# Práctico 3 - GPGPU

Santiago Calvo
*Estudiante de Ingeniería en Computación*
5.055.578-2
Montevideo, Uruguay
santiago.calvo.vello@fing.edu.uy

***Resumen*—Informe de los resultados de la realización del práctico 3 de la materia Computación de Propósito General en Unidades de Procesamiento Gráfico.**

## I. Introducción

Con el objetivo de realizar un estudio de la performance de distintos algoritmos utilizando CUDA, se realizan diversas implementaciones con accesos a memorias distintos.

## II. Instrucciones de ejecución y compilación

La ejecución de los algoritmos se realizó sobre cluster.uy utilizando los archivos bash para cada ejercicio contenido en el entregable.

## III. Ejercicio 1

Para la generación del cuadrado se realizó la escritura utilizando el indice según bloque, su dimensión y el índice de thread. De esta manera se escribe en la dirección necesaria.

Luego para el llamado se dividió el espacio en bloques de $32 \times 32$ teniendo bloques de tamaño 1024. Para la grilla se divide en el $\frac{número\_de\_puntos}{32}$ en cada eje (x,y) cubriendo todas las celdas.

Luego se utilizó bloques y grillas de igual distribución para la generación de la matriz con función $seno$ de los puntos.

## IV. Ejercicio 2

Para la generación del cubo se realizó la escritura utilizando el indice según bloque, su dimensión y el índice de thread. De esta manera se escribe en la dirección necesaria.

Luego para el llamado se dividió el espacio en bloques de $8 \times 8 \times 8$ teniendo bloques de tamaño 512. Posteriormente se divide la grilla en $\frac{número\_de\_puntos}{8}$ en cada eje (x,y,z) cubriendo todas las celdas.

Luego se utilizó bloques y grillas de igual distribución para la generación del cubo con función $tangente$ de los puntos.

## V. Ejercicio 3

Este ejercicio trataba sobre el desarrollo de una función con trés órdenes de acceso distintas.

- 1. $S(y,x) = \sum_{\pi}^{z=-\pi} f(x,y,z)$
- 2. $S(z,x) = \sum_{\pi}^{y=-\pi} f(x,y,z)$
- 3. $S(z,y) = \sum_{\pi}^{x=-\pi} f(x,y,z)$

Estas presentan una diferencia importante en los tiempos de ejecución como se puede ver en la figura V. La 3er forma de acceso tiene un tiempo de ejecución de $125ms$ en contraposición a los $16ms$ de las otras funciones.

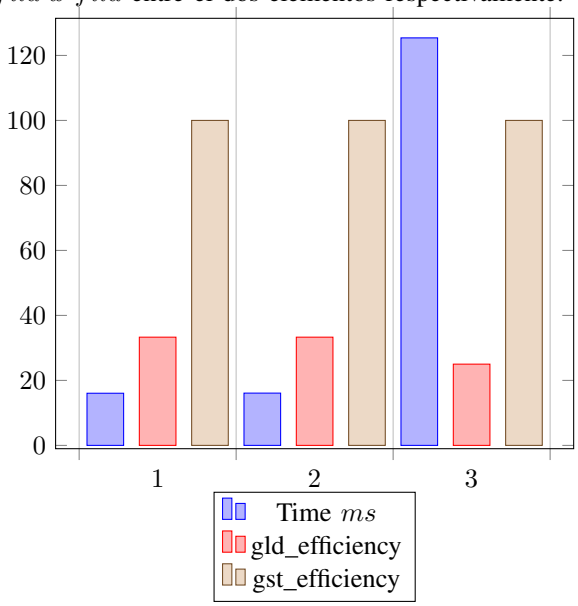
Estos se debe deber a los accesos globales a memoria. Como se puede ver, al utilizar $nvprof$ se obtienen los valores de eficiencia de estos accesos nombrados:

- gld_efficiency: Relación de lecturas globales a memoria con lecturas requeridas
- gst_efficiency: Relación de escrituras a memoria con escritras a memoria requeridas

En la figura V se puede ver como las lecturas tienen una particularmente baja en el tercer método lo cual se manifiesta como menor tiempo de ejecución dado que los datos deben ser obtenidos de memoria global cada vez.

Al acceder a direcciones contiguas se aprovecha el principio de localidad espacial permitiendo aprovechar las lecturas, necesitando luego menos accesos globales. En el caso del tercer algorítmo no se está aprovechando a su máximo los accesos dado que las direcciones accedidas están .ªpacidas"por la manera en que se organiza el cubo del cual se leen los datos.

Los accesos por el eje Z e Y no permiten aprovechar el uso de la memoria dado su separación. Tienen distancia de $fila$ y $fila\ x\ fila$ entre el dos elementos respectivamente.



