# Computación de Alto Rendimiento en GPU

Trabajo Práctico

Gimenez, Christian

05 Abril 2019

## Índice

# 1. Convenciones y Consideraciones

Cada resolución de los ejercicios propuestos para este trabajo se localiza un una carpeta individual con el nombre correspondiente. 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 referidos al ejercicio en el cluster.
- Los resultados obtenidos en archivos .out y .err en un subdirectorio denominado "results".
- Un subdirectorio "imgs" con scripts para procesar los archivos de resultados y generar un CSV utilizable por el programa gnuplot. Los gráficos de barras y líneas producidos se encuentran en este mismo directorio.

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 (GeForce GTX 980) y

**Grupo T** a las máquinas con placas Tesla V100.

## 1.1. 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. Al utilizar el comando `nvprof`, por defecto la salida del profiler se realiza sobre la salida de error y la salida del programa al que se le está testeando se realiza sobre la estándar.

Los nombres de los archivos de la salida estándar del programa poseen el siguiente formato:

```
Nombre_programa-Grupo-Tipo_Profiling-Numero_prueba.out
```

De forma análoga, el nombre de archivo de los de la salida de error poseen el mismo formato pero con extensión `.err`.

Los tipos de profiling utilizados para cada prueba son tres: "none" indicando que no se utilizó `nvprof` sin parámetros, "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). Este archivo contendrá datos de profiling de evento puesto que guardará la salida de error del comando `nvprof --events all ./suma_matrices_A`. La salida estándar de suma matrices estará en el archivo `suma_matrices_A-G-events-1.out` y poseerá los resultados de la ejecución programa `suma_matrices_A`.

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

## 1.2. Disponibilidad de los Códigos y los Resultados

Los códigos fuentes y los resultados se encuentran disponibles en un repositorio público en Bitbucket. La dirección Web para acceder a éste es la siguiente: `https://bitbucket.org/cnngimenez/curso-gpu`. También, se puede encontrar los fuentes de este informe y los códigos utilizados para procesar la salida y generar los gráficos utilizados en el presente escrito.

En dicho repositorio se utiliza el sistema de control de versiones denominado Mercurial. Su utilización es similar al de los más conocidos y está disponible para instalar en todos los repositorios de las distribuciones

de GNU/Linux. Más información acerca de su uso puede encontrase en su sitio oficial `https://www.mercurial-scm.org/`.

Para obtener una copia del repositorio se utiliza el comando "clone" de Mercurial de la siguiente manera:

```
hg clone https://bitbucket.org/cnngimenez/curso-gpu
```

Además, el informe se ha escrito utilizando Org-mode, el cual se encuentra disponible por defecto en el editor Emacs. Los beneficios de éste complemento es que permite exportar el documento en varios formatos: LaTeX, PDF, HTML, etc. Utilizando esta característica, se ha exportado una versión PDF disponible para su descarga en `https://bitbucket.org/cnngimenez/curso-gpu/downloads/` y una versión HTML publicada en el sitio `http://crowd.fi.uncoma.edu.ar/~christian.gimenez/cursos/gpu/`.

## 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 [1].

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 de hilos 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 Cuadro 1.

Cuadro 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. hilos por bloque | 1024 | 1024 |
| Max. dim. de hilos | (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. Más información acerca de las especificaciones técnicas de estas placas gráficas se encuentran disponibles en la página oficial de NVIDIA® y GeForce [2, 3, 4].

## 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 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 hilos en X e Y).

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

Utilizando varios valores para estos parámetros se realizan varias ejecuciones para medir sus tiempos de ejecución. En la Cuadro 2 se presentan las distintas pruebas realizadas y sus parámetros. Las pruebas iniciales se enfocan en ver qué sucede con los tiempos de ejecución al incrementar la cantidad de datos. Las pruebas 7, 8 y 9 se incrementan las veces en que se repite la suma pero con pocos datos. Las tres siguientes tratan de usar una buena cantidad de datos y variando la repetición de la suma. Por último, se repiten las pruebas pero aumentando la cantidad de hilos dentro de la configuración de los bloques.

Cuadro 2: Parámetros utilizados en las distintas ejecuciones de la suma de vectores.

| Núm. | tam | veces | gridx | gridy | blockx | blocky |
|---|---|---|---|---|---|---|
| 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 |
| 13 | 10000000 | 300 | 19532 | 1 | 512 | 1 |
| 14 | 10000000 | 300 | 13021 | 1 | 768 | 1 |
| 15 | 10000000 | 300 | 9766 | 1 | 1024 | 1 |
| 16 | 1000 | 1 | 1 | 1 | 1024 | 1 |
| 17 | 10000 | 1 | 10 | 1 | 1024 | 1 |
| 18 | 100000 | 1 | 98 | 1 | 1024 | 1 |
| 19 | 1000000 | 1 | 977 | 1 | 1024 | 1 |
| 20 | 10000000 | 100 | 9766 | 1 | 1024 | 1 |
| 21 | 10000000 | 200 | 9766 | 1 | 1024 | 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 6, se requiere 39063 bloques de 256 hilos para poder realizar el cálculo de 10000000 elementos del vector, a pesar de que hayan quedado 128 hilos 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 experimento.

Obsérvese los experimentos del 1 al 6, en los cuales 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. Esto nos indica que la tranferencia de datos penaliza de forma significativa los tiempos de ejecución con respecto al tiempo de uso de la GPU.

Para el caso de los experimentos numerados entre 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. Los resultados obtenidos en estas tres purebas nos muestra que el uso de la GPU no se justifica para pocos datos.

Los experimentos de 9 al 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.

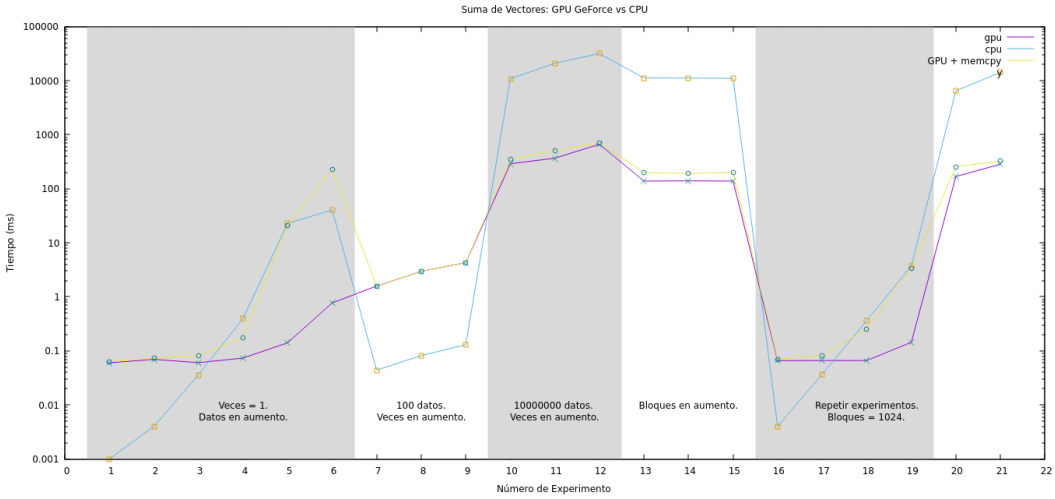


Figura 1: Resultados de los tiempos de la suma de vectores.

En los experimentos del 13 al 15 se aumenta la cantidad de hilos por bloque y se preserva la cantidad de datos y de repeticiones de forma constante. Los resultados no varían mucho manteniendose constante.

Es importante ver el aprovechamiento del hardware al comparar el experimento 15 con el 12. Ambos poseen la misma cantidad de datos y se repite la misma cantidad, pero la cantidad de hilos por bloque es diferente. El experimento 15 resolvió el problema en menos tiempo de ejecución, haciendo uso del hardware de forma más eficiente. Del 16 al 21 se repiten los experimentos utilizando 1024 hilos por bloque. Aprovechando el máximo de paralelismo se obtiene una mejora en especial cuando se aumentan la cantidad de operaciones que la GPU debe hacer. Por ejemplo, al comparar la prueba 11 con la 21 se obtiene una mejora de $510{,}152 - 324{,}5559 = 185{,}5961$ ms.

## 4. Suma de Matrices

La suma de matrices consiste en trasladar la suma de vectores a una dimensión más. En este caso, se debe considerar en distribuir el procesamiento a hilos usando grillas y bloques con dos dimensiones. Análogo al programa de suma de vectores, el tamaño de la matriz y la cantidad de bloques y grillas se puede configurar con los argumentos quedando la sinopsis del comando de la siguiente forma:

```
./suma_matrices filas cols grid_x grid_y block_x block_y
```

En el código del kernel, se realiza un chequeo para evitar la ejecución de hilos cuyos índices estén fuera del tamaño del vector. Esta situación puede producirse cuando el tamaño de la matriz no es múltiplo de la cantidad de hilos por bloque en la dimensión determinada.

