Proyecto #3: CUDA

**Contenido**

[Introducción 4](#_Toc136551258)

[Resultados: 4](#_Toc136551259)

[Metodología de medición de tiempos: 4](#_Toc136551260)

[Bitácora de resultados: 4](#_Toc136551261)

[Anexos: 5](#_Toc136551262)

[Anexo 1: Seguimiento de guía: 5](#_Toc136551263)

[Anexo 2) Link a repositorio de las implementaciones 10](#_Toc136551264)

[Referencias 11](#_Toc136551265)

# Introducción

Se implementaron varias versiones un kernel de CUDA que calcula la transformación de Hough para un buffer que representa una imagen bidimensional en blanco y negro. Estas versiones fueron, una versión inocente como control y dos versiones que implementan optimizaciones, una con el uso de memoria constante y otra con el uso de memoria constante, así como memoria compartida. Sin embargo, se determinó que ninguna de estas optimizaciones es aplicable al problema y que no proveen un speedup significativo y en algunos casos pueden tener el efecto contrario.

# Resultados:

## Metodología de medición de tiempos:

Para tomar los tiempos del desempeño de cada versión se utilizaron eventos de CUDA (Harris, 2023) y un Makefile dedicado a correr el programa 10 veces seguidas, con el cuidado de esperar un tiempo prudente (dos segundos) entre cada corrida del programa; se encontró que esto era necesario puesto que, de no ser así, los tiempos de ejecución tendieron a ser espontáneamente más rápidos, probablemente porque el driver de CUDA logra algún speedup cuando se requiere la misma memoria repetidamente. Esto no ocurrió cuando se corría el programa repetidamente a mano y se razonó que la lentitud del usuario podría ser la causa de la consistencia. Finalmente, para reducir la desviación estándar entre cada corrida, se corrió el kernel 400 veces para que el tiempo de ejecución fuera mayor a un segundo. Esto hace que el resultado final sea incorrecto, pero no afecta a la cantidad de instrucciones que se ejecuta cada vez que se corre el kernel, por lo que sigue siendo adecuado para la medición de tiempos.

Se tomaron mediciones par 4 versiones del programa: Normal, Memoria constante, Memoria compartida con 256 hilos por bloque, Memoria compartida 1024 hilos por bloque.

## Bitácora de resultados:

# Anexos:

## Anexo 1: Seguimiento de guía:

1.a Cálculo de la fórmula para crear el gloID en el kernel

A picture containing text, screenshot, font

Description automatically generated

Se usó una fórmula genérica para distinguir a cada hilo individualmente basada en desplazar el identificador del hilo dentro del bloque por la cantidad de hilos presentes en los bloques que preceden al bloque actual en el orden. Es mejor usar esta fórmula porque es lo mínimo que se necesita para distinguir a cada hilo individualmente. Existen fórmulas más complicadas que toman en cuenta la dimensionalidad de la cuadrícula, pero no es necesario usarlas dado que el kernel opera linealmente.

1.b Hace falta la liberación de memoria al final del programa.

A screenshot of a computer code

Description automatically generated with low confidence

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

Siguiendo la documentación oficial:

A screenshot of a computer screen

Description automatically generated with low confidence

A screen shot of a computer code

Description automatically generated with low confidence

A picture containing text, screenshot, font

Description automatically generated

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.

La librería pgm transformó a la imagen de un vector de dos dimensiones en un vector de solo una dimensión al colocar todos los pixeles de corrido al partir la imagen en filas y colocando una fila seguida de la otra. Entendiendo esto, podemos revertir esa misma transformación al dividir el identificador del hilo por el ancho de la imagen y usando el residuo como la coordenada en horizontal y al cociente como la vertical. Finalmente, se traslada la transformación de un rango entre 0 a 1, a un rango entre -0.5 y 0.5 al restar la mitad de del ancho y del alto de las coordenadas horizontal y vertical, respectivamente; con el objetivo de que el origen del plano (0, 0) sea el centro de la imagen.

4 Generación de imágenes.

Para esto se escribió un kernel adicional, que toma el máximo del resultado para cada pixel y llena un buffer que representa una imagen BGR en memoria. Luego se copia este buffer de VRAM a RAM y se lo escribe como una imagen en el formato BMP según la especificación para imágenes de 24 bits por pixel. (Liesh, 2023)

\_\_global\_\_ void makeImage(unsigned char\* pic, int w, int h,

    int\* acc, float rMax, float rScale,

    float\* d\_Cos, float\* d\_Sin, char3\* out, int thresholdMul = 6)

{

    int gloID = blockIdx.x \* blockDim.x + threadIdx.x;

    if (gloID > w \* h) return;      // in case of extra threads in block

    int xCent = w / 2;

    int yCent = h / 2;

    int xCoord = gloID % w - xCent;

    int yCoord = yCent - gloID / w;

    out[gloID] = make\_char3(pic[gloID],

                            pic[gloID],

                            pic[gloID]); //pgm y bmp guardan los pixeles de manera inversa

    \_\_shared\_\_ int max[THREADS\_PER\_BLOCK];

    max[threadIdx.x] = 0;

    for (int tIdx = 0; tIdx < degreeBins; tIdx++)

    {

        float r = xCoord \* d\_Cos[tIdx] + yCoord \* d\_Sin[tIdx];

        int rIdx = (r + rMax) / rScale;

        const int val = \*(acc + (rIdx \* degreeBins + tIdx));

        max[threadIdx.x] = val > max[threadIdx.x] ? val : max[threadIdx.x];

    }

    if (max[threadIdx.x] > w\*thresholdMul)

        out[gloID] = make\_char3(   out[gloID].x   +      max[threadIdx.x] \* 37 % 100,//blue

                                   out[gloID].y    +    max[threadIdx.x] \* 91 % 100,//green

                                   out[gloID].z  +    max[threadIdx.x] \* 51 % 100); //red

}

El color de cada pixel resultante es calculado en base en el puntaje máximo que alguna de las líneas obtuvo para ese pixel dado que este será único para toda línea distinguiblemente única; y es mezclado usando *additive blending* para mostrar las partes de la imagen sobre las que se detectó una línea. (Learn OpenGl ES, 2023) Un ejemplo de una imagen resultante es la siguiente:

A picture containing colorfulness, screenshot, art, line

Description automatically generated

5 Uso de memoria constante

Para esto se declaró memoria constante y se la inicializó con cudaMemcpyToSymbol()

\_\_constant\_\_ float d\_constCos[degreeBins];

\_\_constant\_\_ float d\_constSin[degreeBins];

\_\_global\_\_ void GPU\_HoughTranConst(unsigned char\* pic, int w, int h,

    int\* acc, float rMax, float rScale)

{

    int gloID = blockIdx.x \* blockDim.x + threadIdx.x;

    if (gloID > w \* h) return;      // in case of extra threads in block

    int xCent = w / 2;

    int yCent = h / 2;

    int xCoord = gloID % w - xCent;

    int yCoord = yCent - gloID / w;

    if (pic[gloID] == 0)

        return;

    for (int tIdx = 0; tIdx < degreeBins; tIdx++)

    {

        //TODO utilizar memoria constante para senos y cosenos

        float r = xCoord \* d\_constCos[tIdx] + yCoord \* d\_constSin[tIdx];

        int rIdx = (r + rMax) / rScale;

        //debemos usar atomic, pero que race condition hay si somos un thread por pixel? explique

        //R: porque el acumulador no es un vector en el espacio de los pixels, sino en el espacio de pesos para líneas

        //Las threads no tienen una relación 1 a 1 con la memoria en este espacio.

        atomicAdd(acc + (rIdx \* degreeBins + tIdx), 1);

    }

}

No fue necesario pasar los punteros al kernel porque están declarados en el namespace global.

