Instituto Politécnico Nacional

“Escuela Superior de Cómputo”

Alumno:

- Ignacio Cortés Atzin Maxela

Unidad de aprendizaje: Cómputo paralelo

Profesor: Luis Alberto Ibáñez Zamora

Grupo: 6BM2

“Reporte de práctica”

Índice

Comparación de tipos de memoria en CUDA

En CUDA, el modelo de memoria distingue entre varios tipos, siendo los más importantes: los registros (privados por hilo), la memoria global (accesible por todos pero lenta), la memoria constante (de solo lectura) y la memoria compartida (rápida y accesible solo dentro de un bloque de hilos). La diferencia clave entre la memoria global y la memoria compartida radica en la velocidad y en la eficiencia del acceso a datos: la global es más lenta y produce redundancia cuando varios hilos acceden al mismo dato, mientras que la compartida permite almacenar localmente bloques de datos, reduciendo accesos redundantes y mejorando el rendimiento.

Estrategia de optimización para multiplicación de matrices

Se propone usar memoria compartida para realizar multiplicación de matrices más eficiente. Cada bloque de hilos carga submatrices (tiles) de A y B en memoria compartida. Posteriormente, los hilos utilizan estos datos almacenados localmente para realizar los cálculos de forma más rápida, sincronizando entre etapas para garantizar coherencia. Esta estrategia reduce la necesidad de múltiples accesos a la memoria global y mejora considerablemente el rendimiento computacional.

Implementación en PyCUDA y validación

El código presentado genera dos matrices aleatorias A y B, compila un kernel en C usando PyCUDA, y lo ejecuta en la GPU utilizando memoria compartida (__shared__). Cada hilo se encarga de un elemento de la matriz resultante C. Al finalizar, se recupera la matriz C desde la GPU y se compara con la multiplicación equivalente realizada en CPU usando numpy.dot. Se mide el error entre ambos resultados, el cual es mínimo, debido a diferencias menores por el orden de operaciones en punto flotante, lo cual valida la corrección del cálculo en GPU.

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

# Tamaño de la matriz (n x n)
TILE_WIDTH = 16
MATRIX_SIZE = 32  # Debe ser múltiplo de TILE_WIDTH
assert MATRIX_SIZE % TILE_WIDTH == 0

# Inicializa matrices de entrada
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)

# Kernel CUDA con memoria compartida
kernel_code = f"""
#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];
```

```
            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];

        __syncthreads();
    }}

    if (row < width && col < width)
        C[row * width + col] = Pvalue;
}}
"""

# Compilación del kernel
mod = SourceModule(kernel_code)
matrixmul = mod.get_function("MatrixMulShared")

# Reservar memoria en la GPU
A_gpu = cuda.mem_alloc(A.nbytes)
B_gpu = cuda.mem_alloc(B.nbytes)
C_gpu = cuda.mem_alloc(C.nbytes)

# Copiar datos a la GPU
cuda.memcpy_htod(A_gpu, A)
cuda.memcpy_htod(B_gpu, B)

# Ejecutar kernel
grid_dim = (MATRIX_SIZE // TILE_WIDTH, MATRIX_SIZE // TILE_WIDTH, 1)
block_dim = (TILE_WIDTH, TILE_WIDTH, 1)

matrixmul(A_gpu, B_gpu, C_gpu, np.int32(MATRIX_SIZE),
          block=block_dim, grid=grid_dim)

# Copiar resultado de vuelta
cuda.memcpy_dtoh(C, C_gpu)

# Verificar resultado con NumPy
C_cpu = np.dot(A, B)
error = np.max(np.abs(C - C_cpu))

print("Error máximo GPU vs NumPy:", error)
```

```
Error máximo GPU vs NumPy: 2.861023e-06
```

Primero, el programa importa las bibliotecas necesarias: pycuda para interactuar con la GPU, y numpy para generar y manipular matrices en la CPU. Define el tamaño de las matrices (32×32), asegurándose de que sea múltiplo del tamaño de bloque (TILE_WIDTH = 16), que es el tamaño del fragmento (tile) que se cargará en memoria compartida.

A continuación, se inicializan dos matrices A y B con valores aleatorios y se crea una matriz C vacía para almacenar el resultado. Luego se define un **kernel CUDA**, es decir, una función escrita en C que se ejecutará en la GPU. Este kernel realiza la multiplicación de matrices cargando submatrices (tiles) de A y B en la **memoria compartida** (más rápida que la memoria global), para reducir accesos repetidos a memoria lenta. Cada hilo de GPU calcula un solo elemento de la matriz resultado C, y se sincroniza con los demás hilos del bloque con __syncthreads().

Una vez definido el kernel, se compila con SourceModule y se obtienen los punteros a la función CUDA. Se reserva memoria en la GPU para las tres matrices y se copian los datos de A y B desde la CPU. Luego, se lanza el kernel con una configuración de bloques y subprocesos que cubren toda la matriz. Cuando la ejecución termina, se copia el resultado C de vuelta a la CPU.

Finalmente, se calcula el producto de A y B en CPU usando numpy.dot() y se compara con

el resultado obtenido en la GPU. Se imprime el **error máximo absoluto** entre ambos resultados, que normalmente será muy pequeño, confirmando que la implementación en GPU es correcta.