Además, debido a la consideración de alineamiento de 32 hilos por warp, es posible que hayan hilos 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 hilos repartidos en 4 × 4 (16) bloques de 32 × 32 (1024) hilos, restando 6 bloques y 240 hilos asignados pero sin utilizar. Por consiguiente, el condicional que figura en el siguiente código del kernel fue incluido para cumplir con este propósito.

```
if (c < cols && f < filas){
  int i = c + (f * cols);
  MatC[i] = MatA[i] + MatB[i];
}
```

Se han ejecutado varias pruebas con diferentes objetivos. En el cuadro 3 se muestran los distintos tamaños de las matrices y la configuración de la grilla y sus bloques. Las pruebas del 1 al 6 se enfocan en ver qué sucede entre los tiempos de CPU y GPU a medida que la cantidad de datos crece. Para las 7 y 8 se busca ver qué sucede ante una grilla de configuración lineal contra una matricial. Y desde la prueba 9 hasta la 12 se desea probar configuracione de grillas desalineadas o con exceso de hilos para ver si esto afecta a la performance. Los resultados se muestran expresados en la Figura 2. En este gráfico se puede apreciar que para una matriz de hasta 5000 × 5000 elementos es beneficioso utilizar la CPU. Esto es debido a que la gran penalización que conlleva el traslado de la información, desde y hacia el dispositivo de la GPU, no compensa con la cantidad de operaciones que la GPU debe realizar.

Cuadro 3: 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 | 32 | 32 |
| 6 | 10000 | 10000 | 313 | 313 | 32 | 32 |
| 7 | 5000 | 5000 | 157 | 5000 | 1024 | 1 |
| 8 | 5000 | 5000 | 157 | 157 | 32 | 32 |
| 9 | 5000 | 5000 | 500 | 5000 | 1000 | 1 |
| 10 | 5000 | 5000 | 314 | 314 | 16 | 16 |
| 11 | 5000 | 5000 | 1000 | 1000 | 5 | 50 |
| 12 | 5000 | 5000 | 10000 | 10000 | 5 | 5 |

Los experimentos del 1 al 6 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 7 y el 8, se denota un leve aumento en los tiempos del GPU al intentar distribuir los datos en 32 × 32. La cantidad de threads que se ejecutaron son 157×5000×1024 = 804625000 y 157×157×32×32 = 25240576 respectivamente. Se requieren 25000000 threads para realizar la suma, sobrando 55462500 y 240576 threads respectivamente.

Analizando las métricas proveídas por el programa `nvprof` en el Cuadro 4 se puede deducir por qué aumentan los tiempos al estar desalineadas y mal configuradas las grillas.

El valor de ocupación alcanzada (*achieved occupancy*) nos muestra una proporción del promedio de los warps activos por ciclos activos al máximo número de warps soportado por un multiprocesador, la diferencia es de 0.28, indicando que hubo inactividad.

Además, se puede diferenciar que hubo una gran cantidad de instrucciones enteras, de control de flujo y otras para el test 7 en comparación con las que más nos interesan que son las de punto flotante. Esto se debe al condicional y al cálculo de la columna y fila de los threads sobrantes.

La cantidad de memoria utilizada, a primera vista, aparenta que es mayor para el test 8 que para el 7. Esto se debe a que la cantidad de threads del primer test es menor, pero a diferencia de la prueba 7, la mayoría de ellos realizan realmente la suma entre dos valores que están en memoria. El test 7 posee más threads pero una gran cantidad no realiza la suma y, como consecuencia, no necesita de los datos de memoria.

En definitiva, el Test 7, que posee los más threads desocupados y desalineados, realiza más trabajo y desaprovecha los recursos de la placa realizando instrucciones que no tienen relación con la suma de matrices. Incluso la utilización de la memoria se ve desaprovechada, puesto que muchos de los threads no acceden a ella resultando en una caída en la tasa de productividad (*Global Load/Store Throughput*) de la memoria.

Cuadro 4: Métricas obtenidas con nvprof para comparar las configuraciones desalineada contra la alineada.

| | Test 7 | Test 8 |
|---|---|---|
| Achieved Occupancy | 0.512730 | 0.790190 |
| Instruction per Warp | 14.5 | 29.923567 |
| | | |
| Global Load Throughput | 29.917GB/s | 104.81GB/s |
| Global Store Throughput | 14.959GB/s | 52.406GB/s |
| | | |
| Control-Flow Func. Unit Utilization | Low (1) | Low (1) |
| Load/Store Func. Unit Utilization | Mid (4) | Low (1) |
| Single-Precision Func. Unit Utilization | Mid (4) | Low (2) |
| | | |
| Instruction Executed | 364240000 | 23602752 |
| dram_utilization | Low (2) | High (8) |
| | | |
| FP Instructions(Single) | 25000000 | 25000000 |
| Integer Instructions | 6705720000 | 476924608 |
| Control-Flow Instructions | 803840000 | 25240576 |
| Load/Store Instructions | 75000000 | 75000000 |
| Misc. Instructions | 4019200000 | 126202880 |
| | | |
| FLOP Efficiency (Peak Single) | 0.06 % | 0.30 % |

Para el resto de las pruebas, se intentan con alineamientos fuera de los 32 hilos 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 $10000 \times 10000$ elementos y sólo se utilizan $5 \times 5$ hilos, se nota un incremento importante.

## 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:

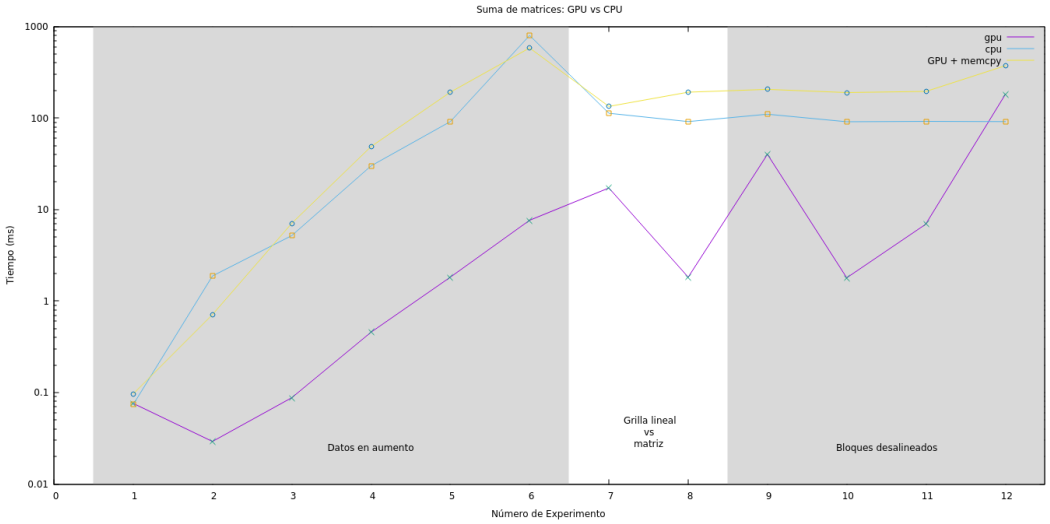


Figura 2: Resultados de los tiempos de ejecutar la suma de matrices para diferentes tamaños.

- 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 Cuadro 5 se pueden observar el número de experimento y sus respectivos parámetros. Las pruebas del 1 al 7 utilizan tamaños fijos del arreglo, pero varían 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. La finalidad de esta distribución de tests es determinar cuándo es más eficiente utilizar la GPU: si es más eficiente a medida que el filtro crece en tamaño o a medida que la cantidad de datos aumenta.

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 y el filtro se mantiene a partir del experimento 8, incluso, el tiempo de transferencia se encuentra en un crecimiento proporcional al tamaño de la entrada.

En la Figura 4 se aprecia el mismo comportamiento para cuando se utilizan tipos de datos double. Sin embargo, al comparar los tiempos en forma numérica (véase la Cuadro 9 y 10 en el Apéndice "Resultados de la Convolución 1D") se detecta mayormente una tendencia a superar los valores 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 5, 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.

