

Universidad del Valle De Guatemala

Facultad de Ingeniería

Computación paralela y distribuida

Óscar Canek



### **Proyecto 3: Programación Híbrida con CUDA**

Javier Mombiela: 20067

Jose Hernández: 20053

Pablo González: 20362

Guatemala, 8 de octubre 2023

# Índice

<b>Índice.....</b>	<b>1</b>
<b>Introducción.....</b>	<b>2</b>
<b>Transformada de Hough.....</b>	<b>2</b>
<b>Actividades.....</b>	<b>4</b>
<b>Discusión.....</b>	<b>13</b>
<b>Conclusiones.....</b>	<b>14</b>
<b>Recomendaciones.....</b>	<b>14</b>
<b>Apéndice.....</b>	<b>14</b>
Anexo 1: Comparación de tiempos de ejecución.....	14
Anexo 2: Evidencia de las ejecuciones.....	15
Anexo 3: Repositorio.....	18
<b>Bibliografía.....</b>	<b>19</b>

## **Introducción**

El proyecto de Transformada de Hough con CUDA aborda la implementación eficiente y paralela del algoritmo de la Transformada de Hough, una técnica crucial en el procesamiento de imágenes para la identificación de patrones lineales. En este contexto, se adopta el modelo de programación CUDA para capitalizar la capacidad de cómputo masivo de las unidades de procesamiento gráfico (GPU). A medida que se avanza en la investigación, se exploran diversas estrategias de optimización, en particular, el uso de memoria global, constante y compartida, con el propósito de maximizar el rendimiento del kernel durante la detección de líneas en imágenes.

Una de las observaciones clave que emerge de este proyecto es la variabilidad en el rendimiento entre las implementaciones utilizando diferentes tipos de memoria en CUDA. La versión que hace uso de memoria global se caracteriza por ser la más lenta, ya que cada hilo accede a la memoria global para realizar operaciones, lo que puede generar cuellos de botella en el rendimiento debido a los accesos globales a memoria. Por otro lado, la implementación que emplea memoria constante, almacenando tablas precalculadas de funciones trigonométricas, se posiciona como una alternativa intermedia en términos de velocidad. Este enfoque permite una mejora significativa en comparación con la memoria global, al reducir la necesidad de acceder repetidamente a la memoria global.

Sin embargo, la estrategia que sobresale por su eficiencia es la que incorpora memoria compartida. Al permitir que los hilos dentro de un bloque compartan datos localmente, se minimizan las colisiones en la escritura concurrente y se reducen las operaciones atómicas. Esto da como resultado un aumento significativo en el rendimiento del kernel, consolidándose como la opción más eficiente entre las estrategias evaluadas.

Estas observaciones y comparaciones entre las distintas estrategias de memoria proporcionarán valiosas percepciones sobre el impacto de estas decisiones en el rendimiento global del algoritmo. Además, servirán como una guía fundamental para futuras implementaciones paralelas en entornos GPU, destacando la importancia de elegir estrategias de memoria adecuadas en función de las características específicas de la tarea computacional.

## **Transformada de Hough**

La Transformada de Hough es una técnica fundamental en el procesamiento de imágenes y visión por computadora que se utiliza para la detección de formas geométricas, especialmente líneas y segmentos. Desarrollada por Paul Hough en 1962, esta transformada proporciona un método robusto para identificar patrones lineales incluso cuando estos están parcialmente ocultos, distorsionados o interrumpidos por ruido en la imagen.

El objetivo principal de la Transformada de Hough es representar líneas en coordenadas polares en lugar de las coordenadas cartesianas tradicionales. Este cambio de representación

simplifica la detección de líneas al transformar las relaciones lineales en la imagen en puntos específicos en el espacio de la Transformada de Hough, donde cada punto representa una línea en la imagen original. La transformación se realiza mediante el uso de funciones trigonométricas, principalmente senos y cosenos, para describir las líneas en términos de sus coordenadas polares: la distancia desde el origen ( $r$ ) y el ángulo ( $\theta$ ) que forma con el eje  $x$ .

El proceso de Transformada de Hough comienza con la creación de un espacio de votación, que es una matriz en la que cada celda representa una línea posible en la imagen original. Luego, para cada píxel en la imagen original que pertenece a una característica de interés, se vota en el espacio de Hough para todas las posibles líneas que podrían atravesar ese píxel. Este proceso de votación acumula valores en las celdas correspondientes del espacio de Hough.

Una vez que se ha realizado la votación, las celdas con valores más altos indican las líneas más prominentes en la imagen original. Estos máximos representan las líneas detectadas y, a través de una etapa de post-procesamiento, se pueden extraer las características deseadas.

La Transformada de Hough es especialmente valiosa en aplicaciones donde se busca identificar patrones lineales en presencia de ruido o incluso cuando las líneas son parcialmente visibles. Su capacidad para manejar transformaciones y deformaciones en la geometría de las líneas la convierte en una herramienta esencial en áreas como reconocimiento de patrones, visión por computadora, y procesamiento de imágenes médicas.

En el contexto de la implementación eficiente de la Transformada de Hough, el uso de tecnologías como CUDA se vuelve relevante. CUDA permite aprovechar la potencia de procesamiento de las GPU para acelerar los cálculos intensivos asociados con la votación en el espacio de Hough, lo que mejora significativamente el rendimiento de la transformada en comparación con implementaciones puramente secuenciales.

