Instituto Politécnico Nacional

“Escuela Superior de Cómputo”

Integrantes:

- Diaz Ruiz Israel
- Ignacio Cortés Atzin Maxela
- Ríos Rivera Fernanda Anahí

Unidad de aprendizaje: Computo paralelo

Profesor: Luis Alberto Ibáñez Zamora

Grupo: 6BM2

“Reporte de práctica 2”

índice

Índice de figuras

Marco teórico

1. Computación Paralela y GPUs

La computación paralela es un modelo de procesamiento que permite ejecutar múltiples operaciones simultáneamente, lo que resulta útil para tareas que requieren gran capacidad de cálculo. Las unidades de procesamiento gráfico (GPU) están diseñadas específicamente para este tipo de procesamiento, ya que contienen miles de núcleos capaces de ejecutar tareas en paralelo de manera masiva. A diferencia de la CPU, que está optimizada para tareas secuenciales complejas, la GPU es ideal para operaciones matemáticas repetitivas y masivas, como la multiplicación de matrices, el procesamiento de imágenes y los algoritmos de inteligencia artificial.

2. CUDA: Plataforma de Cómputo en GPU

CUDA (Compute Unified Device Architecture) es una plataforma y modelo de programación desarrollada por NVIDIA que permite a los desarrolladores aprovechar la potencia de las GPU para ejecutar algoritmos generales, no limitados al ámbito gráfico. CUDA permite escribir programas en C, C++ o Python (mediante bibliotecas como PyCUDA) que se dividen en unidades de trabajo llamadas hilos, agrupadas en bloques y grids. Esto facilita el paralelismo masivo y el aprovechamiento de todos los recursos internos de la GPU.

3. Multiplicación de Matrices como Caso de Estudio

La multiplicación de matrices es una operación fundamental en muchas áreas como la computación científica, el aprendizaje automático, la visión por computadora y el procesamiento de señales. Su estructura repetitiva y predecible la hace especialmente apta para la ejecución paralela en una GPU. El código proporcionado utiliza CUDA para realizar esta operación en la GPU y compara el rendimiento con la ejecución en CPU mediante NumPy, lo que permite observar diferencias significativas en tiempo de ejecución para distintos tamaños de matriz.

4. Optimización con Memoria Compartida

En CUDA, existen varios tipos de memoria. Una de las más rápidas es la memoria compartida, que puede ser utilizada por los hilos de un mismo bloque para almacenar datos temporalmente. El uso de memoria compartida mejora el rendimiento al reducir la necesidad de acceder a la memoria global, que es más lenta. El código implementa una técnica conocida como tiling, que divide las matrices en bloques pequeños para cargarlos en memoria compartida y procesarlos localmente. Esta estrategia es común en algoritmos de alto rendimiento porque permite aprovechar mejor la arquitectura interna de la GPU.

5. Evaluación del Rendimiento

La comparación entre los tiempos de ejecución de la GPU y la CPU permite cuantificar la eficiencia del cómputo paralelo. Además, se valida que los resultados obtenidos por ambos métodos sean equivalentes en términos de precisión. Este tipo de evaluación es fundamental al desarrollar aplicaciones que buscan ser aceleradas por hardware.

Implementación del código

Implementar y analizar el rendimiento de la multiplicación de matrices cuadradas usando PyCUDA en dos contextos:

1. En la GPU utilizando memoria compartida.
2. En la CPU utilizando numpy.dot() como referencia.

El objetivo particular es medir y comparar los tiempos de ejecución y la precisión de los resultados, además de familiarizarse con el modelo de memoria de CUDA y el flujo de programación PyCUDA.

Objetivos específicos:

1. Implementar un kernel CUDA que:

- Use memoria compartida para cargar submatrices (tiles) de A y B.
- Calcule la multiplicación matricial $C=A \cdot B$ en la GPU.

2. Reservar memoria en la GPU, transferir las matrices de entrada A y B desde la CPU, y copiar el resultado C de vuelta a la CPU.

3. Ejecutar y medir los siguientes tiempos de ejecución:

- Tiempo de compilación del kernel CUDA.
- Tiempo de asignación de memoria en la GPU.
- Tiempo de transferencia CPU → GPU (A y B).
- Tiempo de ejecución del kernel CUDA.
- Tiempo de transferencia GPU → CPU (C).
- Tiempo de ejecución de numpy.dot(A, B) en CPU.