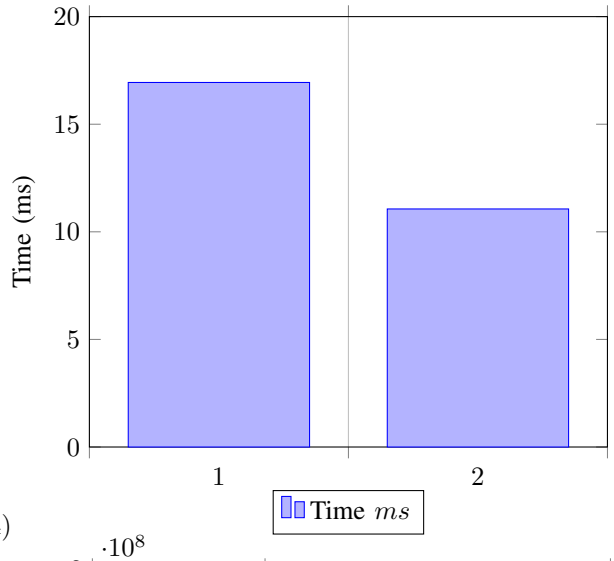

## VI. Ejercicio 4

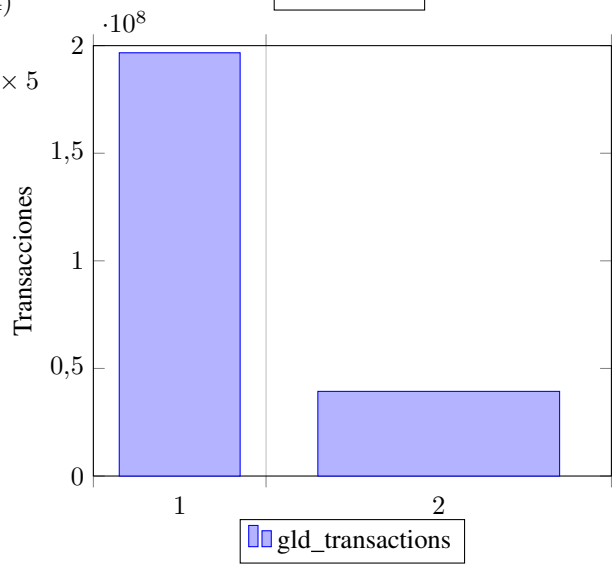
En este ejercicio se realizó un algoritmo de tipo stencil. Este algoritmo realiza lecturas de múltiples elementos dentro de un mismo thread para realizar un cálculo final. Por esto se verán los accesos a memoria realizados por este.

Utilizando $nvprof$ se visualizó la cantidad de transacciones que se realizaron a memoria global, visualizables en la figura VI como la barra 1.

Esté número es coherente con la cantidad de transacciónes esperadas aunque levemente menor.

$$
\begin{aligned}
tamaño\_fila = tamaño\_columna &= \frac{2 \times \pi}{0{,}001} = 6280 \\
tamaño\_matriz &= tamaño\_fila \times tamaño\_columna \\
&= 39438400 \\
transacciones\_esquinas &= 3 \times 4 = 12 \\
transacciones\_edges &= 2 \times 4 \times \\
&\quad (tamaño\_fila + tamaño\_columna - 4) \\
&= 100448 \\
transacciones\_centro &= (tamaño\_f - 2) \times (tamaño\_c - 2) \times 5 \\
&= 197066420 \\
transacciones &= t\_esquinas + t\_edges + t\_centro \\
&= 12 + 100448 + 197066420 \\
&= 197166880
\end{aligned}
$$




Sin embargo esta cantidad de accesos podriamos aprovechar el uso de warps aprovechando una mayor cantidad de datos por transacción si sincronizamos los warps a la hora de leer de la siguiente manera:

```
__shared__ double matrix_point[32][32];
unsigned int threadIdx_x = threadIdx.x + 1;
unsigned int threadIdx_y = threadIdx.y + 1;

matrix_point[threadIdx_y -1][threadIdx_x -1] =
  matrix[i * num_points + j];

__syncwarp();
```

De esta forma guardamos una porción importante de la lectura en memoria Shared de mayor velocidad.

Al comparar ambas implementaciones (encontradas en ej4.cu y ej4_improved.cu), vemos una diferencia de transacciones y de acceso importante pues se realiza solo un acceso a memoria en vez de 5 por thread (reduciendo 5 veces las transacciones necesarias).

Nota, este ejercicio fue corrido en una maquina personal dado falta de acceso a ClusterUY. Las especificaciónes son:

- CPU: Intel i7 3.6 GHz
- GPU: GTX 1060 3GB GDDR5
- RAM: 16 GB 2133 MHz

En las siguientes figuras se tiene las dos implementaciones, 1 sin optimizaciones y 2 con optimizaciones.