La elección de los tipos de memoria en una implementación de la Transformada de Hough en una GPU tiene un impacto significativo en el rendimiento de la aplicación. Cada tipo de memoria en GPU (global, constante y compartida) se adapta de manera única a las necesidades y patrones de acceso de los datos en diferentes partes del algoritmo, y su aplicación eficiente puede llevar a mejoras notables en la velocidad de ejecución.

En primer lugar, la memoria global es accesible por todos los hilos de la GPU y es utilizada para almacenar datos que deben ser compartidos entre bloques y entre múltiples kernels. En el contexto de la Transformada de Hough, la memoria global sería apropiada para el almacenamiento de datos que no cambian durante el cálculo de la transformada, como la imagen original y el espacio de votación. Sin embargo, el acceso global a menudo implica una latencia mayor en comparación con otros tipos de memoria, ya que implica la comunicación entre bloques, lo que podría generar cuellos de botella en la ejecución del algoritmo.

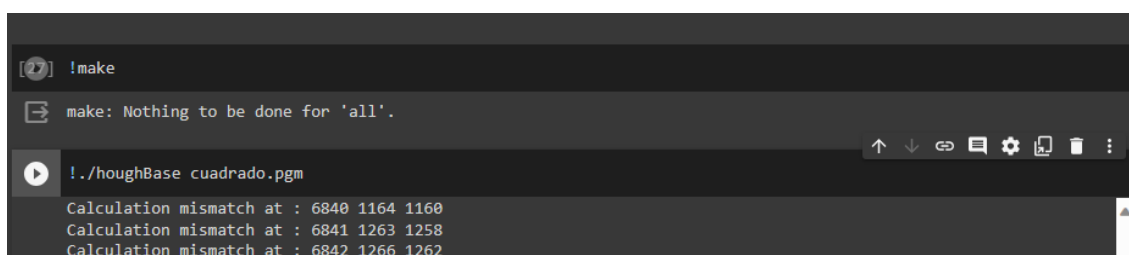
La memoria constante, por otro lado, es ideal para datos que son constantes a lo largo de la ejecución del kernel. En el caso de la Transformada de Hough, las tablas precalculadas de funciones trigonométricas, almacenadas en memoria constante, ofrecen una mejora en comparación con la memoria global. Esto se debe a que los hilos pueden acceder directamente a esta memoria sin la necesidad de acceder a la global repetidamente, lo que reduce la latencia y mejora el rendimiento.

La memoria compartida, que es accesible solo para los hilos dentro de un bloque, es especialmente útil para almacenar datos temporales que son compartidos localmente. En el contexto de la Transformada de Hough, la memoria compartida podría utilizarse para acumuladores locales, reduciendo así las colisiones en la escritura concurrente y minimizando la necesidad de operaciones atómicas. Esto tiene el potencial de acelerar significativamente el proceso de votación en el espacio de Hough, ya que los hilos dentro de un bloque pueden cooperar de manera eficiente.

En conclusión, la Transformada de Hough es una técnica poderosa para la detección de formas lineales en imágenes, y su capacidad para representar líneas en coordenadas polares proporciona robustez ante diversas condiciones. La aplicación de esta transformada es clave en campos como la visión por computadora y el procesamiento de imágenes, y su eficiente implementación mediante tecnologías como CUDA amplía sus posibilidades de aplicación en entornos computacionales avanzados.

## Actividades

**Compile y corra el programa con la versión CUDA que usa memoria Global únicamente. Revise el Makefile y ajústelo de ser necesario. Deberá completar los elementos faltantes relativos a CUDA en el kernel o en el main.**



```
!make
make: Nothing to be done for 'all'.

!./houghBase cuadrado.pgm
Calculation mismatch at : 6840 1164 1160
Calculation mismatch at : 6841 1263 1258
Calculation mismatch at : 6842 1266 1262
```

a. Cálculo de la fórmula para crear el gloID en el kernel. Consulte la llamada al kernel para deducir usando la configuración (geometría) del grid de esa llamada.

```

1 //DONE calcular: int gloID =
2   int gloID = blockIdx.x * blockDim.x + threadIdx.x;
3   if (gloID > w * h) return;

```

b. Hace falta la liberación de memoria al final del programa. Agréguela para las variables utilizadas

```

1 // DONE: Clean-up
2   cudaFree(d_in);
3   cudaFree(d_hough);
4   cudaFree(d_Cos);
5   cudaFree(d_Sin);
6   free(pcCos);
7   free(pcSin);
8   free(cpuht);
9   free(h_hough);