## 6. Convolución 1D Usando Varias 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 6 se muestra la ubicación de las diferentes memorias con respecto a los bloques

Cuadro 5: Experiencias a realizarse y los parámetros 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 |

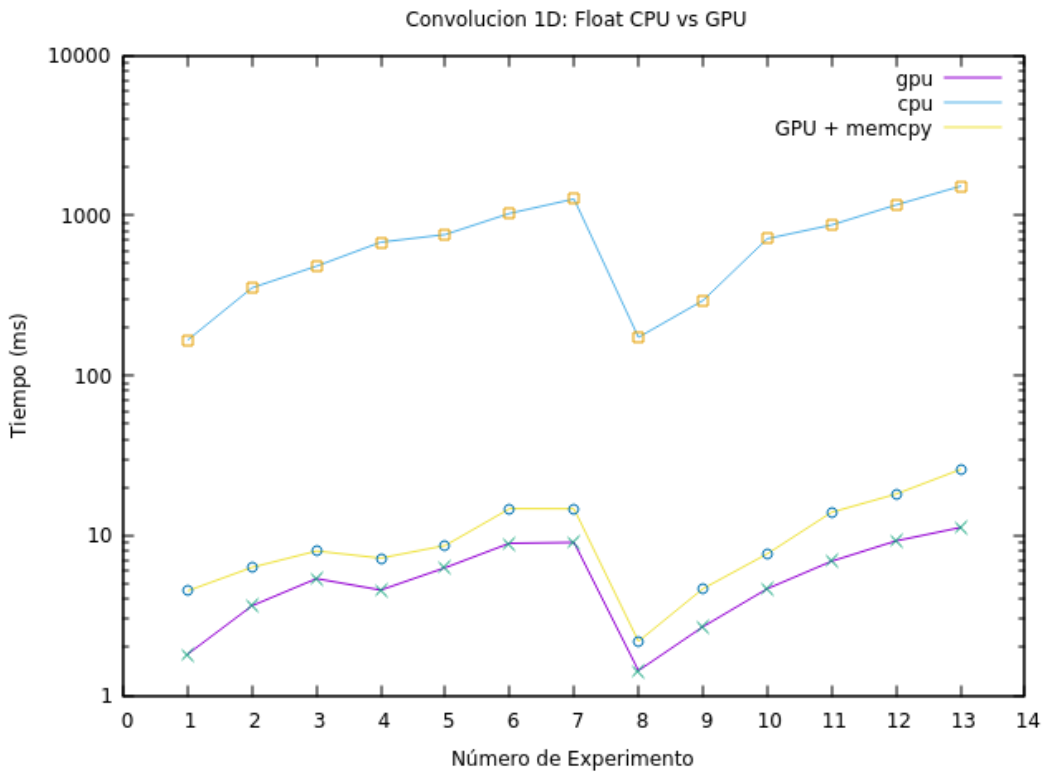


Figura 3: Resultados comparativos entre CPU y GPU con tipos de datos float.

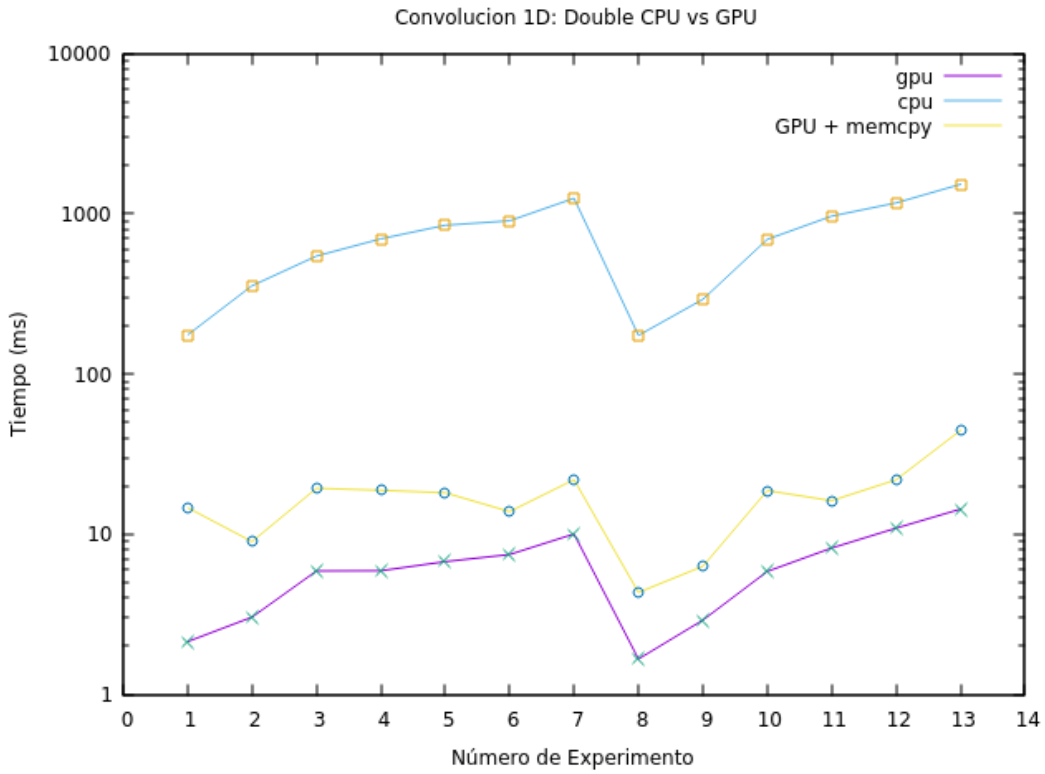


Figura 4: Resultados de ejecutar la convolución 1D usando tipos de datos double.

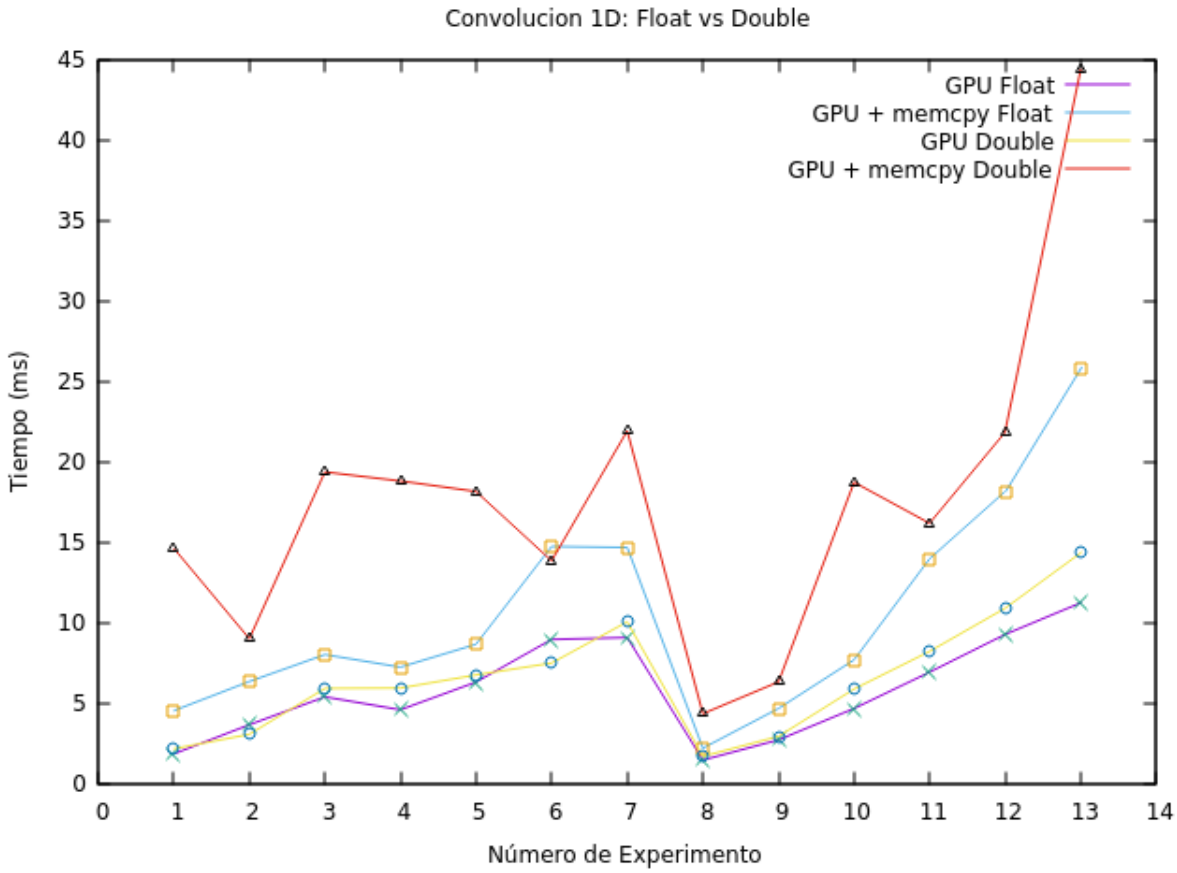


Figura 5: Comparativa entre la convolución realizada con Double y Floats.

y GPU.

