TEC | Tecnológico de Costa Rica

Instituto Tecnológico de Costa Rica
Área Académica de Ingeniería en Computadores
CE 4302 — Arquitectura de Computadores II

Taller 4:
CUDA

Fecha de asignación:	28 de abril 2023	Fecha de entrega:	05 de mayo 2023
Grupo:	4 personas (proyecto 2)	Profesor:	Luis Barboza Artavia

Integrantes:

Alejandro Díaz Pereira

Bertha Brenes Brenes

Ignacio Vargas Campos

Carlos Adrián Araya Ramírez

Shakime Richards Sparks

1. Investigación

Para comprender mejor CUDA, realice una pequeña búsqueda para responder las siguientes preguntas:

1. ¿Qué es CUDA?

Significa Compute Unified Device Architecture, la cual fue desarrollada por NVIDIA Compute Unified Device Architecture para trabajar tareas de computación paralela las cuales utilizan GPU (Unidades de procesamiento gráfico) en tareas de procesamiento intensivo en paralelo, como el procesamiento de imágenes, la simulación científica, el aprendizaje profundo y el análisis de datos [1].

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

Un kernel CUDA es una función especializada que está diseñada para ser ejecutada en paralelo por un gran número de hilos en la GPU utilizando la plataforma de cómputo paralelo CUDA. En la arquitectura CUDA, la GPU se utiliza para realizar cálculos altamente paralelos, lo que puede acelerar significativamente ciertos tipos de tareas. La función del kernel se define utilizando un lenguaje similar a C con extensiones específicas proporcionadas por CUDA y se invoca con un número específico de bloques y hilos por bloque, y es responsabilidad del programador definir esta

organización en función del problema que se está resolviendo. Cada hilo ejecuta el mismo código, pero opera sobre datos diferentes.

Los kernels CUDA se utilizan con frecuencia para tareas intensivas en cómputo que pueden ser paralelizadas, como operaciones de matriz, procesamiento de imágenes, simulaciones y algoritmos de aprendizaje automático. Al aprovechar la arquitectura masivamente paralela de la GPU, los kernels CUDA pueden lograr aceleraciones significativas en comparación con la ejecución del mismo código en una CPU [1].

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

Cuando se lanza un programa CUDA, se crean múltiples hilos y se organizan en bloques de hilos. Estos bloques de hilos se agrupan en una malla (grid) y a cada hilo se le asigna una identificación única de hilo, lo que le permite acceder y procesar una porción diferente de los datos de entrada y cada bloque es atendido por un multiprocesador. El sistema de tiempo de ejecución CUDA administra la ejecución del kernel en la GPU, distribuyendo la carga de trabajo entre los recursos de cómputo disponibles. Esto permite que los hilos de un mismo bloque puedan sincronizar su ejecución y compartir datos de forma eficiente usando memoria local de baja latencia [2].

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

La plataforma Jetson Nano es un sistema en módulo desarrollado por NVIDIA para aplicaciones de inteligencia artificial y computación de borde. Está compuesto por un procesador ARM Cortex-A57 de cuatro núcleos, una GPU NVIDIA Maxwell de 128 núcleos y 4 GB de memoria LPDDR4.

La arquitectura de la plataforma Jetson Nano se divide en dos partes principales: la CPU y la GPU. La CPU es responsable de las tareas de procesamiento general, mientras que la GPU se encarga de las tareas de cómputo intensivo, como el procesamiento de imágenes y la detección de objetos.

Además, la plataforma Jetson Nano cuenta con una amplia variedad de puertos y conectores, incluyendo USB 3.0, HDMI y GPIO, lo que la hace ideal para una amplia variedad de aplicaciones de inteligencia artificial y robótica.

5. ¿Cómo se compila un código CUDA?