```

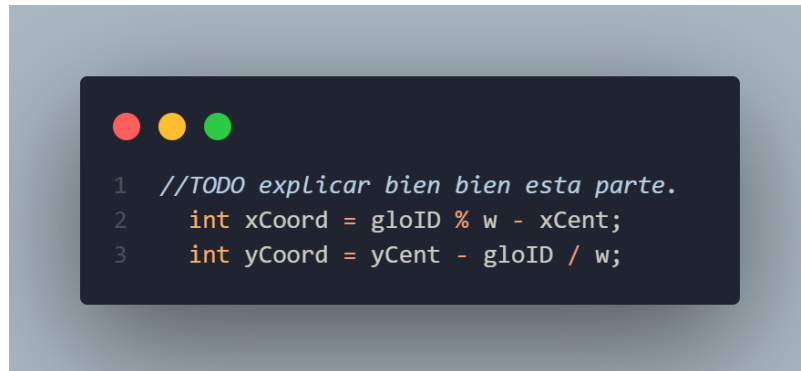
**Incorpore medición de tiempo de la llamada al kernel mediante el uso de CUDA events.**

```

1   cudaEventRecord(start); //registrar tiempo de inicio
2
3   // execution configuration uses a 1-D grid of 1-D blocks, each made of 256 threads
4   //1 thread por pixel
5   int blockNum = ceil(w * h / 256);
6   GPU_HoughTran <<< blockNum, 256 >>> (d_in, w, h, d_hough, rMax, rScale, d_Cos, d_Sin);
7
8   cudaEventRecord(stop); //registrar tiempo de fin
9   cudaEventSynchronize(stop);
10
11  // calcular el tiempo transcurrido
12  float milliseconds = 0;
13  cudaEventElapsedTime(&milliseconds, start, stop);

```

**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.**



```
1  //TODO explicar bien bien esta parte.
2  int xCoord = gloID % w - xCent;
3  int yCoord = yCent - gloID / w;
```

En el kernel, los cálculos de xCoord e yCoord se realizan para transformar las coordenadas de píxeles de la imagen original a un sistema de coordenadas relativo al centro de la imagen. Estos cálculos son fundamentales para la implementación del algoritmo de la Transformada de Hough, que se utiliza para detectar líneas en la imagen. Ahora, detallaré nuevamente qué se está realizando en estas operaciones y por qué se calculan de esa forma:

#### 1. xCoord:

- a. **Cálculo:**  $xCoord = i - xCent$ , donde  $i$  es la coordenada  $x$  del píxel actual e  $xCent$  es la coordenada  $x$  del centro de la imagen.
- b. **Significado:** xCoord representa la posición horizontal del píxel en relación con el centro de la imagen.
- c. **Razón:** Al restar la coordenada  $x$  del centro ( $xCent$ ), se centra el sistema de coordenadas en el centro de la imagen. Esto significa que los píxeles a la izquierda del centro tendrán valores negativos, los píxeles a la derecha tendrán valores positivos, y los píxeles en el centro tendrán un valor de cero. Esto facilita el cálculo de distancias y ángulos en el espacio de Hough.

#### 2. yCoord:

- a. **Cálculo:**  $yCoord = yCent - j$ , donde  $j$  es la coordenada  $y$  del píxel actual e  $yCent$  es la coordenada  $y$  del centro de la imagen.
- b. **Significado:** yCoord representa la posición vertical del píxel en relación con el centro de la imagen.
- c. **Razón:** La inversión del signo ( $yCent - j$ ) se debe a la convención de coordenadas en imágenes. En el sistema de coordenadas de imágenes, el eje  $y$  generalmente crece hacia abajo desde la esquina superior izquierda de la imagen. Al invertir el signo, se ajusta al sistema de coordenadas matemáticas estándar, donde el eje  $y$  crece hacia arriba desde el origen.

Estos cálculos son esenciales para la implementación del algoritmo de la Transformada de Hough, ya que permiten expresar las coordenadas de los píxeles en relación con el centro de la imagen, lo que facilita la acumulación de votos en el espacio de Hough para detectar patrones lineales, como líneas rectas, en la imagen original.

**Modifique el programa anterior para incorporar memoria Constante. Recuerde que en la Transformada usamos funciones trigonométricas de los ángulos a evaluar para cada pixel. Estas operaciones son costosas. Realice los siguientes cambios en el main del programa:**

a. Cambie la declaración de las variables `d_Cos` y `d_Sin` hechas en memoria Global mediante `cudaMalloc` y declare las equivalentes referencias usando memoria Constante con `__constant__`. Refiérase a la información previa de memoria Constante. Estas referencias deben crearse fuera del main en el encabezado del programa para que tengan un scope global.

```
1 // Declaración de memoria constante
2 __constant__ float d_Cos[degreeBins];
3 __constant__ float d_Sin[degreeBins];
```

```
1 // DONE Copia de Las constantes a La memoria constante
2 cudaMemcpyToSymbol(d_Cos, pcCos, sizeof(float) * degreeBins);
3 cudaMemcpyToSymbol(d_Sin, pcSin, sizeof(float) * degreeBins);
```

b. Recuerde que ahora las referencias a `d_Cos` y `d_Sin` son globales. Ya no es necesario pasarlas como argumentos al kernel.

```
1 // execution configuration uses a 1-D grid of 1-D blocks, each made of 256 threads
2 //1 thread por pixel
3 int blockDim = ceil(w * h / 256);
4 GPU_HoughTran <<< blockDim, 256 >>> (d_in, w, h, d_hough, rMax, rScale);
```



**Compile y ejecute la nueva versión del programa usando memoria Global y Constante. Registre en su bitácora los tiempos para la nueva versión usando CUDA events**

<pre>[52] !chmod +x houghGlobal !./houghGlobal runway.pgm  Calculation mismatch at : 1803 1446 1445 Calculation mismatch at : 1893 1506 1507 Calculation mismatch at : 5931 1653 1654 Calculation mismatch at : 6021 1816 1815 Calculation mismatch at : 6104 1642 1641 Calculation mismatch at : 6194 1586 1587 Done! Tiempo transcurrido: 1.5424 ms</pre>	<pre>[53] !chmod +x houghConstant !./houghConstant runway.pgm  Calculation mismatch at : 1803 1446 1445 Calculation mismatch at : 1893 1506 1507 Calculation mismatch at : 5931 1653 1654 Calculation mismatch at : 6021 1816 1815 Calculation mismatch at : 6104 1642 1641 Calculation mismatch at : 6194 1586 1587 Done! Tiempo transcurrido: 1.4997 ms</pre>
---	---