4. Calcular el error numérico máximo entre el resultado obtenido en GPU y el de NumPy para validar la exactitud de la implementación.

5. Variar el tamaño de la matriz (por ejemplo: 32×32, 128×128, 512×512) y observar el impacto en tiempos y errores.

Este código realiza la multiplicación de matrices cuadradas utilizando tanto la CPU como la GPU para comparar su rendimiento. Empleando PyCUDA, se define un kernel CUDA que implementa la multiplicación de matrices con memoria compartida (shared memory) para optimizar el acceso a los datos. Para cada tamaño definido en la lista sizes, se generan dos matrices aleatorias A y B, y se multiplica A×B tanto en la GPU como en la CPU. En la GPU, los datos se transfieren a la memoria del dispositivo, se ejecuta el kernel con una configuración basada en bloques de 16×16 hilos, y luego se recupera el resultado. En paralelo, se calcula la misma operación con NumPy en la CPU. Finalmente, se mide y registra el tiempo de ejecución de cada enfoque y el error máximo entre los resultados, mostrando una tabla comparativa que permite evaluar la eficiencia del cómputo en GPU frente al procesamiento tradicional en CPU.

```python
import pycuda.autoinit
import pycuda.driver as cuda
import numpy as np
from pycuda.compiler import SourceModule
import time
import pandas as pd

TILE_WIDTH = 16
sizes = [128, 256, 512]  # Puedes agregar más tamaños

kernel_template = """
#define TILE_WIDTH {TILE_WIDTH}
__global__ void MatrixMulShared(float *A, float *B, float *C, int width)
{{
    __shared__ float A_tile[TILE_WIDTH][TILE_WIDTH];
    __shared__ float B_tile[TILE_WIDTH][TILE_WIDTH];

    int row = blockIdx.y * TILE_WIDTH + threadIdx.y;
    int col = blockIdx.x * TILE_WIDTH + threadIdx.x;
    float Pvalue = 0;

    for (int t = 0; t < (width + TILE_WIDTH - 1)/TILE_WIDTH; ++t) {{
        if (row < width && t * TILE_WIDTH + threadIdx.x < width)
            A_tile[threadIdx.y][threadIdx.x] = A[row * width + t * TILE_WIDTH + threadIdx.x];
        else
            A_tile[threadIdx.y][threadIdx.x] = 0;

        if (col < width && t * TILE_WIDTH + threadIdx.y < width)
            B_tile[threadIdx.y][threadIdx.x] = B[(t * TILE_WIDTH + threadIdx.y) * width + col];
        else
            B_tile[threadIdx.y][threadIdx.x] = 0;

        __syncthreads();

        for (int k = 0; k < TILE_WIDTH; ++k)
            Pvalue += A_tile[threadIdx.y][k] * B_tile[k][threadIdx.x];
```

Figura 1 Se observan las librerías

```
results = []

for MATRIX_SIZE in sizes:
    A = np.random.randn(MATRIX_SIZE, MATRIX_SIZE).astype(np.float32)
    B = np.random.randn(MATRIX_SIZE, MATRIX_SIZE).astype(np.float32)
    C = np.zeros_like(A)

    mod = SourceModule(kernel_template.format(TILE_WIDTH=TILE_WIDTH))
    matrixmul = mod.get_function("MatrixMulShared")

    A_gpu = cuda.mem_alloc(A.nbytes)
    B_gpu = cuda.mem_alloc(B.nbytes)
    C_gpu = cuda.mem_alloc(C.nbytes)

    cuda.memcpy_htod(A_gpu, A)
    cuda.memcpy_htod(B_gpu, B)

    grid_dim = (MATRIX_SIZE // TILE_WIDTH, MATRIX_SIZE // TILE_WIDTH, 1)
    block_dim = (TILE_WIDTH, TILE_WIDTH, 1)

    start_gpu = time.time()
    matrixmul(A_gpu, B_gpu, C_gpu, np.int32(MATRIX_SIZE), block=block_dim, grid=grid_dim)
    cuda.Context.synchronize()
    end_gpu = time.time()

    cuda.memcpy_dtoh(C, C_gpu)

    start_cpu = time.time()
    C_cpu = np.dot(A, B)
    end_cpu = time.time()

    error = np.max(np.abs(C - C_cpu))
```

