Universidad del Valle de Guatemala
Computación Paralela y Distribuida
Proyecto no. 3

Javier Emilio Alvarez Cifuentes
18051
15/11/2023

Introducción

Con el objetivo de profundizar en nuestro conocimiento de programación CUDA, particularmente en el contexto de la computación paralela y la gestión de la memoria mediante la implementación y optimización de la Transformada de Hough para la detección de líneas en imágenes, buscamos explorar las complejidades de la jerarquía de memoria de CUDA, incluyendo la memoria global, constante y compartida. Este proyecto no solo sirve como ejercicio práctico en programación CUDA, sino también como una plataforma para ilustrar las significativas diferencias de rendimiento que varios tipos de memoria y técnicas de optimización pueden introducir en un entorno de computación paralela.

Antecedentes:

La Transformada de Hough es una técnica popular en procesamiento de imágenes utilizada para la detección de formas simples, como líneas, círculos y elipses. Originalmente desarrollada para detectar líneas en fotografías de cámaras de burbujas, desde entonces ha encontrado una amplia aplicación en varios campos, como visión por computadora y procesamiento digital de imágenes. El algoritmo funciona mapeando puntos en el espacio de la imagen a un espacio de parámetros y luego encontrando máximos locales en este espacio de parámetros, que corresponden a los candidatos más probables para la presencia de una forma particular en la imagen original.

CUDA (Compute Unified Device Architecture) es una plataforma de computación paralela y un modelo de interfaz de programación de aplicaciones (API) creada por NVIDIA. Permite a los desarrolladores de software utilizar una unidad de procesamiento gráfico (GPU) compatible con CUDA para procesamiento de propósito general (un enfoque conocido como GPGPU, General-Purpose computing on Graphics Processing Units). La programación CUDA implica entender cómo utilizar de manera efectiva la arquitectura de la GPU, incluidas sus diversas formas de memoria, para optimizar tareas computacionales.

En este proyecto, la implementación de la Transformada de Hough sirve como un banco de pruebas para experimentar con diferentes aspectos de la programación CUDA. El enfoque está en cómo se pueden aprovechar diferentes tipos de memoria para mejorar el rendimiento y cómo una tarea tradicionalmente intensiva en la CPU puede acelerarse utilizando la computación GPU.

Metodología:

El proceso de desarrollo comenzó con el código del algoritmo de la Transformada de Hough que se nos fue proporcionado. Este algoritmo, tradicionalmente ejecutado en una CPU, fue adaptado y optimizado para ejecutarse en una GPU utilizando la plataforma CUDA de NVIDIA. La metodología involucra varios pasos clave: entender el algoritmo, implementar una versión básica en CUDA y luego optimizar progresivamente esta implementación aprovechando diferentes tipos de memoria de la GPU.

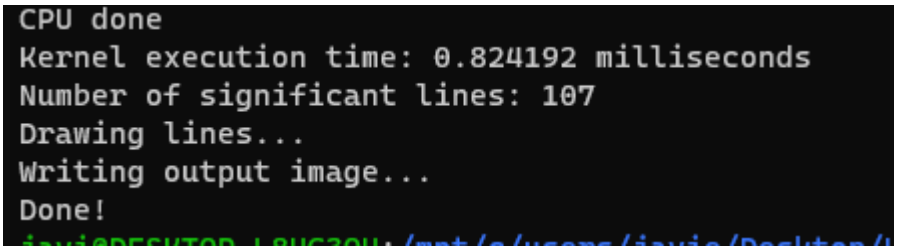
Primero se analizó el algoritmo de la Transformada de Hough para identificar sus características computacionales, centrándose en el código existente y que era lo que necesitaba para que funcionara. Luego de esto se buscó corregir cualquier error que podría existir dentro del código y agregar las funciones útiles que se nos mencionan para poder terminar el funcionamiento funcional del programa. Luego de esto se procedio a trabajar con el entrono de CUDA, realizando sus configuraciones pertinentes para que este funcionara de manera eficaz y sin tener problemas. Finalmente se realizaron los 3 programas que se nos pedía realizar. La primera tarea que se realizó involucró el uso efectivo de la memoria global.
El principal desafío fue minimizar la transferencia de memoria entre el host (CPU) y el dispositivo (GPU) y asegurar patrones eficientes de acceso a la memoria en la GPU.
Integración de Memoria Constante:

El siguiente paso fue aprovechar la memoria constante para datos de acceso frecuente que no cambian durante la ejecución del kernel (por ejemplo, valores precalculados de seno y coseno).
Esto implicó modificar el kernel para usar memoria constant y adaptar la transferencia de datos para usar cudaMemcpyToSymbol.
Utilización de Memoria Compartida:

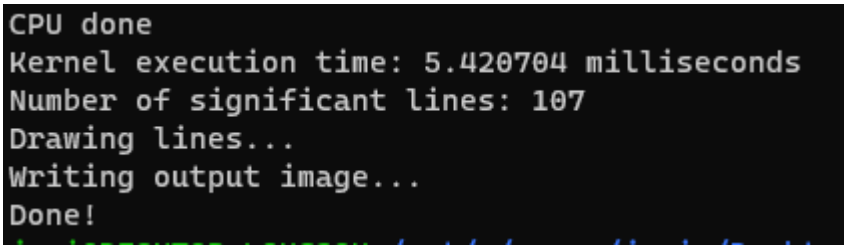
Se logró una optimización adicional utilizando memoria compartida para reducir los accesos a memoria global.
Esto requirió rediseñar el kernel para incluir un acumulador local en memoria compartida y la implementación de puntos de sincronización para garantizar la integridad de los datos.

Resultados

Programa Base con memoria global:



Programa con memoria constante:



Programa con memoria compartida:

```
CPU done
Kernel execution time: 0.712448 milliseconds
Number of significant lines: 107
Drawing lines...
Writing output image...
Done!
```

Iteración	Base (ms)	Constante (ms)	Compartida (ms)
1	0.824192	5.420704	0.712448
2	1.169088	5.346592	0.795040
3	0.807104	5.338976	0.731680
4	0.817376	5.814240	0.740800
5	0.885984	5.320384	0.779456
6	0.830048	5.585216	0.717152
7	0.820832	5.513376	0.784032
8	0.807168	5.339264	0.729440
9	0.887744	5.968672	0.666464
10	0.815424	5.449472	0.676992
Promedio	0.866496	5.509690	0.733350

Como podemos observar, la ejecución más rápida es la de la memoria compartida, pero siendo realistas la diferencia entre cada una es tan pequeña que se percibe todo de manera inmediata. Algunas de las razones que la memoria constante es más lenta puede ser su naturaleza, ya que la memoria constante se almacena en caché y se optimiza para escenarios en los cuales todos los hilos leen el mismo valor. En este caso no necesariamente los mismos hilos leen el mismo valor, lo que crea que exista también lentitud. Otra posible razón de la lentitud es una sobrecarga de operaciones atómicas ya que el atomicAdd es parte del programa para evitar las race conditions pero puede crear lentitud por la cantidad de operaciones que se están realizando.
En resumen, la rapidez de la memoria compartida proviene de su naturaleza en el chip y local al bloque, lo que reduce las transacciones de memoria global y se ajusta bien al patrón de acumulación de la Transformada de Hough. La versión de memoria constante, aunque teóricamente más rápida para datos de solo lectura, puede tener una sobrecarga de operaciones atómicas, ralentizando su ejecución.

Conclusiones

- En este proyecto se encontraron retos que se relacionaban más a la parte de hardware y del manejo del tiempo que conceptuales. Se siguió la guía proveída pero en la segunda entrega no colocamos la llamada a cudaMemcpyToSymbol lo cual causó resultados erróneos en los tiempos de ejecución para memoria constante. Esto se soluciono al agregar la llamada adecuadamente y cambiar la bitácora de resultados a los valores nuevos.

