



**Instituto Tecnológico de Costa Rica
Área Académica de Ingeniería en Computadores
Arquitectura de Computadores II
CE4302**

Tipo de Asignación: Taller

**Título:
Taller I - CUDA**

**Autores:
Carrillo Salazar Juan Pablo,
Esquivel Sanchez Jonathan Daniel,
Mendoza Mata Jose Fabian,
Ortiz Vega Angelo Jesus**

**Responsable Técnico:
Barboza Artavia Luis Alonso**

**Fecha de Entrega: 03 de abril de 2024
Cartago, Costa Rica**

Investigación

1. ¿Qué es CUDA?

CUDA es una plataforma de computación paralela y un modelo de programación desarrollado por NVIDIA para realizar computación general en unidades de procesamiento gráfico (GPU). Permite a los desarrolladores acelerar significativamente aplicaciones informáticas aprovechando el poder de las GPUs.

Referencia:

- NVIDIA CUDA Zone (<https://developer.nvidia.com/cuda-zone>)

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

En CUDA, un kernel es una función que se ejecuta en la GPU y realiza operaciones de computación en paralelo. Se define en el código del programa como una función que se marca con el modificador `__global__`. Cuando se invoca un kernel, se ejecuta en paralelo por múltiples hilos en la GPU.

Referencia:

- NVIDIA CUDA C Programming Guide
(<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#kernels>)

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

El trabajo a procesar en CUDA se maneja mediante la división del problema en hilos y bloques. Los hilos son los elementos básicos de ejecución y se organizan en bloques. Los bloques son grupos de hilos que se ejecutan simultáneamente en un multiprocesador de la GPU.

La asignación de hilos y bloques se realiza especificando el número de bloques y el número de hilos por bloque en la configuración del lanzamiento del kernel. Los bloques se distribuyen a través de los multiprocesadores de la GPU, y los hilos dentro de un bloque se programan para ejecutarse en los multiprocesadores de manera eficiente.

La gestión del trabajo a procesar en CUDA se basa en aprovechar al máximo la arquitectura paralela de la GPU, distribuyendo la carga de trabajo entre los múltiples núcleos de procesamiento disponibles y minimizando la comunicación entre la CPU y la GPU para obtener un rendimiento óptimo.

Referencias:

- NVIDIA CUDA C Programming Guide
 - (<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#thread-hierarchy>)
 - (<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#execution-configuration>)

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

La arquitectura de la plataforma Jetson Nano a nivel de hardware se compone de los siguientes elementos:

- a. **Procesador y GPU:** Jetson Nano está equipado con un procesador quad-core ARM Cortex-A57 y una GPU NVIDIA Maxwell con 128 núcleos CUDA. Estos componentes proporcionan la potencia de cómputo necesaria para ejecutar múltiples redes neuronales en paralelo.
- b. **Memoria:** El Jetson Nano incluye 4 GB de memoria LPDDR4, que proporciona un amplio espacio para almacenar datos y ejecutar aplicaciones de inteligencia artificial.
- c. **Almacenamiento:** La plataforma cuenta con un conector para tarjeta microSD, que se utiliza para cargar el sistema operativo y otras aplicaciones en la memoria del dispositivo. Esto permite una fácil configuración y actualización del software.
- d. **Conectividad:** Jetson Nano ofrece varias opciones de conectividad, incluyendo USB 3.0, USB 2.0, HDMI, Ethernet Gigabit y GPIO, lo que facilita la integración con una amplia gama de dispositivos y periféricos.
- e. **Consumo de energía:** Una de las características destacadas de Jetson Nano es su eficiencia energética. La plataforma puede funcionar con tan solo 5 watts de potencia, lo que la hace adecuada para aplicaciones de bajo consumo y portátiles.

Referencias:

- Jetson Nano Developer Kit
(<https://developer.nvidia.com/embedded/jetson-nano-developer-kit>)
- Jetson Download Center
(<https://developer.nvidia.com/embedded/downloads#?search=Jetson%20Nano>)

5. ¿Cómo se compila un código CUDA?