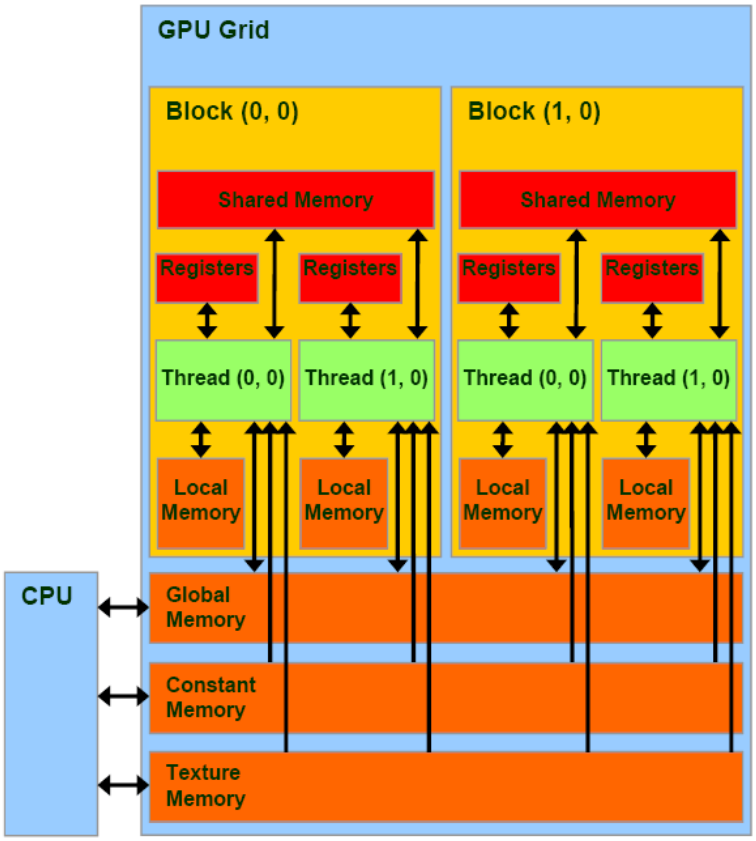


Figura 6: Estructura del GPU y la ubicación de sus memorias.

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 más rápido 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 Cuadro 1 indica que disponemos de más de 10GiB de memoria global, 64MiB de memoria constante y 48MiB de memoria compartida.

## 6.1. 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 hacer referencia al espacio de memoria. Luego, se implementa la convolución accediendo 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];
  }

