Contents

1 Archivos de Salida **1**

2 Propiedades de las Placas Gráficas **2**

3 Suma de Vectores **3**

4 Suma de Matrices **4**

5 Convolución 1D **6**
5.1 Implementación 11
5.2 Resultados 12

6 Filtro de Imágen **14**
6.1 Resultados 14

7 Filtro Sobel **20**
7.1 Resultados 21

8 Apéndice **23**
8.1 Resultados de la Suma de Vectores 23
8.1.1 Grupo G 23
8.1.2 Grupo T 23
8.2 Resultados de la Suma de Matrices 24
8.3 Resultados de la Convolución 1D 24
8.3.1 Grupo G Float 24
8.3.2 Grupo G Double 24

Cada resolución de los ejercicios propuestos para este trabajo se localiza un una carpeta. Cada carpeta contiene:

- El o los códigos necesarios para cada prueba.
- Un script de compilación "compile.sh"
- Un script denominado "run.sh" para ejecutar todos los programas.
- Los resultados obtenidos en archivos .out.

Considerando que el cluster CDER posee dos placas gráficas diferentes, se propone como convención denominar a:

Grupo G a las máquinas del cluster con placas GeForce GTX Titan y

Grupo T a las máquinas con placas Tesla V100.

1 Archivos de Salida

Cada ejecución producirá dos archivos de salida: El archivo de la salida estándar del programa y el correspondiente a la salida de error usado por el profiler.

Los nombres de los archivos de la salida estándar del programa poseen el siguiente formato: `~Nombre_programa-Grup`

Los nombres de los archivos del profiler tienen el siguiente patrón: `Nombre_programa-Grupo-Tipo_Profiling-Numer`
Los tipos de profiling utilizados para cada prueba son tres: "none" indicando que no se utilizó `nvprof`, "events" que corresponde con `nvprof --events all` y metrics con `nvprof --metrics all`.

Por ejemplo, el archivo `suma_matrices_A-G-events-1.err` refiere a la ejecución del programa suma_matrices_A, sobre las computadoras del grupo G (máquinas con GeForce GTX Titan) y contendrá datos de profiling de eventos.

Puede consultar el script run.sh para determinar qué comandos se ejecutó para cada archivo de salida.

2 Propiedades de las Placas Gráficas

En primera instancia, se analizarán las placas gráficas disponibles dentro del cluster CDER a utilizar. Para ello, se realizará una aplicación que solicite los datos necesarios al sistema por medio de los pasos que se explicarán a continuación.

Primero, es necesario obtener la cantidad de dispositivos que puedan realizar cómputos. Esto se realiza con la función `cudaGetDeviceCount(&count)`. Luego, por cada dispositivo se solicitarán sus detalles por medio de la función `cudaGetDeviceProperties(&prop, dev)`, dónde `dev` es el número de dispositivo.

Los datos obtenidos serán contenidos dentro dentro de la variable `prop`, la cual es de tipo `cudaDeviceProp`. Dicha estructura esta compuesta por varios campos que representan cada una de la información que corresponde al detalle del dispositivo: nombre, versión, cantidad de memoria global, etc. Una descripción completa se puede encontrar en su documentación Web de CUDA [?].

A continuación se describen algunos campos de relevancia que fueron utilizados en el programa. Supóngase que `prop` es una variable de tipo `cudaDeviceProp`.

prop.name Nombre que identifica al dispositivo.

prop.totalGlobalMem Memoria global disponible (en bytes).

prop.totalConstMem Memoria constante disponible (en bytes).

prop.maxThreadPerBlock Máximo número the threads por bloque.

prop.maxThreadsDim Un arreglo de 3 elementos con el máximo tamaño de cada dimensión del bloque.

prop.maxGridSize Un arreglo de 3 elementos con el tamaño máximo de cada dimensión de la grilla.

prop.multiProcessorCount Número de multiprocesadores en el dispositivo.

Al ejecutar el script `run.sh` dentro del cluster se producirán una salida por cada grupo. Para el grupo T se genera el siguiente resultado.

```
--- General Information for device 0 ---
Compute name: Tesla V100-PCIE-16GB
Compute capability: 7.0
Clock rate: 1380000
 --- Memory Information for device 0 ---
Total global mem: 16914055168
Total constant Mem: 65536
 --- MP Information for device 0 ---
Multiprocessor count: 80
Shared mem per mp: 49152
Registers per mp: 65536
Threads in warp: 32
Max threads per block: 1024
Max thread dimensions: (1024, 1024, 64)
Max grid dimensions: (2147483647, 65535, 65535)
```

Y para el grupo G, el siguiente.

```
--- General Information for device 0 ---
Compute name: GeForce GTX TITAN X
Compute capability: 5.2
Clock rate: 1076000
 --- Memory Information for device 0 ---
Total global mem: 12806062080
Total constant Mem: 65536
 --- MP Information for device 0 ---
Multiprocessor count: 24
Shared mem per mp: 49152
Registers per mp: 65536
Threads in warp: 32
Max threads per block: 1024
Max thread dimensions: (1024, 1024, 64)
Max grid dimensions: (2147483647, 65535, 65535)
```

Utilizando estos datos obtenemos la Tabla comparativa 1.

Table 1: Detalles de la placas gráficas obtenidos mediante el programa 1.

Característica	GeForce GTX Titan X	Tesla V100
Multiprocesadores	24	80
Mem. global	11.92 GiB	15.75 GiB
Mem. constante	64 MiB	64 MiB
Mem. compartida (shared)	48 MiB	48 MiB
Max. threads por bloque	1024	1024
Max. dim. de thread	(1024, 1024, 64)	(1024, 1024, 64)
Max. dim. de grilla	(2147483647, 65535, 65535)	(2147483647, 65535, 65535)

Por consiguiente, se espera que el grupo T ofrezca un mejor rendimiento de un programa si se aprovecha completamente el dispositivo con respecto al grupo G. Con respecto a la cantidad de hilo para paralelizar, ambos ofrecen el mismo tamaño de grilla y de hilos por bloque.

3 Suma de Vectores

En la suma de vectores se propone sumar de forma secuencial una serie de números en CPU y posteriormente realizarlas en paralelo.

El código se realizó para tomar tiempos de GPU y CPU aún si el programa `nvprof` no funcionase. Además, se modificó para que los ciertos parámetros puedan ser ingresados como argumentos del programa, quedando la sinopsis de la siguiente forma:

```
./suma_vectores tam veces blockx blocky gridx gridy
```

De esta manera, se puede modificar los siguientes atributos por cada llamada:

tam Tamaño del vector.

veces Veces que se repite el experimento.

blockx, blocxy Formato del bloque (cantidad de threads en X e Y).

gridx, gridy Formato de la grilla (cantidad de bloques en X e Y).

Se experimentó con las siguientes variaciones:

Núm.	tam	veces	blockx	blocky	gridx	gridy
1	100	1	1	1	256	1
2	1000	1	4	1	256	1
3	10000	1	40	1	256	1
4	100000	1	391	1	256	1
5	1000000	1	3907	1	256	1
6	10000000	1	39063	1	256	1
7	100	100	1	1	256	1
8	100	200	1	1	256	1
9	100	300	1	1	256	1
10	10000000	100	39063	1	256	1
11	10000000	200	39063	1	256	1
12	10000000	300	39063	1	256	1

Además, se tuvo cuidado de reservar la cantidad de recursos necesaria para poder llevar a cabo la tarea. Por ejemplo: en el experimento número 2, se requiere 39063 bloques de 256 threads para poder realizar el cálculo de 10000000 elementos del vector, a pesar de que hayan quedado 128 threads libres de un bloque.

Para estudiar los resultados, se procederá, primeramente, a comparar los tiempos de CPU con respecto a los de GPU sin considerar la transmisión de datos. Luego, se considerarán todos los tiempos.

En el gráfico de la Figura 1 se muestra los de GPU, CPU y GPU sumado la copia a memoria para cada experimenteo.

Obsérvese la el experimento del 1 al 6, el cual se nota una creciente en los tiempos de CPU debido al aumento de la cantidad del tamaño del vector. Sin embargo, el tiempo de GPU tiende a aumentar en un ritmo menor aparentando incluso a ser casi constante al principio. Si se observa el tiempo de "GPU + memcpy" se aprecia un aumento considerable en los tiempos, incluso superando el tiempo de CPU.

Para el caso de los experimentos numerados ente el 7 y el 9, se testea baja cantidad de datos pero la cantidad de ejecución del algoritmo en aumento. Tanto en GPU como en CPU el crecimiento es estable aunque los tiempos de GPU son mayores. La copia de datos al dispositivo no afecta a los tiempos de forma considerable debido a la cantidad de datos.

Los experimentos de 9 a 11, los tiempos para la GPU son muy favorables. La cantidad de datos es considerable y las operaciones a realizar se repiten de forma creciente logrando que el CPU las realice de forma secuencial y la GPU paralela. Aquí se aprecia un beneficio en el uso de las tarjetas gráficas puesto que las operaciones de copiado de datos se realizan al principio y al final del experimento una única vez y se aprovecha el paralelismo en la gran cantidad de operaciones.

4 Suma de Matrices

En el código del kernel, se realiza un chequeo para evitar la ejecución de threads cuyos índices estén fuera del tamaño del vector. Esta situación puede producirse cuando el tamaño de la matriz no coincide con la cantidad de threads por bloques. Además, debido a la consideración de alineamiento de 32 threads por warp, es posible que hayan threads restantes al asignar un bloque para completar el tamaño de la matriz. Por ejemplo: si la matriz es de 100 filas y 100 columnas, se requieren 10000 threads repartidos en 10 bloques de 32 × 32 threads, restando del último bloque 240 threads asignados pero sin utilizar. Por consiguiente, el siguiente condicional fue incluido para cumplir con este propósito.

```
if (c < cols && f < filas){
  // ...
}
```

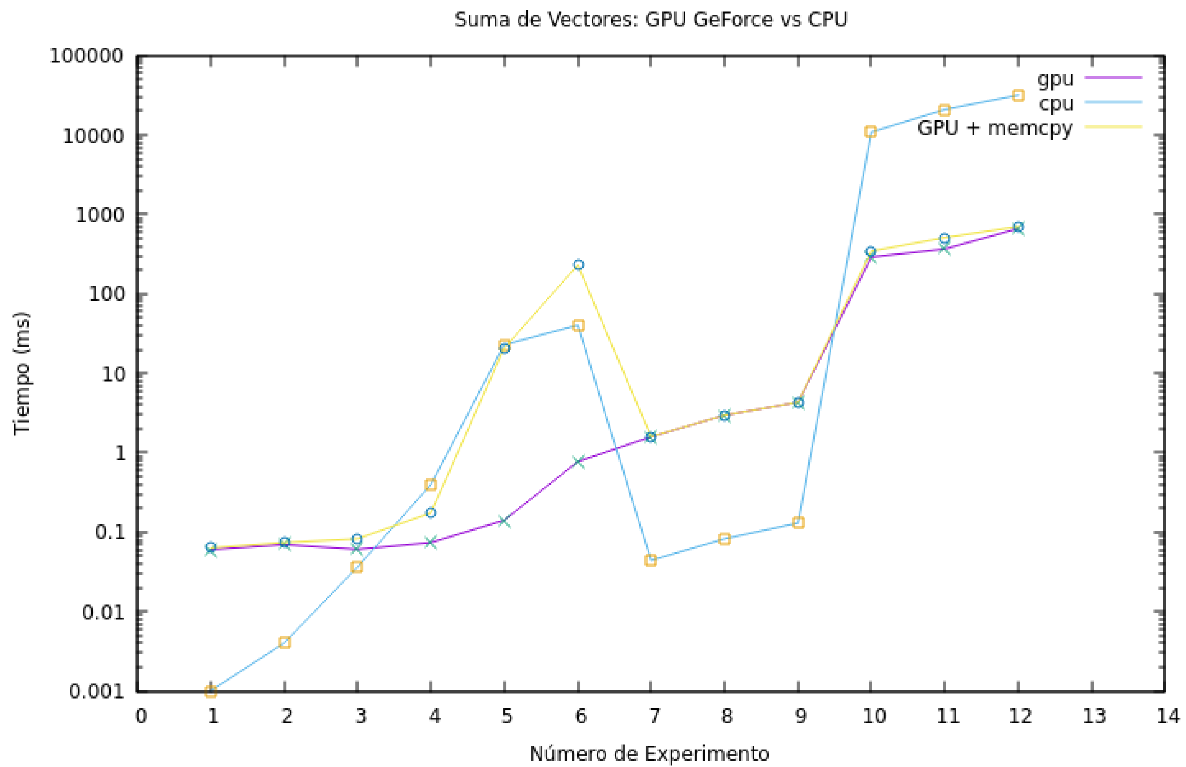


Figure 1: Resultados de los tiempos de la suma de vectores.

Table 2: Número y configuración utilizada para cada experimento. .

Num	Filas	Cols.	Grid. X	Grid. Y	Bloque X	Bloque Y
1	100	100	4	4	32	32
2	500	500	16	16	32	32
3	1000	1000	32	32	32	32
4	2500	2500	79	79	32	32
5	5000	5000	157	157	1024	1
6	5000	5000	157	157	32	32
7	5000	5000	500	30	1000	1
8	5000	5000	314	314	16	16
9	5000	5000	1000	1000	5	50
10	5000	5000	10000	10000	5	5

Se han ejecutado las pruebas con los tamaños y la configuración de la grilla expresadas en la Tabla 2. Los resultados se muestran expresados en la Figura 2. En este gráfico se puede apreciar que para una matriz de 5000 × 5000 es beneficioso utilizar la CPU. Esto es debido al traslado de la información hacia el dispositivo de la GPU.

Los experimentos del 1 al 4 se enfocan en aumentar gradualmente la cantidad de datos manteniendo la grilla de forma uniforme y alineada. Se aprecia un aumento considerable en el trabajo de la CPU comparado con el de la GPU. Esta última realiza la suma dentro del milisegundo, casi diez veces más rápido que la velocidad del CPU. Sin embargo, el precio por trasladar los datos hace que se pierda este beneficio.

En el caso del 5 y el 6, se denota un leve aumento al intentar distribuir los datos en 32 × 32.

Para el resto de las pruebas, se intentan con alineamientos fuera de los 32 threads que contienen un warp. Esto lleva a un tiempo de GPU que supera a los previos debido a que la placa requiere de más tiempo por ciclo para realizar la misma tarea que podría hacer en uno. En especial cuando se solicita una grilla de 10000 × 10000 elementos y sólo se utilizan 5 × 5 threads, se nota un incremento importante.

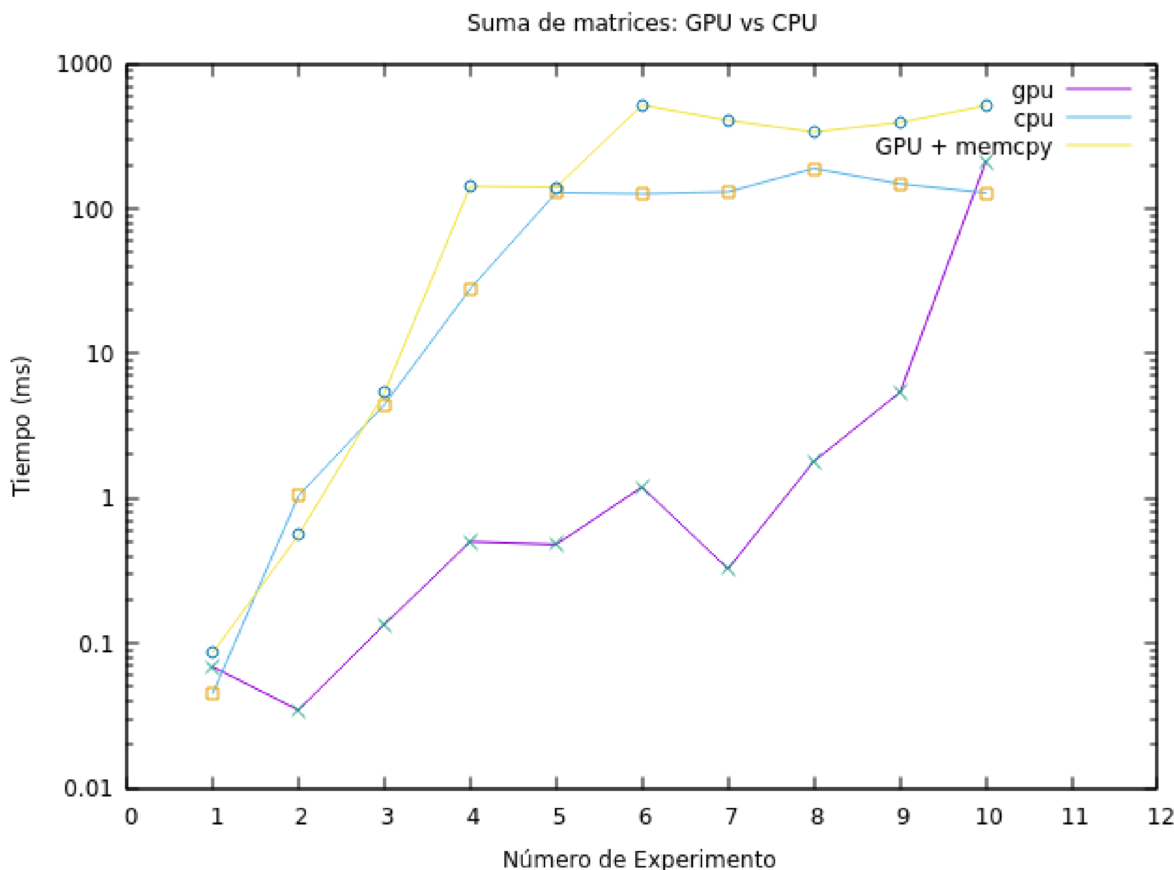


Figure 2: Resultados de los tiempos de ejecutar la suma de matrices para diferentes tamaños.

5 Convolución 1D

La suma de matrices requería una traslación de datos importante que hacía que la eficiencia del procesamiento paralelo no compensara los tiempos de copia de datos. En este caso, la convolución requiere de más trabajo de procesamiento y por consecuencia, un trabajo de paralelismo necesario que puede compensar los tiempos de transferencia.

Las pruebas a realizarse son las siguientes:

- Diferencias en tiempos a medida que el filtro crece.
- Diferencias en tiempos a medida que los datos crecen, pero el filtro se mantiene fijo.
- Diferencias en tiempos con pruebas similares, pero utilizando tipos de datos double.

En la Tabla 3 se pueden observar el número de experimento y sus respectivos parámetros. Las pruebas del 1 al 7 utilizan tamaños fijos del arreglo, pero varian el del filtro. Las del 8 al 13, fijan el tamaño del filtro para testear los tiempos a medida que crecen la cantidad de datos. Luego, se repiten las pruebas pero usando tipo de datos double.

Table 3: Experiencias a realizarse y los parametros utilizados.

Num.	Tipo de Dato	Tam. Arreglo	Tam. Filtro
1	float	1280000	32
2	float	1280000	64
3	float	1280000	100
4	float	1280000	128
5	float	1280000	160
6	float	1280000	200
7	float	1280000	256
8	float	320000	128
9	float	640000	128
10	float	1280000	128
11	float	1920000	128
12	float	2560000	128
13	float	3200000	128
14	double	1280000	32
15	double	1280000	64
16	double	1280000	100
17	double	1280000	128
18	double	1280000	160
19	double	1280000	200
20	double	1280000	256
21	double	320000	128
22	double	640000	128
23	double	1280000	128
24	double	1920000	128
25	double	2560000	128
26	double	3200000	128

En la Figura 3 se observa una comparación entre los tiempos de CPU y GPU para resolver la convolución. Como se puede observar, los tiempos de CPU son muy elevados con respecto a los de GPU sumada la transferencia de los datos. Los experimentos del 1 al 7 notan un crecimiento a medida que aumenta el tamaño del filtro, aunque no superan los 10ms. Un crecimiento más estable se aprecia a medida que se aumentan los datos a partir del experimento 8, incluso, el tiempo de transferencia se encuentra en aumento. Las experiencias

En la Figura **??** se aprecia el mismo comportamiento. Sin embargo, al comparar los tiempos (véase Apéndice "Resultados de la Convolución 1D") se detecta mayormente una tendencia a superar los valores

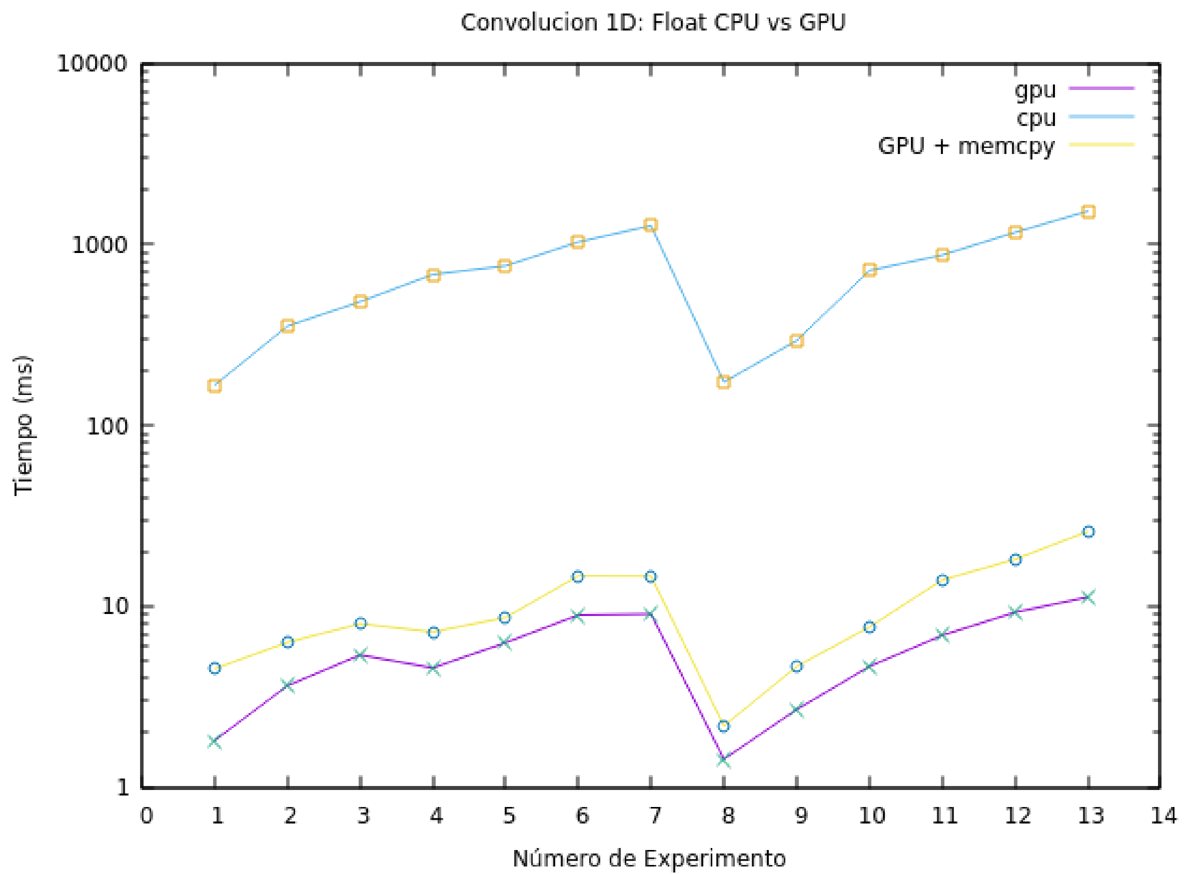
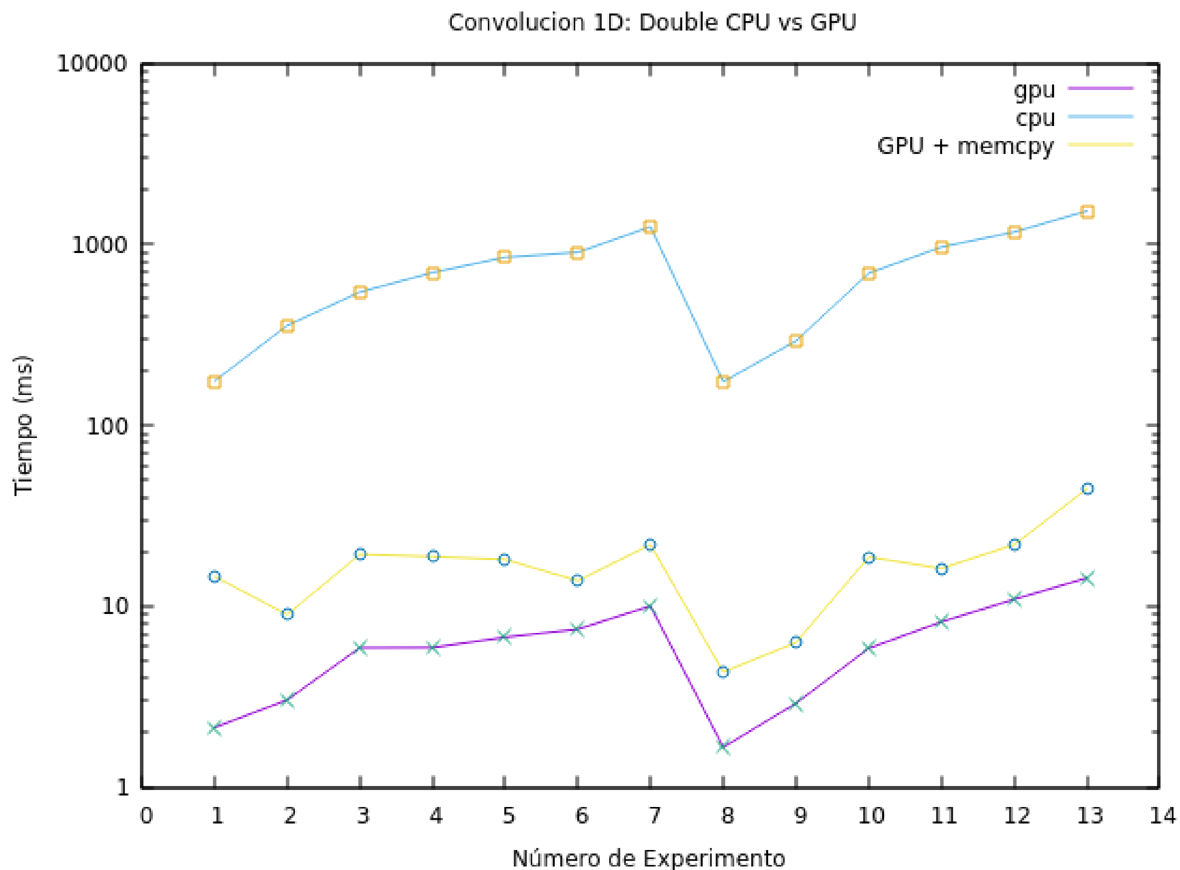


Figure 3: Resultados comparativos entre CPU y GPU con tipos de datos float.

de los tiempos con respecto a los de punto flotante. Por ejemplo, el tiempo de uso de GPU para float en las experiencias 15 y 19 son las únicas que superan al de double.



Para una mejor comparación, se presenta la Figura 4, la cual muestra la diferencia entre las ejecuciones de las mismas experiencias usando ambos tipos de datos. Aunque los tiempos de GPU son similares al principio, a partir de las pruebas 8 se nota un incremento estable en los tiempos. Se puede concluir, que el tiempo de ejecución de double supera al de float tanto en GPU como en GPU sumado el tiempo de transferencia, haciéndose más notorio a medida que la cantidad de datos crece.

* Convolución 1D y Memorias

Para el siguiente caso, se probará copiar el filtro desde el host a dos tipo de memorias disponibles en la placa gráfica. En la Figura 5 se muestra la ubicación de las diferentes memorias con respecto a los bloques y GPU.

La memoria constante es una memoria global de capacidad más reducida que posee una caché para agilizar su acceso. Puede ser accedida por todos los threads y su acceso es rápido más comparado con la memoria global. Obsérvese que el host puede acceder para inicializarla con información de forma directa. La memoria compartida reside dentro del bloque y puede ser utilizada por sus threads. La cercanía con el thread la hace muy ágil, pero es de tamaño muy reducido y sólo puede ser accedida por el thread lo que lo hace responsable de su inicialización.

Se propone implementar la convolución con el filtro residiendo en las tres memorias: global, compartida y constante. En nuestro caso, la Tabla 1 indica que disponemos de más de 10GiB de memoria global, 64MiB de memoria constante y 48MiB de memoria compartida.

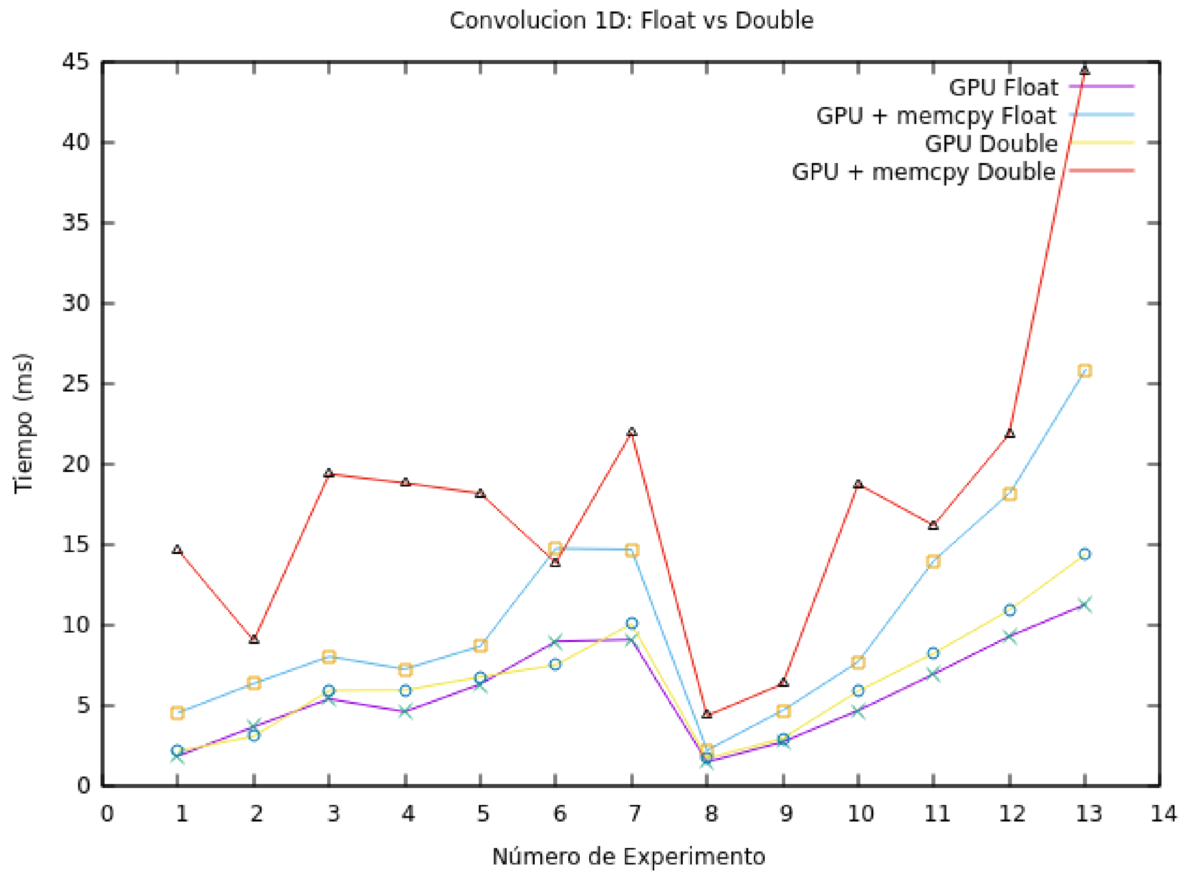


Figure 4: Comparativa entre la convolución realizada con Double y Floats.

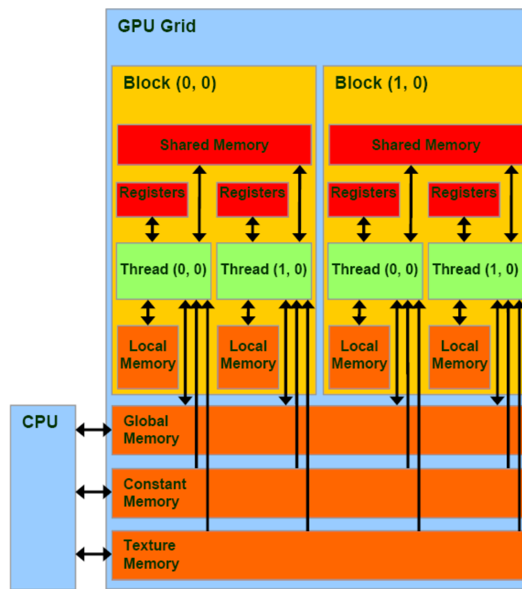


Figure 5: Estructura del GPU y la ubicación de sus memorias.

5.1 Implementación

El código propuesto posee las tres implementaciones y la correspondiente a la secuencial para realizar la comparativa de los resultados. La implementación de memoria global es la misma que la presentada en la sección anterior.

En el siguiente código, se implementa la convolución con el filtro en memoria constante. Primero, se declara una variable para referenciar el espacio de memoria. Luego, se implementa la convolución accediento a esta variable.

```
// declaracion del filtro en memoria constante
__constant__ FLOAT d_filtro_constant[MAX_FILTER];

/**
 convolucion utilizando el filtro en memoria constante
*/
__global__ void conv_gpu_constant_memory (const FLOAT* input, FLOAT* output,
                                          const int n, const int m)
{
  int j = blockIdx.x * blockDim.x + threadIdx.x;

  output[j] = 0.0;
  for (int i = 0; i < m; i++){
    output[j] += d_filtro_constant[i] * input[i+j];
  }
}
```

Finalmente, previo a la ejecución del kernel, es necesario copiar la información del filtro del host a la memoria por medio de la siguiente instrucción.

```
cudaMemcpyToSymbol(d_filtro_constant, h_filter, size_filter);
```

La memoria compartida difiere de esta implementación. En el siguiente código se observa la función que implementa la convolución. Obsérvese que primero se reserva el espacio en la memoria compartida y, posteriormente, se copia el filtro de forma paralela: cada thread copia un número float del filtro. Luego, cuando todos los threads hayan terminado la copia, se procede a la convolución.

Esta diferencia es importante pues no existe un comando que copie el filtro desde la memoria del host a la memoria compartida.

```
/* Solucion que solo sirve para Nh menor a tamaño de bloque */
__global__ void conv_gpu_shared_memory(const FLOAT *input, FLOAT *output,
                                       const FLOAT *filter,
                                       const int n, const int m)
{

  int tidx = blockIdx.x * blockDim.x + threadIdx.x; // global
  int id = threadIdx.x;

  __shared__ float filter_sm[MAX_FILTER];

  // lleno el vector de memoria compartida con los datos del filtro
  // Cada thread copia un float del filtro.
  if (id < m){
    filter_sm[id] = filter[id];
  }

  // todos los threads del bloque se deben sincronizar antes de seguir
  __syncthreads();


  /*
   Barro vector input (tamaño N) y para cada elemento j hasta N hago la
   operacion de convolucion: elemento i del vector filter por elemento
   i+j del vector input.
  */
```

```
    output[tidx] = 0.0;
    for(int i = 0; i < m; i++){
      output[tidx] += filter_sm[i] * input[i+tidx];
    }

  }
```

5.2 Resultados

En el gráfico de la Figura 6 se observa los resultados de la convolución utilizando los tres tipos de memorias y usando un arreglo de tipo float. Los parámetros de los experimentos son los mismos que fueron utilizados en las sección anterior (véase Tabla 3).

Como se puede apreciar, el uso de la memoria global conlleva un aumento considerable en el tiempo de ejecución, incluso hasta el doble con respecto a las otras dos. Para el caso de la memoria compartida y constante mejoran considerablemente sus tiempo, en especial a medida que se aumenta la cantidad de datos entre los experimentos 8 a 13.

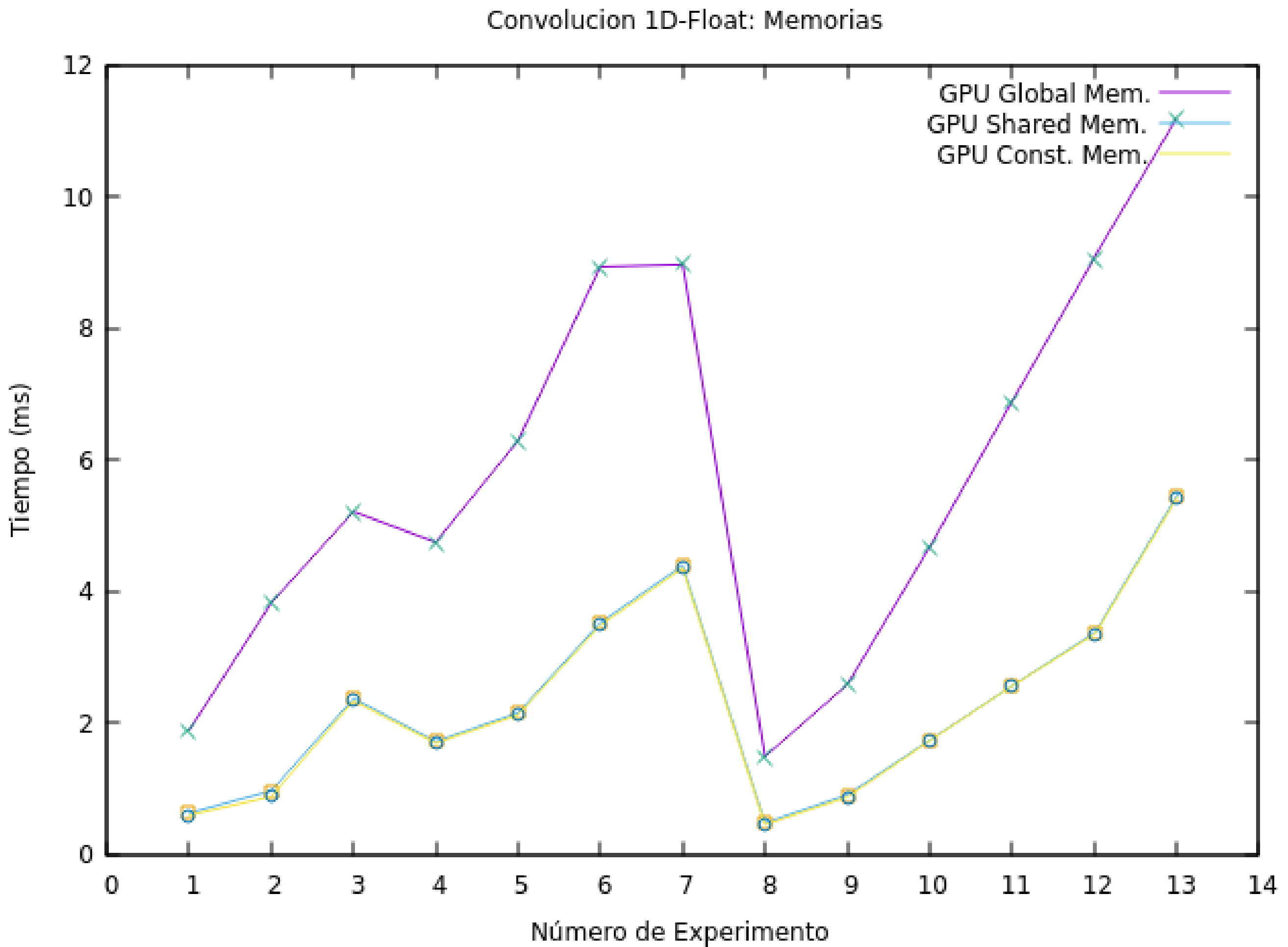


Figure 6: Resultados de la convolución usando distintos tipos de memorias.

Utilizando tipos de datos double, se aprecia curvas similares para la memoria global. En el caso de la memoria compartida, se observa un aumento con respecto a la constante, en especial si el filtro está desalineado (experimento 6). En el experimento 13 se observa la diferencia de los tres tipos de memoria, quedando como mejor candidato para este caso el tipo de memoria constante.

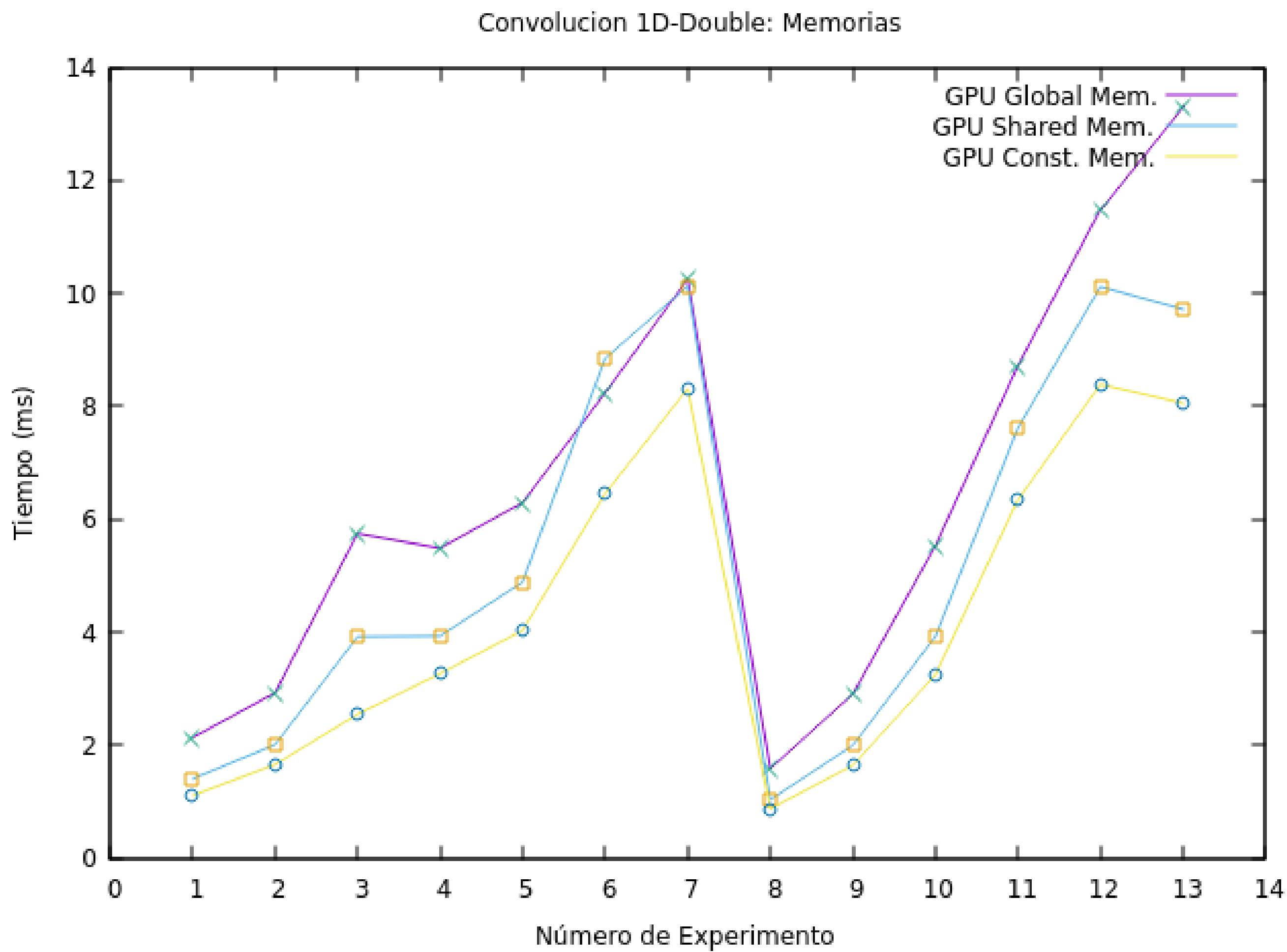


Figure 7: Resultados de la convolución usando distintos tipos de memorias y tipos de datos double.

6 Filtro de Imágen

Para esta sección se plantea la aplicación de dos filtros diferentes a una imágen. El formato de ésta última es en escala de grises y de una magnitud de 750x499 píxeles, pudiéndose considerar como una matriz de números enteros. El filtro se aplica utilizando la técnica de convolución 2D.

Los filtros son matrices de 3×3 mostrados a continuación. El primero corresponde al filtro promedio y el segundo al filtro enfocado.

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

La matriz del filtro es aplicada utilizando la técnica de la convolución 2D. El código para aplicar el filtro promedio para cada pixel de la salida es el siguiente. Considere lo siguiente:

- La imagen de entrada se encuentra en el arreglo de una dimensión `d_imagen_in` (la posición de columna y fila debe calcularse para convertirla a una dimensión).
- `FILTRO_RCOLS` y `FILTRO_RFILAS` son las cantidades de columnas y filas que posee el filtro.
- `d_filtro` es un arreglo unidimensional con los valores del filtro a aplicar.

```
  int k, l;
  float aux = 0.0;
  for(k=-1; k <= 1; k++) { // fila filtro
    for (l = -1; l <= 1; l++) {// col filtro
      int fx = FILTRO_RCOLS + l;
      int fy = FILTRO_RFILAS + k;
      aux += d_filtro[fx + fy * FILTRO_COLS] *
d_imagen_in[(myCol+l) + (myRow+k) * X];
    }
   }

  d_imagen_out[myCol + myRow * cols] = aux;
```

Para el filtro promedio, la aplicación de la matriz con valores 1 debe ser dividida por la cantidad de elemenos que posee el filtro (9). Por ello, su función sólo modifica la última línea del código anterior por:

```
d_imagen_out[myCol + myRow * cols] = aux / TAM_FILTRO;
```

6.1 Resultados

La imágen original se muestra en la Figura 8. La salida de la aplicación del kernel del filtro promedio se muestran en la Figura 9 y la del filtro enfoque en la Figura 10. La salida de las funciones paralelas son idénticas a las de las secuenciales.

En cuanto a los tiempos de ejecución, son mostrados en la Figura 11. Para cualquiera de los casos, utilizar la solución secuencial requiere de una gran cantidad de tiempo de ejecución comparada con la paralela. Además, se puede apreciar que la cantidad de tiempo utilizado para copiar los datos necesarios a la GPU supera considerablemente al del tiempo de la ejecución del kernel.

Para comparar los tiempos de un filtro con respecto al otro. En la Figura 12 se presenta sólo la utilización de GPU de ambos filtros. Es curioso el hecho de que el filtro promedio posea menos tiempo de ejecución a pesar de tener una división extra.

Figure 8: Imagen original usada de entrada para los filtros.

Figure 9: Salida de la aplicación del kernel del filtro promedio.

Figure 10: Salida de la aplicación del kernel del filtro enfocado.

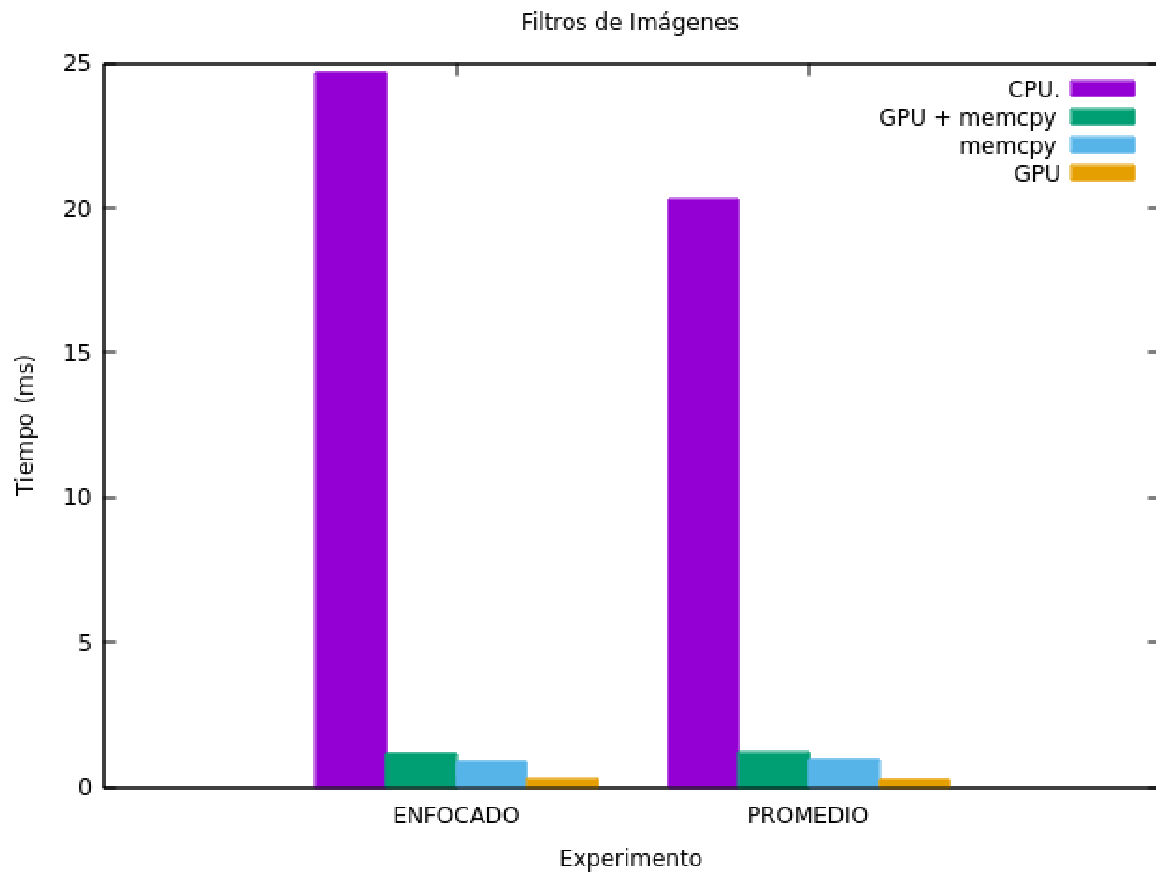


Figure 11: Resultados de aplicar el filtro enfocado y promedio en la imagen.

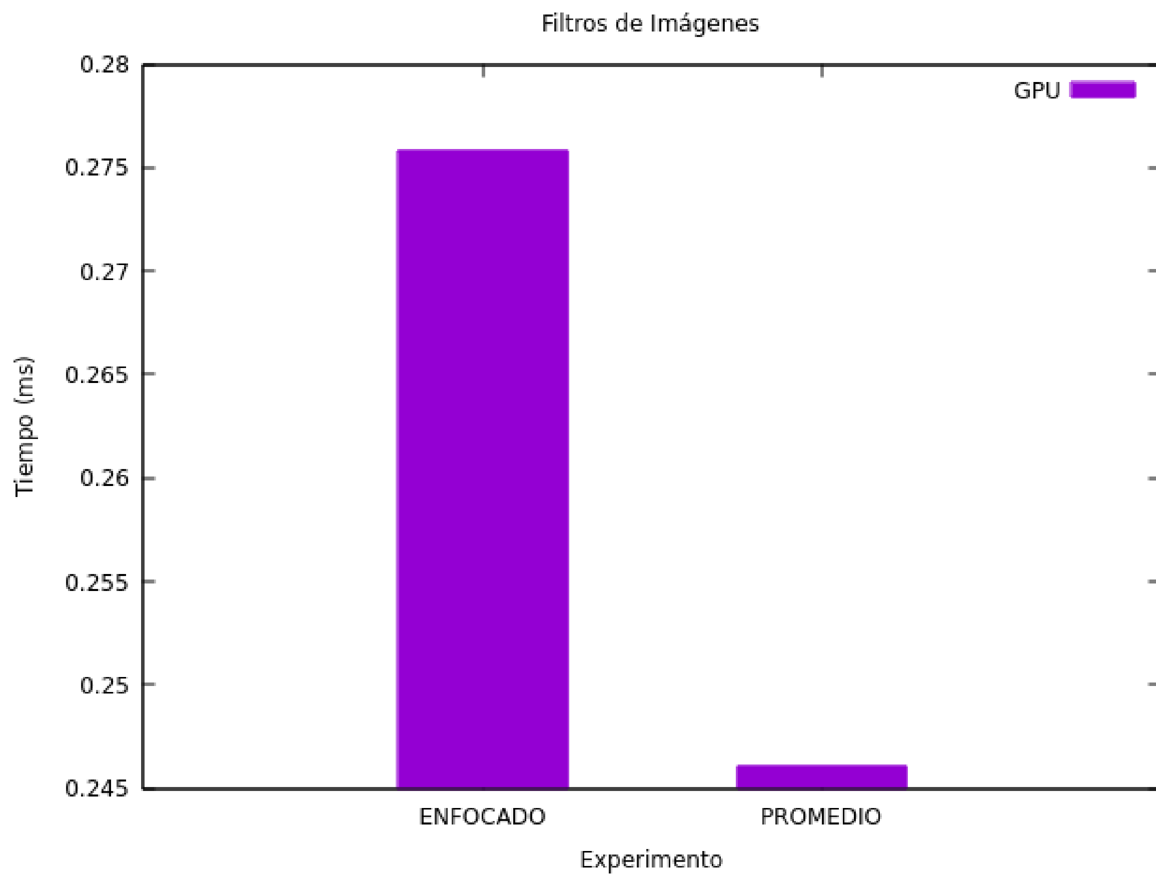


Figure 12: Tiempos de ejecución de los kernels de cada filtro (sólo el uso de GPU).

7 Filtro Sobel

El filtro Sobel requiere de la aplicación de dos kernels de 3 × 3 sobre la imagen original generando las matrices G_x y G_y. Si suponemos que $*$ es la aplicación de la convolución, entonces las siguientes son las expresiones que se deben implementar.

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * A \qquad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} * B$$

En el archivo `imagen.cu` se encuetran las funciones necesarias para aplicar el filtro Sobel. La función `kernel_aplicar_filtro` aplica uno de los filtros para obtener G_x o G_y dependiendo del valor del parámetro `hor`. Los filtros están dentro de un espacio de memoria constante referenciados por `d_filtro_hor` y `d_filtro_ver`.

El siguiente código corresponde con ésta función. Obsérvese la similitud con la convolución, el cual aplica el filtro apropiado. Para que sea utilizable por la GPU, se paraleliza teniendo en cuenta que se desea obtener el resultado de un pixel de la salida por cada thread.

```
__constant__ float d_filtro_hor[9];
__constant__ float d_filtro_ver[9];

__global__ void kernel_aplicar_filtro(float* d_imagen_in, float* d_imagen_out,
                                      bool hor, int filas, int cols){

  int icol = threadIdx.x + blockIdx.x * blockDim.x;
  int irow = threadIdx.y + blockIdx.y * blockDim.y;


  if ((icol < cols-1) && (irow < filas-1)
      && (icol > 0) && (irow > 0)){
    float aux = 0.0;

    // Aplicar el filtro al pixel [icol, irow]
    for (int k = -1; k <= 1; k++){ // filas
      for (int l = -1; l <= 1; l++){ // cols
        int fx = l + 1;
        int fy = k + 1;

        if (hor){
          aux += d_filtro_hor[fx + fy * 3] *
            d_imagen_in[(icol + l) + (irow + k) * cols];
        }else{
          aux += d_filtro_ver[fx + fy * 3] *
            d_imagen_in[(icol + l) + (irow + k) * cols];
        }

      }
    }

    d_imagen_out[icol + irow * cols] = aux;
  }
}
```

El condicional para determinar si aplicar un filtro u otro depende del parámetro `hor`. Es recomendado por CUDA evitar las diferentes ejecuciones dentro de un warp. Esto se debe a que la arquitectura de los procesadores gráficos consiste mayormente en Single Instruction Multiple Data (SIMD) lo cual consiste en ejecutar una instrucción en múltiple datos por unidad de tiempo. En este caso, el valor de `hor` es verdadero o falso en toda la ejecución del warp, evitando que algunos threads vayan por una ramificación haciendo necesario varios ciclos para ejecutar cada una de las ramificaciones.

Luego de obtener las matrices, se requiere calcular la matriz $G = \sqrt{G_x^2 + G_y^2}$. Ésta se logra por medio de la función `kernel_calcular_g`. La paralelización se realiza de la misma manera que los filtros: se calcula la formula indicadapor cada pixel de salida en cada thread.

```
__global__ void kernel_calcular_g(float *d_g_x, float *d_g_y,
  float *d_imagen_out,
  int filas, int cols){
  int icol = threadIdx.x + blockIdx.x * blockDim.x;
  int irow = threadIdx.y + blockIdx.y * blockDim.y;

  if ((icol < cols-1) && (irow < filas-1)
      && (icol > 0) && (irow > 0)){

    int idx = icol + irow * cols;
    d_imagen_out[idx] = (float) sqrt
      (
       (float) powf(d_g_x[idx], 2)
       + (float) powf (d_g_y[idx], 2)
       );
  }
}
```

Finalmente, la función `aplicar_filtro_sobel_par` ejecuta todos estos kernels de forma ordenada. En el siguiente retazo de código se muestra las secciones más importantes: la inicialización del filtro en memoria host, la copia de los filtro a la memoria constante de la GPU, la reserva de espacio para los resultados de G_x y G_y, la disposición de la grilla y sus bloques y los llamados a los kernels para obtener los resultados.

```
  inicializar_filtro_sobel_horizontal(h_filtro_hor, 9);
  inicializar_filtro_sobel_vertical(h_filtro_ver, 9);

  cudaMemcpyToSymbol(d_filtro_hor, h_filtro_hor, sizeof(float) * 9);
  cudaMemcpyToSymbol(d_filtro_ver, h_filtro_ver, sizeof(float) * 9);

  float *d_g_x, *d_g_y;
  cudaMalloc((void **) &d_g_x, size_img);
  cudaMalloc((void **) &d_g_y, size_img);

  dim3 blocklayout(16, 16);
  dim3 gridlayout(cols /blocklayout.x + (cols  % blocklayout.x ? 1 : 0),
                filas/blocklayout.y + (filas % blocklayout.y ? 1 : 0));


  kernel_aplicar_filtro<<<gridlayout, blocklayout>>>(d_imagen_in, d_g_x,
                                                   true, filas, cols);
  kernel_aplicar_filtro<<<gridlayout, blocklayout>>>(d_imagen_in, d_g_y,
                                                   false, filas, cols);
  kernel_calcular_g<<<gridlayout, blocklayout>>>(d_g_x, d_g_y, d_imagen_out,
                                             filas, cols);
```

7.1 Resultados

Para su ejecución se utilizó la misma imagen de entrada propuesta en la sección anterior (Figura 8). La salida aplicando el filtro Sobel de forma paralela se muestra en la Figura 13. La imagen de la aplicación secuencial es la misma.

En cuanto a los tiempos de ejecución, en la Figura 14 se presentan los de ambos placas G y T. Como se puede observar, es notoria la diferencia entre el uso del CPU y el GPU, mejorando éste último aún con el traslado de la imagen del host al dispositivo.

Figure 13: Resultado de aplicar el filtro Sobel de forma paralela.

Filtro Sobel

CPU.
GPU + memcpy
memcpy
GPU

Tiempo (ms)

0
10
20
30
40
50
60

0
1

Experimento

Figure 14: Tiempos de ejecución de la aplicación del filtro de Sobel en CPU y GPU.

8 Apéndice

8.1 Resultados de la Suma de Vectores

8.1.1 Grupo G

num	tam	veces	cpu	gpu	HtoD	DtoH	memcpy	total_gpu
1	100	1	0.001	0.060032	0.002048	0.001184	0.003232	0.063264
2	1000	1	0.004	0.069568	0.002656	0.00144	0.004096	0.073664
3	10000	1	0.036	0.060608	0.012768	0.008832	0.0216	0.082208
4	100000	1	0.396	0.0736	0.06816	0.032416	0.100576	0.174176
5	1000000	1	22.845	0.14096	1.6634	18.864	20.5274	20.66836
6	10000000	1	40.348	0.76944	201.95	26.895	228.845	229.61444
7	100	100	0.044	1.58166	0.001952	0.00112	0.003072	1.584732
8	100	200	0.082	2.95117	0.00208	0.001184	0.003264	2.954434
9	100	300	0.13	4.29638	0.002048	0.001184	0.003232	4.299612
10	10000000	100	10846.7	288.979	27.469	27.799	55.268	344.247
11	10000000	200	20878.2	366.811	19.131	124.21	143.341	510.152
12	10000000	300	31552.3	662.707	12.777	26.068	38.845	701.552

8.1.2 Grupo T

num	tam	veces	cpu	gpu
1	100	1	0	0.042976
2	1000	1	0.003	0.045056
3	10000	1	0.033	0.043872
4	100000	1	0.334	0.048864
5	1000000	1	3.13	0.063456
6	10000000	1	31.53	0.20416
7	100	100	0.035	1.03117
8	100	200	0.069	1.98525
9	100	300	0.106	2.9952
10	10000000	100	3186.68	15.7568
11	10000000	200	6312.74	31.0619
12	10000000	300	9534.76	46.7993

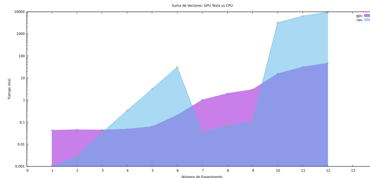


Figure 15: Resultados de la suma de vectores en las tarjetas gráficas Tesla.

8.2 Resultados de la Suma de Matrices

8.3 Resultados de la Convolución 1D

8.3.1 Grupo G Float

num	tam_data	tam_filter	cpu	gpu	HtoD	DtoH	memcpy	total_gpu
1	1280000	32	166.144559	1.810112	0.56989	2.1192	2.68909	4.499202
2	1280000	64	352.529806	3.63408	0.57037	2.1118	2.68217	6.31625
3	1280000	100	481.500205	5.36736	0.62276	1.9976	2.62036	7.98772
4	1280000	128	681.105225	4.558016	0.6471	2.0083	2.6554	7.213416
5	1280000	160	759.491343	6.283264	0.60014	1.751	2.35114	8.634404
6	1280000	200	1029.78404	8.932768	3.629	2.1487	5.7777	14.710468
7	1280000	256	1266.530555	9.05232	3.6633	1.9336	5.5969	14.64922
8	320000	128	172.972962	1.427776	0.42218	0.32771	0.74989	2.177666
9	640000	128	293.857043	2.688544	1.0201	0.92958	1.94968	4.638224
10	1280000	128	715.717712	4.643712	0.86717	2.1597	3.02687	7.670582
11	1920000	128	871.812922	6.894464	4.3098	2.7357	7.0455	13.939964
12	2560000	128	1162.075456	9.245152	5.6464	3.2468	8.8932	18.138352
13	3200000	128	1519.545263	11.207872	9.89	4.7189	14.6089	25.816772

8.3.2 Grupo G Double

vs_float	num	tam_data	tam_filter	cpu	gpu	HtoD	DtoH	memcpy	total_gpu
1	14	1280000	32	175.309864	2.129664	8.43	4.0529	12.4829	14.612564
2	15	1280000	64	355.01804	3.034016	1.345	4.6045	5.9495	8.983516
3	16	1280000	100	544.46133	5.890368	8.6345	4.8191	13.4536	19.343968
4	17	1280000	128	696.174747	5.902752	7.9031	4.9791	12.8822	18.784952
5	18	1280000	160	846.687975	6.724256	7.8314	3.5796	11.411	18.135256
6	19	1280000	200	903.357921	7.46576	2.3105	4.0445	6.355	13.82076
7	20	1280000	256	1247.355298	10.02384	7.778	4.0936	11.8716	21.89544
8	21	320000	128	174.005531	1.658464	1.4504	1.2117	2.6621	4.320564
9	22	640000	128	292.378909	2.906496	1.4104	1.9768	3.3872	6.293696
10	23	1280000	128	692.077252	5.84944	7.7513	5.1069	12.8582	18.70764
11	24	1920000	128	965.685331	8.190496	3.205	4.7574	7.9624	16.152896
12	25	2560000	128	1167.45247	10.879584	2.4406	8.5013	10.9419	21.821484
13	26	3200000	128	1523.375297	14.314592	21.272	8.7521	30.0241	44.338692