Procesamiento de imgenes con OpenCV y CUDA

William Salazar

Pereira, Colombia

Abstract

El fin de este documento es comprobar la diferencia de rendimiento que implica la utilizacin de memoria compartida y memoria constante de un algoritmo en CUDA, por qu sucede este aumento o disminucin en la velocidad de procesamiento y que ventajas o desventajas nos da usar memoria compartida y memoria constante en procesamiento paralelo, esto para el procesamiento de imgenes con OPENCV y su funcin para aplicar el filtro Sobel a imagenes. El lenguaje de programacin usado para realizar estas pruebas fue c++/cuda.

Keywords: `CUDA, OpenCV, GPU, Imagen`

1. Introduccin

Table 1: Imagenes utilizadas en las pruebas

Imagen	Dimensiones
Crash.jpg	640*360
Bikes.jpg	640*480
moon.jpg	2600*2910
GrayRock.jpg	3888*2592
Landscape.jpg	5184*3456

A continuacin se realizarn una prueba de tiempos a diferentes implementaciones del filtro Sobel utilizando procesamiento secuencial y paralelo, estas se realizarn utilizando

[☆] Fully documented templates are available in the elsarticle package on CTAN.
[1] Since 2017.

memoria constante, compartida y ambas. Las dimensiones de las imagenes utilizadas en todas la pruebas estn en la **Tabla 1**.

2. Operador sobel en OpenCV funcion secuencial

El filtro sobel es usado en el procesamiento de imagenes principalmente para la deteccin de bordes, OpenCV cuenta con funciones predeterminadas para realizar esta operacin, la implementacin de esta es la siguiente:

```
cvtColor(image, gray_image_opencv, CV_BGR2GRAY);
        Sobel(gray_image_opencv,grad_x, CV_8UC1, 1, 0,3, 1, 0, BORDER_DEFAULT);
        convertScaleAbs(grad_x, abs_grad_x);
```

3. Operador sobel realizado en CUDA

El operador utiliza dos kernels de $3x3$ elementos para aplicar convolucin a la imagen original para calcular aproximaciones a las derivadas, un kernel para los cambios horizontales y otro para las verticales. En la siguiente implementacin slo se utilizar un Kernel en el cual se hagan cambios verticales.

```
__global__ void sobelCuda(unsigned char *imageInput,
int width, int height, unsigned int maskWidth,
char *M, unsigned char *imageOutput){
    unsigned int row;
    unsigned int col;
    row = blockIdx.y*blockDim.y+threadIdx.y;
    col = blockIdx.x*blockDim.x+threadIdx.x;

    int valor = 0;

    int N_row = row - (maskWidth/2);
    int N_col = col - (maskWidth/2);

    for(int i = 0; i < maskWidth; i++){
        for(int j = 0; j < maskWidth; j++ ){
            if((N_col + j >=0 && N_col + j < width)
            && (N_row + i >=0 && N_row + i < height))
```

```
            {
                valor += imageInput[(N_row + i)*width
                +(N_col + j)] * M[i*maskWidth+j];
            }
        }
    }
    imageOutput[row*width+col] = setNumber(valor);
}
```

La funcin ”setNumber” al final, garantiza que los nmeros estn en el rango de 0 a 255.

3.1. Memoria constante

Para que el anterior kernel pueda utilizar memoria constante, se debe saber previamente que variable u estructura es utilizada frecuentemente sin sufrir cambios, para el caso anterior sera la matriz M.

```
__constant__ float constMask[MASK_WIDTH*MASK_WIDTH];
//some code...
cudaMemcpyToSymbol(constMask, h_Mask, maskWidth*maskWidth*sizeof(float));
```

Aunque muestra matriz sea declarada como constante se debe, no se inicializa desde el host, para ello debemos crear un matriz en el host y transferir sus datos hacia nuestra matriz constante utilizando la funcin cudaMemcpyToSymbol, una vez hecho esto no hace falta pasar muestra matriz como parmetro en nuestro kernel para poder utilizarla.

