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 3”

índice

índice 2
Índice de figuras 2
Marco teórico 3
Implementación del código 3
Estimar π con Monte Carlo 3
Objetivos Específicos 3
Puntos para desarrollar 4
Parámetros sugeridos 4
Cuestionario 9
Referencias 11

Índice de figuras

Figura 1 Se importan los módulos necesarios 5
Figura 2 Ejecución del kernel 5
Figura 3 Implementacion en CPU 6
Figura 4 Continuación del código 7
Figura 5 Continuación del código 8
Figura 6 Resultados 8

Marco teórico

El método de Monte Carlo es una técnica probabilística ampliamente utilizada para resolver problemas numéricos complejos mediante simulaciones aleatorias. En particular, puede emplearse para estimar el valor del número π considerando que el área de un cuarto de círculo unitario inscrito en un cuadrado de lado 1 se aproxima a π/4. Al generar puntos aleatorios dentro de ese cuadrado y contar cuántos caen dentro del círculo, es posible estimar π con la fórmula: $\pi \approx 4 \times$ (puntos dentro del círculo) / (puntos totales).

En este programa, se implementa el método de Monte Carlo en la GPU utilizando **CUDA a través de PyCUDA**, una interfaz que permite compilar y ejecutar kernels CUDA directamente desde Python. Se define un kernel llamado monteCarloPi, que aprovecha funciones de generación aleatoria de CUDA (curand_uniform) para generar puntos (x, y) en el dominio $[0,1]\times[0,1]$ y cuenta cuántos están dentro del círculo unidad. El resultado parcial de cada hilo se acumula de forma segura utilizando la operación atómica atomicAdd, evitando condiciones de carrera al sumar los resultados al contador global.

El uso de múltiples **bloques** y **hilos** (en este caso 128 bloques de 256 hilos) permite ejecutar millones de simulaciones en paralelo, logrando una alta eficiencia de cómputo. Además, se mide el tiempo de ejecución en la GPU para demostrar su rendimiento comparado con métodos tradicionales. Esta implementación es un claro ejemplo del uso de cómputo paralelo para resolver problemas matemáticos de forma eficiente mediante técnicas estocásticas y programación de GPU.

Implementación del código

Estimar π con Monte Carlo

Implementar el algoritmo de Monte Carlo para estimar el valor de π, ejecutándolo de manera paralela en GPU mediante CUDA, y evaluar el rendimiento y la eficiencia de ejecución usando herramientas de análisis como Nsight Compute.

Objetivos Específicos

- Comprender la paralelización del método de Monte Carlo.
- Implementar kernels CUDA usando múltiples hilos.
- Usar generadores de números pseudoaleatorios en GPU.
- Analizar la ocupación de recursos de hardware con Nsight.
- Comparar rendimiento vs. versión secuencial en CPU.

Descripción del Problema

Se quiere estimar el valor de π mediante el método de Monte Carlo, que consiste en lanzar puntos aleatorios dentro de un cuadrado de lado 1. Si un punto aleatorio cae dentro del círculo de radio 1 centrado en el origen, se cuenta como válido. El valor estimado de π se calcula como:

$$\pi \approx 4 \cdot \frac{\text{puntos dentro del círculo}}{\text{total de puntos generados}}$$

Puntos para desarrollar

A. **Escribir un kernel CUDA** donde:

- Cada hilo genera varios pares (x,y)(x,y)(x,y) aleatorios.
- Cuenta cuántos puntos caen dentro del círculo.
- Usa atomicAdd() para acumular resultados en memoria global.

B. **Compilar y ejecutar el código CUDA**, midiendo el tiempo de ejecución para distintas cantidades de puntos.

C. **Comparar con una implementación secuencial en CPU** y anotar diferencias de tiempo y precisión.

D. Analizar **con Nsight Compute**:

- Uso de SMs
- Warp occupancy
- Divergencia de warps
- Tiempo de ejecución

Parámetros sugeridos

- Total, de puntos: 10610^6106, 10710^7107, 10810^8108
- Puntos por hilo: 1,000 – 10,000
- Tamaño de bloque: 128 – 512 hilos
- Usa curand o valores precargados en CPU (si se hace en PyCUDA)

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

# Código CUDA como string
cuda_code = """
#include <curand_kernel.h>

extern "C"
__global__ void monteCarloPi(int *count, int samples, unsigned long seed) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    curandState state;
    curand_init(seed, tid, 0, &state);

    int local_count = 0;
    for (int i = 0; i < samples; ++i) {
        float x = curand_uniform(&state);
        float y = curand_uniform(&state);
        if (x*x + y*y <= 1.0f) local_count++;
    }

    atomicAdd(count, local_count);
}
"""
```

Figura 1 Se importan los módulos necesarios