Figura 2 Se crea la matriz

```
    end_gpu = time.time()

    cuda.memcpy_dtoh(C, C_gpu)

    start_cpu = time.time()
    C_cpu = np.dot(A, B)
    end_cpu = time.time()

    error = np.max(np.abs(C - C_cpu))

    results.append({
        "Tamaño": f"{MATRIX_SIZE}×{MATRIX_SIZE}",
        "Tiempo GPU (ms)": round((end_gpu - start_gpu) * 1000, 3),
        "Tiempo CPU (ms)": round((end_cpu - start_cpu) * 1000, 3),
        "Error máximo": f"{error:.2e}"
    })

# Mostrar tabla
df = pd.DataFrame(results)
print(df.to_string(index=False))
```

```
  Tamaño  Tiempo GPU (ms)  Tiempo CPU (ms) Error máximo
128×128           0.086            0.158     0.00e+00
256×256           0.172            0.466     0.00e+00
512×512           0.837            3.975     1.14e-04
```

Figura 3 Resultado

La implementación demuestra de forma efectiva cómo el uso de CUDA y memoria compartida en la GPU puede acelerar significativamente la multiplicación de matrices en comparación con la CPU, especialmente a medida que crece el tamaño de las matrices. Gracias al uso de "tiles" y sincronización eficiente entre hilos, el kernel optimizado logra reducir el tiempo de cómputo manteniendo una alta precisión, con errores numéricos mínimos debidos a la aritmética de punto flotante. Esta comparación valida el potencial del cómputo paralelo en GPU para tareas intensivas en datos, como operaciones matriciales, siendo una estrategia clave para aplicaciones científicas, gráficas o de aprendizaje profundo.

Colab

https://colab.research.google.com/drive/1VovM8T2yeNc420bxDJsIPLHq1SZLsfwb?usp=sharing

Cuestionario

1. ¿Qué es un Streaming Multiprocessor (SM) en la arquitectura CUDA y cuál es su función dentro de la GPU?

Un SM es una unidad de cómputo dentro de una GPU NVIDIA que agrupa varios núcleos CUDA (SP) y unidades de gestión de memoria y ejecución. Su función principal es ejecutar múltiples hilos simultáneamente, gestionando la ejecución, la memoria compartida y el despacho eficiente de instrucciones

2. ¿Qué diferencia existe entre un SM y un SP (Scalar Processor o núcleo CUDA) y cómo se relacionan?

Un SM es un bloque funcional que contiene múltiples SP (CUDA Cores). Los SP son los ejecutores que realizan operaciones aritméticas o lógicas en cada hilo. Mientras el SM organiza la ejecución en grupos (warps), los SP procesan las instrucciones del warp.

3. ¿Cómo puedes inspeccionar desde Python (usando PyCUDA) las capacidades de tu GPU, como número de SMs, tamaño de memoria compartida o hilos por bloque?

Con PyCUDA puedes llamar a cuda.Device(idx).get_attributes(), obteniendo atributos como MULTIPROCESSOR_COUNT, MAX_SHARED_MEMORY_PER_BLOCK, MAX_THREADS_PER_BLOCK, entre otros. Esto permite conocer cuántos SMs hay, cuanta memoria compartida y hilos máximos por bloque.

4. ¿Qué es el "compute capability" de una GPU y cómo influye en las funcionalidades que puedes utilizar?

El *compute capability* (versión de arquitectura CUDA) indica las funcionalidades soportadas: número de registros, soporte a memoria unificada, tipos de sincronia, tensor cores, etc. Determina qué características y optimizaciones puede usar tu kernel, así como compatibilidad con herramientas y librerías.

5. Explica qué es el tamaño de **warp** y qué impacto tiene sobre la eficiencia de ejecución de los hilos.

El warp en CUDA es una unidad de ejecución compuesta por 32 hilos que se ejecutan *simultáneamente* bajo el mismo PC (program counter) en un SM (Streaming Multiprocessor). Todos los hilos dentro de un warp comparten la misma instrucción en un ciclo de reloj, lo que permite aprovechar el paralelismo SIMT (Single Instruction Multiple Thread). Una mala configuración del tamaño de bloque puede saturar o desperdiciar recursos, impactando directamente en la eficiencia del kernel.

6. ¿Qué herramientas ofrece NVIDIA para analizar el rendimiento de los kernels CUDA y la ocupación de recursos como SMs y warps?