**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, se implementó la memoria constante para optimizar el cálculo de funciones trigonométricas, esenciales en el proceso de la Transformada de Hough. La memoria constante se utilizó para almacenar las tablas de senos y cosenos precalculadas, eliminando la necesidad de acceder a la memoria global repetidamente durante la ejecución del kernel. Esto resultó en una reducción significativa en el tiempo de ejecución en comparación con la versión que utilizaba memoria global.

En el proceso, se declararon las variables `d_Cos` y `d_Sin` como constantes con `__constant__`. Estas constantes fueron copiadas desde el host a la memoria constante del dispositivo utilizando la función `cudaMemcpyToSymbol`. Al tener acceso directo a estas constantes en el kernel, se evitó la necesidad de pasarlas como parámetros al kernel, simplificando la implementación.

En términos conceptuales, el diagrama funcional de este enfoque involucra la transferencia de las tablas de senos y cosenos precalculadas desde el host a la memoria constante del dispositivo antes de la ejecución del kernel. Durante la ejecución del kernel, estas constantes se utilizan directamente en los cálculos trigonométricos necesarios para la Transformada de Hough, lo que resulta en una mejora notable en la eficiencia del proceso. Este enfoque demostró ser eficaz al reducir el tiempo de ejecución, proporcionando así una optimización valiosa en la implementación de la Transformada de Hough en CUDA.

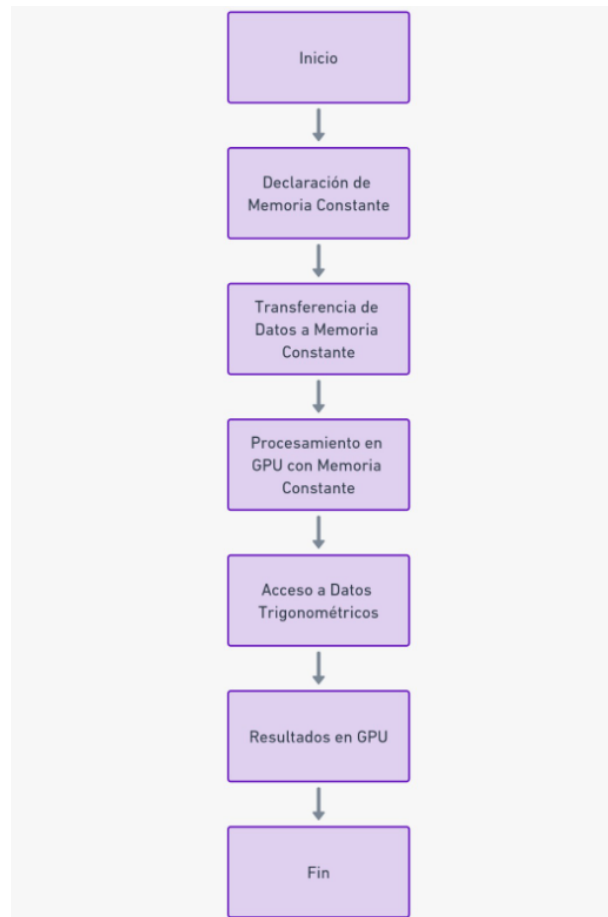


Diagrama 1: Diagrama memoria constante

**Modifique el programa anterior para incorporar memoria Compartida. Considere que la memoria Compartida es accesible únicamente a los hilos de un mismo bloque.**

a. Defina en el kernel un locID usando los IDs de los hilos del bloque.

```
1 // a. Definir locID usando los IDs de los hilos del bloque
2 int locID = threadIdx.x;
```

b. Defina en el kernel un acumulador local en memoria compartida llamado localAcc, que tenga degreeBins \* rBins elementos.



```
1 // b. Definir un acumulador local en memoria compartida
2 __shared__ int localAcc[degreeBins * rBins];
```

c. Inicialice a 0 todos los elementos de este acumulador local. Recuerde que la memoria Compartida solamente puede manejarse desde el device (kernel).



```
1 // c. Inicializar a 0 todos los elementos del acumulador local
2 localAcc[locID] = 0;
```

d. Incluya una barrera para los hilos del bloque que controle que todos los hilos hayan completado el proceso de inicialización del acumulador local.



```
1 // d. Barrera para asegurar que todos los hilos hayan completado la inicialización
2 __syncthreads();
```

e. Modifique la actualización del acumulador global acc para usar el acumulador local localAcc. Para coordinar el acceso a memoria y garantizar que la operación de suma sea completada por cada hilo, use una operación de suma atómica.



```
1 // e. Actualizar el acumulador global acc usando el acumulador local localAcc
2 atomicAdd(&localAcc[rIdx * degreeBins + tIdx], 1);
```

f. Incluya una segunda barrera para los hilos del bloque que controle que todos los hilos hayan completado el proceso de incremento del acumulador local.

```
1 // f. Barrera para asegurar que todos los hilos hayan completado el proceso de incremento del acumulador local
2 __syncthreads();
```

g. Agregue un loop al final del kernel que inicie en locID hasta degreeBins \* rBins . Este loop sumará los valores del acumulador local localAcc al acumulador global acc.

```
1 // g. Loop para sumar los valores del acumulador local localAcc al acumulador global acc
2 for (int i = locID; i < degreeBins * rBins; i += blockDim.x)
3 {
4     atomicAdd(&acc[i], localAcc[i]);
5 }
```