Una vez instalado el CUDA Toolkit, se puede utilizar el compilador **nvcc** para compilar el código CUDA. El proceso de compilación implica especificar la ubicación de los archivos de inclusión y bibliotecas necesarios, así como las opciones de optimización, arquitectura y enlazado. También es importante asegurarse de que cualquier biblioteca externa utilizada en el código esté instalada y configurada correctamente para su uso con CUDA. Para compilar un código se utiliza el siguiente comando:

$$nvcc\ my\_code.cu\ -o\ my\_executable$$

Esto compilará el archivo **my_code.cu** y generará un ejecutable llamado **my_executable**. Una vez compilado el código CUDA y generado el archivo ejecutable, para ejecutar el programa en la GPU compatible se debe abrir una terminal y navegar hasta el directorio donde se encuentra el archivo ejecutable. Luego, el programa puede ser ejecutado en la GPU utilizando el siguiente comando:

$$./\ executable$$

Como se observa, el nombre del archivo ejecutable debe ser el que se le haya dado durante la generación con el compilador **nvcc**. Es importante verificar que la GPU esté correctamente instalada y configurada en el sistema antes de ejecutar el programa [3].

NOTA: Es necesario tener en cuenta que el código CUDA tiene que ser diseñado para ser paralelizable para que el programa se ejecute en la GPU. Si el código no se puede paralelizar, entonces el programa se ejecutará en la CPU en lugar de la GPU, lo que puede ser menos eficiente.

5. Análisis

Como primera prueba de CUDA, se va a trabajar con el archivo *vecadd.cu*, que utiliza un *kernel* en GPU para una operación de suma. Antes de continuar, debe asegurarse de entender correctamente qué es un kernel y cómo funcionan en el ámbito de CUDA.
Con base en el código de ejemplo:

1. Analice el código vecadd.cu.
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?

La operación que se realiza con los vectores de entrada es una suma vectorial, se suman los elementos de los vectores **a** y **b** de forma paralela y el resultado se almacena en el vector c. Esta operación se realiza tanto en la CPU como en la GPU, en la función **vecAdd_h** y en la función del kernel **vecAdd**, respectivamente. La diferencia es que la CPU realiza esta operación de manera secuencial, mientras que la GPU lo hace en paralelo mediante hilos.

Los vectores de entrada se identifican por la forma en que se almacenan en memoria. Los punteros que apuntan a la memoria del CPU se inicializan utilizando la función malloc, mientras que los punteros que apuntan a la memoria del GPU se inicializan utilizando la función cudaMalloc. Para inicializar los datos, el CPU utiliza un bucle for que recorre todos los elementos de los vectores y los inicializa con valores aleatorios. Para copiar los datos de los vectores de entrada desde la memoria del CPU a la memoria del GPU, se utiliza la función cudaMemcpy, que copia los valores de a y b y los almacena en **a_d** y **b_d**, respectivamente.

Respecto a la forma en que se realiza el procesamiento en paralelo, en la CPU, la suma vectorial se realiza en la función **vecAdd_h**, que recibe los vectores a y b como argumentos, realiza la suma vectorial elemento a elemento y guarda el resultado en el vector c2. Esto se hace en un ciclo for que recorre todos los elementos de los vectores. En la GPU, la suma vectorial se realiza en la función del kernel **vecAdd**, que también recibe los vectores a y b, así como el vector c donde se guardará el resultado de la suma. Para realizar el procesamiento paralelo en el GPU, se utilizan los conceptos de bloques y rejillas (grids). Cada bloque es un grupo de hilos que ejecutan la función del kernel de manera paralela, lo cual implica que la función del kernel se ejecuta en paralelo en varios hilos en la GPU, esto permite acelerar la operación de suma vectorial en comparación con la CPU.

3. Realice la compilación del código vecadd.cu.

```
heutlett@heutlett-PC:~/TEC 2023/Arqui 2/Taller 4/Taller4_CUDA/src$ nvcc vecadd.cu -o vecadd
vecadd.cu: In function ‘int main(int, char**)’:
vecadd.cu:74:23: warning: ‘cudaError_t cudaThreadSynchronize()’ is deprecated [-Wdeprecated-decl
arations]
     cudaThreadSynchronize();
                       ^
/usr/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?

