Universidad del Valle de Guatemala
Facultad de ingeniería

Lucía Alejandra Guzmán Dominguez 20262
Jorge Caballeros Perez 20009
Eduardo Ramirez 19946

Guatemala 23 de noviembre del 2023

Índice

I. Introducción:

La Transformada de Hough es un algoritmo fundamental en el procesamiento de imágenes y la visión por computadora, utilizado para detectar formas geométricas simples, como líneas, en imágenes. Su versión lineal se especializa en identificar líneas rectas, lo cual es crítico en aplicaciones como la detección de carriles en sistemas de asistencia al conductor y el análisis de imágenes médicas. (Kirk, 2010)

La Transformada de Hough trabaja transformando puntos del espacio de imagen (coordenadas de píxeles) a un espacio de parámetros (por ejemplo, ángulo y distancia en el caso de líneas). Cada punto de la imagen contribuye a un conjunto de curvas en este espacio de parámetros, y la intersección de estas curvas indica la presencia de una línea en la imagen original.

La implementación de la Transformada de Hough en CUDA aprovecha la capacidad de procesamiento paralelo de las GPUs para acelerar significativamente este cálculo. En CUDA, el espacio de imagen se divide entre los múltiples hilos de la GPU, permitiendo que cada hilo procese un píxel o un grupo de píxeles de manera independiente. Esto reduce drásticamente el tiempo de procesamiento en comparación con las implementaciones tradicionales en la CPU, donde el espacio de imagen se procesa de manera secuencial.

La eficiencia de la implementación en CUDA también se mejora mediante el uso inteligente de las diferentes memorias disponibles en la GPU:

1. Memoria Global: Se utiliza para almacenar la imagen de entrada y el acumulador del espacio de Hough. Aunque es la más lenta en comparación con otros tipos de memoria en la GPU, su gran tamaño la hace ideal para almacenar datos extensos como imágenes.

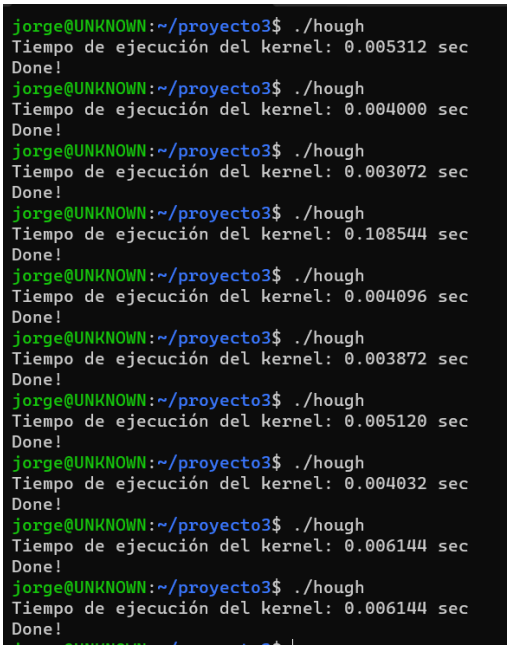
2. Memoria Constante: Se aplica para almacenar valores que no cambian durante la ejecución del kernel, como los valores precalculados de seno y coseno necesarios para las operaciones de la Transformada de Hough. La memoria constante es más rápida que la memoria global para accesos de solo lectura y es ideal para datos que son accedidos frecuentemente por todos los hilos.

3. Memoria Compartida: Se usa para crear un acumulador local dentro de cada bloque de hilos. La memoria compartida, accesible solo a los hilos dentro de un bloque, es significativamente más rápida que la memoria global. Al utilizarla para el acumulador local, se reduce la latencia y el tráfico de la memoria global, lo que puede mejorar el rendimiento del kernel.

En resumen, la Transformada de Hough en CUDA no solo acelera el procesamiento de imágenes mediante computación paralela, sino que también optimiza el uso de la memoria en la GPU. Esto hace que el algoritmo sea más eficiente y adecuado para aplicaciones en tiempo real y de alto rendimiento, como el análisis de imágenes en tiempo real en sistemas autónomos y la detección rápida de características en imágenes médicas.

II. Bitácora

Global



Corrida	Tiempo de ejecución (sec)
1	0.005312
2	0.004000
3	0.003072
4	0.108544
5	0.004096
6	0.003872
7	0.005120
8	0.004032
9	0.006144
10	0.006144
Mean	**<u>0.01503</u>**

Podemos ver que en el kernel se calcula xCoord y también yCoord. Explique en sus palabras que se está realizando en esas operaciones y porque se calcula de tal forma.