**Compile y ejecute la nueva versión del programa usando memoria Global, Compartida y Constante. Registre en su bitácora los tiempos para la nueva versión usando CUDA events.**

```
[52] !chmod +x houghGlobal
!./houghGlobal runway.pgm
```

```
Calculation mismatch at : 1803 1446 1445
Calculation mismatch at : 1893 1506 1507
Calculation mismatch at : 5931 1653 1654
Calculation mismatch at : 6021 1816 1815
Calculation mismatch at : 6104 1642 1641
Calculation mismatch at : 6194 1586 1587
Done!
Tiempo transcurrido: 1.5424 ms
```

```
[53] !chmod +x houghConstant
!./houghConstant runway.pgm
```

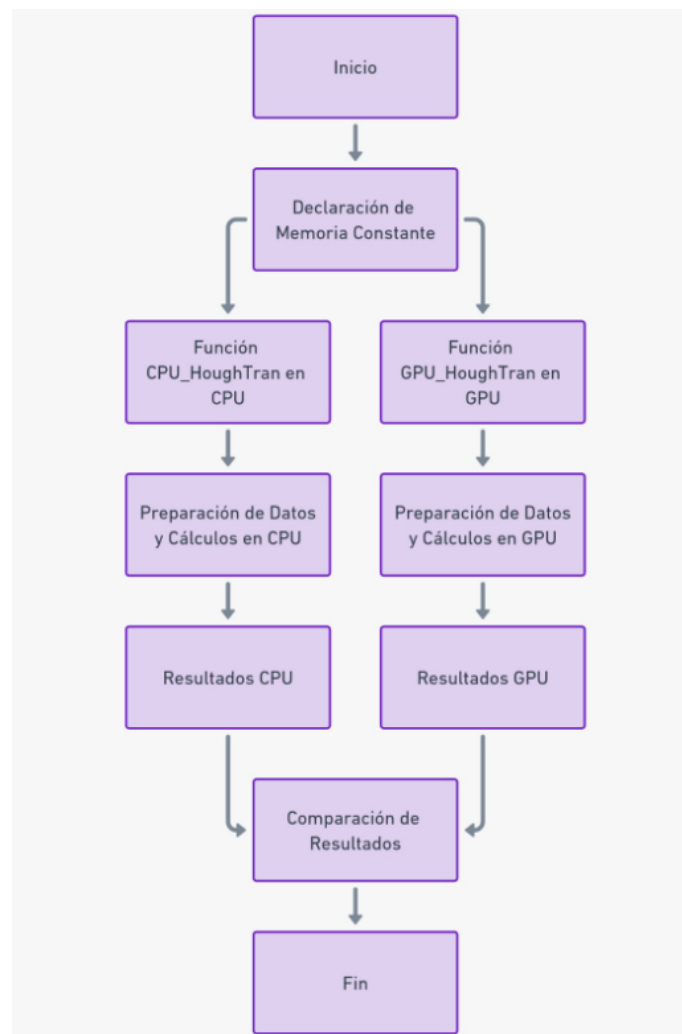
```
Calculation mismatch at : 1803 1446 1445
Calculation mismatch at : 1893 1506 1507
Calculation mismatch at : 5931 1653 1654
Calculation mismatch at : 6021 1816 1815
Calculation mismatch at : 6104 1642 1641
Calculation mismatch at : 6194 1586 1587
Done!
Tiempo transcurrido: 1.4997 ms
```

```
[67] !chmod +x houghShared
!./houghShared runway.pgm
```

```
Done!
Tiempo transcurrido: 1.1817 ms
```

**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. 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, se implementó memoria compartida para mejorar el rendimiento del kernel. La memoria compartida se utilizó para almacenar un acumulador local, denominado localAcc, que tiene dimensiones de  $\text{degreeBins} * \text{rBins}$ . Este acumulador local se inicializa a cero para cada hilo dentro de un bloque y se sincroniza con una barrera para asegurar que todos los hilos completen la inicialización antes de su uso. Durante la fase de procesamiento, cada hilo realiza operaciones de actualización en el acumulador local basadas en los píxeles relevantes de la imagen. Luego, se sincroniza nuevamente para garantizar que todos los hilos hayan completado sus operaciones antes de sumar los valores del acumulador local al acumulador global mediante operaciones atómicas. Este diseño reduce las colisiones en la escritura concurrente y minimiza las operaciones atómicas, mejorando la eficiencia del kernel. Los resultados muestran que la implementación de memoria compartida logra una notable mejora en el tiempo de ejecución, con una reducción significativa en comparación con las versiones que utilizan memoria global y constante.



*Diagrama 2: Diagrama memoria compartida*

## Discusión

La comparación de los tiempos de ejecución promedio de los tres programas que implementan la Transformada de Hough en CUDA revela diferencias significativas en términos de eficiencia y rendimiento. Los resultados muestran que la versión que utiliza memoria compartida supera a las implementaciones con memoria global y constante, lo que sugiere una optimización efectiva en el manejo de la memoria y el acceso a los datos.