```python
# Compilar el kernel
mod = SourceModule(cuda_code, no_extern_c=True, options=["--use_fast_math"])
monteCarloPi = mod.get_function("monteCarloPi")

# Parámetros
threads = 256
blocks = 128
samples_per_thread = 10000
total_samples = threads * blocks * samples_per_thread

# Inicializar memoria en GPU
count_gpu = cuda.mem_alloc(np.int32().nbytes)
cuda.memcpy_htod(count_gpu, np.int32(0))

# Medir tiempo GPU
start_gpu = time.time()

# Ejecutar kernel
monteCarloPi(
    count_gpu,
    np.int32(samples_per_thread),
    np.uint64(int(time.time())),
    block=(threads, 1, 1),
    grid=(blocks, 1)
)
cuda.Context.synchronize()

end_gpu = time.time()

# Recuperar resultado
h_count = np.empty(1, dtype=np.int32)
cuda.memcpy_dtoh(h_count, count_gpu)

# Calcular Pi
pi_gpu = 4.0 * h_count[0] / total_samples
print(f"[GPU] Pi ~ {pi_gpu:.6f} en {1000*(end_gpu - start_gpu):.2f} ms")
```

```
[GPU] Pi ~ 3.141564 en 5.20 ms
```

Figura 2 Ejecución del kernel

Esta parte del código realiza la ejecución del kernel CUDA desde Python y calcula una estimación del número π utilizando la GPU.

```
[ ]  # Versión CPU
     start_cpu = time.time()

     count_cpu = 0
     for _ in range(total_samples):
         x = np.random.rand()
         y = np.random.rand()
         if x*x + y*y <= 1.0:
             count_cpu += 1

     end_cpu = time.time()
     pi_cpu = 4.0 * count_cpu / total_samples
     print(f"[CPU] Pi ~ {pi_cpu:.6f} en {1000*(end_cpu - start_cpu):.2f} ms")
```

```
[CPU] Pi ~ 3.141711 en 397582.26 ms
```

Figura 3 Implementación en CPU

Este bloque implementa el mismo algoritmo de Monte Carlo que la versión en GPU, pero usando un solo hilo en la CPU. Su objetivo principal es comparar rendimiento (tiempo de ejecución) y precisión con la versión paralelizada en GPU, mostrando las ventajas del cómputo paralelo para tareas intensivas en simulaciones. El resultado puede mejorar si se usara nsight compute de manera local.

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

    cuda_code = """
    #include <curand_kernel.h>

    extern "C"
    __global__ void monteCarloPi(int *count, int samples, unsigned long seed) {
        int tid = blockIdx.x * blockDim.x + threadIdx.x;
        curandState state;
        curand_init(seed, tid, 0, &state);

        int local_count = 0;
        for (int i = 0; i < samples; ++i) {
            float x = curand_uniform(&state);
            float y = curand_uniform(&state);
            if (x*x + y*y <= 1.0f) local_count++;
        }

        atomicAdd(count, local_count);
    }
    """

    mod = SourceModule(cuda_code, no_extern_c=True)
    monteCarloPi = mod.get_function("monteCarloPi")

    # Parámetros a probar
    threads_list = [64, 128, 256]
    blocks_list = [64, 128]
    samples_per_thread_list = [1000, 5000]

    results = []
```

Figura 4 Continuación del código

```
for threads in threads_list:
    for blocks in blocks_list:
        for samples_per_thread in samples_per_thread_list:
            total_samples = threads * blocks * samples_per_thread

            # GPU
            count_gpu = cuda.mem_alloc(np.int32().nbytes)
            cuda.memcpy_htod(count_gpu, np.int32(0))
            start_gpu = time.time()

            monteCarloPi(
                count_gpu,
                np.int32(samples_per_thread),
                np.uint64(int(time.time())),
                block=(threads, 1, 1),
                grid=(blocks, 1)
            )
            cuda.Context.synchronize()
            end_gpu = time.time()

            h_count = np.empty(1, dtype=np.int32)
            cuda.memcpy_dtoh(h_count, count_gpu)
            pi_gpu = 4.0 * h_count[0] / total_samples
            time_gpu = 1000 * (end_gpu - start_gpu)

            # CPU
            start_cpu = time.time()
            count_cpu = 0
            for _ in range(total_samples):
                x = np.random.rand()
                y = np.random.rand()
                if x*x + y*y <= 1.0:
                    count_cpu += 1
            end_cpu = time.time()
            pi_cpu = 4.0 * count_cpu / total_samples
            time_cpu = 1000 * (end_cpu - start_cpu)
```

```python
            results.append({
                "Threads": threads,
                "Blocks": blocks,
                "Samples/Thread": samples_per_thread,
                "Total Samples": total_samples,
                "GPU Time (ms)": round(time_gpu, 2),
                "GPU Pi": round(pi_gpu, 6),
                "CPU Time (ms)": round(time_cpu, 2),
                "CPU Pi": round(pi_cpu, 6),
            })

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

Figura 5 Continuación del código