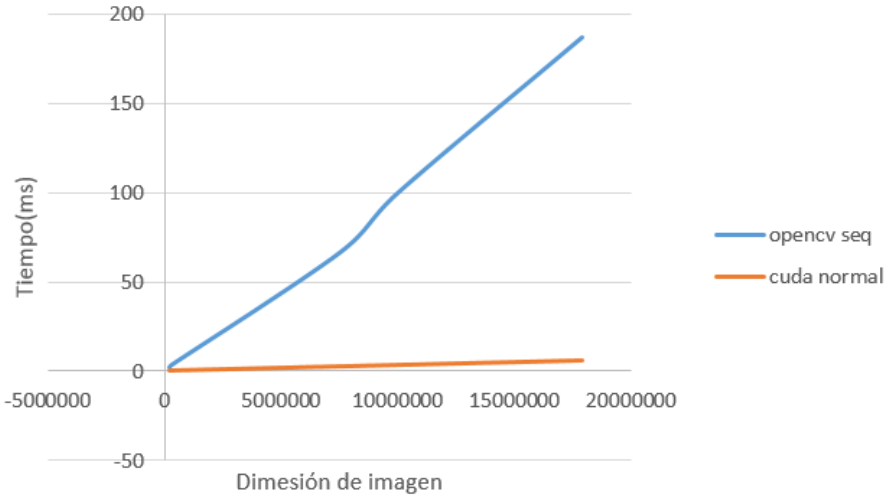
3.2. Memoria compartida

En este implementacin a diferencia del anterior se manda una parte de las matrices a memoria compartida para hacer el acceso a los datos ms rpido que con memoria global.

4. Implementacin en CUDA vs OpenCV secuencial

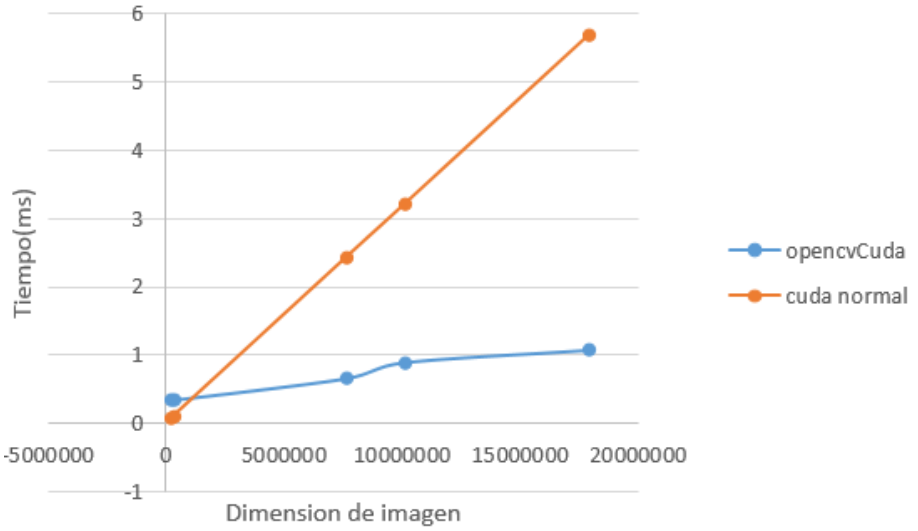
En la siguiente figura se observa que el rendimiento de la implementacin paralela del filtro sobel respecto a su equivalente secuencial de OpenCV es muy superior.

Figure 1: secuencial vs paralelo



Esto se debe principalmente a que la implementacin en CUDA hace que cada hilo en la GPU realicen estas operaciones de manera simultanea a diferencia de la implementacin secuencial de Opencv en la que hace una operacin despus de otra. En la siguiente imagen se muestra una comparativa entre la implementacin del filtro sobel de OpenCV para GPU y nuestra implementacin de sobel en CUDA.

Figure 2: Imagenes utilizadas en las pruebas

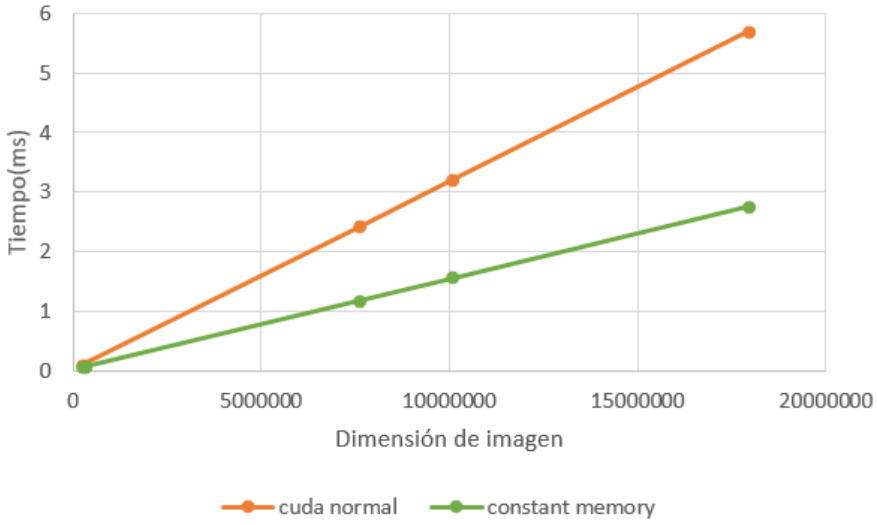


Como se puede ver sobel en OpenCV CUDA tiene un mejor rendimiento que muestra implementacin.