- a. **Instalación del CUDA Toolkit:** Primero, se debe asegurar de tener instalado el CUDA Toolkit en el sistema (preferiblemente en sistemas que cuenten con GPUs de NVIDIA).
- b. **Preparación del entorno:** Luego, se debe abrir una terminal y navegar hasta el directorio que contiene el código fuente CUDA.
- c. **Escribir el código fuente CUDA:** Se debe tener un archivo de código fuente con extensión .cu. Este archivo puede incluir tanto el código del kernel como cualquier código de host necesario para inicializar y gestionar la ejecución del kernel.
- d. **Compilación del código:** Es importante utilizar el compilador de NVIDIA nvcc para compilar el código. Se pueden especificar las opciones de compilación, como la arquitectura de la GPU objetivo, las optimizaciones y las opciones de generación de código, según sea necesario. Por ejemplo, para compilar un archivo llamado *example.cu*, se puede ejecutar el siguiente comando en la terminal:

nvcc -o example example.cu

- e. **Ejecución del programa:** Después de compilar con éxito, puedes ejecutar el programa generado como lo harías con cualquier otro programa en tu sistema.

./example

Referencia:

- Documentación oficial de NVIDIA CUDA (<https://docs.nvidia.com/cuda/>)

Análisis

1. Analice el código vecadd.cu

Anotaciones importantes del código:

```
//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];

//CPU function
void vecAdd_h(int* A1, int* B1, int* C1, int N) {
    for (int i = 0; i < N; i++)
        C1[i] = A1[i] + B1[i];
```

- Se define el kernel vecAdd, que es el código que se ejecutará en la GPU. Este kernel suma los elementos correspondientes de los vectores A y B y almacena el resultado en el vector C. Cada hilo calcula el índice i para acceder a los elementos correspondientes en los vectores de entrada y salida.
- Se define una función vecAdd_h() que realiza la misma operación que el kernel vecAdd, pero en la CPU. Esta función suma los elementos correspondientes de los vectores de entrada A1 y B1 y almacena el resultado en el vector C1.

```
// Data filling
for (int i = 0; i < n; i++)
    a[i] = i, b[i] = i;
```

- Dentro del main, al inicio se inicializan variables para definir el tamaño de los vectores y el número de bloques y hilos por bloque. También se asigna memoria para los vectores en el host (a, b, c y c2).
- Pero en este punto, se llenan los vectores de entradas a y b con valores de índices.

```

printf("Allocating device memory on host..\n");
//GPU memory allocation
cudaMalloc((void**)&a_d, n * sizeof(int));
cudaMalloc((void**)&b_d, n * sizeof(int));
cudaMalloc((void**)&c_d, n * sizeof(int));

printf("Copying to device..\n");
cudaMemcpy(a_d, a, n * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(b_d, b, n * sizeof(int), cudaMemcpyHostToDevice);

clock_t start_d = clock();
printf("Doing GPU Vector add\n");
vecAdd <<<block_no, block_size >>> (a_d, b_d, c_d, n);
cudaCheckError();

//Wait for kernel call to finish
cudaThreadSynchronize();

clock_t end_d = clock();

```

- En el bloque de código anterior, se asigna memoria en la GPU para los vectores de entrada y salida, y copia los datos desde el host a la GPU.
- También se mide el tiempo de ejecución del kernel en la GPU utilizando la función `clock()`. Llama al kernel `vecAdd` con el número de bloques e hilos por bloque calculados anteriormente.