}
```

## 6.2. Resultados

En el gráfico de la Figura 7 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 Cuadro 5).

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 tiempos, en especial a medida que se aumenta la cantidad de datos entre los experimentos 8 a 13.

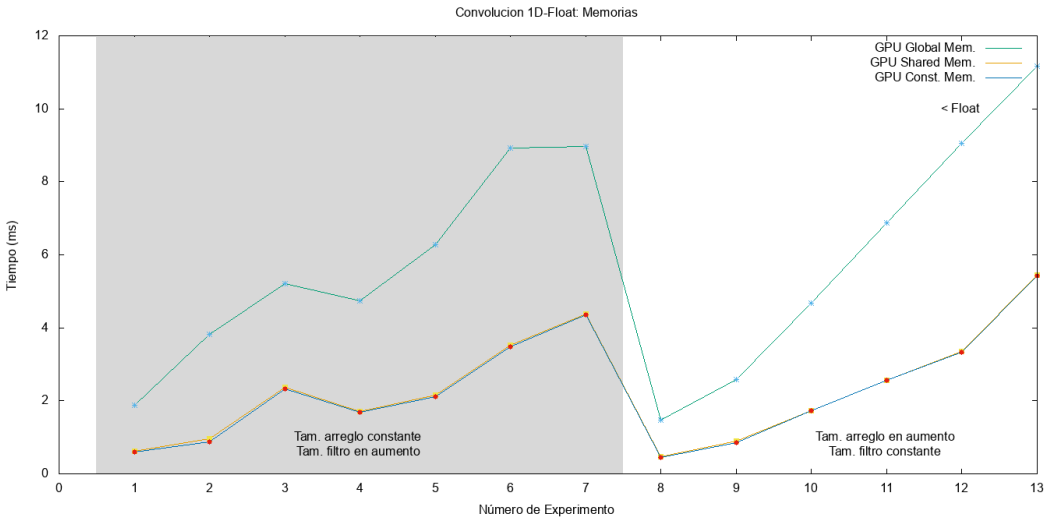


Figura 7: Resultados de la convolución usando distintos tipos de memorias.

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 19). En el experimento 26 se observa la diferencia de los tres tipos de memoria, quedando como mejor candidato para este caso el tipo de memoria constante.

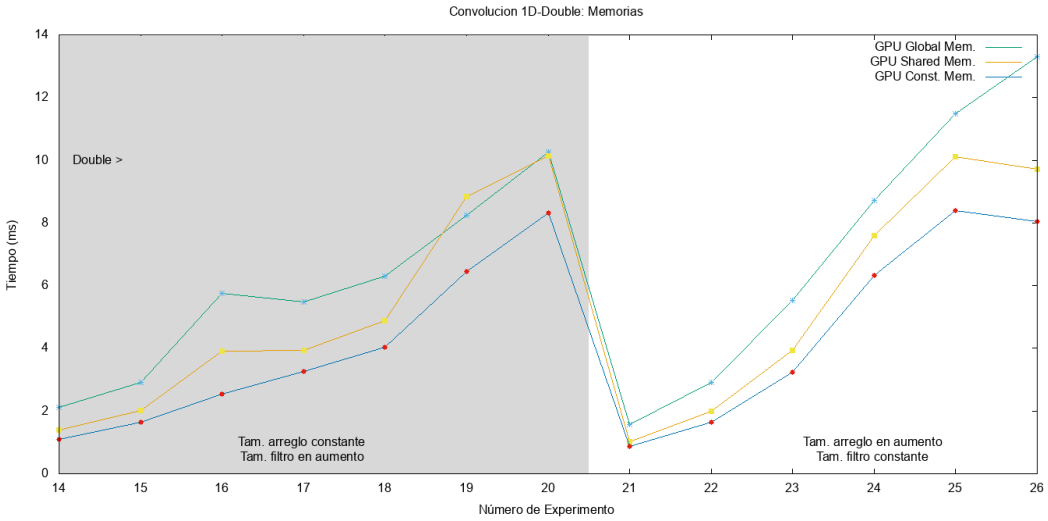


Figura 8: Resultados de la convolución usando distintos tipos de memorias y tipos de datos double.

### 6.2.1. Métricas

Se evaluarán algunas métricas referidas a la utilización de memoria global, constante y compartida y las caché L1 y L2. Es importante reconocer la ubicación de estas memorias puesto que cuanto más cerca del núcleo de procesamiento estén, más rápida se espera que sean (aunque menor es la cantidad que soporta). En la Figura 9 se puede observar la microarquitectura de las placas Fermi, las cuales sus sucesores Kepler y Maxwell están basadas. Las placas del grupo G, GeForce GTX Titan X (GeForce GTX 980) utilizan la microarquitectura Maxwell de la Figura 10 la cual es muy similar a su predecesora.

En ambas figuras se puede observar la ubicación de la caché L2 y de la memoria global (representada

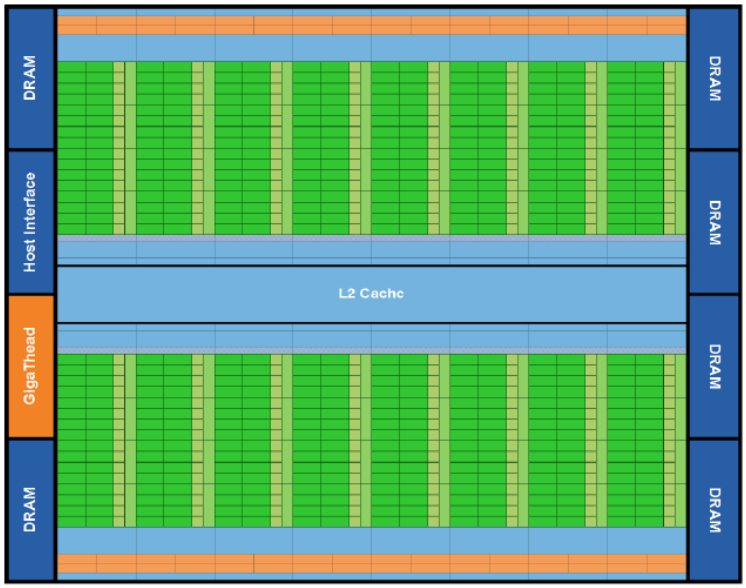


Figura 9: Microarquitectura de las placas Fermi.


Figura 10: Microarquitectura de las placas Maxwell.

como “DRAM” y *“Memory Controller”)*. Ésta se muestra fuera de los *Streaming Multiprocessors* (SM) y de los Maxwell SM (SMM) indicando que es compartida por todos ellos. Se espera que su acceso sea lento comparado con una memoria que está dentro de las unidades SM.

Para el caso de las memorias compartidas y la caché L1, la Figura 11 muestra la estructura interna de un *Streaming Multiprocessor*. Como se puede apreciar, el SM Maxwell de la Figura 12 es similar al SM Fermi pero con mayor cantidad de núcleos, memorias y otros componentes internos. La memoria caché L1 y la memoria compartida se encuentran dentro de la unidad, próxima al núcleo y a las unidades de ejecución lo que las hace altamente veloces. Se debe remarcar que en el caso de las Maxwell, la caché L1 se comparte con la memoria interna de textura, a diferencia de las Fermi que comparte con la memoria compartida [5, 6].

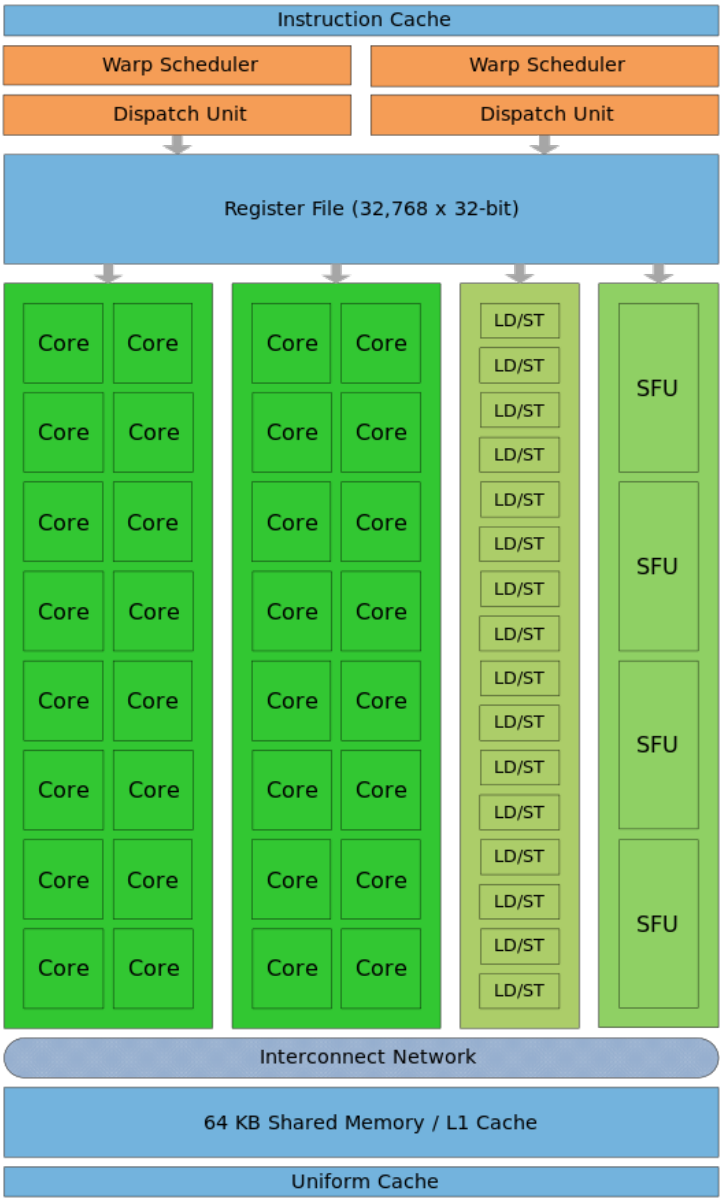


Figura 11: Estructura del SM Fermi.

Para empezar, la Figura 13 muestra la cantidad de transacciones de carga que se realizan a la memoria global. Se puede apreciar que para realizar la misma tarea, la cantidad de transacciones de carga de los

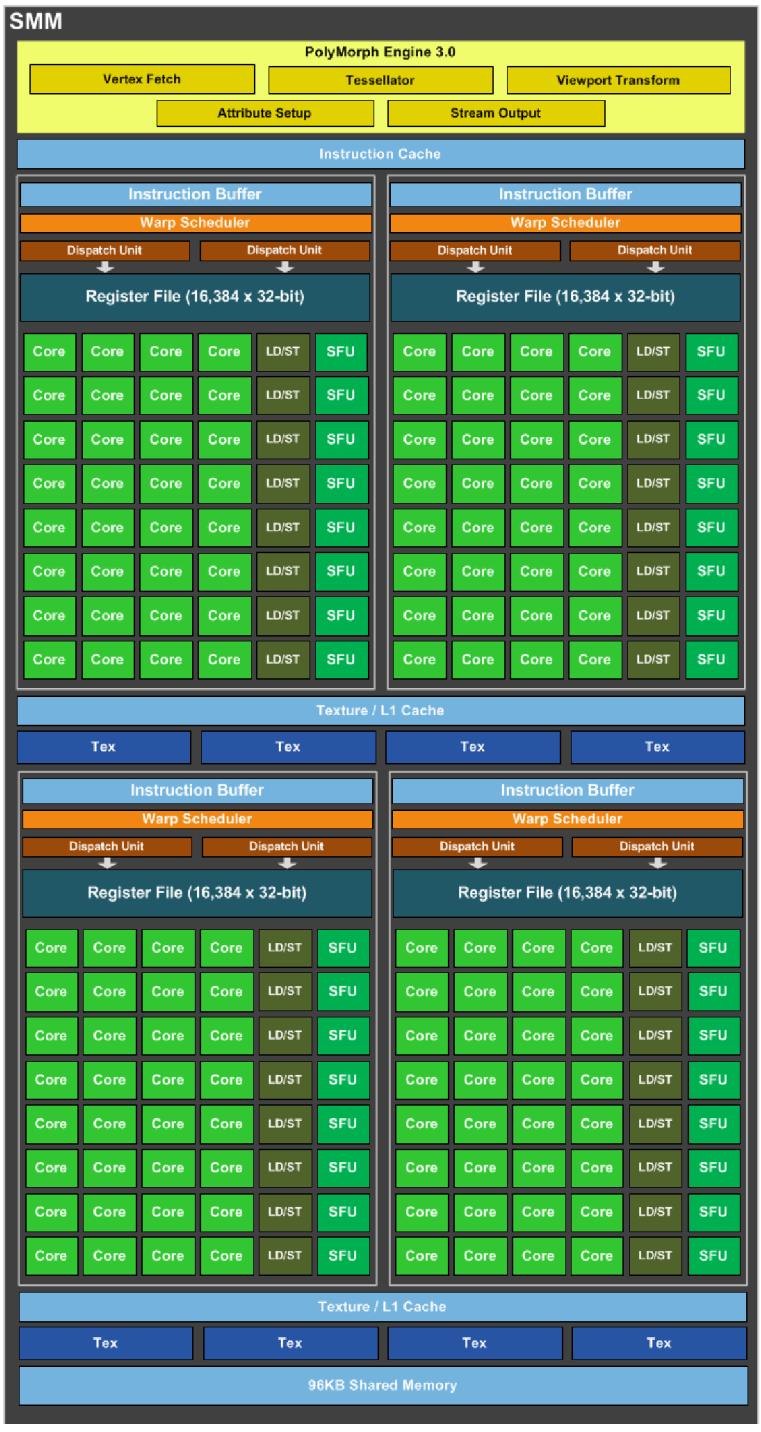


Figura 12: Estructura del SM Maxwell (SMM).

kernels de memoria constante y compartida se ve reducida, debido a que el acceso al filtro no se realiza directamente sobre memoria global. En la memoria constante, una vez cacheado dentro de la SM ya no es necesario consultar la DRAM. Para la memoria compartida, la información se carga dentro de esta memoria una única vez y luego consultada directamente en ella. En la Figura 14 se puede apreciar que todos los kernels acceden las mismas cantidades de veces a la memoria global para almacenar el resultado y su aumento se ve afectado por el tamaño del filtro, pero mayormente por la cantidad de datos a procesar.

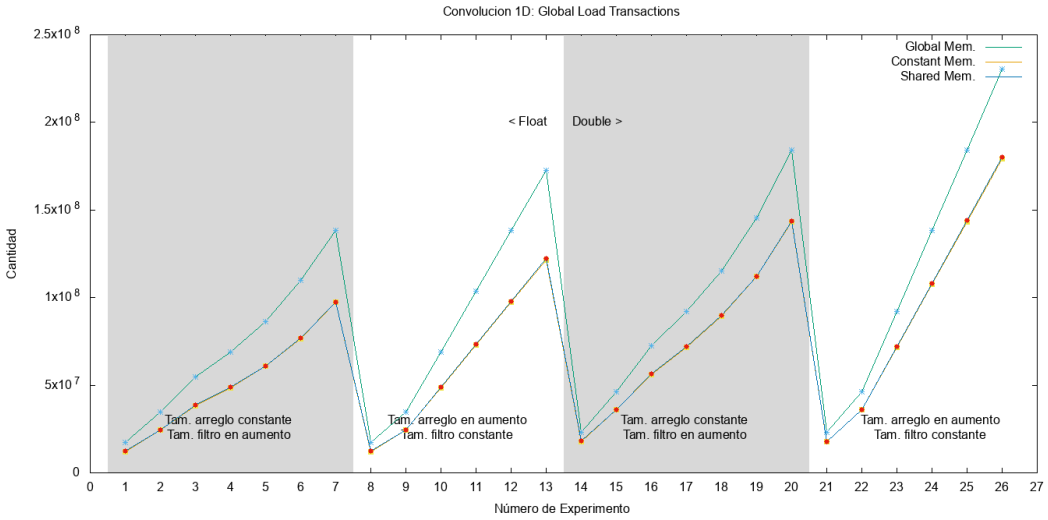


Figura 13: Cantidad de transacciones de carga de a la memoria global.

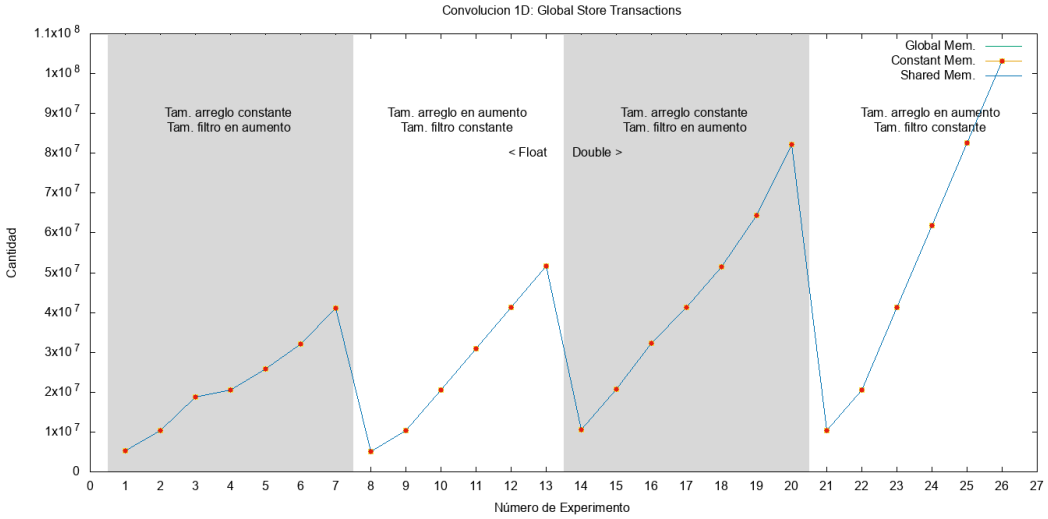


Figura 14: Cantidad de transacciones de almacenamiento hacia la memoria global.

En cuanto a la caché L2, se puede deducir que el kernel de memoria global posee más transacciones debido a que el filtro reside en ella. Sin embargo, para el caso de los kernels que usan la memoria compartida y constante este acceso no lo realizan más que una vez por lo que no afecta significativamente a la cantidad de transacciones sobre esta caché. Además, se puede observar que la cantidad de transacciones se vé afectada tanto por el filtro como por la cantidad de datos (recordar que el filtro también se recorre completo por cada dato de entrada). En la Figura 15 se puede observar gráficamente esta diferencia en la cantidad de transacciones.

