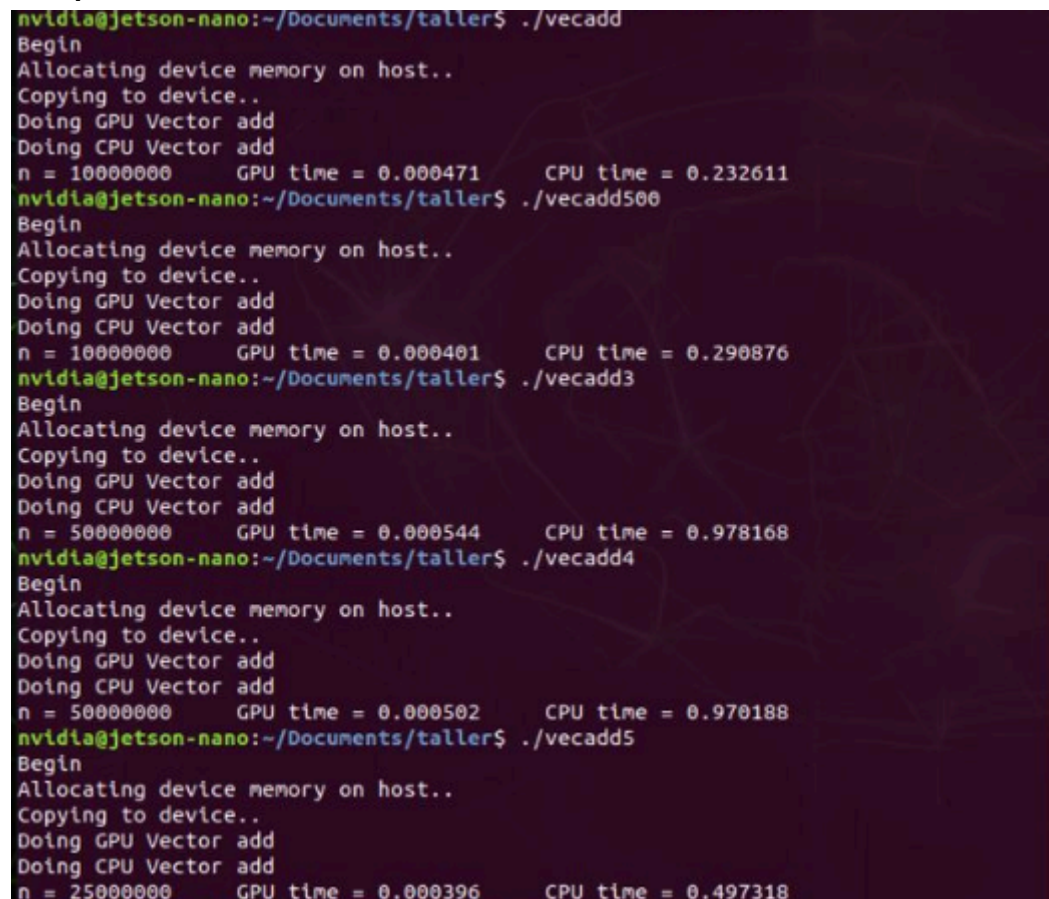
La instrucción `blockIdx.x` identifica el bloque, `blockDim.x` determina la dimensión del mismo y `threadIdx.x` sería el identificador del hilo.

El procesamiento en paralelo es gracias a que CUDA organiza los hilos en bloques, y los bloques en un grid o cuadrícula, cada hilo de la GPU procesa una parte del vector, lo que permite que el proceso de la suma de vectores en este caso se de en paralelo y agilice la solución del problema a resolver.

- **Realice la ejecución de la aplicación con `vecadd`. ¿Qué hace finalmente la aplicación?**

La suma paralela desde la GPU de dos vectores, y una vez finalizada se realiza la misma a nivel CPU. Imprime los datos de tiempo tomados por cada cálculo para que el usuario pueda realizar su respectiva comparación.