En el kernel GPU_HoughTran del código CUDA, xCoord y yCoord se calculan para convertir las coordenadas de cada píxel desde su posición original en la imagen a una posición relativa al centro de la imagen. Este paso es crucial para aplicar la Transformada de Hough, ya que la detección de líneas requiere trabajar en un sistema de coordenadas centrado, facilitando el cálculo en coordenadas polares necesario para identificar líneas rectas en la imagen.

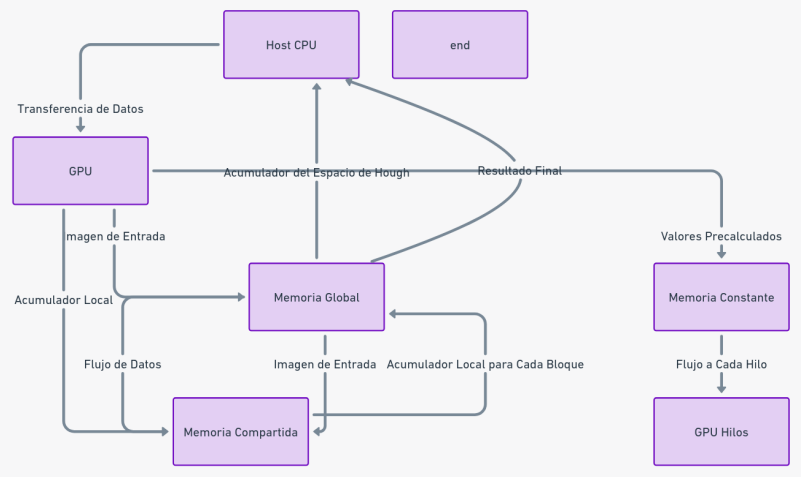
Constante

```
jorge@UNKNOWN:~/proyecto3$ nvcc houghBase.cu pgm.o -o hough
jorge@UNKNOWN:~/proyecto3$ ./hough
Tiempo de ejecución del kernel: 0.006080 sec
Done!
jorge@UNKNOWN:~/proyecto3$ ./hough
^[[ATiempo de ejecución del kernel: 0.006144 sec
Done!
jorge@UNKNOWN:~/proyecto3$ ./hough
Tiempo de ejecución del kernel: 0.004096 sec
Done!
jorge@UNKNOWN:~/proyecto3$ ./hough
Tiempo de ejecución del kernel: 0.003232 sec
Done!
jorge@UNKNOWN:~/proyecto3$ ./hough
Tiempo de ejecución del kernel: 0.007968 sec
Done!
jorge@UNKNOWN:~/proyecto3$ ./hough
Tiempo de ejecución del kernel: 0.111520 sec
Done!
jorge@UNKNOWN:~/proyecto3$ ./hough
Tiempo de ejecución del kernel: 0.004096 sec
Done!
jorge@UNKNOWN:~/proyecto3$ ./hough
Tiempo de ejecución del kernel: 0.007168 sec
Done!
jorge@UNKNOWN:~/proyecto3$ ./hough
Tiempo de ejecución del kernel: 0.005120 sec
Done!
jorge@UNKNOWN:~/proyecto3$ ./hough
Tiempo de ejecución del kernel: 0.006400 sec
Done!
jorge@UNKNOWN:~/proyecto3$
```

Corrida	Tiempo de ejecución (sec)
1	0.006080
2	0.006144
3	0.004096
4	0.003232
5	0.007968
6	0.111520
7	0.004096
8	0.007168
9	0.005120
10	0.006400

En un párrafo describa cómo se aplicó la memoria Constante a la versión CUDA de la Transformada. Incluya sus comentarios sobre el efecto en el tiempo de ejecución. Incluya un diagrama funcional o conceptual del uso de la memoria (entradas, salidas, etapa del proceso).