En cuanto al hit-rate de la caché, La Figura 16 muestra un incremento del 10 % para el kernel que

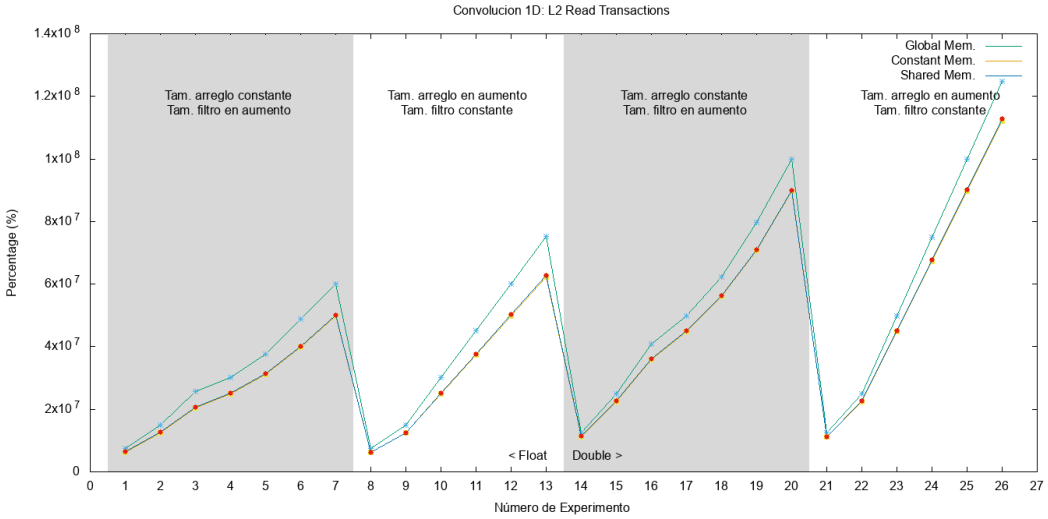


Figura 15: Cantidad de transacciones de realizadas a la caché L2.

utiliza la memoria global y el resto se mantiene similar y casi constante. Una explicación probable es que el uso del filtro en memoria global, y considerando que éste se accede varias veces entre un rango reducido de direcciones, hace que aumente la cantidad de transacciones que se benefician de la caché L2.

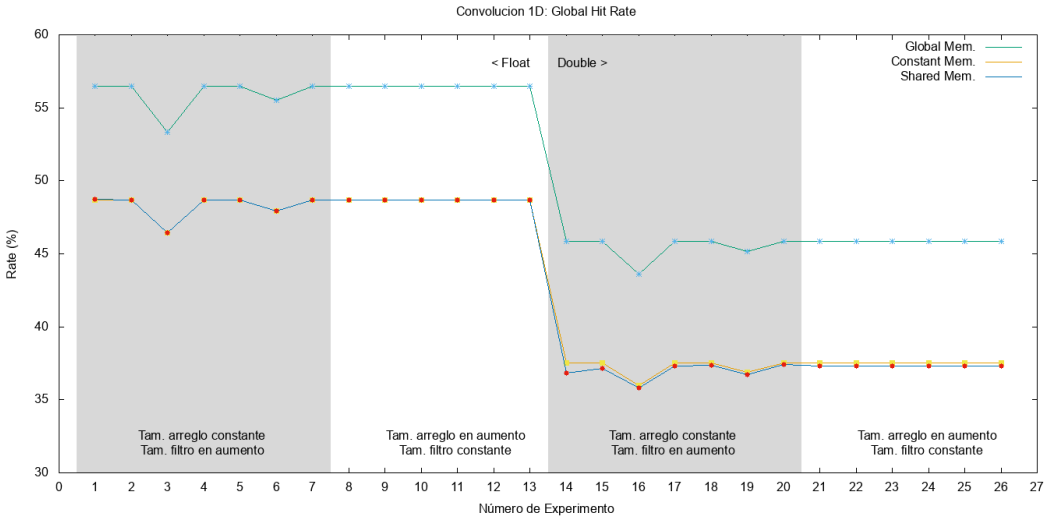


Figura 16: Porcentaje de aciertos a la caché global.

Sin embargo, el acceso a la caché L2 es de menor velocidad que el de la L1 y la memoria compartida. Además, la cantidad de transacciones realizadas en L2 por el kernel que usa la memoria global es mucho mayor al de los otros dos. Esto indica que los kernels que utilizan memoria compartida y constante realizan dichas transacciones sobre otra memoria que no es la global ni la caché L2.

# 7. Filtro de Imagen

Para esta sección se plantea la aplicación de dos filtros diferentes a una imagen. 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. Los filtros son matrices de $3 \times 3$ 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 píxel de la salida se muestra a continuación. 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 elementos que posee el filtro. Por ello, su implementación es similar a la anterior modificando la última línea del código por:

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

### 7.1. Resultados

La imagen original se muestra en la Figura 17. La salida de la aplicación del kernel del filtro promedio se muestran en la Figura 18 y la del filtro enfoque en la Figura 19. 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 20. 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 21 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.

## 8. Filtro Sobel

El filtro Sobel requiere de la aplicación de dos kernels de $3 \times 3$ 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.

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

Figura 17: Imagen original usada de entrada para los filtros.

Figura 18: Salida de la aplicación del kernel del filtro promedio.

Figura 19: Salida de la aplicación del kernel del filtro enfocado.

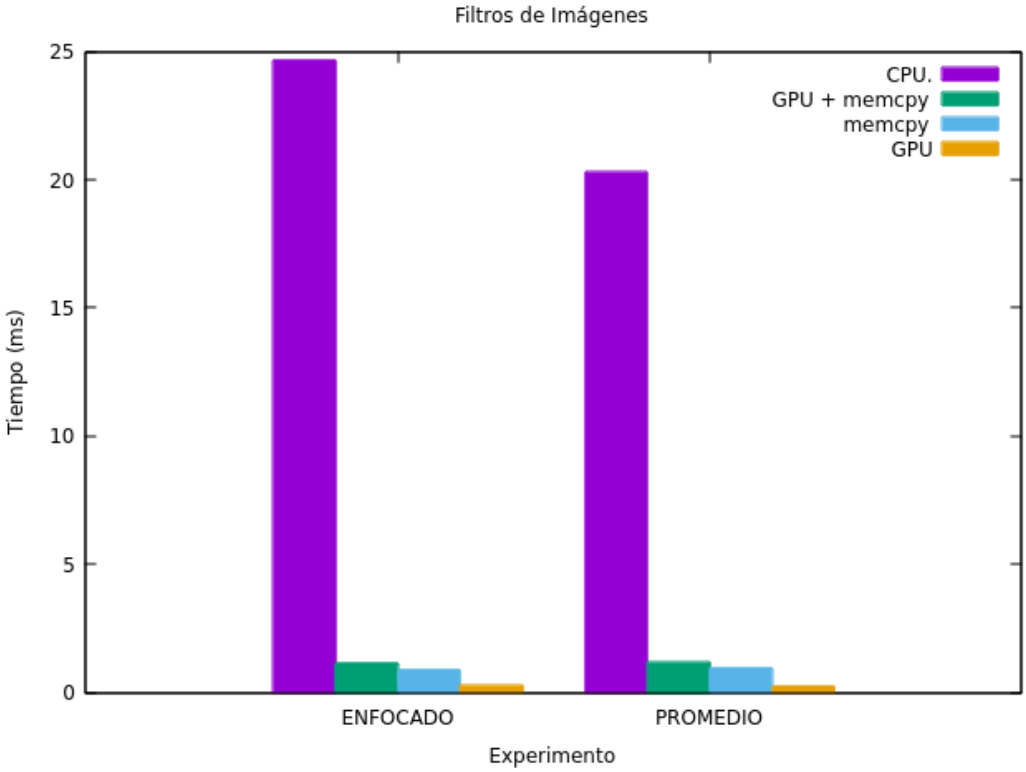


Figura 20: Resultados de aplicar el filtro enfocado y promedio en la imagen.

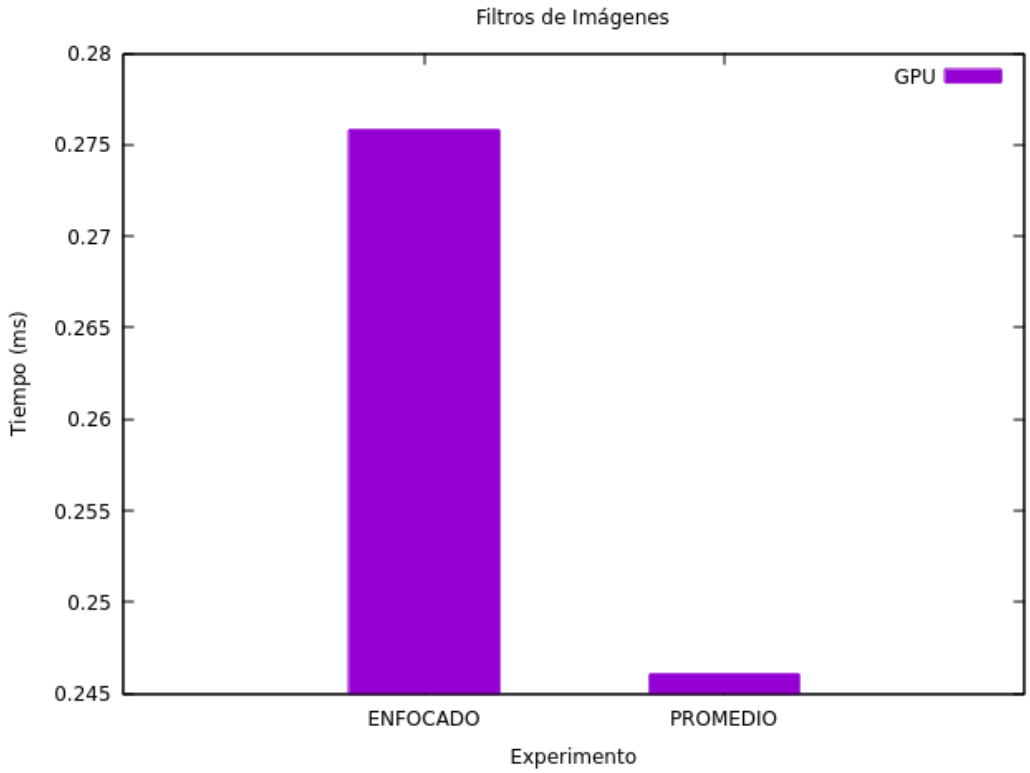


Figura 21: Tiempos de ejecución de los kernels de cada filtro (sólo el uso de GPU).

En el archivo `imagen.cu` se encuentran las funciones necesarias para aplicar el filtro Sobel de forma secuencial y en `sobel.cu` se implementan las paralelas. La función `kernel_aplicar_filtro` aplica uno de los filtros para obtener $G_x$ o $G_y$ dependiendo del valor booleano del parámetro de entrada `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 de las secciones anteriores. Para que sea utilizable por la GPU, se paraleliza teniendo en cuenta que se desea obtener el resultado de un píxel 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 (ramificaciones o *branch*) 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 sobre 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 un *branch* haciendo necesario varios ciclos para ejecutar cada una de las ramificaciones. Esto puede ser corroborado al ver las métricas resultantes del comando `nvprof`. En el Cuadro 6 muestra dicha métrica indicando un 100 % de eficiencia en el kernel `kernel_aplicar_filtro` y 0 eventos de ramificaciones divergentes. En cuanto al kernel `kernel_calcular_g`, se obtiene una pequeña divergencia debido a que hay threads que no realizan los cálculos dentro del condicional. Estos threads son los sobrantes generadon por la configuración de la grilla y bloques.

Cuadro 6: Métrica y evento correspondientes a la divergencia de los branch en los kernels ejecutados sobre el Grupo G.

| | Branch Efficiency (Métrica) | Divergent Branch (Evento) |
|---|---|---|
| kernel_calcular_g | 93.39 % | 5941 |
| kernel_aplicar_filtro | 100.00 % | 0 |

Luego de obtener $G_x$ y $G_y$, 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 indicada por cada píxel 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);
```