Comandos CUDA importantes:

- **cudaMalloc:** Esta función se utiliza para asignar memoria en el dispositivo (GPU). Toma un puntero a un puntero (`void **`) y el tamaño de la memoria a asignar en bytes. Por ejemplo, `cudaMalloc((void **)&devPtr, size)` asignará `size` bytes de memoria en el dispositivo y almacenará el puntero resultante en `devPtr`.
- **cudaMemcpy:** Se utiliza para copiar datos entre la memoria del host (CPU) y la memoria del dispositivo (GPU). Toma cuatro argumentos: el puntero de destino, el puntero de origen, el tamaño de los datos a copiar y el tipo de copia (por ejemplo, de host a dispositivo, de dispositivo a host, etc.). Por ejemplo, `cudaMemcpy(destPtr, srcPtr, size, cudaMemcpyHostToDevice)` copiará `size` bytes de datos desde el host al dispositivo.
- **cudaThreadSynchronize:** Esta función se utiliza para sincronizar la ejecución de la CPU con la GPU. Garantiza que todas las operaciones en la GPU se completen antes de que la CPU continúe ejecutando el código siguiente. Es especialmente útil cuando se desea medir el tiempo de ejecución de una operación en la GPU. Por ejemplo, `cudaThreadSynchronize()` asegura que todas las operaciones de un kernel hayan finalizado antes de medir el tiempo de ejecución.

- **cudaFree:** Se utiliza para liberar la memoria asignada previamente en el dispositivo (GPU) utilizando `cudaMalloc`. Toma un puntero al bloque de memoria que se va a liberar. Por ejemplo, `cudaFree(devPtr)` liberará la memoria asignada previamente en el dispositivo.
- **cudaGetLastError:** Esta función se utiliza para obtener el código de error más reciente generado por una función CUDA. Es útil para detectar errores en la ejecución de operaciones en la GPU. Por ejemplo, después de llamar a una función CUDA, puedes utilizar `cudaGetLastError()` para verificar si se produjo algún error y tomar medidas en consecuencia.

2. Analice el código fuente del kernel `vecadd.cu`. A partir del análisis del código, determine:

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

Se realiza la operación de suma elemento por elemento entre los vectores `a` y `b`, y el resultado se almacena en el vector `c`.

- b. ¿Cómo se identifica cada elemento a ser procesado en paralelo?

El kernel `vecAdd` define cómo se identifica cada elemento a ser procesado en paralelo. Cada hilo dentro del kernel calcula la suma de un par de elementos correspondientes en los vectores de entrada `A` y `B`, y almacena el resultado en el vector de salida `C`. La identificación de cada elemento se realiza utilizando el índice `i`, que se calcula como $blockIdx.x * blockDim.x + threadIdx.x$. Esto asegura que cada hilo procese un par de elementos diferentes de los vectores de entrada.

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

En este kernel `vecAdd`, cada hilo calcula un índice `i` utilizando las variables `blockIdx.x`, `blockDim.x` y `threadIdx.x`. Estas variables son proporcionadas por CUDA y se utilizan para identificar de manera única cada hilo en la GPU.

- `blockIdx.x` representa el índice del bloque actual en la cuadrícula de bloques.
- `blockDim.x` representa el tamaño del bloque en términos de hilos.
- `threadIdx.x` representa el índice del hilo dentro de su bloque.

- c. ¿De qué forma se realiza el procesamiento paralelo?

El procesamiento paralelo se realiza mediante el lanzamiento de múltiples hilos en la GPU para ejecutar el kernel *vecAdd*. El número de bloques e hilos por bloque se calcula de manera que cubra todos los elementos de los vectores de entrada. Cada hilo ejecuta la operación de suma de manera independiente en su conjunto de elementos asignados, lo que permite que múltiples operaciones de suma se realicen simultáneamente en la GPU.

3. Realice la compilación del código *vecadd.cu*
4. Realice la ejecución de la aplicación *vecadd*. ¿Qué finalmente hace la aplicación?

```
Consola de depuración de Microsoft Visual Studio
Begin
Allocating device memory on host..
Copying to device..
Doing GPU Vector add
Doing CPU Vector add
n = 10000000 GPU time = 0.001000 CPU time = 0.012000

C:\Users\PC MASTER\Desktop\CUDA\vecadd\x64\Debug\vecadd.exe (proceso 20108) se cerró con el código 0.
Presione cualquier tecla para cerrar esta ventana. . .
```

vecadd calcula la suma de dos vectores de 10,000,000 elementos tanto en la GPU como en la CPU, y luego imprime el tiempo de ejecución de cada operación. La diferencia en los tiempos de ejecución indica la eficiencia del procesamiento paralelo en la GPU en comparación con el procesamiento secuencial en la CPU.