```
heutlett@heutlett-PC:~/TEC 2023/Arqui 2/Taller 4/Taller4_CUDA/src$ ./vecadd
Begin
Allocating device memory on host..
Copying to device..
Doing GPU Vector add
Doing CPU Vector add
n = 10000000     GPU time = 0.001304     CPU time = 0.050415
```

Este código realiza la suma de dos vectores, "A" y "B", almacenados en el host (CPU), y almacena el resultado en un tercer vector "C". Para acelerar la operación, utiliza el paralelismo de la GPU a través de la tecnología CUDA de NVIDIA. Para ello, el programa define una función kernel, "vecAdd()", que se ejecuta en la GPU y suma los vectores "A" y "B", almacenándolos en "C".

El programa también realiza la misma operación de suma en el host (CPU) utilizando la función "vecAdd_h()". Finalmente, el programa mide el tiempo de ejecución de ambas operaciones, tanto en la GPU como en el host, y los imprime junto con el tamaño del bloque utilizado en la ejecución del kernel.

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

N = 1000

```
heutlett@heutlett-PC:~/TEC 2023/Arqui 2/Taller 4/Taller4_CUDA/src$ ./vecadd
Begin
Allocating device memory on host..
Copying to device..
Doing GPU Vector add
Doing CPU Vector add
n = 1000         GPU time = 0.000029     CPU time = 0.000004     Block size = 250
```

```
heutlett@heutlett-PC:~/TEC 2023/Arqui 2/Taller 4/Taller4_CUDA/src$ ./vecadd
Begin
Allocating device memory on host..
Copying to device..
Doing GPU Vector add
Doing CPU Vector add
n = 1000         GPU time = 0.000038     CPU time = 0.000003     Block size = 512
```

N = 10000

```
heutlett@heutlett-PC:~/TEC 2023/Arqui 2/Taller 4/Taller4_CUDA/src$ ./vecadd
Begin
Allocating device memory on host..
Copying to device..
Doing GPU Vector add
Doing CPU Vector add
n = 10000        GPU time = 0.000039     CPU time = 0.000091     Block size = 250
```

```
heutlett@heutlett-PC:~/TEC 2023/Arqui 2/Taller 4/Taller4_CUDA/src$ ./vecadd
Begin
Allocating device memory on host..
Copying to device..
Doing GPU Vector add
Doing CPU Vector add
n = 10000        GPU time = 0.000029     CPU time = 0.000039     Block size = 512
```

N = 100000

```
heutlett@heutlett-PC:~/TEC 2023/Arqui 2/Taller 4/Taller4_CUDA/src$ ./vecadd
Begin
Allocating device memory on host..
Copying to device..
Doing GPU Vector add
Doing CPU Vector add
n = 100000       GPU time = 0.000082     CPU time = 0.000404     Block size = 250
```

```
heutlett@heutlett-PC:~/TEC 2023/Arqui 2/Taller 4/Taller4_CUDA/src$ ./vecadd
Begin
Allocating device memory on host..
Copying to device..
Doing GPU Vector add
Doing CPU Vector add
n = 100000       GPU time = 0.000053     CPU time = 0.000375     Block size = 512
```

6. Ejercicios prácticos

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

Código:

mult.cu vecadd.cu

mult.cu > main()

```cuda
#include <stdio.h>

#define N 4


__global__ void matrixMult(float *a, float *b, float *c) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0;

    for (int k = 0; k < N; k++) {
        sum += a[row * N + k] * b[k * N + col];
    }

    c[row * N + col] = sum;
}

int main() {
    float *h_a, *h_b, *h_c;
    float *d_a, *d_b, *d_c;

    h_a = (float*) malloc(N * N * sizeof(float));
    h_b = (float*) malloc(N * N * sizeof(float));
    h_c = (float*) malloc(N * N * sizeof(float));

    for (int i = 0; i < N * N; i++) {
        h_a[i] = i;
        h_b[i] = i + 1;
    }

    printf("Matriz a:\n");
    for (int i = 0; i < N * N; i++) {
        printf("%.2f ", h_a[i]);
        if ((i + 1) % N == 0) {
            printf("\n");
        }
    }
    printf("\n");

    printf("Matriz b:\n");
    for (int i = 0; i < N * N; i++) {
        printf("%.2f ", h_b[i]);
        if ((i + 1) % N == 0) {
            printf("\n");
        }
    }
    printf("\n");
```