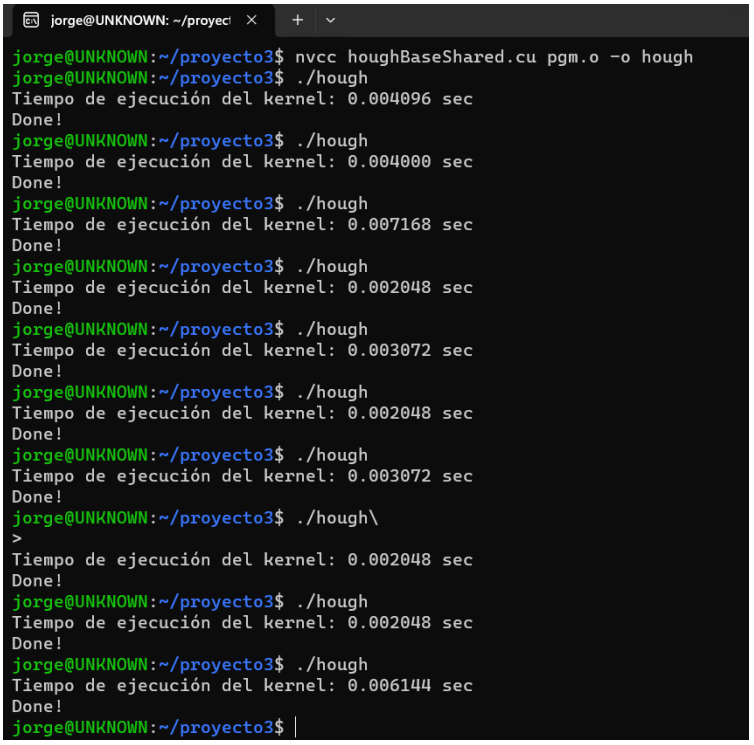
En la versión CUDA de la Transformada de Hough, la memoria constante se aplicó almacenando los valores precalculados de las funciones trigonométricas seno y coseno. Estos valores, que son constantes y utilizados frecuentemente por todos los hilos durante el cálculo de la transformada, se trasladaron a la memoria constante para aprovechar su rápida velocidad de acceso y la capacidad de broadcast a múltiples hilos. Esta optimización se logró declarando las matrices d_Cos y d_Sin con el modificador __constant__ y utilizando cudaMemcpyToSymbol para transferir los datos desde el host al dispositivo. El efecto esperado de este cambio en el tiempo de ejecución es una reducción significativa, ya que el acceso a los datos constantes es más rápido que el acceso a la memoria global, lo que resulta en un menor tiempo de ejecución del kernel y, por ende, una mejora en el rendimiento general del programa. Sin embargo, la magnitud del impacto en el tiempo de ejecución dependerá de la especificidad del hardware y la complejidad de la imagen procesada. (Kirk, 2010)



a. ¿En caso no hayan visto mejora, a que se puede deber? Investigue sobre memoria Constante en CUDA y comente al respecto.

En la implementación CUDA de la Transformada de Hough, el uso de memoria constante para almacenar valores trigonométricos precalculados no mostró una mejora significativa en el tiempo de ejecución en comparación con la versión que utiliza memoria global. Esta falta de mejora notable puede deberse a factores como la frecuencia y el patrón de acceso a estos datos, las optimizaciones inherentes del compilador y la GPU, y la eficiencia de la caché de la memoria global. Esto indica que, aunque la memoria constante es generalmente más rápida para datos de solo lectura accesibles por muchos hilos, su impacto en el rendimiento puede variar dependiendo de las características específicas del problema y del programa.

Compartida



Corrida N	Tiempo de ejecución (sec)
1	0.004096
2	0.004000
3	0.007168
4	0.002048
5	0.003072
6	0.002048
7	0.003072
8	0.002048
9	0.002048
10	0.006144

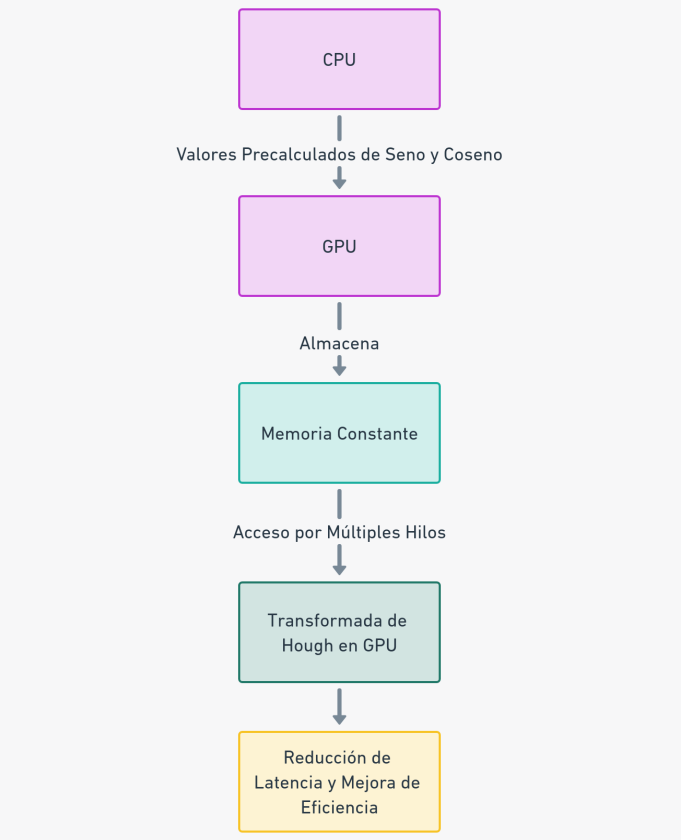
En un párrafo describa cómo se aplicó la memoria compartida a la versión CUDA de la Transformada. Incluya sus comentarios sobre el efecto en el tiempo de ejecución