5. Implementacin en CUDA con memoria constante vs CUDA simple

Para mejorar el rendimiento de la anterior implementacin de ha decidido utilizar la memoria constante.

Figure 3: Cuda normal vs Cuda con memoria constante

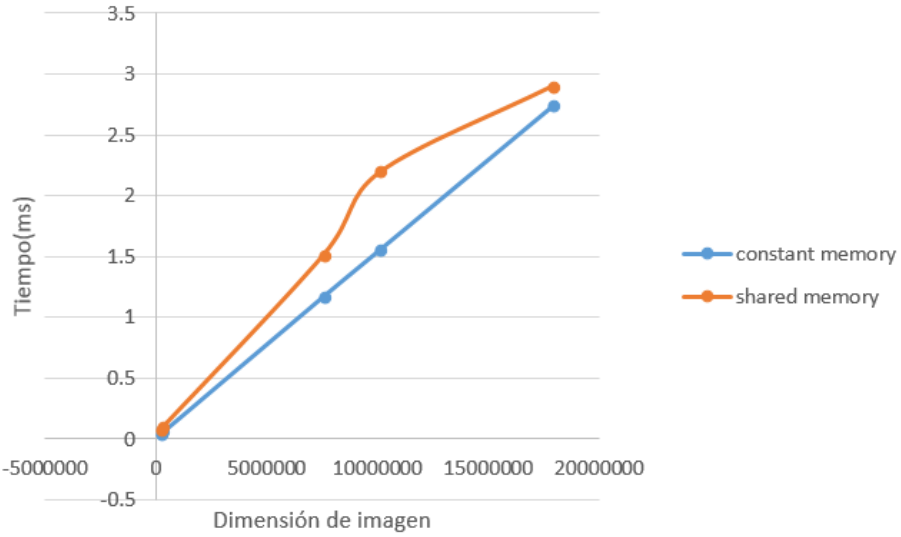


Como se puede apreciar el rendimiento con memoria constante aumenta en gran cantidad respecto a su implementacin inicial, pero esto tiene una limitante y es que en la mayora de dispotivos se encuenta una memoria constante de 64 KB, esta memoria constante se almacena en la memoria del dispositivo y debido a esto la lectura de datos en memoria constante es muy rpida gracias a esto el rendimiento del algoritmo aumenta.

6. Implementacin en CUDA con memoria constante vs CUDA con memoria compartida

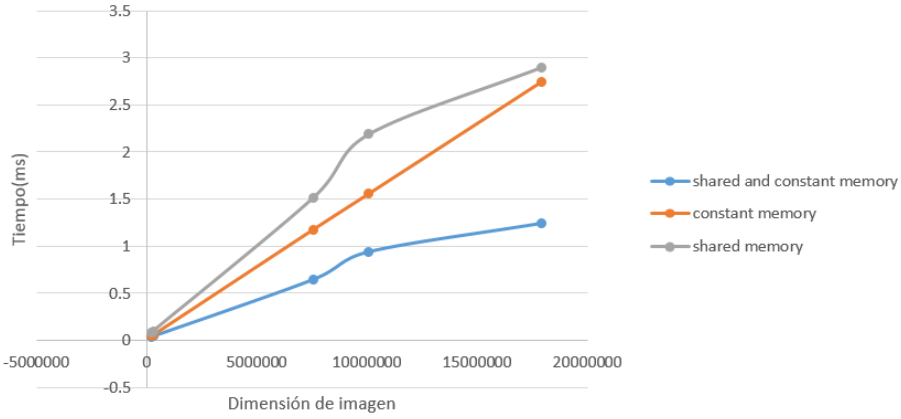
Debido a que est en la GPU, la memoria compartida es mucho ms rpida que la memoria local y global. De hecho, la latencia de memoria compartida es aproximadamente 100 veces menor que la latencia de memoria global no almacenada en cach. La memoria compartida se asigna por bloque de subprocesos, por lo que todos los subprocesos en el bloque tienen acceso a la misma memoria compartida. Los hilos pueden acceder a los datos de la memoria compartida cargada desde la memoria global por otros hilos dentro del mismo bloque de hilos.