NVIDIA proporciona un conjunto de herramientas potentes para analizar y optimizar el rendimiento de kernels CUDA y la ocupación de recursos como SMs y warps. En el nivel de kernel, Nsight Compute permite un perfilado detallado con métricas precisas sobre latencia, uso de registros, optimización de memoria, eficiencia de warp/SM, y ofrece análisis guiados por expertos con sugerencias automáticas. A nivel de aplicación, Nsight Systems brinda un perfilado holístico que abarca CPU, GPU, transferencias de memoria, trazas de eventos CUDA/NVTX y sincronización entre procesos, ideal para identificar cuellos de botella globales. Herramientas históricas como nvprof y Visual Profiler aún permiten medir métricas clave como sm_efficiency (tiempo de SM activo) y warp_execution_efficiency (uso efectivo de hilos dentro de warps). Además, la API CUPTI posibilita análisis personalizado dentro de las aplicaciones, integrándose con estas herramientas para extraer métricas específicas en tiempo de ejecución. En conjunto, este ecosistema hace posible diagnosticar tanto problemas de divergencia de warps y baja ocupación, como ineficiencias en memoria o latencia, para maximizar el rendimiento en GPU.

7. ¿Qué papel juega la **memoria compartida** dentro de un SM, y por qué su uso puede mejorar el rendimiento de un kernel?

La memoria compartida es un buffer de baja latencia dentro de cada SM que permite que los hilos de un mismo bloque compartan datos sin pasar por la memoria global, reduciendo latencias y tráfico. Cuando se usa correctamente (por ejemplo, en multiplicación con "tiles"), mejora significativamente el rendimiento.

8. ¿Cómo evoluciona la arquitectura CUDA en las familias más recientes de NVIDIA (Ampere, Hopper, Blackwell)? Menciona al menos dos innovaciones.

- Tensor Cores especializados para cálculos de IA (Ampere+).
- NVLink más rápido y mayor ancho de banda HBM.
- Soporte nativo a *cooperative groups* y grid-wide sync en Hopper/H100/Blackwell

9. ¿Qué ventajas ofrece la ejecución de kernels cooperativos (cooperative groups) y en qué escenarios se usarían?

Permiten sincronizar hilos dentro de subgrupos arbitrarios o incluso entre bloques e incluso en múltiples GPUs (*multi-grid*). Útiles para algoritmos como reducciones globales, persistentes o paralelización dinámica.Por ejemplo, permiten realizar operaciones globales en una sola invocación, evitando múltiples kernel-launches.

10. Imagina que estás desarrollando un algoritmo que requiere muchos accesos a memoria. ¿Qué atributos de la GPU inspeccionarías para asegurarte de que es adecuada?

Si estás desarrollando un algoritmo que requiere muchos accesos a memoria, es fundamental evaluar atributos clave de la GPU que impactan directamente en el rendimiento. Debes considerar el ancho de banda de la memoria global, ya que un mayor ancho permite mover más datos por segundo, así como el tamaño de las cachés L1 y L2, que reducen la latencia de acceso a datos reutilizados. La memoria compartida por SM también es crucial, pues permite una comunicación rápida entre hilos del mismo bloque. Además, una alta cantidad de registros por hilo evita recurrir a memoria local lenta, y una buena capacidad de hilos y warps activos por SM ayuda a ocultar latencias mediante paralelismo masivo. Finalmente, tecnologías modernas como transferencias asíncronas (TMA) entre memoria global y compartida, presentes en arquitecturas como Hopper, optimizan el flujo de datos y mejoran la eficiencia general del algoritmo. Evaluar estos factores te permite seleccionar una GPU adecuada para cargas de trabajo intensivas en memoria.

Referencias

- Kim, J. Y. (2007). Computación paralela en unidades de procesamiento gráfico (GPU).
- Gong, S. (2025, marzo 3). *Streaming Multiprocessor (SM)*. Recuperado de Steven Gong.
- NVIDIA Corporation. (2024). *CUDA C++ Programming Guide*. Recuperado de la documentación oficial de NVIDIA.
- Jiang, Y. (2024, enero 29). *Understanding Streaming Multiprocessors (SM), blocks, threads and warps in CUDA programming*. *Medium*.
- Cornell Virtual Workshop. (s. f.). *GPU Characteristics – Compute Capability*. Cornell University.
- Modal. (s. f.). *What is a Streaming Multiprocessor? | GPU Glossary*. Modal.
- Stack Overflow. (2015, diciembre). Difference between cuda core & streaming multiprocessor [Mensaje en foro]. En *NVIDIA Developer Forums*.