### 8.1. Resultados

Para su ejecución se utilizó la misma imagen de entrada propuesta en la sección anterior (Figura 17). La salida aplicando el filtro Sobel de forma paralela se muestra en la Figura 22. La imagen de la aplicación secuencial es la misma.

Figura 22: Resultado de aplicar el filtro Sobel de forma paralela.

En cuanto a los tiempos de ejecución, en la Figura 23 se presentan los de ambas 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.

## 9. Conclusiones y Trabajos Futuros

Este informe describe los resultados de los distintos ejercicios propuestos en el curso. Se aprecia que a medida que se avanza con los ejercicios la utilización del GPU cobra más relevancia.

En los primeros ejercicios, la cantidad de operaciones a realizar por dato es muy poca, de forma que no compensa el traslado de la información a la GPU con la utilización de la misma. Este balance entre transferencia de datos y uso de GPU, a veces, no logra ser suficiente como para considerar utilizar CUDA.

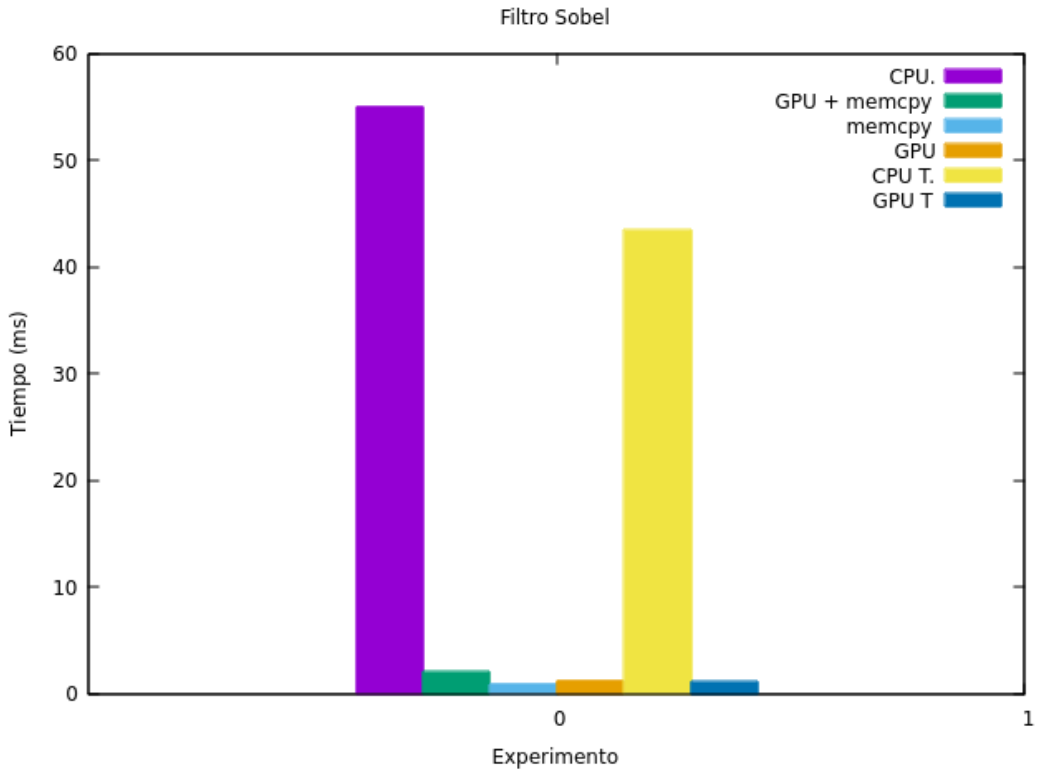


Figura 23: Tiempos de ejecución de la aplicación del filtro de Sobel en CPU y GPU.