Bibliografía

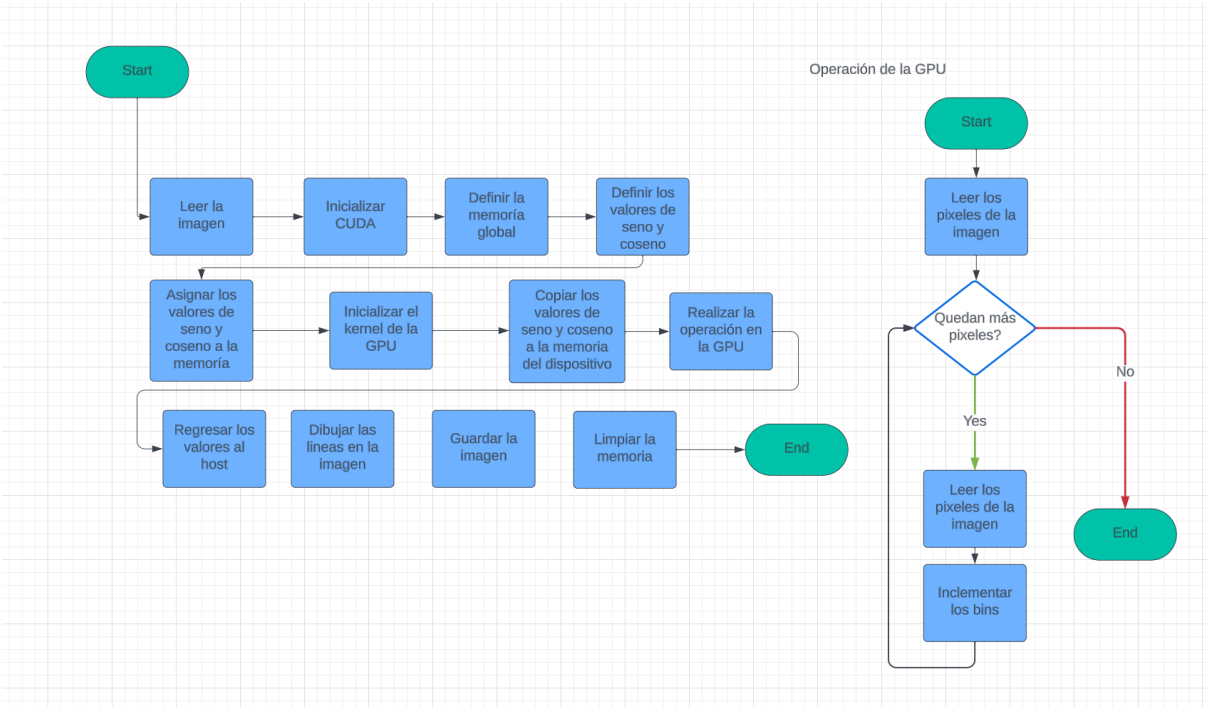
- Harris, M. (2012). How to Implement Performance Metrics in CUDA C/C++. Extraido de: https://developer.nvidia.com/blog/how-implement-performance-metrics-cuda-cc/

- Harris, M. (2013). Using Shared Memory in CUDA C/C++ Extraído de: https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/

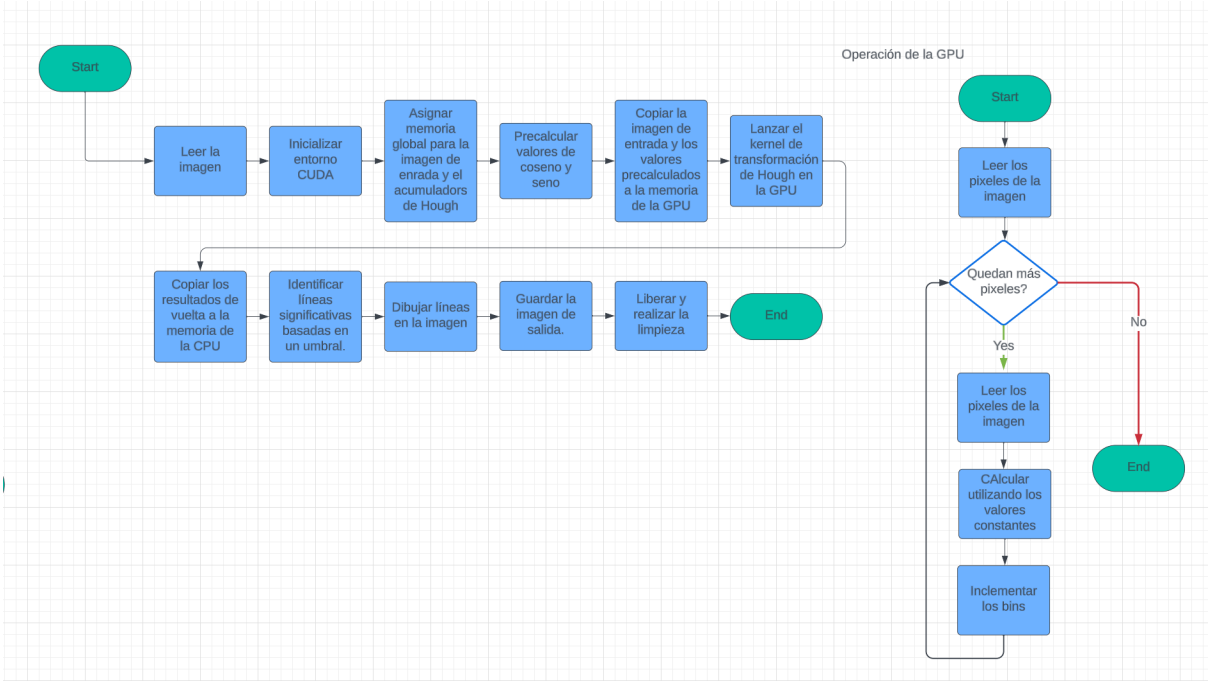
- Lee, S. (2020). Lines Detection with Hough Transform. Extraído de: https://towardsdatascience.com/lines-detection-with-hough-transform-84020b3b1549

Anexos

- Diagrama de flujo programa base:



- Diagrama de flujo programa con memoria constante:



- Diagrama de flujo programa con memoria compartida: