1. Antecedentes

CUDA es el API más usado para el modelo paralelo de computación acelerada (híbrida). A pesar de estar restringido a hardware Nvidia, su amplia diversidad de funciones le ha hecho el estándar en modelos acelerados de escritorio y datacenter. Una de las características de la computación acelerada mediante GPUs es la versatilidad que ofrece al poseer diferentes niveles y tipos de memoria, además de ofrecer una gran cantidad de recursos para realizar cómputo SPMD.

2. Objetivos y Competencias

- Conocer una aplicación de la memoria Constante de la GPU.
- Aprovechar las características de las memorias Global, Compartida y Constante en un problema común en análisis de gráficos.

3. Transformada de Hough

Para este proyecto usaremos como ejemplo una implementación en el modelo CUDA del algoritmo para la Transformada de Hough. Esta transformación es una técnica computacional para la detección de líneas rectas en una imagen. La forma generalizada de esta transformación también puede usarse para detectar bordes en figuras más complejas, tanto las que pueden tener representación analítica como las que no tienen representación analítica.

En este proyecto haremos uso de la versión lineal de la Transformada de Hough, que se aplica a imagenes blanco y negro (la cuales usualmente son resultado de otro proceso de deteccion de bordes, como Canny, Sobel, etc.). Esta transformación es parte de los procesos dentro de los filtros para la detección de bordes de una imagen. De forma resumida, esta transformación es un sistema de votación que ayuda a detectar si un pixel es parte o no de una línea, asignándole un peso. El pixel califica su pertenencia dentro de un número discreto de líneas. Las líneas que reciben más votos al analizar todos los pixeles, son las que son elegidas para detección.

El grupo de coordenadas en un plano Cartesiano de un punto perteneciente a una línea recta están restringidas por la ecuación pendiente-intercepto de la línea:

$$y = pendiente \cdot x + punto\ de\ intercepción \qquad (1.1)$$

Si r es la distancia del origen hacia la línea y θ el ángulo perpendicular a la línea, entonces tal ecuación de línea, ilustrado en la ***Figura 1***, puede expresarse mediante la siguiente fórmula:

$$y = -\frac{cos(\theta)}{sin(\theta)}x + \frac{r}{sin(\theta)} \quad (1.2)$$

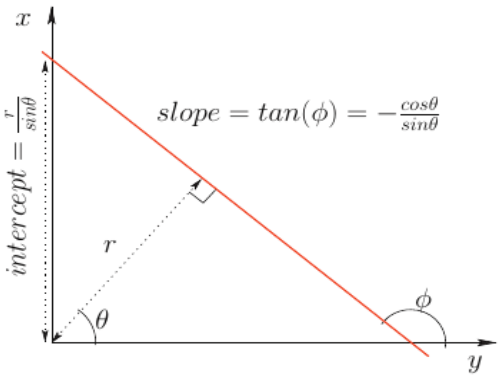

Figura 1. *Descripción de la linea en el plano bidimensional*

Podemos expresar la ecuación (1.2) de la siguiente forma:

$$r(\theta) = x \cdot cos(\theta) + y \cdot sin(\theta) \quad (1.3)$$

Al aplicar la Transformada de Hough, cada pixel "iluminado" (x, y) de la imagen puede pertenecer a una familia completa de líneas, como lo ilustra la ***Figura 2***. Iterando sobre los posibles ángulos θ, podemos calcular el correspondiente $r(\theta)$. mediante la ecuación (1.3). Para poder hacer computable este cálculo, usamos un número discreto y limitado de ángulos con un incremento constante (i.e.: de 0 a 180 grados en incrementos de 1 grado).

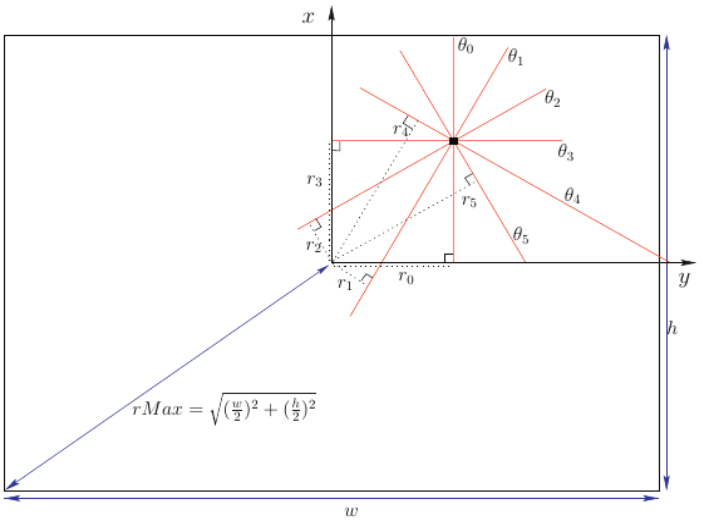

Figura 2. *Ejemplo de seis pares de parámetros* (θ_i, r_i) *asociados a un pixel (•). La distancia máxima en una figura WxH está dada por rMax, asumiendo como orígen del eje el centro de la imágen.*

El componente final de la Transformada es un acumulador de "votaciones" o pesos. Este se expresa en forma de una matriz de contadores con índices θ y $r(\theta)$, e incrementados cada vez que un pixel encaja en la ecuación de la línea correspondiente. Una discretización similar a θ es aplicada al cálculo de las distancias $r(\theta)$, para poder acceder a los componentes acumuladores de la matriz de pesos. El número de contenedores o "bins" usados para $r(\theta)$ generalmente está definido por la precisión deseada. Pero en el caso de CUDA en realidad está restringido por el tamaño de la memoria Compartida, ya que para beneficiarnos de la velocidad de acceso a esta memoria, mantenemos un acumulador local por cada bloque de hilos.

En una versión simple de la implementación de la Transformada Lineal de Hough, cabe notar que el origen del plano se asume como el centro de la imagen. Esto define el rango de las distancias a cualquier línea como [*−rMax, rMax*] (***Figura 2***).

Estos son los puntos claves de la versión secuencial (CPU) de la implementación de la transformación lineal en el ***Listado 1***:

- El número de bins para θ es 90 y para $r(\theta)$ es 100. El incremento de θ es de 2 grados, y luego convertimos los ángulos a radianes.
- Definimos la estructura para la matriz acumuladora en la línea 11 como un arreglo de $[rBins * 180/ degreeInc]$ elementos.
- Inicializamos a 0 todos los elementos del acumulador en la línea 12 mediante la instrucción `memset()`.
- Los loops externos en las líneas 17 y 18 iteran sobre los pixeles de la imagen.
- Si el pixel está encendido, cumplirá con la condición en la línea 21 y actualizará la matriz acumuladora. Usamos los índices bidimensionales (i, j) para calcular el índice lineal idx ("cortamos" la imagen en filas y las colocamos de forma consecutiva).
- Convertimos en las líneas 23 y 24 los índices (i, j) a las coordenadas $(xCoord, yCoord)$ asumiendo el centro de la imagen como el orígen del plano.
- El loop en las líneas 26 a 33 nos servirá para discretizar y calcular los valores (θ, r) de las líneas posibles. Esto producirá los índices $(tIdx, rIdx)$ en el plano transformado que nos servirá para actualizar la matriz acumuladora en la línea 30.
- Discretizamos r en la línea 29 y agregamos un offset con rMax y una escala rScale. Esto hace que $rIdx$ empiece en 0 al mapear el rango $[- rMax, rMax]$ a $[0, rBins)$.

```
1  // File: hough/hough.cu
2  . . .
3  const int degreeInc = 2;
4  const int degreeBins = 90;
5  const int rBins = 100;
6  const float radInc = degreeInc * M_PI / 180;
7  //*****************************************************************
8  void CPU_HoughTran (unsigned char *pic, int w, int h, int **acc)
9  {
10   float rMax = sqrt (1.0 * w * w + 1.0 * h * h) / 2;
11   *acc = new int[rBins * 180 / degreeInc];
12   memset (*acc, 0, sizeof (int) * rBins * 180 / degreeInc);
13   int xCent = w / 2;
14   int yCent = h / 2;
15   float rScale = 2 * rMax / rBins;
16
17   for (int i = 0; i < w; i++)
18     for (int j = 0; j < h; j++)
19       {
20         int idx = j * w + i;
21         if (pic[idx] > 0)
22           {
23             int xCoord = i - xCent;
24             int yCoord = yCent - j;  // y-coord has to be reversed
25             float theta = 0;         // actual angle
26             for (int tIdx = 0; tIdx < degreeBins; tIdx++)
27               {
28                 float r = xCoord * cos (theta) + yCoord * sin (theta);
29                 int rIdx = (r + rMax) / rScale;
30                 (*acc)[rIdx * degreeBins + tIdx]++;
31                 theta += radInc;
32               }
33           }
34       }
35 }
```

Listado 1. *Implementación secuencial simple de la versión lineal de la Transformada de Hough*

4. Memoria Constante

En CPUs, esta es una memoria off-chip dedicada a almacenar datos de acceso frecuente y que posee dos características importantes: 1) puede ser transferida a la memoria Cache de cada Symmetric Multiprocessor y 2) puede hacer broadcast de un valor a todos los hilos de un Warp. El límite actual de la memoria Constante es de 64KB, mientras que los GPUs de Nvidia tienen 8KB de Cache por SM.

La memoria Constante también sirve para almacenar los valores de configuración pasados al Kernel, con un límite de 4KB disponibles de la memoria Constante para estos valores. La memoria Constante puede asignarse en el device desde el host mediante el especificador `__constant__`:

```
__constant__ float d_Cos[degreeBins];
```

Esto aparta en el device un arreglo float de *degreeBins* número de elementos. En dispositivos más recientes y a partir de la arquitectura Fermi, las tarjetas tienen una memoria caché L2 que tiene menos latencia y satisface muchas de las necesidades de transacciones de memoria. La ventaja de la memoria constante es el tamaño de 64KB, superior al de la caché L2.

La memoria Constante es “fija” desde el punto de vista del dispositivo. El host puede manejar esta memoria antes de la llamada al Kernel. Por lo tanto, el host es capaz de trasladar los contenidos de memoria de host a la memoria Constante mediante la instrucción `cudaMemcpyToSymbol`:

```
__constant__ float d_Cos[degreeBins];
…
float *pcCos = (float *) malloc (sizeof (float) * degreeBins);
…
cudaMemcpyToSymbol(d_Cos, pcCos, sizeof (float) * degreeBins);
```

Recordemos que los procesadores son eficientes para operaciones aritméticas simples, pero las operaciones trigonométricas requieren dispositivos complejos y varios ciclos de proceso para ejecutarse. Por tal razón, cualquier operación trigonométrica es costosa a nivel computacional. La versión lineal de la Transformada de Hough realiza cálculos de senos y cosenos sobre un número definido de ángulos θ y siempre usará estos valores no importando qué pixel estemos analizando. Una vez calculados, estos valores son solamente de lectura. Por lo tanto, la memoria Constante es un buen candidato para almacenar los valores de seno y coseno de todos los ángulos que se usarán en esta transformación.

5. Actividades

Trabajando con el código base hough.cu, realice las siguientes tareas:

1. Compile y corra el programa con la versión CUDA que usa memoria **Global** únicamente. Revise el Makefile y ajustelo 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.
    b. Hace falta la liberación de memoria al final del programa. Agreguela para las variables utilizadas.
2. Incorpore medición de tiempo de la llamada al kernel mediante el uso de CUDA events. Para esto consulte la documentación y ejemplos en el siguiente sitio de Nvidia:
    a. https://developer.nvidia.com/blog/how-implement-performance-metrics-cuda-cc/
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.
4. 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.
    b. Recuerde que ahora las referencias a d_Cos y d_Sin son globales. Ya no es necesario pasarlas como argumentos al kernel.
    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(rad) y sin(rad) del host a la memoria Constante del device usando `cudaMemcpyToSymbol`.
5. 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.

6. 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).
7. 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.
    b. Defina en el kernel un acumulador local en memoria compartida llamado `localAcc`, que tenga `degreeBins * rBins` elementos.
    c. Inicialize a 0 todos los elementos de este acumulador local. Recuerde que la memoria Compartida solamente puede manejarse desde el device (kernel).
    d. Incluya una barrera para los hilos del bloque que controle que todos los hilos hayan completado el proceso de inicialización del acumulador local.
    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.
    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.
    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`.
8. 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.
9. 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).

6. Evaluación

	Informe − 25 puntos	**valor**
1	Mínimo 1, máximo 4 páginas de contenido sobre el algoritmo, la implementación en CUDA, las aplicaciones de las diferentes memorias de GPU en la versión lineal de la Transformación de Hough.	3
2	Formato según guía de informes UVG: carátula, índice, introducción, cuerpo, citas textuales / píe de página, conclusiones / recomendaciones, apéndice con material suplementario y al menos 3 citas bibliográficas relevantes y confiables	3
3	Bitácora con mediciones de tiempo, un mínimo de 10, para: ● Versión del kernel con memoria Global únicamente ● Versión del kernel con memoria Constante y Global ● Versión del kernel con los tres tipos de memoria	6
4	Uso de memoria Constante: ● Un párrafo explicando la forma como se usó la memoria Constante para la versión CUDA del algoritmo. ● Un diagrama funcional o conceptual ilustrando el efecto en el algoritmo al usar memoria Constante	5
5	Uso de memoria Compartida: ● Un párrafo explicando la forma como se usó la memoria Compartida para la versión CUDA del algoritmo ● Un diagrama funcional o conceptual ilustrando el efecto en el algoritmo al usar memoria Compartida.	5
6	Conclusión / recomendación − resumir los retos encontrados y las soluciones para la implementación acelerada de la Transformación Lineal de Hough en CUDA	3

	Programa y presentación − 75 puntos	**valor**
1	**Entrega preliminar 1** (Viernes 27 de mayo) ***(Actividades 1, 2, 3)***: ● Versión CUDA del programa con uso de memoria Global ● Medición de tiempos de esta versión mediante Cuda Events (10 mediciones) ● Fórmula correcta para el cálculo del gloID	10
2	**Entrega preliminar 2** (Miércoles 1 de junio) ***(Actividades 4, 5, 6)***: ● Versión CUDA del programa con uso de memoria Global y memoria Constante ● Medición de tiempos de esta versión mediante Cuda Events ● Primer versión del informe con: ○ Bitácoras de pruebas para kernel con memoria Global y kernel con memorias Global y Constante. ○ Información sobre el uso de memoria Constante en la versión de la Transformación de Hough.	20
2	**Entrega final** (Viernes 3 de junio) ***(Actividades 7, 8, 9)***: ● Versiòn CUDA con memoria Global, Constante y Compartida ● Mediciòn de tiempos ● Documento final	30
3	Documentación y comentarios explicativos sobre las partes importantes del programa. Diseñe su propio estilo de documentación y úselo consistentemente en todo el programa	5
4	Uso de barreras para el control de hilos de bloque	5
5	Liberación y destrucción de memoria asignada en CPU y GPU. Reinicio de recursos en el dispositivo	5

7. Entregar en Canvas

Entrega Parcial 1: (Viernes 27 de Mayo)

1. Versión CUDA del programa con uso de memoria Global funcional (.cu, .cpp, .h, etc)
2. Medición de tiempos de esta versión mediante Cuda Events (10 mediciones mínimo)

Entrega Parcial 2: (Miércoles 1 de Junio)

3. Versión CUDA del programa con uso de memoria Global y memoria Constante funcional (.cu, .cpp, .h, etc)
4. Medición de tiempos de esta versión mediante Cuda Events
5. Primera versión del informe en formato PDF

Entrega Final y presentación: (Viernes 13 de Mayo).

1. Versión CUDA con memoria Global, Constante y Compartida funcional (.cu, .cpp, .h, etc).
2. Mediciòn de tiempos
3. Documento final del informe en formato PDF

Se les recuerda que **no** se acepta solamente link a repositorio. Deben subir su código.