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



```
nvidia@jetson-nano:~/Documents/taller$ ./vecadd
Begin
Allocating device memory on host..
Copying to device..
Doing GPU Vector add
Doing CPU Vector add
n = 10000000 GPU time = 0.000471 CPU time = 0.232611
nvidia@jetson-nano:~/Documents/taller$ ./vecadd500
Begin
Allocating device memory on host..
Copying to device..
Doing GPU Vector add
Doing CPU Vector add
n = 10000000 GPU time = 0.000401 CPU time = 0.290876
nvidia@jetson-nano:~/Documents/taller$ ./vecadd3
Begin
Allocating device memory on host..
Copying to device..
Doing GPU Vector add
Doing CPU Vector add
n = 50000000 GPU time = 0.000544 CPU time = 0.978168
nvidia@jetson-nano:~/Documents/taller$ ./vecadd4
Begin
Allocating device memory on host..
Copying to device..
Doing GPU Vector add
Doing CPU Vector add
n = 50000000 GPU time = 0.000502 CPU time = 0.970188
nvidia@jetson-nano:~/Documents/taller$ ./vecadd5
Begin
Allocating device memory on host..
Copying to device..
Doing GPU Vector add
Doing CPU Vector add
n = 25000000 GPU time = 0.000396 CPU time = 0.497318
```

Imagen 2. Comparación del programa `vecadd.cu` para distintas cantidades de hilos y tamaños de vector.

Hilos por bloque	N (Iteraciones)	Tiempo CPU [s]	Tiempo GPU [s]
250	50 000 000	0.978168	0.000544
500	50 000 000	0.970188	0.000502
375	25 000 000	0.497318	0.000396
250	10 000 000	0.232611	0.000471
500	10 000 000	0.290876	0.000401

Comparando los casos donde n es igual pero el número de hilos por bloque varía se puede observar que el tiempo de ejecución de la GPU es mejor al tener 500 hilos por lo que parece que tener más en el bloque puede ser beneficioso para esta tarea. Es interesante analizar que para el quinto caso el tiempo de ejecución es el menor de todos incluso siendo muchos más datos que en los primeros dos casos, por lo que puede ser difícil en todas las aplicaciones definir directamente cuales son los mejores parámetros, ya que por ejemplo es este caso parece que los 375 hilos por bloques representaron una mejora en el tiempo de ejecución del programa. Por último al comparar los casos donde se tiene el mismo número de hilos por bloque se tuvo un resultado que podía ser esperado donde el tiempo de ejecución fue menor para los casos con menor n.

3. Resultados ejercicios prácticos.

A continuación, se muestran los valores del ejercicio 1 de la parte práctica, el cual corresponde a una multiplicación de matrices 4x4 donde se puede observar que la ejecución en GPU fue menor en cada una de las pruebas realizadas con diferentes hilos por bloque y la cantidad de iteraciones realizadas.

Hilos por bloque	N (Iteraciones)	Tiempo CPU [s]	Tiempo GPU [s]
250	1 000 000	0.013567	0.000125
500	1 000 000	0.017575	0.000136
250	5 000 000	0.067305	0.000635
500	5 000 000	0.069557	0.000621
375	2 500 000	0.033703	0.000593

Por otra parte, para el caso del ejercicio práctico 2 el cual pretendía la implementación de un filtro de detección de bordes de manera serial, utilizando CUDA y con Neon ARM. Primeramente, se realizó un programa para descomponer la imagen utilizada en un archivo de texto con sus píxeles, el cual iba a ser utilizado en cada uno de los siguientes programas

a realizar y probar. Una vez realizado esto se procedió a programar cada uno de los casos, para los cuales se pudieron obtener los siguientes valores.

Imagen	Serial (ms)	Cuda (ms)	Neon ARM (ms)
1	19	89	1
2	38	80	1
3	19	94	1
4	19	83	1
5	655	84	1