Figure 4: memoria compartida vs memoria constante



Como se puede apreciar no hay mucha diferencia entre el uso de memoria constante y memoria compartida en CUDA por lo menos para las imagenes usadas en estas pruebas, en los puntos finales de la grfica se puede observar que la implementacin con memoria compartida tiende a ser logaritmica mientras que la de memoria constante sigue de forma lineal, sin embargo hacen falta ms pruebas para asegurar esta afirmacin.

Figure 5: grfica de implementacin con memoria compartida y constante



La utilizacin de memoria constante y compartida de forma conjunta aumenta el

redimiento de forma considerable respecto a sus versiones por separado.

7. Implementacin en CUDA con memoria compartida y memoria constante vs OpenCV con CUDA

Figure 6: Imagenes utilizadas en las pruebas

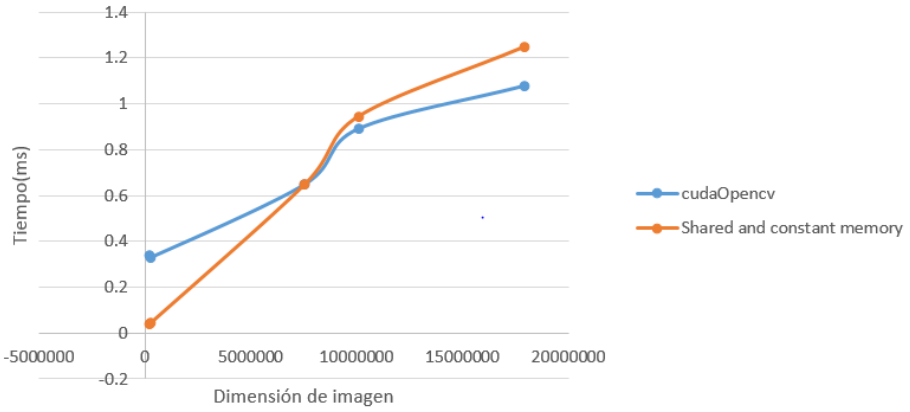
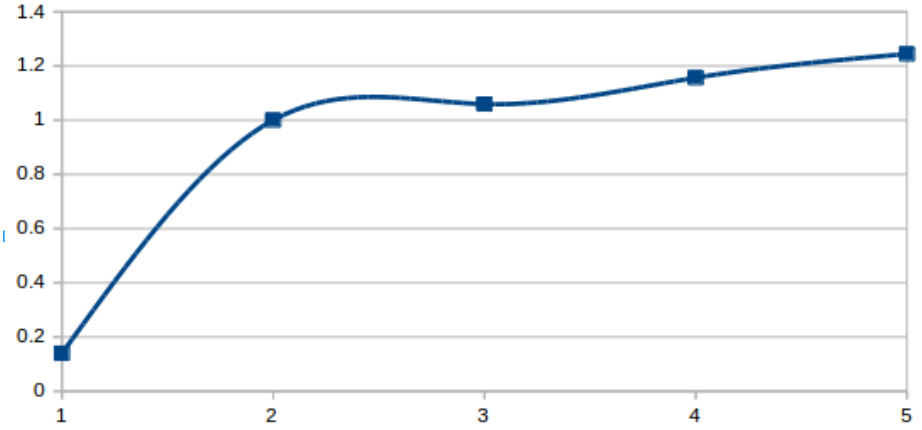


Figure 7: Aceleracin del filtro sobel en OpenCV:CUDA respecto a nuestra implementacin con memoria compartida y memoria constante



8. Conclusin

- Como se pudo apreciar, el rendimiento general de la GPU para el procesamiento masivo de datos es muy superior al de la CPU.

- El utilizar memoria compartida en GPU aumenta considerablemente el rendimiento, pero se debe tener en cuenta el tamao de la memoria compartida en la GPU.
- Al igual que con el uso de memoria compartida, al usar memoria constante se debe tener encuenta que se tiene un almacenamiento limitado, en general de 64 KB.
- Aunque la memoria compartida tiene 100 veces menos latencia que la memoria constante esto no determina si una aplicacin ser mucho ms rpida con memoria compartida que con memoria constante para todos los casos.
- La memoria compartida slo es accesible por los hilos de un mismo bloque.