La implementación con memoria global, que presenta un tiempo promedio de 1.5356, utiliza la memoria global de la GPU para almacenar tanto la imagen de entrada como el acumulador resultante. Aunque este enfoque es sencillo y fácil de implementar, tiene limitaciones en términos de rendimiento. Cada hilo accede directamente a la memoria global para leer y escribir datos, lo que puede resultar en cuellos de botella debido a la latencia asociada con la transferencia de datos entre la CPU y la GPU. Además, la dependencia constante de la memoria global puede generar conflictos y retrasos en la ejecución paralela, ya que varios hilos compiten por el mismo recurso de memoria.

En contraste, la implementación con memoria constante, que muestra un tiempo promedio de 1.5118, mejora el rendimiento al almacenar los valores precalculados de cosenos y senos en la memoria constante de la GPU. Este enfoque minimiza la necesidad de acceder repetidamente a la memoria global para obtener estos valores trigonométricos, lo que debería reducir el tiempo de ejecución. Sin embargo, a pesar de la optimización en el acceso a los datos trigonométricos, esta implementación sigue dependiendo en gran medida de la memoria global para la imagen de entrada y el acumulador, limitando su capacidad para aprovechar al máximo el paralelismo.

La implementación con memoria compartida, que exhibe un tiempo promedio de 1.1803, demuestra un rendimiento sobresaliente en comparación con las otras dos variantes. En este enfoque, cada bloque de hilos tiene su propia memoria compartida (localAcc) para reducir la latencia y mejorar la eficiencia en el acceso a los píxeles de la imagen. Esta estrategia minimiza las colisiones en el acceso a la memoria, ya que los hilos dentro de un bloque cooperan para procesar datos locales antes de realizar una operación de suma atómica para actualizar el acumulador global. La memoria compartida, al ser más rápida y con menor latencia que la global, permite un acceso eficiente a los datos locales y reduce los cuellos de botella, lo que se traduce en un tiempo de ejecución más corto.

En términos de jerarquía de memoria y eficiencia, la memoria compartida se sitúa en un nivel intermedio entre la memoria global y la memoria constante. Aunque cada enfoque tiene sus propias ventajas y limitaciones, la elección entre ellos debe basarse en la naturaleza específica del problema y las características de acceso a la memoria. La optimización exitosa depende de encontrar un equilibrio adecuado entre el paralelismo, el acceso a la memoria y la cooperación eficiente entre los hilos.

## Conclusiones

1. **Impacto de la Jerarquía de Memoria:** La elección de memoria compartida en la implementación de la Transformada de Hough en CUDA demuestra que la jerarquía de memoria tiene un papel crítico en el rendimiento, destacando la necesidad de estrategias optimizadas para el acceso a datos locales y la minimización de colisiones en la escritura concurrente.
2. **Coordinación Efectiva de Hilos en Bloques:** La eficiencia de la memoria compartida resalta la importancia de la coordinación entre hilos dentro de bloques en arquitecturas CUDA, permitiendo una colaboración eficiente que minimiza las operaciones en la memoria global y reduce los cuellos de botella asociados con el acceso a la memoria.
3. **Adaptación del Enfoque al Problema:** La elección entre memoria global, constante o compartida debe basarse en las características específicas del problema, subrayando la necesidad de adaptar estrategias de manejo de memoria para maximizar el rendimiento según las demandas únicas de cada tarea en sistemas GPU.

## Recomendaciones

Se recomienda realizar un perfilamiento continuo de la aplicación CUDA utilizando herramientas como NVIDIA Nsight Systems para identificar y abordar posibles cuellos de botella en el rendimiento y optimizar la implementación. Además, se sugiere explorar estrategias híbridas que combinen distintos tipos de memoria (global, constante y compartida) para adaptarse de manera óptima a las necesidades específicas de acceso a los datos en diferentes etapas del algoritmo, potencialmente mejorando el rendimiento global de la aplicación.

## Apéndice

### Anexo 1: Comparación de tiempos de ejecución

Tabla 1: Comparaciones de tiempo de las ejecuciones

Tiempo Versión Global (ms)	Tiempo Versión Constante (ms)	Tiempo Versión Compartida (ms)
1.5370	1.5033	1.1817
1.5367	1.5156	1.1858
1.5256	1.5174	1.1796
1.5365	1.5154	1.1804

1.5386	1.5196	1.1817
1.5305	1.5030	1.1776
1.5325	1.5093	1.1772
1.5340	1.5216	1.1790
1.5396	1.4930	1.1800
1.5360	1.5100	1.1796