Dicha ejecución, así como los datos obtenidos pueden ser visualizados en la siguiente imagen donde puede visualizarse la compilación de los programas y el tiempo de ejecución obtenido.

```
tcuda@jetson:~/Documents$ g++ -o edgeDetSerial edgeDetSerial.c && ./edgeDetSerial
Tiempo de ejecución: 19 ms
tcuda@jetson:~/Documents$ g++ -o edgeDetSerial1 edgeDetSerial.c && ./edgeDetSerial1
Tiempo de ejecución: 38 ms
tcuda@jetson:~/Documents$ g++ -o edgeDetSerial3 edgeDetSerial.c && ./edgeDetSerial3
Tiempo de ejecución: 19 ms
tcuda@jetson:~/Documents$ g++ -o edgeDetSerial4 edgeDetSerial.c && ./edgeDetSerial4
Tiempo de ejecución: 19 ms
tcuda@jetson:~/Documents$ g++ -o edgeDetSerial5 edgeDetSerial.c && ./edgeDetSerial5
Tiempo de ejecución: 655 ms
tcuda@jetson:~/Documents$ g++ -o edgeDetSerial5 edgeDetSerial.c && ./edgeDetSerial5
Tiempo de ejecución: 656 ms
tcuda@jetson:~/Documents$ g++ -o edgeDetNeon edgeDetNeon.c && ./edgeDetNeon
Tiempo de ejecución: 1 ms
tcuda@jetson:~/Documents$ g++ -o edgeDetNeon2 edgeDetNeon.c && ./edgeDetNeon2
Tiempo de ejecución: 1 ms
tcuda@jetson:~/Documents$ g++ -o edgeDetNeon3 edgeDetNeon.c && ./edgeDetNeon3
Tiempo de ejecución: 1 ms
tcuda@jetson:~/Documents$ g++ -o edgeDetNeon4 edgeDetNeon.c && ./edgeDetNeon4
Tiempo de ejecución: 1 ms
tcuda@jetson:~/Documents$ g++ -o edgeDetNeon5 edgeDetNeon.c && ./edgeDetNeon5
bash: ++: command not found
tcuda@jetson:~/Documents$ ++ -o edgeDetNeon5 edgeDetNeon.c && ./edgeDetNeon5
bash: ++: command not found
tcuda@jetson:~/Documents$ g++ -o edgeDetNeon5 edgeDetNeon.c && ./edgeDetNeon5
Tiempo de ejecución: 1 ms
tcuda@jetson:~/Documents$ nvcc edgeDetCuda.cu -o edgeDetCuda && ./edgeDetCuda
Tiempo de ejecución: 89 ms
tcuda@jetson:~/Documents$ nvcc edgeDetCuda.cu -o edgeDetCuda2 && ./edgeDetCuda2
Tiempo de ejecución: 80 ms
tcuda@jetson:~/Documents$ nvcc edgeDetCuda.cu -o edgeDetCuda3 && ./edgeDetCuda3
Tiempo de ejecución: 94 ms
tcuda@jetson:~/Documents$ nvcc edgeDetCuda.cu -o edgeDetCuda4 && ./edgeDetCuda4
Tiempo de ejecución: 83 ms
tcuda@jetson:~/Documents$ nvcc edgeDetCuda.cu -o edgeDetCuda5 && ./edgeDetCuda5
Tiempo de ejecución: 84 ms
tcuda@jetson:~/Documents$
```

Imagen 3. Comparación de aplicación de filtro para 5 imágenes distintas con tiempo de ejecución obtenido

Los datos obtenidos en la tabla anterior dan un panorama amplio entre cada una de las tres implementaciones, donde puede observarse lo efectivo que resulta la ejecución del programa utilizando Neon ARM, el cual mantiene valores de ejecución de 1ms, muy por debajo de las otras dos implementaciones. El valor tan bajo de tiempo de ejecución al utilizar Neon ARM es bien respaldado por la teoría, ya que esta implementación posee instrucciones especiales que utilizan el paralelismo con el objetivo de hacer sistemas más

eficientes que aprovechen al máximo los recursos del GPU, siendo ampliamente utilizado en aplicaciones multimedia en plataformas ARM. Por otra parte, existe una comparación interesante entre la implementación serial y la cuda, donde los tiempos de ejecución de las primeras 4 imágenes resultan menores al utilizar una implementación serial en comparación con una CUDA, siendo estas imágenes de tamaño normal y donde podría llegar a concluirse que para imágenes sencillas el costo de realizar el proceso de paralelismo propio de CUDA resulta más costoso que beneficioso y una implementación directa termina siendo más favorable, sin embargo los valores cambian a la hora de utilizar una imagen de mayor tamaño, caso para el cual los valores de CUDA se mantienen similares en comparación a las demás imágenes y la implementación serial aumenta su tiempo de ejecución en sobremanera. De lo anterior puede ser concluido que la implementación CUDA y sus beneficios empiezan a ser positivos a medida que se aumenta la complejidad del programa y la cantidad de datos, lo que es respaldado por la teoría de paralelismo el cual resulta cada vez más beneficioso a medida que se aumenta la cantidad de datos y el tamaño del vector de paralelización a utilizar.

4. Bibliografía.

<https://www.ichec.ie/academic/national-hpc/documentation/tutorials/compiling-program-cuda>
<https://developer.nvidia.com/embedded/jetson-nano#:~:text=Jetson%20Nano%20is%20a%20small,graphics%2C%20multimedia%2C%20and%20more.>