



Universidad del Valle de Guatemala
Facultad de Ingeniería
Departamento de Ciencias de la Computación
CC3069 Computación Paralela y Distribuida
Catedrático: Miguel Novella
Ciclo 1 de 2023

Proyecto 3 - Programación Híbrida con CUDA

Bryann Alfaro 19372
Raul Jimenez 19017
Donaldo Garcia 19683

Link al repo: <https://github.com/bryannalfaro/proyecto3paralela>

Link al video: <https://youtu.be/Uayu8b77wDg>

Guatemala, 01 de junio de 2023

ÍNDICE

ÍNDICE	1
INTRODUCCIÓN	2
ANTECEDENTES	2
Primeros equipos gráficos, Apple y CUDA	2
¿Qué es CUDA?	3
Ejemplos de usos	4
Otras herramientas	4
DESARROLLO	5
CONCLUSIONES	12
RECOMENDACIONES	12
RETOS	12
APÉNDICE	13
Instalación de CUDA toolkit en Windows	13
ANEXO 1 - Bitácora	15
Tiempos con kernel global	16
Tiempos con kernel global y constante	17
Tiempos con kernel global, constante y compartido	20
ANEXO 2 - Uso de memoria constante	21
Explicación	21
Diagrama	22
Mejora en tiempo	22
ANEXO 3 - Uso de memoria compartida	23
Explicación	23
Diagrama	23
Mejora en tiempo	24
LITERATURA CITADA	25

INTRODUCCIÓN

La realización de cálculos complejos se vuelve algo necesario en el día a día , debido a la gran cantidad de información que se genera y se necesita procesar para poder obtener datos valiosos. El poder tener computacionalmente una forma efectiva de realizar estos cálculos puede representar un gran cambio. La utilización de GPUs hace más robusto este proceso.

En cuanto a hardware NVIDIA, una forma de sencilla de interactuar con esto es por medio de CUDA (Compute Unified Device Architecture) la cual es una plataforma de computación de manera paralela que incluye herramientas de desarrollo para codificar en GPUs de NVIDIA. (NVIDIADeveloper, s.f)

En este proyecto se utiliza la paralelización por núcleos que provee CUDA para poder realizar el cálculo del algoritmo para la transformada de Hough la cual permite computacionalmente detectar líneas rectas en una imagen.

Se busca además, el poder conocer y observar las características acerca de los niveles de memoria que ofrece la utilización de GPUs tanto memoria global, constante y compartida.

ANTECEDENTES

Primeros equipos gráficos, Apple y CUDA

En el año 2001, la empresa Apple hizo el lanzamiento de un equipo el cual incluía gráficos de NVIDIA. Sin embargo, a pesar de que la relación que existía entre NVIDIA y APPLE, en el año 2004 ocurrió un retraso en el lanzamiento del monitor Cinema HD Display debido a que NVIDIA no tuvo la capacidad de brindar el chip gráfico GeForce 6800 Ultra DDL para que el monitor pudiera funcionar de manera adecuada (Fernández, 2019).

Seguidamente, en el año 2007, NVIDIA enviaba a sus clientes chipsets de la gama G84 y G86. Estos fueron colocados incluso en algunos MacBook Pro, como lo fue el caso de la Early 2008. El problema fue que estos chips presentaban varios defectos provocando la distorsión de imagen o incluso no mostrar nada al conectarse a monitores externos (Fernández, 2019).

Esta situación provocó una demanda contra NVIDIA y Apple tuvo que absorber las consecuencias de esta situación y reemplazar y asumir el costo de los equipos. Alrededor de este tiempo NVIDIA desarrolló también una manera de controlar las comunicaciones entre equipo y memoria por GPU. Este controlador mejoraba en gran medida el rendimiento gráfico, sin embargo, provocó un conflicto con INTEL y que duró hasta 2011 afectando nuevamente a APPLE (Fernández, 2019).

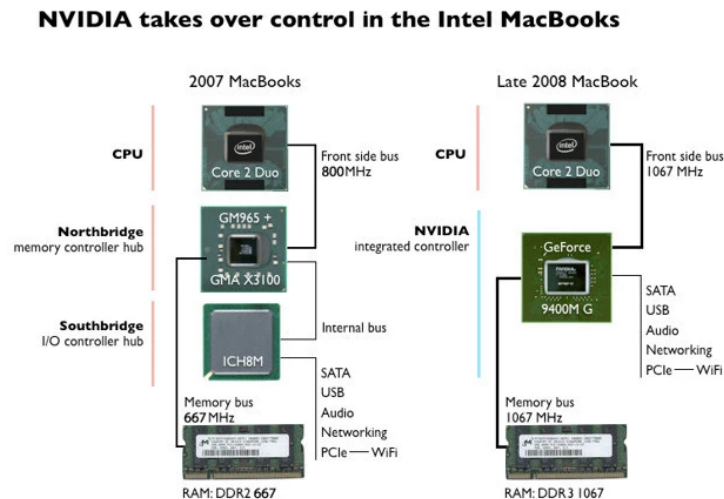


Imagen 1. Tecnología NVIDIA ignorando NorthBridge y SouthBridge. Extraído de: Fernandez, 2019

Otro conflicto ocurrido fue al momento de implementar chips gráficos en el iPhone, ya que Apple no confiaba en ATI o en NVIDIA por lo que solicitó a Samsung este dispositivo. Sin embargo, NVIDIA demandó a Qualcomm por inventar el término GPU. El detonante de esta pelea entre Apple y NVIDIA fue el consumo energético que los chips de NVIDIA presentan ya que si bien las MacBook Pro presentan problemas térmicos utilizando GPU de AMD, con NVIDIA serían problemas serios (Fernández, 2019).

Cuando Apple dejó de incluir los controladores de GPU NVIDIA, la relación entre ambas compañías se cerró completamente. Esto también afectó indirectamente al estándar de la industria CUDA (Fernández, 2019).

CUDA es una librería cerrada y perteneciente a NVIDIA para poder realizar cálculos computacionales. Esta ha demostrado tener mejores resultados que otras opciones como OpenCL. En cuanto a Apple, no le parece el soporte a librerías cerradas que no tienen el control, por lo que crearon su estándar Metal. Sin embargo CUDA es generalizado y ha limitado el desarrollo de Apple profesionalmente (Fernández, 2019).

¿Qué es CUDA?

CUDA por sus siglas en inglés Compute Unified Device Architecture fue desarrollada por NVIDIA para sacarle el mayor provecho posible a las unidades de procesamiento básico

(GPU). Esta es una plataforma de programación paralela que permite a los desarrolladores escribir código en varios lenguajes entre ellos C, C++ y Fortran, para ser ejecutado por la GPUs NVIDIA. La gran ventaja de esta plataforma es que permite aprovechar al máximo el paralelismo y así acelerar el rendimiento de las aplicaciones por medio de herramientas y extensiones que se encuentran incluidas. (NVIDIA, s.f)

La arquitectura CUDA se basa en el modelo de programación de "computación heterogénea", donde la CPU y la GPU trabajan en conjunto para realizar tareas de computación de manera eficiente. La GPU se encarga de las tareas intensivas en paralelo, mientras que la CPU se encarga de las tareas secuenciales y de la gestión general del sistema. Esta combinación de potencia de procesamiento de la GPU y la flexibilidad de la CPU permite acelerar una amplia gama de aplicaciones, incluyendo simulaciones científicas, análisis de datos, inteligencia artificial y más. (NVIDIA, s.f)

Ejemplos de usos

CUDA ofrece un amplio rango de aplicaciones en diferentes campos, permitiendo el procesamiento masivo de datos y la aceleración de algoritmos computacionalmente intensivos. Un ejemplo destacado es su aplicación en la simulación de fluidos y dinámica de fluidos computacional. Mediante CUDA, los científicos e ingenieros pueden utilizar GPUs para simular y visualizar el comportamiento de fluidos de manera más rápida y precisa. Esto resulta especialmente útil en campos como la ingeniería aeroespacial, la meteorología y la industria del petróleo y gas, donde se requieren cálculos complejos para entender y predecir el flujo de fluidos en diferentes condiciones. (Lanchas, L. 2017)

Otro gran ejemplo del uso de CUDA es para el aprendizaje de máquina y la inteligencia artificial. Estos algoritmos de aprendizaje automático requieren de múltiples cálculos y operaciones matemáticas simultáneas y en algunos casos incluso en paralelo, lo que lo hace apto para usar GPUs. Con esta plataforma los desarrolladores pueden entrenar modelos de forma más rápida y manejar grandes conjuntos de datos con una mayor eficiencia. Esto incluso ha favorecido grandes proyectos en áreas de reconocimiento de imágenes, procesamiento de lenguaje natural y la visión por computadora.

Otras herramientas

CUDA permite el uso de programación paralela por medio de la GPU, sin embargo esta no es la única plataforma que permite esto. Existen otras herramientas y frameworks que ofrecen la capacidad de procesamiento de GPU. Uno de los ejemplos más conocidos es OpenCL, esta plataforma permite acelerar el rendimiento de aplicaciones debido a que es

compatible con una amplia gama de dispositivos y fabricantes, lo cual brinda una mayor flexibilidad en comparación de CUDA que está únicamente limitado por GPUs de NVIDIA.

Otra de las grandes herramientas es Tensor Flow, el cual es un framework de aprendizaje automático desarrollado por Google. Este framework proporciona una interfaz sencilla y eficiente para el uso de desarrollo de modelos de aprendizaje de máquina y soporta el uso de GPU. Aunque inicialmente el objetivo inicial de este framework era ser utilizado en combinación con NVIDIA y CUDA con el paso del tiempo ha evolucionado y hoy en día brinda soporte a varias bibliotecas y APIs. Lo cual permite aprovechar el poder de las GPUs en varios dispositivos. (Tensor Flow, s.f.)

DESARROLLO

La Transformada de Hough es una técnica de extracción de características utilizada en la visión artificial y el procesamiento de imágenes para detectar formas geométricas simples, como líneas, círculos y elipses, en una imagen. Fue propuesto por Richard Duda y Peter Hart en 1972.

La transformada de Hough funciona asignando puntos en un espacio de imagen a un espacio de parámetro, donde cada punto en el espacio de imagen corresponde a una curva o una línea en el espacio de parámetro. El espacio de parámetros está definido por los parámetros de la forma geométrica que se detecta, como la pendiente y la intersección de una línea, o el centro y el radio de un círculo.

Por ejemplo, en el caso de la detección de líneas, cada punto (x, y) en el espacio de la imagen se transforma en una línea en el espacio de parámetros, definido por la ecuación $y = mx + b$. El espacio de parámetros se discretiza en contenedores, y para cada punto (x, y) en el espacio de la imagen, la línea correspondiente en el espacio de parámetros incrementa el contenedor que intersecta. El contenedor con el recuento más alto representa la línea que mejor se ajusta a los puntos de la imagen.

Hough Transform es resistente al ruido y puede manejar la oclusión parcial y los espacios en las formas detectadas. Es ampliamente utilizado en diversas aplicaciones, como la detección de carriles en la conducción autónoma, el reconocimiento de objetos y la segmentación de imágenes.

También hay variaciones de Hough Transform, como Probabilistic Hough Transform, que muestrea aleatoriamente puntos del espacio de la imagen para acelerar el cálculo, y Generalized Hough Transform, que puede detectar formas más complejas más allá de líneas y círculos.

En general, Hough Transform es una herramienta poderosa para detectar formas geométricas simples en imágenes y ha sido ampliamente adoptada en aplicaciones de procesamiento de imágenes y visión por computadora.

La implementación de Hough Transform en CUDA implica aprovechar las capacidades de cómputo paralelo de las GPU de NVIDIA para acelerar el cómputo. CUDA es una plataforma de computación

paralela y un modelo de programación que permite a los desarrolladores aprovechar el poder de las GPU para la computación de propósito general.

Para implementar la Transformada de Hough en CUDA, se pueden seguir los siguientes pasos:

1. Asignación de memoria: asigna memoria en la GPU para la imagen de entrada y la matriz de acumuladores utilizada en Hough Transform.
2. Transferencia de datos: transfiera los datos de la imagen de entrada desde la CPU a la memoria de la GPU.
3. Definición de kernel: define un kernel CUDA, que es una función que varios subprocesos ejecutarán en paralelo en la GPU. Cada hilo procesa un píxel diferente de la imagen de entrada.
4. Ejecución de subprocesos: inicia el núcleo CUDA, especificando la cantidad de subprocesos que se ejecutarán en paralelo. Cada subproceso realizará el cálculo de la Transformada de Hough para su píxel asignado.
5. Actualizaciones del acumulador: dentro del núcleo CUDA, para cada píxel de borde en la imagen de entrada, calcule la curva o línea correspondiente en el espacio de parámetros e incremente el contenedor correspondiente en la matriz del acumulador.
6. Sincronización: Sincroniza los subprocesos para asegurarse de que se completen todos los cálculos.
7. Transferencia de resultados: transfiera la matriz de acumuladores de la memoria de la GPU a la memoria de la CPU.
8. Postprocesamiento: analiza la matriz de acumuladores en la CPU para determinar las formas detectadas, como líneas o círculos, y sus parámetros.

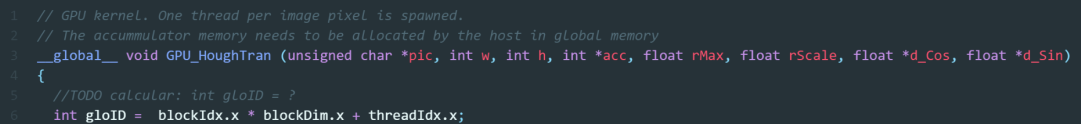
Al implementar Hough Transform en CUDA, la naturaleza paralela de la computación GPU puede acelerar significativamente la computación en comparación con una implementación secuencial basada en CPU. La gran cantidad de subprocesos que se ejecutan simultáneamente en la GPU permite un procesamiento eficiente de los píxeles de la imagen y las actualizaciones del acumulador, lo que resulta en una detección más rápida de las formas en la imagen.

Es importante tener en cuenta que los detalles de implementación específicos pueden variar según el lenguaje de programación y la arquitectura de GPU utilizada. CUDA proporciona un marco de programación y bibliotecas que permiten a los desarrolladores utilizar de manera eficiente la GPU para varios cálculos, incluida la transformación de Hough.

Hough Transform es un algoritmo computacionalmente intensivo, y aprovechar los diferentes tipos de memoria disponibles en una GPU puede afectar significativamente su rendimiento. En la versión lineal de Hough Transform, los principales tipos de memoria GPU que se pueden utilizar son la memoria global, la memoria compartida y la memoria constante.

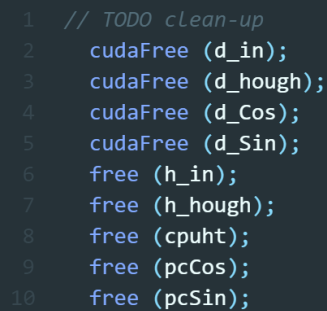
1. Compile y corra el programa con la versión CUDA que usa memoria Global únicamente. Revise el Makefile y ajústelo de ser necesario. Deberá completar los elementos faltantes relativos a CUDA en el kernel o en el main.

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 // GPU kernel. One thread per image pixel is spawned.
2 // The accumulator memory needs to be allocated by the host in global memory
3 __global__ void GPU_HoughTran (unsigned char *pic, int w, int h, int *acc, float rMax, float rScale, float *d_Cos, float *d_Sin)
4 {
5     //TODO calcular: int gloID = ?
6     int gloID = blockIdx.x * blockDim.x + threadIdx.x;
```

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



```
1 // TODO clean-up
2 cudaFree (d_in);
3 cudaFree (d_hough);
4 cudaFree (d_Cos);
5 cudaFree (d_Sin);
6 free (h_in);
7 free (h_hough);
8 free (cpuht);
9 free (pcCos);
10 free (pcSin);
```

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


```

1  int blockNum = ceil ((double)w * (double)h / (double)256);
2  cudaEventRecord(start);
3  GPU_HoughTran <<< blockNum, 256 >>> (d_in, w, h, d_hough, rMax, rScale, d_Cos, d_Sin);
4  cudaEventRecord(stop);
5
6
7  // get results from device
8  cudaMemcpy (h_hough, d_hough, sizeof (int) * degreeBins * rBins, cudaMemcpyDeviceToHost);
9  cudaEventSynchronize(stop);
10 float milliseconds = 0;
11
12 cudaEventElapsedTime(&milliseconds, start, stop);
13 printf("Tiempo de ejecucion: %f ms\n", milliseconds);
14

```

3. 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.

Para calcular estos valores se hace uso de los índices globales. Para Xcoord lo que se hace es sacar el mod del índice con el width y luego se le resta el centro de X para poder centrar las coordenadas sabiendo que el centro de la imagen está en Xcent. Luego para YCoord se divide el id global dentro del ancho de la imagen y luego esto se le resta al centro para centrar las coordenadas. Estos son los píxeles específicos en relación con el centro de la imagen para calcular la transformada de Hough.

4. Ya que estamos en temas de Computer Vision, acá aplica fuertemente el dicho de “una imagen vale más que mil palabras”. Modifique el programa y adicione como salida una imagen (.jpg o .png) con las líneas detectadas (Es decir, la imagen de entrada con las líneas dibujadas a color encima de la imagen blanco y negro). Para no dibujar todas las líneas posibles, dibuje solamente aquellas cuyo “peso” sea mayor a un Threshold arbitrario (i.e.: dibujas aquellas cuyo peso > threshold, o bien dibujar aquellas cuyo peso > promedio de pesos + 2 stdvs, etc).

5. 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
2 __constant__ float d_Cos[degreeBins];
3 __constant__ float d_Sin[degreeBins];
4 //TODO Kernel memoria Constante
5 __global__ void GPU_HoughTranConst(unsigned char *pic, int w, int h, int *acc, float rMax, float rScale )
6 {
7     //TODO calcular: int gloID = ?
8     int gloID = blockIdx.x * blockDim.x + threadIdx.x;
9
10    if (gloID > w * h) return;    // in case of extra threads in block
11
12    int xCent = w / 2;
13    int yCent = h / 2;
14
15    //TODO explicar bien esta parte. Dibujar un rectangulo a modo de imagen sirve para visualizarlo mejor
16    int xCoord = gloID % w - xCent;
17    int yCoord = yCent - gloID / w;
18
19    //TODO eventualmente usar memoria compartida para el acumulador
20
21    if (pic[gloID] > 0)
22    {
23        for (int tIdx = 0; tIdx < degreeBins; tIdx++)
24        {
25            //TODO utilizar memoria constante para senos y cosenos
26            //float r = xCoord * cos(tIdx) + yCoord * sin(tIdx); //probar con esto para ver diferencia en tiempo
27            float r = xCoord * d_Cos[tIdx] + yCoord * d_Sin[tIdx];
28            int rIdx = (r + rMax) / rScale;
29            //debemos usar atomic, pero que race condition hay si somos un thread por pixel? explique
30            atomicAdd (acc + (rIdx * degreeBins + tIdx), 1);
31        }
32    }
33 }

```

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

```

1 //TODO Kernel memoria Constante
2 __global__ void GPU_HoughTranConst(unsigned char *pic, int w, int h, int *acc, float rMax, float rScale )
3 {
4     //TODO calcular: int gloID = ?
5     int gloID = blockIdx.x * blockDim.x + threadIdx.x;
6
7     if (gloID > w * h) return;    // in case of extra threads in block
8
9     int xCent = w / 2;
10    int yCent = h / 2;
11
12    //TODO explicar bien esta parte. Dibujar un rectangulo a modo de imagen sirve para visualizarlo mejor
13    int xCoord = gloID % w - xCent;
14    int yCoord = yCent - gloID / w;
15
16    //TODO eventualmente usar memoria compartida para el acumulador
17
18    if (pic[gloID] > 0)
19    {
20        for (int tIdx = 0; tIdx < degreeBins; tIdx++)
21        {
22            //TODO utilizar memoria constante para senos y cosenos
23            //float r = xCoord * cos(tIdx) + yCoord * sin(tIdx); //probar con esto para ver diferencia en tiempo
24            float r = xCoord * d_Cos[tIdx] + yCoord * d_Sin[tIdx];
25            int rIdx = (r + rMax) / rScale;
26            //debemos usar atomic, pero que race condition hay si somos un thread por pixel? explique
27            atomicAdd (acc + (rIdx * degreeBins + tIdx), 1);
28        }
29    }
30 }

```

```

1  cudaEventRecord(start);
2  GPU_HoughTranConst <<< blockNum, 256 >>> (d_in, w, h, d_hough, rMax, rScale);
3  cudaEventRecord(stop);

```

c. Al usar memoria Constante, la forma de trasladar datos del host al device cambia. Modifique los respectivos enunciados para trasladar los valores precalculados de $\cos(\text{rad})$ y $\sin(\text{rad})$ del host a la memoria Constante del device usando `cudaMemcpyToSymbol`.

```

1
2  cudaMemcpyToSymbol(d_Cos, pcCos, sizeof (float) * degreeBins);
3  cudaMemcpyToSymbol(d_Sin, pcSin, sizeof (float) * degreeBins);
4

```

8. Modifique el programa anterior para incorporar memoria Compartida. Considere que la memoria Compartida es accesible únicamente a los hilos de un mismo bloque. Realice los siguientes cambios y/o ajustes:

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

```

5  if (gridID > w * h) return; // exit
6  int locID = threadIdx.x; //local ID
  int xCent = w / 2;
  int yCent = h / 2;

```

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

```

__shared__ int localAcc[rBins * degreeBins]; // local accumulator
int i;
//loop to initialize localAcc

```

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

```

int i,
//loop to initialize localAcc
for (i = locID; i < rBins * degreeBins; i+= blockDim.x){
    localAcc[i] = 0;
}
__syncthreads ();

if (pic[gloID] > 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.

```

    localAcc[i] = 0;
}
__syncthreads ();

if (pic[gloID] > 0)

```

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.

```

if (pic[gloID] > 0)
{
    for (int tIdx = 0; tIdx < degreeBins; tIdx++)
    {
        // we calculate the radius
        float r = xCoord * dCos[tIdx] + yCoord * dSin[tIdx];
        // we calculate the index of the radius
        int rIdx = (r + rMax) / rScale;
        // Because it is done based on the angles, it may be that at some point they will
        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.

```

if (pic[gloID] > 0)
{
    for (int tIdx = 0; tIdx < degreeBins; tIdx++)
    {
        // we calculate the radius
        float r = xCoord * dCos[tIdx] + yCoord * dSin[tIdx];
        // we calculate the index of the radius
        int rIdx = (r + rMax) / rScale;
        // Because it is done based on the angles, it may be that at
        atomicAdd (localAcc + (rIdx * degreeBins + tIdx), 1);
    }
}
__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.

```
//loop to add localAcc to acc
for (i = locID; i < rBins * degreeBins; i += blockDim.x){
    atomicAdd (&acc[i], localAcc[i] );
}
}
```

CONCLUSIONES

1. La presencia de GPUs en los computadores actuales puede brindar una nueva perspectiva y capacidad computacional para realizar acciones y cálculos complejos permitiendo obtener resultados que ayuden a un avance en la tecnología.
2. Se puede concluir que es necesaria la comprensión de las partes secuenciales y paralelas para poder sacarles el mayor provecho tanto a la CPU como a la GPU.
3. Se puede concluir que el uso de memoria consta ayuda en tareas que requieren tener un acceso a estos datos para realizar varios cálculos consecutivos.

RECOMENDACIONES

1. Comprender las capacidades que ofrece CUDA para la interacción con GPUs y las distintas memorias para poder realizar programas más poderosos y que tengan buen rendimiento computacionalmente.
2. Entender a profundidad el algoritmo de Hough para poder graficar las líneas con más peso.
3. Comprender la teoría de la memoria adecuadamente para poder utilizarlas dentro de los programas

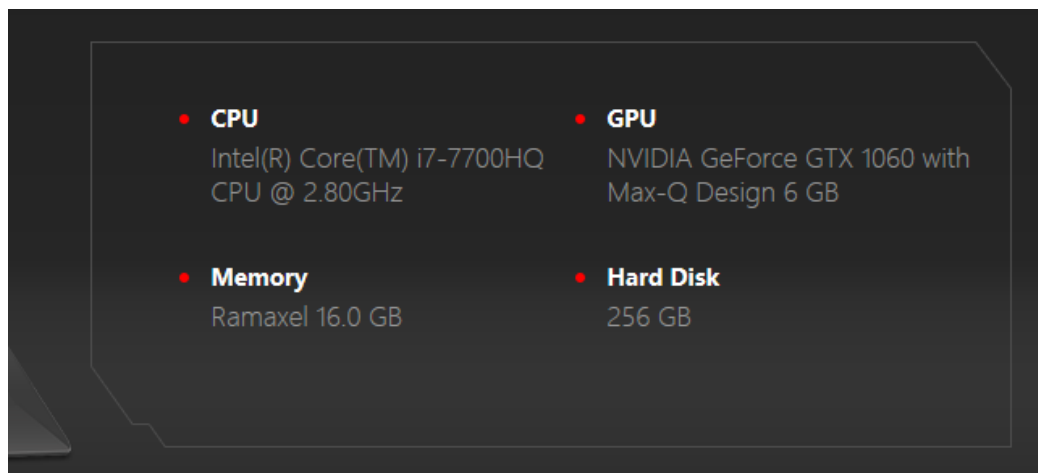
RETOS

1. Al inicio del proyecto nos encontramos con un problema en el código base. El cual constaba en que las variables que arroja el CPU no eran las mismas que daba el GPU. Esto se logró resolver haciendo uso de casteo de las variables que se le enviaban al kernel.
2. Otro reto que encontramos fue el comprender la funcionalidad e importancia de los distintos tipos de memorias, logramos entender por medio de lectura y documentación y desarrollar mejor el programa.
3. Un reto que afrontamos al final del último programa , era un mismatch en los cálculos del acumulador de CPU y GPU, sin embargo se soluciona aumentando la cantidad de hilos y bloque del kernel.

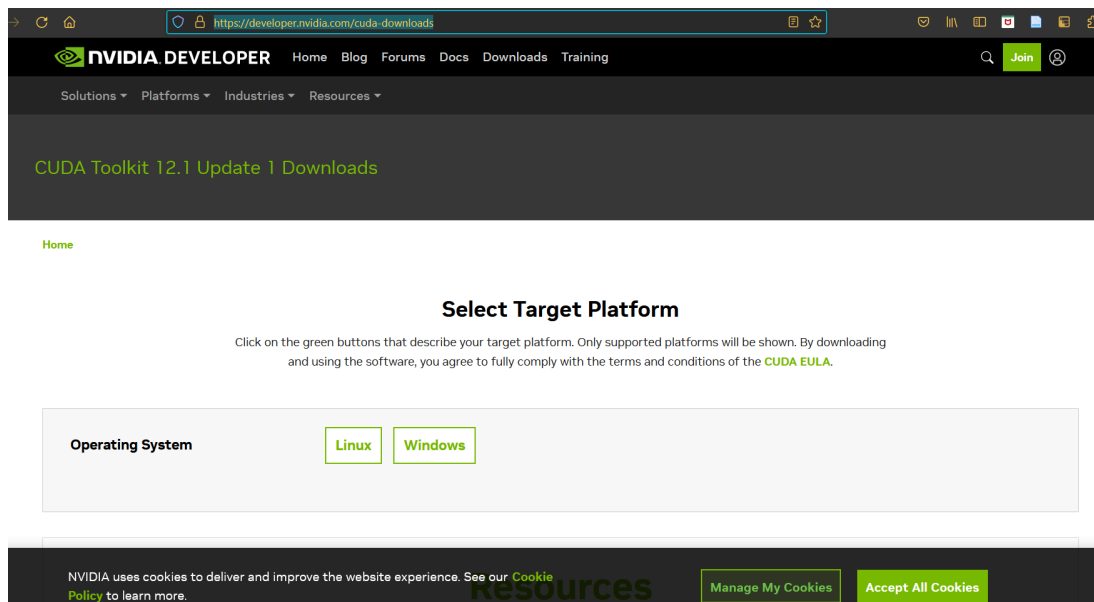
APÉNDICE

Instalación de CUDA toolkit en Windows

Para poder realizar una instalación de CUDA es importante en primer lugar asegurarse de tener NVIDIA. Esto puede visualizarse en las especificaciones del computador.



Luego, dirigirse a la página de CUDA Toolkit: <https://developer.nvidia.com/cuda-downloads>



Escoger las opciones de acuerdo al sistema operativo y arquitectura

Select Target Platform

Click on the green buttons that describe your target platform. Only supported platforms will be shown. By downloading and using the software, you agree to fully comply with the terms and conditions of the [CUDA EULA](#).

Operating System	Linux Windows
Architecture	x86_64
Version	10 11 Server 2019 Server 2022
Installer Type	exe (local) exe (network)

Luego se podrá realizar la instalación del exe

Download Installer for Windows 10 x86_64

The base installer is available for download below.

►Base Installer

[Download \(3.2 GB\)](#) 

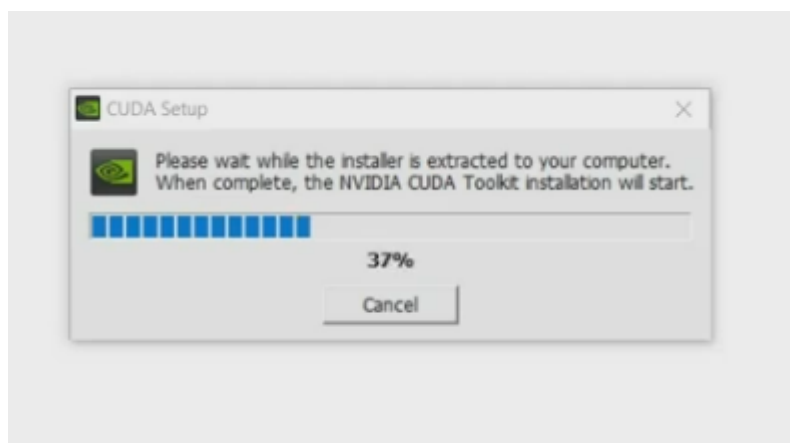
Installation Instructions:

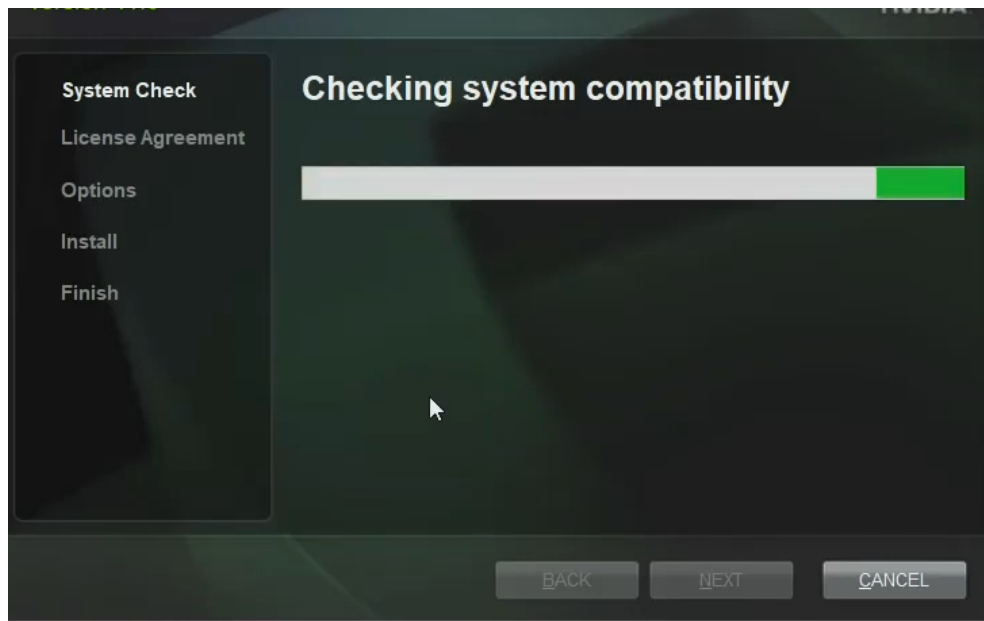
1. Double click cuda_12.1.1_531.14_windows.exe
2. Follow on-screen prompts

The checksums for the installer and patches can be found in [Installer Checksums](#).

For further information, see the [Installation Guide for Microsoft Windows](#) and the [CUDA Quick Start Guide](#).

Luego ejecutar el instalador y seguir los pasos que ahí se mencionan para tener completa la instalación.





ANEXO 1 - Bitácora

Tiempos con kernel global

```
PS C:\Users\Bryann\Desktop\9no semestre\Proyecto3Paralela> nvcc houghBase.cu pgm.cpp -o houghBase
se
houghBase.cu
tmpxft_00005cf0_00000000-10_houghBase.cudafe1.cpp
pgm.cpp
Creating library houghBase.lib and object houghBase.exp
PS C:\Users\Bryann\Desktop\9no semestre\Proyecto3Paralela> ./houghBase.exe ./cuadrosHough.pgm
Reading ./cuadrosHough.pgm
Image size is 300 x 300
Tiempo de ejecucion: 0.993184 ms
Done!
PS C:\Users\Bryann\Desktop\9no semestre\Proyecto3Paralela> ./houghBase.exe ./cuadrosHough.pgm
Reading ./cuadrosHough.pgm
Image size is 300 x 300
Tiempo de ejecucion: 1.005696 ms
Done!
PS C:\Users\Bryann\Desktop\9no semestre\Proyecto3Paralela> ./houghBase.exe ./cuadrosHough.pgm
Reading ./cuadrosHough.pgm
Image size is 300 x 300
Tiempo de ejecucion: 0.988448 ms
Done!
PS C:\Users\Bryann\Desktop\9no semestre\Proyecto3Paralela> ./houghBase.exe ./cuadrosHough.pgm
Reading ./cuadrosHough.pgm
Image size is 300 x 300
Tiempo de ejecucion: 1.015776 ms
Done!
PS C:\Users\Bryann\Desktop\9no semestre\Proyecto3Paralela> ./houghBase.exe ./cuadrosHough.pgm
Reading ./cuadrosHough.pgm
Image size is 300 x 300
Tiempo de ejecucion: 1.013056 ms
Done!
PS C:\Users\Bryann\Desktop\9no semestre\Proyecto3Paralela> ./houghBase.exe ./cuadrosHough.pgm
Reading ./cuadrosHough.pgm
Image size is 300 x 300
Tiempo de ejecucion: 0.978784 ms
Done!
PS C:\Users\Bryann\Desktop\9no semestre\Proyecto3Paralela> ./houghBase.exe ./cuadrosHough.pgm
Reading ./cuadrosHough.pgm
Image size is 300 x 300
Tiempo de ejecucion: 1.009280 ms
Done!
PS C:\Users\Bryann\Desktop\9no semestre\Proyecto3Paralela> ./houghBase.exe ./cuadrosHough.pgm
Reading ./cuadrosHough.pgm
Image size is 300 x 300
Tiempo de ejecucion: 1.009312 ms
Done!
PS C:\Users\Bryann\Desktop\9no semestre\Proyecto3Paralela> ./houghBase.exe ./cuadrosHough.pgm
Reading ./cuadrosHough.pgm
Image size is 300 x 300
Tiempo de ejecucion: 1.000256 ms
Done!
PS C:\Users\Bryann\Desktop\9no semestre\Proyecto3Paralela> ./houghBase.exe ./cuadrosHough.pgm
Reading ./cuadrosHough.pgm
Image size is 300 x 300
Tiempo de ejecucion: 0.977152 ms
Done!
PS C:\Users\Bryann\Desktop\9no semestre\Proyecto3Paralela> 
```

Iteración	Tiempo (milisegundos)
1	0,9932
2	1,0057

3	0,9884
4	1,0158
5	1,0131
6	0,9788
7	1,0093
8	1,0093
9	1,0003
10	0,9772
Promedio	0,9991

Tabla 1. Mediciones con memoria global.

Tiempos con kernel global y constante

```

PS C:\Users\Bryann\Desktop\9no semestre\Proyecto3Paralela> nvcc houghBaseConstante.cu pgm.cpp -
o houghBaseConstante
  houghBaseConstante.cu
  tmpxft_00001914_00000000-10_houghBaseConstante.cudafe1.cpp
  pgm.cpp
  Creating library houghBaseConstante.lib and object houghBaseConstante.exp
PS C:\Users\Bryann\Desktop\9no semestre\Proyecto3Paralela> ./houghBaseConstante.exe ./cuadrosHo
ugh.pgm
  Reading ./cuadrosHough.pgm
  Image size is 300 x 300
  Tiempo de ejecucion: 0.763680 ms
  Done!
PS C:\Users\Bryann\Desktop\9no semestre\Proyecto3Paralela> ./houghBaseConstante.exe ./cuadrosHo
ugh.pgm
  Reading ./cuadrosHough.pgm
  Image size is 300 x 300
  Tiempo de ejecucion: 0.804320 ms
  Done!
PS C:\Users\Bryann\Desktop\9no semestre\Proyecto3Paralela> ./houghBaseConstante.exe ./cuadrosHo
ugh.pgm
  Reading ./cuadrosHough.pgm
  Image size is 300 x 300
  Tiempo de ejecucion: 0.754880 ms
  Done!
PS C:\Users\Bryann\Desktop\9no semestre\Proyecto3Paralela> ./houghBaseConstante.exe ./cuadrosHo
ugh.pgm
  Reading ./cuadrosHough.pgm
  Image size is 300 x 300
  Tiempo de ejecucion: 0.762912 ms
  Done!
PS C:\Users\Bryann\Desktop\9no semestre\Proyecto3Paralela> ./houghBaseConstante.exe ./cuadrosHo
ugh.pgm
  Reading ./cuadrosHough.pgm
  Image size is 300 x 300
  Tiempo de ejecucion: 0.752688 ms
  Done!
PS C:\Users\Bryann\Desktop\9no semestre\Proyecto3Paralela> ./houghBaseConstante.exe ./cuadrosHo
ugh.pgm
  Reading ./cuadrosHough.pgm
  Image size is 300 x 300
  Tiempo de ejecucion: 0.795456 ms
  Done!
PS C:\Users\Bryann\Desktop\9no semestre\Proyecto3Paralela> ./houghBaseConstante.exe ./cuadrosHo
ugh.pgm
  Reading ./cuadrosHough.pgm
  Image size is 300 x 300
  Tiempo de ejecucion: 0.764992 ms
  Done!
PS C:\Users\Bryann\Desktop\9no semestre\Proyecto3Paralela> ./houghBaseConstante.exe ./cuadrosHo
ugh.pgm
  Reading ./cuadrosHough.pgm
  Image size is 300 x 300
  Tiempo de ejecucion: 0.756672 ms
  Done!
PS C:\Users\Bryann\Desktop\9no semestre\Proyecto3Paralela>
PS C:\Users\Bryann\Desktop\9no semestre\Proyecto3Paralela> ./houghBaseConstante.exe ./cuadrosHo
ugh.pgm
  Reading ./cuadrosHough.pgm
  Image size is 300 x 300
  Tiempo de ejecucion: 0.777888 ms
  Done!
PS C:\Users\Bryann\Desktop\9no semestre\Proyecto3Paralela> ./houghBaseConstante.exe ./cuadrosHo
ugh.pgm
  Reading ./cuadrosHough.pgm
  Image size is 300 x 300
  Tiempo de ejecucion: 0.770304 ms
  Done!

```

Iteración	Tiempo (milisegundos)
1	0,7637
2	0,8043
3	0,7549
4	0,7629

5	0,7526
6	0,7955
7	0,7650
8	0,7567
9	0,7779
10	0,7703
Promedio	0,7704

Tabla 2. Mediciones con memoria global y constante

Tiempos con kernel global, constante y compartido

```
PS C:\Users\Bryann\Desktop\9no semestre\Proyecto3Paralela> ./houghBaseConstanteShared.exe
e ./cuadrosHough.pgm
Reading ./cuadrosHough.pgm
Image size is 300 x 300
Tiempo de ejecucion: 0.371840 ms
Done!
PS C:\Users\Bryann\Desktop\9no semestre\Proyecto3Paralela> ./houghBaseConstanteShared.exe
e ./cuadrosHough.pgm
Reading ./cuadrosHough.pgm
Image size is 300 x 300
Tiempo de ejecucion: 0.449248 ms
Done!
PS C:\Users\Bryann\Desktop\9no semestre\Proyecto3Paralela> ./houghBaseConstanteShared.exe
e ./cuadrosHough.pgm
Reading ./cuadrosHough.pgm
Image size is 300 x 300
Tiempo de ejecucion: 0.458816 ms
Done!
PS C:\Users\Bryann\Desktop\9no semestre\Proyecto3Paralela> ./houghBaseConstanteShared.exe
e ./cuadrosHough.pgm
Reading ./cuadrosHough.pgm
Image size is 300 x 300
Tiempo de ejecucion: 0.468896 ms
Done!
PS C:\Users\Bryann\Desktop\9no semestre\Proyecto3Paralela> ./houghBaseConstanteShared.exe
e ./cuadrosHough.pgm
Reading ./cuadrosHough.pgm
Image size is 300 x 300
Tiempo de ejecucion: 0.439552 ms
Done!
PS C:\Users\Bryann\Desktop\9no semestre\Proyecto3Paralela> ./houghBaseConstanteShared.exe
e ./cuadrosHough.pgm
Reading ./cuadrosHough.pgm
Image size is 300 x 300
Tiempo de ejecucion: 0.439712 ms
Done!
PS C:\Users\Bryann\Desktop\9no semestre\Proyecto3Paralela> ./houghBaseConstanteShared.exe
e ./cuadrosHough.pgm
Reading ./cuadrosHough.pgm
Image size is 300 x 300
Tiempo de ejecucion: 0.431552 ms
Done!
PS C:\Users\Bryann\Desktop\9no semestre\Proyecto3Paralela> ./houghBaseConstanteShared.exe
e ./cuadrosHough.pgm
Reading ./cuadrosHough.pgm
Image size is 300 x 300
Tiempo de ejecucion: 0.430016 ms
Done!
PS C:\Users\Bryann\Desktop\9no semestre\Proyecto3Paralela> ./houghBaseConstanteShared.exe
e ./cuadrosHough.pgm
Reading ./cuadrosHough.pgm
Image size is 300 x 300
Tiempo de ejecucion: 0.376736 ms
Done!
PS C:\Users\Bryann\Desktop\9no semestre\Proyecto3Paralela> ./houghBaseConstanteShared.exe
e ./cuadrosHough.pgm
Reading ./cuadrosHough.pgm
Image size is 300 x 300
Tiempo de ejecucion: 0.380896 ms
Done!
```

Iteración	Tiempo (milisegundos)
1	0,3718
2	0,4492
3	0,4588

4	0,4689
5	0,4396
6	0,4397
7	0,4316
8	0,4300
9	0,3767
10	0,3809
Promedio	0,4247

Tabla 3. Mediciones con memoria global, constante y compartida

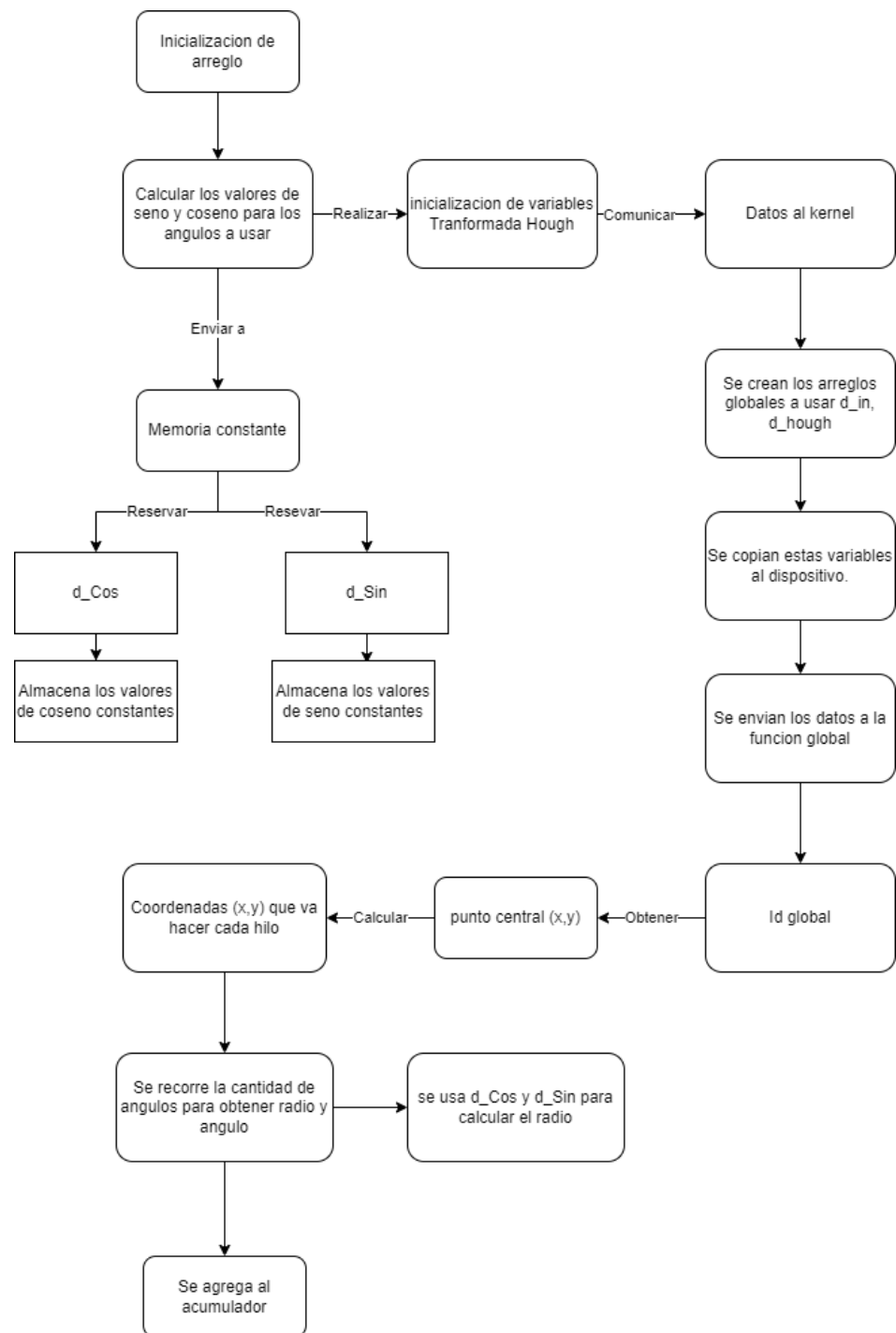
ANEXO 2 - Uso de memoria constante

Explicación

En el programa de memoria constante lo que se hizo fue almacenar los valores de seno y coseno en un arreglo de tamaño *degreeBins* en donde se tiene cada una de estas posiciones, para que nuestro bloques no tuvieran que estar recalculando estos valores constantemente para los pixeles. Estos valores son necesarios para los cálculos de la transformada de Hough y se utilizaron en la partes de memoria global como *d_Cos* y *d_Sen*.