Tabla 2: Promedios de tiempo de las ejecuciones

Versión	Promedio de tiempo (ms)
Global	1.5356
Constante	1.5118
Compartida	1.1803

## Anexo 2: Evidencia de las ejecuciones

### Evidencia de la versión global

```
!chmod +x houghGlobal
!./houghGlobal runway.pgm
```

```
Calculation mismatch at : 1803 1446 1445
Calculation mismatch at : 1893 1506 1507
Calculation mismatch at : 5931 1653 1654
Calculation mismatch at : 6021 1816 1815
Calculation mismatch at : 6104 1642 1641
Calculation mismatch at : 6194 1586 1587
Done!
Tiempo transcurrido: 1.5370 ms
```

```
!chmod +x houghGlobal
!./houghGlobal runway.pgm
```

```
Calculation mismatch at : 1803 1446 1445
Calculation mismatch at : 1893 1506 1507
Calculation mismatch at : 5931 1653 1654
Calculation mismatch at : 6021 1816 1815
Calculation mismatch at : 6104 1642 1641
Calculation mismatch at : 6194 1586 1587
Done!
Tiempo transcurrido: 1.5367 ms
```

```
!chmod +x houghGlobal
!./houghGlobal runway.pgm
```

```
Calculation mismatch at : 1803 1446 1445
Calculation mismatch at : 1893 1506 1507
Calculation mismatch at : 5931 1653 1654
Calculation mismatch at : 6021 1816 1815
Calculation mismatch at : 6104 1642 1641
Calculation mismatch at : 6194 1586 1587
Done!
Tiempo transcurrido: 1.5256 ms
```

```
!chmod +x houghGlobal
!./houghGlobal runway.pgm
```

```
Calculation mismatch at : 1803 1446 1445
Calculation mismatch at : 1893 1506 1507
Calculation mismatch at : 5931 1653 1654
Calculation mismatch at : 6021 1816 1815
Calculation mismatch at : 6104 1642 1641
Calculation mismatch at : 6194 1586 1587
Done!
Tiempo transcurrido: 1.5365 ms
```



```
!chmod +x houghGlobal
!./houghGlobal runway.pgm

Calculation mismatch at : 1803 1446 1445
Calculation mismatch at : 1893 1506 1507
Calculation mismatch at : 5931 1653 1654
Calculation mismatch at : 6021 1816 1815
Calculation mismatch at : 6104 1642 1641
Calculation mismatch at : 6194 1586 1587
Done!
Tiempo transcurrido: 1.5386 ms
```

```
!chmod +x houghGlobal
!./houghGlobal runway.pgm

Calculation mismatch at : 1803 1446 1445
Calculation mismatch at : 1893 1506 1507
Calculation mismatch at : 5931 1653 1654
Calculation mismatch at : 6021 1816 1815
Calculation mismatch at : 6104 1642 1641
Calculation mismatch at : 6194 1586 1587
Done!
Tiempo transcurrido: 1.5305 ms
```

```
!chmod +x houghGlobal
!./houghGlobal runway.pgm

Calculation mismatch at : 1803 1446 1445
Calculation mismatch at : 1893 1506 1507
Calculation mismatch at : 5931 1653 1654
Calculation mismatch at : 6021 1816 1815
Calculation mismatch at : 6104 1642 1641
Calculation mismatch at : 6194 1586 1587
Done!
Tiempo transcurrido: 1.5325 ms
```

```
!chmod +x houghGlobal
!./houghGlobal runway.pgm

Calculation mismatch at : 1803 1446 1445
Calculation mismatch at : 1893 1506 1507
Calculation mismatch at : 5931 1653 1654
Calculation mismatch at : 6021 1816 1815
Calculation mismatch at : 6104 1642 1641
Calculation mismatch at : 6194 1586 1587
Done!
Tiempo transcurrido: 1.5340 ms
```

```
!chmod +x houghGlobal
!./houghGlobal runway.pgm

Calculation mismatch at : 1803 1446 1445
Calculation mismatch at : 1893 1506 1507
Calculation mismatch at : 5931 1653 1654
Calculation mismatch at : 6021 1816 1815
Calculation mismatch at : 6104 1642 1641
Calculation mismatch at : 6194 1586 1587
Done!
Tiempo transcurrido: 1.5396 ms
```

```
!chmod +x houghGlobal
!./houghGlobal runway.pgm

Calculation mismatch at : 1803 1446 1445
Calculation mismatch at : 1893 1506 1507
Calculation mismatch at : 5931 1653 1654
Calculation mismatch at : 6021 1816 1815
Calculation mismatch at : 6104 1642 1641
Calculation mismatch at : 6194 1586 1587
Done!
Tiempo transcurrido: 1.5360 ms
```