6 El funcionamiento y toma de tiempos se evidencian en la sección de resultados.

7 No hubo una mejora, esto se discute en la sección de discusión.

8 Modificación para el uso de memoria compartida:

El identificador local es simplemente threadIDx.x puesto que solo es necesario distinguir al hilo de los demás hilos del bloque al que pertenece. Sin embargo, esto no es suficiente para repartir equitativamente el trabajo entre todos los hilos puesto que la cantidad de direcciones de memoria supera con creces la cantidad máxima de hilos por bloque. Para resolver esto, se intentó repartir el trabajo de la manera más equitativa posible entre cada hilo y procurando que la memoria a la que cada hilo accesa sea cercana a la memoria a la que accesan los demás hilos del bloque. Esto aumenta la probabilidad de encontrar la memoria en caché y significó un speedup importante a la hora de acumular los resultados finales en VRAM.

En base a las metas planteadas en el párrafo anterior, la implementación de la inicialización fue la siguiente:

A picture containing screenshot, font, text, line

Description automatically generated

Podemos ver que esto logra que cada hilo opere en direcciones de memoria contiguas a los demás hilos del bloque:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Hilos | hilo 1 | hilo 2 | hilo 3 | hilo 4 | … | hilo n |
| Memoria | dirección 1 | dirección 2 | dirección 3 | dirección 4 | … | dirección n |

Sin embargo, hacer esto una vez como máximo puede inicializar 1024 direcciones de memoria, es necesario inicializar 9000, razón por la que se introdujo un loop para repetir el proceso desfasando a la memoria n espacios y una condicional que se encarga de procesar el residuo. Esto tiene un resultado como en la tabla de la página siguiente:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Hilos | 1 | 2 | … | n | 1 | 2 | … | n | … | 9000%n |
| Memoria | 1 | 2 | … | n | n+1 | n+2 | … | 2n | … | 9000 |

Luego se hizo la computación como se ha hecho siempre, solo que se acumula en memoria compartida en vez de memoria global.

A computer code on a black background

Description automatically generated with low confidence

Es necesario seguir haciendo uso de atomicAdd porque, aunque se trate de memoria compartida, sigue siendo el caso que a cada hilo le corresponde un pixel, pero por cada pixel se escriben varios valores a la tabla de puntaje resultante. Se sincroniza antes y después de hacer este proceso para respectivamente asegurar que la memoria esté inicializada y que cada hilo haya terminado su trabajo antes de comenzar a acumularlo.

Finalmente se acumuló el resultado local de cada bloque en memoria global:

A picture containing text, screenshot, font

Description automatically generated

Para la acumulación final de los resultados se usa la misma repartición de trabajo que para la inicialización. También es necesario usar atomicAdd, pero en este caso es porque cada hilo con el mismo identificador local escribirá a la misma dirección en memoria global.

## Anexo 2) Link a repositorio de las implementaciones

[JmDeleon2000/Proyecto-3-paralela-CUDA (github.com)](https://github.com/JmDeleon2000/Proyecto-3-paralela-CUDA)

# Referencias

Harris, M. (25 de 5 de 2023). *How to Implement Performance Metrics in CUDA C/C++ | NVIDIA Technical Blog*. Obtenido de NVIDIA Developer: https://developer.nvidia.com/blog/how-implement-performance-metrics-cuda-cc/

Learn OpenGl ES. (1 de 6 de 2023). *additive blending | Learn OpenGL ES*. Obtenido de Learn OpenGl ES: https://www.learnopengles.com/tag/additive-blending/

Liesh, N. (1 de 6 de 2023). *THE BMP FILE FORMAT*. Obtenido de Electrical and Computer Engineering University of Alberta: http://www.ece.ualberta.ca/~elliott/ee552/studentAppNotes/2003\_w/misc/bmp\_file\_format/bmp\_file\_format.htm