```
48
49      cudaMalloc(&d_a, N * N * sizeof(float));
50      cudaMalloc(&d_b, N * N * sizeof(float));
51      cudaMalloc(&d_c, N * N * sizeof(float));
52
53      cudaMemcpy(d_a, h_a, N * N * sizeof(float), cudaMemcpyHostToDevice);
54      cudaMemcpy(d_b, h_b, N * N * sizeof(float), cudaMemcpyHostToDevice);
55
56      dim3 threadsPerBlock(N, N);
57      dim3 numBlocks(1, 1);
58
59      matrixMult<<<numBlocks, threadsPerBlock>>>(d_a, d_b, d_c);
60
61      cudaMemcpy(h_c, d_c, N * N * sizeof(float), cudaMemcpyDeviceToHost);
62
63      printf("Matriz c:\n");
64      for (int i = 0; i < N * N; i++) {
65          printf("%.2f ", h_c[i]);
66          if ((i + 1) % N == 0) {
67              printf("\n");
68          }
69      }
70
71      free(h_a);
72      free(h_b);
73      free(h_c);
74      cudaFree(d_a);
75      cudaFree(d_b);
76      cudaFree(d_c);
77
78      return 0;
79  }
80
```

Ejecución:

```
heutlett@heutlett-PC:~/TEC 2023/Arqui 2/Taller 4/Taller4_CUDA/src$ nvcc mult.cu -o mult
heutlett@heutlett-PC:~/TEC 2023/Arqui 2/Taller 4/Taller4_CUDA/src$ ./mult
Matriz a:
0.00 1.00 2.00 3.00
4.00 5.00 6.00 7.00
8.00 9.00 10.00 11.00
12.00 13.00 14.00 15.00

Matriz b:
1.00 2.00 3.00 4.00
5.00 6.00 7.00 8.00
9.00 10.00 11.00 12.00
13.00 14.00 15.00 16.00

Matriz c:
62.00 68.00 74.00 80.00
174.00 196.00 218.00 240.00
286.00 324.00 362.00 400.00
398.00 452.00 506.00 560.00
```

2. Proponga una aplicación que involucre procesamiento vectorial. Implemente dicha aplicación tanto serial (sin paralelismo) como con CUDA. Mida tiempos de ejecución para diferentes tamaños y/o iteraciones.

Código:

Taller4_CUDA > src > saxpy.cu > saxpy_kernel(float, float *, float *, int)

```cpp
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define N 10000000
#define BLOCK_SIZE 256

__global__ void saxpy_kernel(float a, float *x, float *y, int n) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n) {
        y[i] = a * x[i] + y[i];
    }
}


void saxpy_serial(float a, float* x, float* y, int n)
{
    for (int i = 0; i < n; i++)
    {
        y[i] = a * x[i] + y[i];
    }
}

int main()
{
    float *x, *y, *d_x, *d_y;
    float a = 2.0;
    int size = N * sizeof(float);

    // Allocate memory on host
    x = (float*) malloc(size);
    y = (float*) malloc(size);

    // Initialize vectors on host
    for (int i = 0; i < N; i++)
    {
        x[i] = i;
        y[i] = i * 2;
    }

    // Allocate memory on device
    cudaMalloc((void**) &d_x, size);
    cudaMalloc((void**) &d_y, size);

    // Copy vectors from host to device
    cudaMemcpy(d_x, x, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_y, y, size, cudaMemcpyHostToDevice);

    // Call serial saxpy
    clock_t t_serial_start = clock();
    saxpy_serial(a, x, y, N);

    clock_t t_serial_end = clock();
    double t_serial = ((double)(t_serial_end - t_serial_start)) / CLOCKS_PER_SEC;
```

```
55
56      // Call parallel saxpy
57      clock_t t_parallel_start = clock();
58      saxpy_kernel<<<(N + BLOCK_SIZE - 1) / BLOCK_SIZE, BLOCK_SIZE>>>(a, d_x, d_y, N);
59      cudaDeviceSynchronize();
60      clock_t t_parallel_end = clock();
61      double t_parallel = ((double)(t_parallel_end - t_parallel_start)) / CLOCKS_PER_SEC;
62
63      // Copy result from device to host
64      cudaMemcpy(y, d_y, size, cudaMemcpyDeviceToHost);
65
66      // Free memory on device
67      cudaFree(d_x);
68      cudaFree(d_y);
69
70      // Print execution times
71      printf("Execution for N = %d\n", N);
72      printf("Serial time: %f\n", t_serial);
73      printf("Parallel time: %f\n", t_parallel);
74
75      // Free memory on host
76      free(x);
77      free(y);
78
79      return 0;
80
81  }
```

Ejecución:

```
heutlett@heutlett-PC:~/TEC 2023/Arqui 2/Taller 4/Taller4_CUDA/src$ ./saxpy
Execution for N = 1000
Serial time: 0.000004
Parallel time: 0.000020
heutlett@heutlett-PC:~/TEC 2023/Arqui 2/Taller 4/Taller4_CUDA/src$
heutlett@heutlett-PC:~/TEC 2023/Arqui 2/Taller 4/Taller4_CUDA/src$ nvcc saxpy.cu -o saxpy
heutlett@heutlett-PC:~/TEC 2023/Arqui 2/Taller 4/Taller4_CUDA/src$ ./saxpy
Execution for N = 10
Serial time: 0.000001
Parallel time: 0.000022
heutlett@heutlett-PC:~/TEC 2023/Arqui 2/Taller 4/Taller4_CUDA/src$ nvcc saxpy.cu -o saxpy
heutlett@heutlett-PC:~/TEC 2023/Arqui 2/Taller 4/Taller4_CUDA/src$ ./saxpy
Execution for N = 100
Serial time: 0.000002
Parallel time: 0.000029
heutlett@heutlett-PC:~/TEC 2023/Arqui 2/Taller 4/Taller4_CUDA/src$ nvcc saxpy.cu -o saxpy
heutlett@heutlett-PC:~/TEC 2023/Arqui 2/Taller 4/Taller4_CUDA/src$ ./saxpy
Execution for N = 1000
Serial time: 0.000004
Parallel time: 0.000019
heutlett@heutlett-PC:~/TEC 2023/Arqui 2/Taller 4/Taller4_CUDA/src$ nvcc saxpy.cu -o saxpy
heutlett@heutlett-PC:~/TEC 2023/Arqui 2/Taller 4/Taller4_CUDA/src$ ./saxpy
Execution for N = 10000
Serial time: 0.000024
Parallel time: 0.000021
heutlett@heutlett-PC:~/TEC 2023/Arqui 2/Taller 4/Taller4_CUDA/src$ nvcc saxpy.cu -o saxpy
heutlett@heutlett-PC:~/TEC 2023/Arqui 2/Taller 4/Taller4_CUDA/src$ ./saxpy
Execution for N = 100000
Serial time: 0.000232
Parallel time: 0.000026
heutlett@heutlett-PC:~/TEC 2023/Arqui 2/Taller 4/Taller4_CUDA/src$ nvcc saxpy.cu -o saxpy
^[[Aheutlett@heutlett-PC:~/TEC 2023/Arqui 2/Taller 4/Taller4_CUDA/src$ ./saxpy
Execution for N = 1000000
Serial time: 0.002434
Parallel time: 0.000140
heutlett@heutlett-PC:~/TEC 2023/Arqui 2/Taller 4/Taller4_CUDA/src$ nvcc saxpy.cu -o saxpy
heutlett@heutlett-PC:~/TEC 2023/Arqui 2/Taller 4/Taller4_CUDA/src$ ./saxpy
Execution for N = 10000000
Serial time: 0.024058
Parallel time: 0.001221
heutlett@heutlett-PC:~/TEC 2023/Arqui 2/Taller 4/Taller4_CUDA/src$
```

References

[1] NVIDIA. CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/

[2] J. Roca, "Núcleos CUDA de NVIDIA: ¿qué son y para qué sirven?," HardZone, 21 de junio 2022. https://hardzone.es/marcas/nvidia/nucleos-cuda/

[3] J. Sanders and E. Kandrot, CUDA by Example : An Introduction to General-Purpose GPU Programming. Sydney: Pearson Education, Limited, 2010. https://edoras.sdsu.edu/~mthomas/docs/cuda/cuda_by_example.book.pdf