Evidencia de la versión constante

```
!chmod +x houghConstant
!./houghConstant runway.pgm

Calculation mismatch at : 1803 1446 1445
Calculation mismatch at : 1893 1506 1507
Calculation mismatch at : 5931 1653 1654
Calculation mismatch at : 6021 1816 1815
Calculation mismatch at : 6104 1642 1641
Calculation mismatch at : 6194 1586 1587
Done!
Tiempo transcurrido: 1.5033 ms
```

```
!chmod +x houghConstant
!./houghConstant runway.pgm

Calculation mismatch at : 1803 1446 1445
Calculation mismatch at : 1893 1506 1507
Calculation mismatch at : 5931 1653 1654
Calculation mismatch at : 6021 1816 1815
Calculation mismatch at : 6104 1642 1641
Calculation mismatch at : 6194 1586 1587
Done!
Tiempo transcurrido: 1.5156 ms
```

```
!chmod +x houghConstant
!./houghConstant runway.pgm

Calculation mismatch at : 1803 1446 1445
Calculation mismatch at : 1893 1506 1507
Calculation mismatch at : 5931 1653 1654
Calculation mismatch at : 6021 1816 1815
Calculation mismatch at : 6104 1642 1641
Calculation mismatch at : 6194 1586 1587
Done!
Tiempo transcurrido: 1.5174 ms
```

```
!chmod +x houghConstant
!./houghConstant runway.pgm

Calculation mismatch at : 1803 1446 1445
Calculation mismatch at : 1893 1506 1507
Calculation mismatch at : 5931 1653 1654
Calculation mismatch at : 6021 1816 1815
Calculation mismatch at : 6104 1642 1641
Calculation mismatch at : 6194 1586 1587
Done!
Tiempo transcurrido: 1.5154 ms
```

```
!chmod +x houghConstant
!./houghConstant runway.pgm

Calculation mismatch at : 1803 1446 1445
Calculation mismatch at : 1893 1506 1507
Calculation mismatch at : 5931 1653 1654
Calculation mismatch at : 6021 1816 1815
Calculation mismatch at : 6104 1642 1641
Calculation mismatch at : 6194 1586 1587
Done!
Tiempo transcurrido: 1.5196 ms
```

```
!chmod +x houghConstant
!./houghConstant runway.pgm

Calculation mismatch at : 1803 1446 1445
Calculation mismatch at : 1893 1506 1507
Calculation mismatch at : 5931 1653 1654
Calculation mismatch at : 6021 1816 1815
Calculation mismatch at : 6104 1642 1641
Calculation mismatch at : 6194 1586 1587
Done!
Tiempo transcurrido: 1.5030 ms
```

```
!chmod +x houghConstant
!./houghConstant runway.pgm

Calculation mismatch at : 1803 1446 1445
Calculation mismatch at : 1893 1506 1507
Calculation mismatch at : 5931 1653 1654
Calculation mismatch at : 6021 1816 1815
Calculation mismatch at : 6104 1642 1641
Calculation mismatch at : 6194 1586 1587
Done!
Tiempo transcurrido: 1.5093 ms
```

```
!chmod +x houghConstant
!./houghConstant runway.pgm

Calculation mismatch at : 1803 1446 1445
Calculation mismatch at : 1893 1506 1507
Calculation mismatch at : 5931 1653 1654
Calculation mismatch at : 6021 1816 1815
Calculation mismatch at : 6104 1642 1641
Calculation mismatch at : 6194 1586 1587
Done!
Tiempo transcurrido: 1.5216 ms
```

```
!chmod +x houghConstant
!./houghConstant runway.pgm

Calculation mismatch at : 1803 1446 1445
Calculation mismatch at : 1893 1506 1507
Calculation mismatch at : 5931 1653 1654
Calculation mismatch at : 6021 1816 1815
Calculation mismatch at : 6104 1642 1641
Calculation mismatch at : 6194 1586 1587
Done!
Tiempo transcurrido: 1.4930 ms
```

```
!chmod +x houghConstant
!./houghConstant runway.pgm

Calculation mismatch at : 1803 1446 1445
Calculation mismatch at : 1893 1506 1507
Calculation mismatch at : 5931 1653 1654
Calculation mismatch at : 6021 1816 1815
Calculation mismatch at : 6104 1642 1641
Calculation mismatch at : 6194 1586 1587
Done!
Tiempo transcurrido: 1.5100 ms
```

Evidencia de la versión compartida

```
!chmod +x houghShared
!./houghShared runway.pgm

Done!
Tiempo transcurrido: 1.1817 ms
```

```
!chmod +x houghShared
!./houghShared runway.pgm

Done!
Tiempo transcurrido: 1.1858 ms
```

```
!chmod +x houghShared
!./houghShared runway.pgm

Done!
Tiempo transcurrido: 1.1796 ms
```

```
!chmod +x houghShared
!./houghShared runway.pgm

Done!
Tiempo transcurrido: 1.1804 ms
```

```
!chmod +x houghShared
!./houghShared runway.pgm

Done!
Tiempo transcurrido: 1.1817 ms
```

```
!chmod +x houghShared
!./houghShared runway.pgm

Done!
Tiempo transcurrido: 1.1776 ms
```

```
!chmod +x houghShared
!./houghShared runway.pgm

Done!
Tiempo transcurrido: 1.1762 ms
```

```
!chmod +x houghShared
!./houghShared runway.pgm

Done!
Tiempo transcurrido: 1.1790 ms
```

```
!chmod +x houghShared
!./houghShared runway.pgm

Done!
Tiempo transcurrido: 1.1800 ms
```

```
!chmod +x houghShared
!./houghShared runway.pgm

Done!
Tiempo transcurrido: 1.1796 ms
```

### Anexo 3: Repositorio

Para poder acceder al código fuente completo de este proyecto y explorar en detalle la implementación de los modelos, puede visitar nuestro repositorio en [GitHub](#). Además, para mayor comodidad, proporcionaremos un archivo ZIP separado que contiene todos los archivos necesarios y el código utilizado en este proyecto.

## **Bibliografía**

Autor Desconocido. (s.f.). Hilos en CUDA (II). Recuperado de:

<https://blitzman.gitbooks.io/cuda/content/modelo-de-memoria.html>

Autor Desconocido. (s.f.). Modelo de Memoria. Recuperado de:

<https://blitzman.gitbooks.io/cuda/content/modelo-de-memoria.html>

Lee, S. (1 de mayo de 2020). Lines Detection with Hough Transform. Recuperado:

<https://towardsdatascience.com/lines-detection-with-hough-transform-84020b3b1549>

Arceo, L. (s.f.). Algoritmo rápido de la transformada de Hough para detección de líneas rectas en una imagen. Recuperado de:

<http://www.progmat.uaem.mx:8080/Vol7num2/vol7num2art2.pdf>