En el kernel del GPU usamos un ciclo for para recorrer todo nuestro arreglo de tamaño *degreeBins* usando *tldx* como el ángulo para obtener los valores de coseno y seno de nuestras variables constantes y esto se usó para obtener el valor de la variable de radio. Seguidamente se usó la operación de acumulacion en el arreglo, usando *atomicAdd*, para evitar que existían race conditions en el acceso de nuestra memoria.

Diagrama



Mejora en tiempo

Tomando en cuenta los resultados obtenidos en la tabla 1 y 2, se puede observar una mejora en el tiempo, ya que, utilizando solamente memoria global el tiempo promedio fue de 0.9991 ms y al agregar la característica de memoria constante al kernel, el tiempo promedio fue de 0.7704 ms, representando una disminución en tiempo de 0.2287 ms.

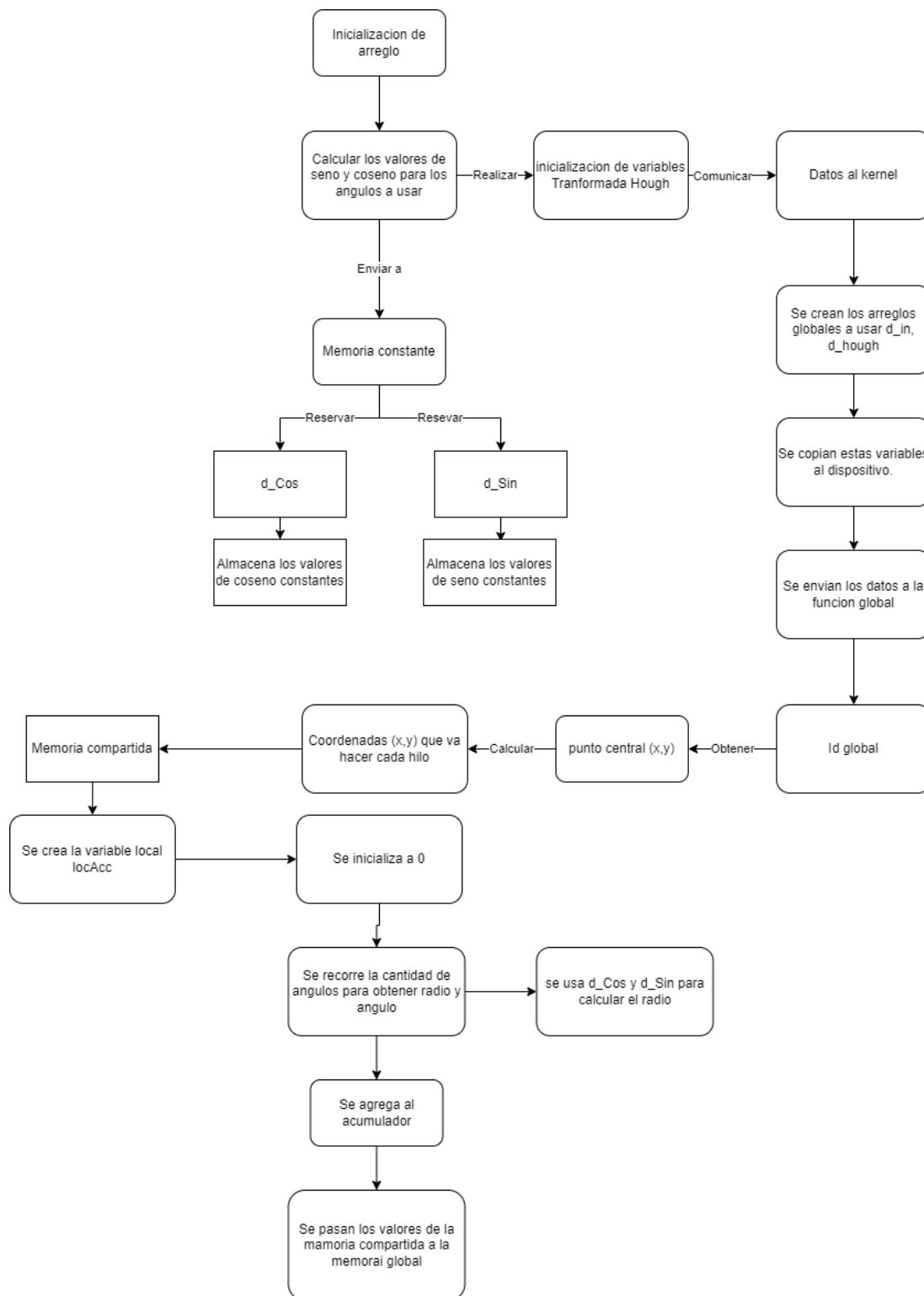
ANEXO 3 - Uso de memoria compartida

Explicación

El programa de memoria compartida lo que hizo fue crear un arreglo de memoria compartida para que cada bloque tuviera un acumulador local. Esto se hace en la línea 80 donde se crea el localAcc y luego de esto se inicializa a 0 cada posición del mismo. Se le coloca una barrera para evitar que hayan race conditions y es este el que se utiliza al momento de encontrar los valores.

En final una vez obtenidos los valores lo que sé hacer es colocar una segunda barrera y trasladar estos valores al arreglo global, acc con una función atómica.

Diagrama



Mejora en tiempo

Tomando en cuenta la tabla 1, 2 y 3 podemos observar que existe una mejora en el tiempo debido a que el promedio de tiempo constante fue de 0.7 ms y el del base fue de 0.9 ms. El tiempo promedio al agregarle memoria compartida y tener constante y global es de 0.4 ms mostrando que este tiene un mejor grado de eficiencia en tiempo de ejecución.

LITERATURA CITADA

NVIDIA Developer. (s.f). CUDA Zone. Extraído de:
<https://developer.nvidia.com/cuda-zone>

Fernández, J. (2019). CUDA, historia del conflicto Apple y NVIDIA. Extraído de:
<https://www.applesfera.com/desarrollo-de-software/cuda-historia-conflicto-apple-nvidia>

NVIDIA. (s.f.). What is CUDA? Extraído de: <https://developer.nvidia.com/what-cuda>

NVIDIA. (s.f.). CUDA Toolkit Documentation. Extraído de:
<https://docs.nvidia.com/cuda/index.html>

NVIDIA. (s.f.). GPU-Accelerated Computing with CUDA. Extraído de:
<https://www.nvidia.com/en-us/data-center/gpu-accelerated-computing/>

Tensor Flow. (s.f.). Tensor Flow. Extraído de: <https://www.tensorflow.org/>

Lanchas, L. (2017) Implementación en CUDA de un algoritmo de flujo óptico denso.
Extraído de:
https://accedacris.ulpgc.es/bitstream/10553/23940/1/0738917_00000_0000.pdf