Sin embargo, a partir de la utilización de la convolución, el traslado de memoria comienza a ser menos considerable comparado con el tiempo de procesamiento. La paralelización en el procesamiento de imágenes hace que la utilización de muchos threads resulte beneficiosa y reduzca tiempos de ejecución logrando una gran ventaja ante el CPU. Es por ello, que en estos últimos ejercicios se observa que CUDA logra un gran avance en cuanto a tiempos de ejecución: realizar un procesamiento que es paralelizable e independiente entre sí sobre grandes cantidades de datos.

Las resoluciones planteadas aquí fueron intentos iniciales para comprender el comportamiento de la GPU, ante variadas tareas. Aunque se pudo apreciar una idea preliminar de lo que sucede sobre cada experiencia, es necesario más pruebas para abarcar con más detalles y certeza las deducciones realizadas. Entonces, como posibles trabajos a considerar es el de realizar varias ejecuciones de un mismo experimento para aumentar la certeza, y evitar errores de medición, y sumar pruebas con parámetros y configuraciones más variados y con tamaños de datos más grandes.

## 10. Referencias

[1] NVIDIA Corporation. All rights reserved. cudaDeviceProp struct reference, 2019. `https://docs.nvidia.com/cuda/cuda-runtime-api/structcudaDeviceProp.html#structcudaDeviceProp` visitado el día 25 de abril del 2019 (captura en `http://archive.today/LrYUE`).

[2] NVIDIA Corporation. All rights reserved. Especificación de GeForce GTX Titan X en GeForce.com, 2019. `https://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan-x/specifications` visitado el día 25 de abril del 2019 (captura en `http://archive.today/4nMP0`).

[3] NVIDIA Corporation. All rights reserved. Especificación Técnica de GeForce GTX Titan X en nvidia.com, 2019. https://www.nvidia.com/en-us/geforce/products/10series/titan-x-pascal/#specs visitado el día 25 de Abril del 2019 (captura en http://archive.today/fhnh1).

[4] NVIDIA Corporation. All rights reserved. Especificación Técnica de Tesla V100 en nvidia.com, 2019. https://www.nvidia.com/en-us/data-center/tesla-v100/ visitado el día 25 de abril del 2019 (captura en http://archive.today/pUhkz).

[5] NVIDIA Corporation. All rights reserved. NVIDIA GeForce GTX 980 Whitepaper, 2014. URL original: https://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF disponible en archive.org: https://web.archive.org/web/20180329030403/https://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF.

[6] NVIDIA Corporation. All rights reserved. NVIDIA's Next Generation Cuda Compute Architecture: Fermi - Whitepaper, 2009. https://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf visitado el día 25 de abril del 2019.

# 11. Apéndice

## 11.1. Resultados de la Suma de Vectores

A continuación se presentan los resultados correspondientes a las pruebas de la suma de vectores, explicada en la sección 3. Primero se muestran los tiempos obtenidos con las placas GeForce Titan X (grupo G) y luego los de las placas Tesla (grupo T).

### 11.1.1. Grupo G

En el Cuadro 7 se muestra los resultados para la suma de vectores de las distintas pruebas utilizando la placa GeForce Titan X. Se muestra el tamaño y la cantidad de repeticiones, el tiempo de CPU, de GPU, el tiempo de copia desde el host al dispositivo, el tiempo de copia desde el dispositivo al host, el total del tiempo de copia y el total en que tarda la copia sumado el del GPU.

El gráfico correspondiente es el de la Figura 1 que se encuentra en la sección 3.

Cuadro 7: Tiempos en milisegundos para la suma de vectores en la GeForce Titan.

| 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 |
| 13 | 10000000 | 300 | 11163.1 | 138.332 | 46.195 | 14.898 | 61.093 | 199.425 |
| 14 | 10000000 | 300 | 11130.7 | 139.087 | 22.241 | 29.143 | 51.384 | 190.471 |
| 15 | 10000000 | 300 | 11003.4 | 138.589 | 46.746 | 15.091 | 61.837 | 200.426 |
| 16 | 1000 | 1 | 0.004 | 0.06592 | 0.002656 | 0.00144 | 0.004096 | 0.070016 |
| 17 | 10000 | 1 | 0.037 | 0.066272 | 0.011136 | 0.004288 | 0.015424 | 0.081696 |
| 18 | 100000 | 1 | 0.362 | 0.066144 | 0.13904 | 0.045504 | 0.184544 | 0.250688 |
| 19 | 1000000 | 1 | 3.737 | 0.144544 | 1.4843 | 1.6867 | 3.171 | 3.315544 |
| 20 | 10000000 | 100 | 6452.6 | 167.942 | 62.398 | 23.94 | 86.338 | 254.28 |
| 21 | 10000000 | 200 | 14213.6 | 283.143 | 9.8989 | 31.514 | 41.4129 | 324.5559 |

### 11.1.2. Grupo T

El Cuadro 8 muestra los resultados de las pruebas realizadas para las placas gráficas Tesla. Se muestra el tamaño del vector, la cantidad de repeticiones, el tiempo de CPU y de GPU. Los tiempos de copia no están disponibles puesto que el programa `nvprof` no retorna dichos datos. La Figura 24 muestra estos resultados utilizando un gráfico de líneas para su mejor interpretación.

Cuadro 8: Resultados de la suma de vectores en la placas gráficas Tesla.

| 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 |
| | | | | |
| 13 | 10000000 | 300 | 9584.69 | 46.4583 |
| 14 | 10000000 | 300 | 9622.15 | 46.6368 |
| 15 | 10000000 | 300 | 9628.86 | 46.5194 |
| | | | | |
| 16 | 1000 | 1 | 0.003 | 0.045312 |
| 17 | 10000 | 1 | 0.038 | 0.050464 |
| 18 | 100000 | 1 | 0.317 | 0.04848 |
| 19 | 1000000 | 1 | 3.365 | 0.06352 |
| | | | | |
| 20 | 10000000 | 100 | 3210.36 | 15.6748 |
| 21 | 10000000 | 200 | 6394.41 | 31.0478 |

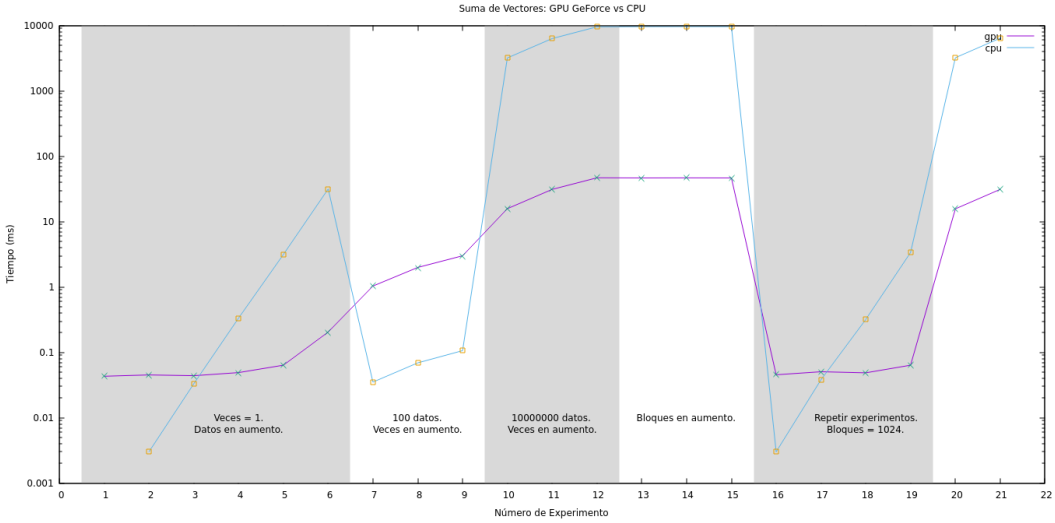


Figura 24: Resultados de la suma de vectores en las tarjetas gráficas Tesla.

## 11.2. Resultados de la Convolución 1D

### 11.2.1. Grupo G Float

El cuadro 9 muestra los resultados de las diferentes pruebas utilizando las placas GeForce Titan X y tipos de datos floats. Las columnas son las siguientes:

**num** Número de prueba.

**tam_data** Tamaño del dato.

**tam_filter** Tamaño del filtro.

**cpu** Tiempo de CPU.

**gpu** Tiempo de GPU sin considerar el de copia.

**HtoD** Tiempo de copia del host al dispositivo.

**DtoH** Tiempo de copia del dispositivo al host.

**memcpy** Suma de los tiempos de copia.

**total_gpu** Suma del tiempo de copia con el de GPU.

Cuadro 9: Tiempos de ejecución en milisegundos al aplicar la convolucion en tipos de datos 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 |

### 11.2.2. Grupo G Double

El Cuadro 10 muestra los resultados de las pruebas de ejecutar la convolución utilizando tipos de datos double en vez de floats. En este caso se utiliza las placas GeForce Titan X. Las columnas son las mismas que las descriptas en la sección 11.2.1 del Apéndice, agregándose las siguientes:

**vs_float** El número de problema que corresponde con el de la versión en punto flotante.

Cuadro 10: Tiempos de ejecución en milisegundos al aplicar la convolucion en tipos de datos 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 |