```
[ ] import pycuda.autoinit
    import pycuda.driver as cuda
    import numpy as np
    import time
    from pycuda.compiler import SourceModule

    # Parámetros
    TILE_WIDTH = 16
    MATRIX_SIZE = 5000  # Ya no importa si no es múltiplo de TILE_WIDTH

    # Crear matrices
    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)

    # Kernel CUDA con manejo de bordes
    kernel_code = f"""
    #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];
```

```
# Compilar kernel
mod = SourceModule(kernel_code)
matrixmul = mod.get_function("MatrixMulShared")

# Reservar memoria en GPU
A_gpu = cuda.mem_alloc(A.nbytes)
B_gpu = cuda.mem_alloc(B.nbytes)
C_gpu = cuda.mem_alloc(C.nbytes)

# Copiar datos
cuda.memcpy_htod(A_gpu, A)
cuda.memcpy_htod(B_gpu, B)

# Medir tiempo en GPU
start_gpu = cuda.Event()
end_gpu = cuda.Event()
start_gpu.record()

# Lanzar kernel
grid_x = (MATRIX_SIZE + TILE_WIDTH - 1) // TILE_WIDTH
grid_y = (MATRIX_SIZE + TILE_WIDTH - 1) // TILE_WIDTH
grid_dim = (grid_x, grid_y, 1)
block_dim = (TILE_WIDTH, TILE_WIDTH, 1)

matrixmul(A_gpu, B_gpu, C_gpu, np.int32(MATRIX_SIZE),
          block=block_dim, grid=grid_dim)

end_gpu.record()
end_gpu.synchronize()
gpu_time_ms = start_gpu.time_till(end_gpu)  # Tiempo en ms

# Copiar resultado de vuelta
cuda.memcpy_dtoh(C, C_gpu)

# Medir tiempo en CPU
start_cpu = time.time()
C_cpu = np.dot(A, B)
end_cpu = time.time()
cpu_time_s = end_cpu - start_cpu
```

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

# Resultados
print(f"Error máximo GPU vs NumPy: {error:.5f}")
print(f"Tiempo GPU: {gpu_time_ms:.2f} ms")
print(f"Tiempo CPU: {cpu_time_s:.2f} s")
```

```
/usr/local/lib/python3.11/dist-packages/google/colab/_variable_inspector.py:27: UserWarning: module in out-of-thread context could not be cleaned up
  globals().clear()
/usr/local/lib/python3.11/dist-packages/google/colab/_variable_inspector.py:27: UserWarning: device_allocation in out-of-thread context could not be cleaned up
  globals().clear()
Error máximo GPU vs NumPy: 0.00134
Tiempo GPU: 482.90 ms
Tiempo CPU: 4.07 s
```

Este programa implementa la multiplicación de dos matrices cuadradas de gran tamaño (5000 × 5000) utilizando CUDA en la GPU a través de PyCUDA. Su propósito principal es comparar el rendimiento y la precisión entre una implementación paralela en GPU y una versión secuencial en CPU. Para ello, se crean matrices aleatorias A y B, y una matriz C vacía para almacenar el resultado. La dimensión de las matrices ya no necesita ser múltiplo exacto del tamaño del bloque (TILE_WIDTH = 16) gracias a una técnica que maneja correctamente los bordes. El núcleo de la operación está en el kernel CUDA definido como una cadena en lenguaje C, el cual divide la tarea en bloques y subprocesos.

Cada bloque trabaja con una submatriz (tile) de A y B, que se carga en memoria compartida, lo que reduce significativamente los accesos a la memoria global (que es más lenta). Durante la ejecución, cada hilo calcula una celda del resultado acumulando productos parciales, y se asegura de sincronizar los hilos para evitar errores de concurrencia. Además, se incluyen

validaciones para evitar accesos fuera de los límites de la matriz, asignando ceros en esos casos.
El código reserva memoria en la GPU para las matrices, copia los datos desde la CPU y lanza el kernel CUDA con una cuadrícula de bloques calculada dinámicamente. Se mide el tiempo de ejecución en GPU mediante eventos CUDA, y al finalizar, se transfiere la matriz resultante C desde la GPU de vuelta a la CPU. A modo de comparación, también se realiza la misma multiplicación en CPU usando numpy.dot() y se mide su tiempo.

Finalmente, el programa calcula el error máximo absoluto entre las dos versiones del resultado para validar que ambas entregan resultados casi idénticos, pese a diferencias en orden de operaciones y redondeo. Los tiempos obtenidos demuestran que la GPU es mucho más rápida, lo cual evidencia el poder del cómputo paralelo en tareas matriciales pesadas. Esta implementación es un ejemplo práctico del uso eficiente de CUDA con manejo de bordes, lo que permite trabajar con matrices de cualquier tamaño sin comprometer la exactitud ni el rendimiento.

Este valor representa la diferencia máxima entre los resultados calculados por la GPU (con PyCUDA) y por la CPU (usando numpy.dot). Aunque no es exactamente cero, este error es muy pequeño y aceptable, considerando que ambas plataformas usan operaciones de punto flotante (float32), las cuales pueden producir ligeras variaciones debido al orden en que se realizan las operaciones. Un error de 0.00134 en matrices de 5000×5000 es insignificante en términos prácticos.

La ejecución de la multiplicación en GPU tomó menos de medio segundo, lo que demuestra el altísimo nivel de paralelismo alcanzado con CUDA. Cada hilo de GPU trabaja sobre una parte de la matriz, aprovechando miles de núcleos de procesamiento y la memoria compartida para acelerar los cálculos.

Conclusión

Este programa compara el rendimiento de una multiplicación de matrices realizada en GPU (con memoria compartida optimizada) frente a la misma operación realizada en CPU usando NumPy. La implementación en GPU permite acelerar el proceso aprovechando la alta paralelización y acceso rápido a memoria compartida, lo que se refleja en un tiempo de ejecución mucho menor. Además, la precisión del cálculo en GPU es muy alta, con un error numérico despreciable respecto a la versión en CPU.

Este ejercicio demuestra la eficiencia y validez del cómputo paralelo con CUDA, especialmente para operaciones matriciales comunes en áreas como gráficos, simulación y aprendizaje profundo.