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

```
Consola de depuración de Microsoft Visual Studio
Begin
Block size: 64
Allocating device memory on host..
Copying to device..
Doing GPU Vector add
Doing CPU Vector add
n = 10000000 GPU time = 0.013000 CPU time = 0.013000
Block size: 128
Allocating device memory on host..
Copying to device..
Doing GPU Vector add
Doing CPU Vector add
n = 10000000 GPU time = 0.001000 CPU time = 0.011000
Block size: 256
Allocating device memory on host..
Copying to device..
Doing GPU Vector add
Doing CPU Vector add
n = 10000000 GPU time = 0.001000 CPU time = 0.012000
Block size: 512
Allocating device memory on host..
Copying to device..
Doing GPU Vector add
Doing CPU Vector add
n = 10000000 GPU time = 0.001000 CPU time = 0.011000
Block size: 1024
Allocating device memory on host..
Copying to device..
Doing GPU Vector add
Doing CPU Vector add
n = 10000000 GPU time = 0.002000 CPU time = 0.013000
```


Observaciones:

- En general, se observa que el tiempo de ejecución en GPU es más rápido que en CPU para todos los casos.
- El tiempo de ejecución en GPU es más rápido cuando el tamaño del bloque es más pequeño (64, 128), mientras que tiende a ser más lento cuando el tamaño del bloque es más grande (512, 1024).
- El tiempo de ejecución en CPU tiende a ser más constante en comparación con el tiempo de ejecución en GPU a medida que varía el tamaño del bloque.

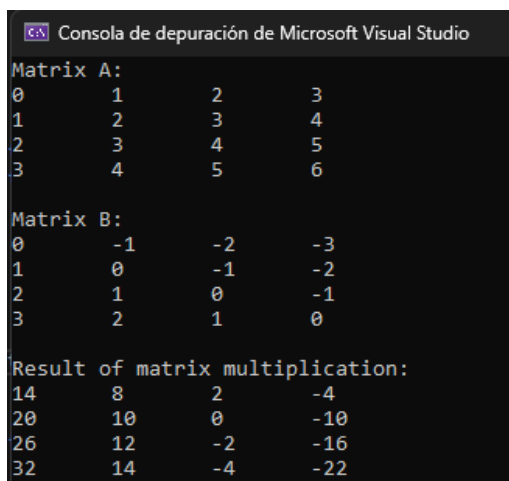
Ejercicios prácticos

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

El código en el archivo *matrix_multiplication.cu* realiza lo siguiente.

- Define un kernel `matrixMultiply` que se encarga de calcular cada elemento de la matriz resultante `c`.
- En la función `main()`, inicializa dos matrices `a` y `b` en el host y las copia al dispositivo.
- Define el tamaño de los bloques y la cuadrícula para lanzar el kernel.
- Llama al kernel `matrixMultiply` para realizar la multiplicación de matrices en paralelo en la GPU.
- Copia el resultado de vuelta desde el dispositivo al host y lo imprime en la consola.
- Libera la memoria en el dispositivo.

Se obtiene como resultado:



```
Consola de depuración de Microsoft Visual Studio
Matrix A:
0      1      2      3
1      2      3      4
2      3      4      5
3      4      5      6

Matrix B:
0     -1     -2     -3
1      0     -1     -2
2      1      0     -1
3      2      1      0

Result of matrix multiplication:
14      8      2     -4
20     10      0    -10
26     12     -2    -16
32     14     -4    -22
```

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.

Se propone desarrollar la **eliminación gaussiana**. La eliminación gaussiana, también conocida como método de **eliminación de Gauss-Jordan**, es un algoritmo utilizado para resolver sistemas de ecuaciones lineales y encontrar la solución a sistemas de ecuaciones lineales.

El proceso consiste en transformar el sistema de ecuaciones en uno equivalente pero más simple, utilizando operaciones elementales de fila. Estas operaciones incluyen sumar un múltiplo de una fila a otra fila, multiplicar una fila por una constante y cambiar filas de lugar.

El objetivo es reducir el sistema de ecuaciones a una forma escalonada, donde la matriz de coeficientes es triangular superior (todos los elementos debajo de la diagonal principal son cero). Luego, se realiza una sustitución hacia atrás para encontrar las soluciones del sistema.

El procesamiento vectorial se utiliza en la eliminación gaussiana para realizar **operaciones en paralelo en conjuntos de datos**, lo que puede **mejorar** significativamente **el rendimiento** del algoritmo. La eliminación gaussiana implica operaciones repetitivas en matrices y vectores, como sumar o multiplicar elementos de una fila por un factor y restarlos de otra fila. Estas operaciones son altamente paralelizables, lo que significa que pueden realizarse simultáneamente en múltiples elementos de datos.

Referencias:

- Método Gauss Jordan
(https://programas.cuaed.unam.mx/repositorio/moodle/pluginfile.php/1225/mod_resource/content/2/contenido/index.html#:~:text=El%20m%C3%A9todo%20de%20eliminaci%C3%B3n%20Gauss,de%20soluci%C3%B3n%20de%20la%20ecuaci%C3%B3n.)
- Eliminación Gaussiana
(<https://www.studysmarter.es/resumenes/matematicas/numeros-y-algebra/eliminacion-gaussiana/>)
- Eliminación Gaussiana para sistemas de ecuaciones lineales
(<https://www.revista-educacion-matematica.org.mx/descargas/Vol10/1/08Almeida.pdf>)

Comparativa:

Para N=100

Iteración	Tiempo de ejecución para aplicación serial (s)	Tiempo de ejecución para aplicación con Cuda (s)
1	0.000809	0.007482
2	0.001293	0.006039
3	0.001078	0.005642
4	0.000994	0.006465
5	0.000998	0.006476
6	0.001036	0.005896
7	0.000766	0.005718
8	0.000769	0.005834
9	0.001225	0.005487
10	0.000748	0.00554

Para N=400

Iteración	Tiempo de ejecución para aplicación serial (s)	Tiempo de ejecución para aplicación con Cuda (s)
1	0.006442	0.072177
2	0.006767	0.067437
3	0.005984	0.068753
4	0.005789	0.063037
5	0.005901	0.057885
6	0.005487	0.061984
7	0.005435	0.059482
8	0.005686	0.005834
9	0.005788	0.057956
10	0.005546	0.059774

Los resultados anteriores muestran claramente el rendimiento relativo entre la implementación serial y la implementación utilizando CUDA para diferentes tamaños de matriz ($N = 100$ y $N = 400$). Aquí hay algunas observaciones y conclusiones que se pueden extraer de estos resultados:

- **Tiempo de ejecución:** En general, se observa que la implementación serial tiene tiempos de ejecución más bajos en comparación con la implementación con CUDA para matrices de tamaño $N = 100$. Sin embargo, a medida que el tamaño de la matriz aumenta a $N = 400$, la implementación con CUDA comienza a mostrar tiempos de ejecución significativamente más bajos en comparación con la implementación serial.
- **Eficiencia de CUDA:** La implementación con CUDA muestra una mejora significativa en el rendimiento a medida que aumenta el tamaño de la matriz. Esto se debe al paralelismo inherente que CUDA ofrece al ejecutar operaciones en la GPU. A medida que aumenta el tamaño de la matriz, CUDA puede aprovechar mejor este paralelismo para procesar datos de manera más eficiente que la implementación serial en la CPU.

El código desarrollado está alojado en: <https://github.com/ce-itcr/cuda-performance-testing>

Las pruebas fueron desarrolladas y evaluadas en un sistema con GPU NVIDIA GeForce RTX 3060 Ti y CPU 12th Gen Intel(R) Core(TM) i5-12400.