```
/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()
/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()
/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()
/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()
/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()
/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()
/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()
/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()
/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()
/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()
/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()
 Threads  Blocks  Samples/Thread  Total Samples  GPU Time (ms)   GPU Pi  CPU Time (ms)   CPU Pi
      64      64            1000        4096000           3.05 3.141737        5429.30 3.141175
      64      64            5000       20480000           0.71 3.141180       25151.75 3.141982
      64     128            1000        8192000           0.56 3.142245       10082.09 3.142512
      64     128            5000       40960000           0.97 3.141639       49153.91 3.141151
     128      64            1000        8192000           0.70 3.142698       10210.06 3.140197
     128      64            5000       40960000           1.03 3.141721       49397.12 3.141295
     128     128            1000       16384000           0.94 3.141367       20214.63 3.142035
     128     128            5000       81920000           1.68 3.141415       99829.96 3.141663
     256      64            1000       16384000           1.03 3.141542       19753.88 3.141754
     256      64            5000       81920000           1.78 3.141574      100757.50 3.141770
     256     128            1000       32768000           1.67 3.142063       39255.66 3.141349
     256     128            5000      163840000           3.00 3.141391      198307.55 3.141642
```

Figura 6 Resultados

Los resultados muestran una comparación clara entre el rendimiento de la GPU y la CPU al calcular π mediante el método de Monte Carlo, bajo distintas configuraciones de hilos, bloques y muestras. Se puede observar que la GPU es consistentemente más rápida que la CPU, con tiempos de ejecución que van desde menos de 1 ms hasta apenas 3 ms, incluso en configuraciones con más de 160 millones de muestras. En cambio, la CPU tarda desde 5 hasta casi 200 segundos para realizar los mismos cálculos, lo que representa una aceleración de hasta 60× o más en favor de la GPU.

Además, la precisión de π estimado es muy similar en ambas plataformas, con diferencias solo en la sexta cifra decimal, lo que indica que la versión en GPU no sacrifica precisión a pesar de su rapidez. También se observa que aumentar la cantidad de hilos y bloques en GPU mejora el aprovechamiento de la paralelización, aunque el tiempo tiende a estabilizarse conforme se incrementan las muestras, mostrando una buena escalabilidad. En conclusión,

estos resultados confirman que CUDA permite realizar simulaciones masivas de manera altamente eficiente y precisa, representando una herramienta clave para cómputo científico.

En conclusión, este código demuestra de manera clara cómo el uso de cómputo paralelo con CUDA en GPU puede mejorar significativamente el rendimiento en tareas intensivas como la estimación de π mediante el método de Monte Carlo. Al comparar distintas configuraciones de hilos, bloques y muestras por hilo, se observa que la GPU logra reducir considerablemente los tiempos de ejecución frente a la CPU, manteniendo una precisión comparable. Esta práctica resalta la eficiencia de la paralelización masiva que ofrece la arquitectura CUDA y su aplicabilidad en simulaciones de gran escala, evidenciando la ventaja de aprovechar recursos de hardware especializados para acelerar procesos computacionales.

Colab

https://colab.research.google.com/drive/1-IAdGrjnsjtwIpq-6QXRaXaNm4wCgMwe?usp=sharing

Cuestionario

1. ¿En qué consiste el método de Monte Carlo para estimar el valor de π y por qué es adecuado para la computación paralela?

Consiste en generar aleatoriamente millones de puntos en un cuadrado [0,1]×[0,1], contar cuántos caen dentro de un cuarto de círculo unidad y usar la proporción para estimar π por: π ≈ 4 × (#dentro/total). Es altamente adecuado para computación paralela porque cada punto se calcula de forma independiente, lo que permite distribuir las simulaciones entre miles de hilos de GPU simultáneamente.

2. Explica cómo se distribuye la carga de trabajo en CUDA cuando se usa un kernel para calcular Monte Carlo en paralelo.

Cada hilo de CUDA ejecuta una porción del total de muestras: se lanzan múltiples bloques con muchos hilos, cada uno genera samples_per_thread puntos y cuenta localmente. Luego agregan sus resultados globalmente. Así, la carga de trabajo se reparte de forma uniforme y sincronizada en toda la GPU.

3. ¿Cuál es la función del curandState en CUDA y por qué es necesaria en este tipo de simulaciones?

curandState es la estructura que gestiona el estado de un generador de números aleatorios en cada hilo. Se inicializa con curand_init() usando una semilla, permitiendo números aleatorios independientes y reproducibles por hilo. Sin esta estructura, los hilos generarían secuencias correlacionadas, invalidando la aleatoriedad necesaria para el método.

4. ¿Qué ventajas ofrece CUDA frente a una implementación secuencial en CPU para este problema?

CUDA permite el cómputo masivo paralelo en miles de hilos, escondiendo latencias de memoria y ejecutando millones de simulaciones en paralelo. Esto resulta en aceleraciones de decenas o cientos de veces frente a una implementación secuencial en CPU.

5. Describe cómo se utiliza atomicAdd() en este contexto y qué problema ayuda a resolver.

Se utiliza para sumar el resultado parcial local de cada hilo al contador global en memoria compartida (count). La operación atómica evita condiciones de carrera, garantizando que los incrementos de diferentes hilos se acumulen sin pérdida ni corrupción de datos.

6. ¿Qué parámetros del kernel (dimensiones de bloque e hilos) afectan el rendimiento de la simulación? ¿Cómo los elegirías?

- Dimensión de bloque (threads): debe ser multiplo de 32 (tamaño de warp) para evitar hilos inactivos.
- Número de bloques: suficiente para saturar todos los SM.
- Samples por hilo: balancea overhead de kernel launches y precisión. Se eligen buscando máxima ocupación de SMs, eficiencia de memoria y cobertura de warps.

7. ¿Qué papel tiene el uso de la **memoria global** en este ejercicio y cómo se podría optimizar?

El único dato extraído de la GPU es el contador global count, por lo que su uso de memoria global es mínimo. Para optimizar, se podría usar memoria compartida por bloque, acumulando dentro del bloque antes de hacer un atomicAdd global, reduciendo contención.

8. ¿Qué métricas puedes analizar con Nsight Compute en la ejecución de un kernel Monte Carlo y qué información útil puedes extraer de ellas?

Puedes analizar:

- sm_efficiency: porcentaje de ciclos en que los SM estuvieron activos.
- warp_execution_efficiency: proporción de hilos activos por warp (identifica divergencia).
- Memory access metrics: coalescencia de accesos globales, transacciones por sector, etc. Estas métricas permiten evaluar cuellos de botella en warp dispatch o latencia de memoria

9. ¿Cómo afectaría una mala generación de números aleatorios a los resultados del experimento de Monte Carlo?

Para un método de Monte Carlo robusto y preciso, se necesita un buen generador que garantice independencia, uniformidad y un periodo lo bastante largo. De lo contrario, los resultados pueden estar seriamente sesgados, tener mayor error y ser estadísticamente poco significativos. apoyar la validez del experimento requiere usar PRNGs modernos y fiables como Mersenne Twister o generadores especialmente diseñados para GPU.

10. Si se quisiera adaptar el método de Monte Carlo para resolver otro problema, como la integración de funciones, ¿qué cambiarías en el enfoque CUDA?

Cambiarías el cálculo dentro del kernel: en lugar de contar puntos dentro de un círculo, calcularías f(x_i)*J (Jacobiano) en un rango definido, y acumularías los resultados. Mantienes la estructura paralela, estados aleatorios y reducción atómica, cambiando solo la definición de la función y el dominio de muestreo.

Referencias

- Urbina, C. (n.d.). Cálculo del pi en paralelo. https://cecilia-urbina.blogspot.com/2012/03/calculo-del-pi-en-paralelo.html
- simulacion_Metodo_Montecarloarlo/README.md at main · JuanaBanda/simulacion_Metodo_Montecarloarlo. (n.d.). GitHub. https://github.com/JuanaBanda/simulacion_Metodo_Montecarloarlo/blob/main/README.md
- Putra, H. (2024). *CUDA code for Monte Carlo estimation of Pi using GPGPU* [Repositorio GitHub]. GitHub. https://github.com/hsaputra/cuda_pi_montecarlo
- Stack Overflow. (2015, enero 8). *Trying to understand nvprof metrics, sm_efficiency and warp_execution_efficiency* [Pregunta en foro]. En NVIDIA Developer Forums. https://stackoverflow.com/questions/40119862/trying-to-understand-nvprof-metrics-sm_efficiency-and-warp-execution-efficiency
- NVIDIA. (2025). *Profiling Guide — Nsight Compute 12.9 documentation*. NVIDIA. https://docs.nvidia.com/nsight-compute/ProfilingGuide