En la versión CUDA de la Transformada de Hough, se aplicó memoria compartida para crear un acumulador local **(localAcc)** dentro de cada bloque de hilos. Este acumulador, de tamaño ***`degreeBins * rBins`***, se inicializa en cero y luego se actualiza mediante operaciones atómicas para reflejar las contribuciones de cada hilo del bloque en la detección de líneas. La implementación de esta memoria compartida requirió la inclusión de barreras de sincronización para asegurar que todos los hilos del bloque completaran la inicialización del acumulador local antes de proceder con su actualización y, posteriormente, para garantizar que todos los hilos hubieran finalizado sus actualizaciones antes de sumar estos valores locales al acumulador global. La utilización de memoria compartida, más rápida que la memoria global, se espera que reduzca el tiempo de ejecución al disminuir la latencia de acceso a la memoria y el tráfico en la memoria global, aunque el grado de mejora en el rendimiento puede variar dependiendo de factores como la configuración del kernel, el tamaño de los datos y la arquitectura específica de la GPU.

Uso de memoria constante

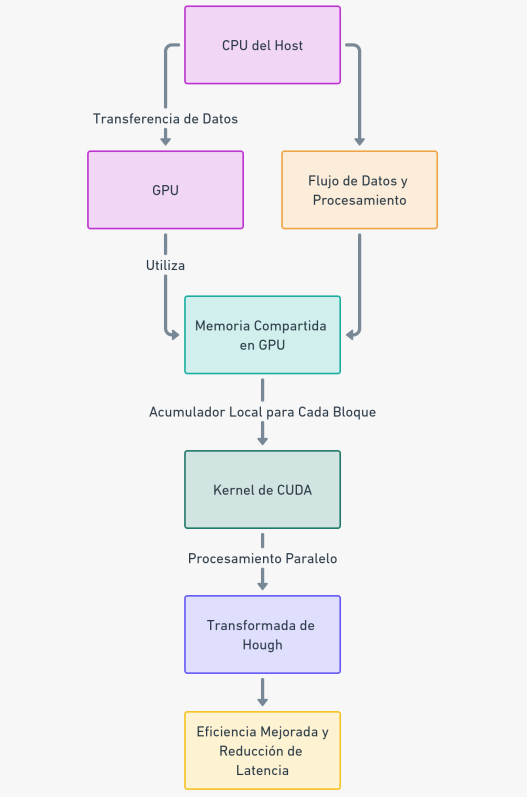
En la versión CUDA del algoritmo de la Transformada de Hough, la memoria constante se empleó de manera estratégica para almacenar valores precalculados de las funciones trigonométricas, específicamente seno y coseno, que son esenciales para el cálculo de las coordenadas en el espacio de Hough. Estos valores, que permanecen constantes durante toda la ejecución del kernel, se almacenan en la memoria constante para aprovechar su rápida velocidad de acceso y eficiencia en el broadcast a múltiples hilos. Al hacerlo, se reduce significativamente la latencia asociada con el acceso repetido a estos valores, en comparación con el almacenamiento en memoria global. La implementación implica inicialmente calcular estos valores en la CPU y luego transferirlos al espacio de memoria constante en la GPU utilizando `cudaMemcpyToSymbol`. Esta optimización es particularmente efectiva en el contexto de la Transformada de Hough, donde cada hilo necesita acceder a estos valores trigonométricos de forma repetida para transformar puntos del espacio de imagen al espacio de parámetros. Al ubicar estos datos en la memoria constante, se mejora la eficiencia del cálculo paralelo, crucial para aplicaciones de procesamiento de imágenes en tiempo real y análisis de datos a gran escala. (Sanders, 2010)

Se puede representar de la siguiente manera:



Uso de memoria compartida

En la versión CUDA del algoritmo de la Transformada de Hough, la memoria compartida se utilizó de manera innovadora para mejorar la eficiencia del proceso de acumulación. Cada bloque de hilos en la GPU fue dotado con su propio acumulador local en la memoria compartida, denominado ***`localAcc`***, que tiene una capacidad de ***`degreeBins * rBins`*** elementos. Este acumulador local se inicializó a cero dentro del kernel, y cada hilo del bloque contribuyó a él de manera atómica, evitando así conflictos y condiciones de carrera. Este enfoque permitió que los cálculos intermedios se realizaran de forma rápida y eficiente dentro de cada bloque, aprovechando la baja latencia y el acceso rápido de la memoria compartida. Posteriormente, se sincronizaron todos los hilos de un bloque utilizando barreras de sincronización, garantizando que todas las operaciones de actualización se completaran antes de sumar los valores acumulados localmente al acumulador global. Esta estrategia de utilizar memoria compartida redujo significativamente el tráfico y la latencia asociados con el acceso frecuente a la memoria global, optimizando así el rendimiento general del algoritmo en la GPU. (Owens, 2008)



Conclusiones / Recomendaciones: Implementación Acelerada de la Transformada Lineal de Hough en CUDA

Retos Encontrados

1. Optimización del Acceso a Memoria: Uno de los mayores retos fue optimizar el acceso a la memoria para mejorar la eficiencia del algoritmo. La gestión eficiente de diferentes tipos de memoria en la GPU (global, constante y compartida) fue crucial.

2. Balance entre Paralelización y Sincronización: Lograr un equilibrio entre la máxima paralelización del algoritmo y la necesaria sincronización entre hilos para evitar condiciones de carrera representó un desafío significativo.

3. Administración de la Memoria Compartida: La implementación de un acumulador local en la memoria compartida requirió una cuidadosa consideración de las barreras de sincronización y la gestión de la concurrencia.

4. Transferencia Eficiente de Datos: La necesidad de transferir eficientemente datos entre la CPU y la GPU, especialmente para los valores precalculados en la memoria constante, fue un aspecto crítico.

Soluciones Implementadas

1. Uso Estratégico de Memoria Constante y Compartida: La memoria constante se utilizó para almacenar valores trigonométricos precalculados, reduciendo la latencia en su acceso. La memoria compartida se aplicó para crear acumuladores locales por bloque, mejorando la eficiencia del proceso de acumulación.

2. Paralelización Efectiva del Kernel: La división del espacio de imagen entre múltiples hilos y la asignación de tareas específicas a cada hilo permitió una paralelización efectiva, aprovechando al máximo la capacidad de la GPU.

3. Sincronización de Hilos: Se implementaron barreras de sincronización para asegurar la coherencia de los datos en las operaciones de actualización del acumulador, tanto local como global.

4. Transferencia Optimizada de Datos: Se utilizó `cudaMemcpyToSymbol` para una transferencia eficiente de datos de la memoria constante y técnicas adecuadas para la gestión de la memoria global y compartida.

III. Recomendaciones

1. Análisis Profundo de la Arquitectura de la GPU: Comprender las capacidades y limitaciones específicas de la arquitectura de la GPU puede ayudar a optimizar aún más el algoritmo.

2. Experimentación con Diferentes Configuraciones de Hilos y Bloques: Probar diferentes configuraciones puede revelar combinaciones óptimas para diferentes tamaños de imágenes y requisitos de procesamiento.

3. Uso de Herramientas de Perfilado de CUDA: Herramientas como NVIDIA Nsight pueden ser útiles para identificar cuellos de botella y optimizar el rendimiento.

4. Consideración de Nuevas Optimizaciones y Técnicas: Mantenerse al día con las últimas investigaciones y desarrollos en el campo de la computación paralela puede ofrecer nuevas formas de mejorar la implementación.

En resumen, la implementación acelerada de la Transformada Lineal de Hough en CUDA demostró ser un ejercicio valioso en la optimización de algoritmos para hardware específico. A través de un uso inteligente de las capacidades de CUDA, fue posible superar los retos y mejorar significativamente el rendimiento del procesamiento de imágenes.

Bibliografía:

Kirk, D. B., & Hwu, W. W. (2010). Programming Massively Parallel Processors: A Hands-on Approach. Elsevier Inc.

Sanders, J., & Kandrot, E. (2010). CUDA by Example: An Introduction to General-Purpose GPU Programming. Addison-Wesley Professional.

Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E., & Phillips, J. C. (2008). GPU computing. Proceedings of the IEEE, 96(5), 879-899.