UNIVERSIDAD DE GRANADA

Práctica Final: Operador Sobel

Arquitectura y Computación de altas prestaciones

Luis González Romero

19 de mayo de 2019

Índice

Índice de figuras

1. Enunciado

- Implementar una versión del problema trabajado en la *P3* con *CUDA* en la GPU.
- Representar gráficamente el tiempo de ejecución total y la ganancia en velocidad en función del número de PCs, contrastando CPU y GPU.

2. Algoritmo

Detector de bordes Sobel.

$$G_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \quad G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Figura 1: Recuerdo del cálculo de los gradientes para un píxel (x,y)

El resultado muestra cómo de abruptamente o suavemente cambia una imagen en cada punto, interpretando si representa un borde y la orientación a la que tiende el borde.

Figura 2: Imagen de entrada

Figura 3: Imagen de salida

3. Implementación secuencial

Solución expuesta en la *P3*, haciendo uso de la biblioteca *OpenCV* para trabajar con imágenes *Mat*.

```
for(int y = 0; y < src.rows; y++){
  for(int x = 0; x < src.cols; x++){
    gx = x_gradient(src, x, y);
    gy = y_gradient(src, x, y);
    sum = abs(gx) + abs(gy);
    sum = sum > 255 ? 255:sum;
    sum = sum < 0 ? 0 : sum;
    dst.at<uchar>(y,x) = sum;
  }
}
```

Figura 4: Tratado secuencial de la imagen

En los métodos para calcular los gradientes, se tienen en cuenta casos especiales para evitar fallos de acceso a memoria.

En esta versión secuencial, lo único que hace el programa es abrir, tratar la imagen y guardarla con el prefijo "*sobel-*".

4. Implementación con CUDA

Partiendo del código secuencial, la función a ejecutar con el kernel se basa en el cálculo de los gradientes de x e y.

```
__global__ void sobel_gpu(const uchar* src, uchar* gpu, const unsigned int width, const unsigned int height)
{
  int x = threadIdx.x + blockIdx.x * blockDim.x;
  int y = threadIdx.y + blockIdx.y * blockDim.y;
  int gx, gy, sum;
  if( x > 0 && y > 0 && x < width-1 && y < height-1) {
    gx = (-1* src[(y-1)*width + (x-1)]) + (-2*src[y*width+(x-1)]) + (-1*src[(y+1)*width+(x-1)]) +
         (    src[(y-1)*width + (x+1)]) + ( 2*src[y*width+(x+1)]) + (   src[(y+1)*width+(x+1)]);
    gy = (    src[(y-1)*width + (x-1)]) + ( 2*src[(y-1)*width+x]) + (   src[(y-1)*width+(x+1)]) +
         (-1* src[(y+1)*width + (x-1)]) + (-2*src[(y+1)*width+x]) + (-1*src[(y+1)*width+(x+1)]);
    sum = abs(gx) + abs(gy);
    sum = sum > 255 ? 255:sum;
    sum = sum < 0 ? 0 : sum;
    gpu[y*width + x] = sum;
  }
}
```

Figura 5: Función a ejecutar por el kernel

Para mi GPU en concreto, he usado la macro *THREADS_DIM* con valor 32.0 para así usar

las 1024 threads que soporta la GPU visto en las properties mostradas en la *P4*, gestionando así el *grid* de *blocks* haciendo uso de estas junto al *width* y *height* de la imagen a tratar.

```
/** set up the dim3's for the gpu to use as arguments (threads per block & num of blocks per grid)**/
dim3 threadsPerBlock(THREADS_DIM, THREADS_DIM, 1);  // THREADS_DIM x THREADS_DIM x 1  -- 32 x 32 = 1024
/** ceil(x) -> the smallest integer greater than or equal to x **/
dim3 numBlocks(ceil(width/THREADS_DIM), ceil(height/THREADS_DIM), 1); // width/GRIDVAL x height/GRIDVAL x 1
```

Figura 6: *threadsPerBlock* y *numBlocks* para el kernel

Para usar la imagen con *cudaMalloc* y *cudaMemcpy* paso la imagen con *uchar*, haciendo uso del puntero a los datos de la imágen *Mat*(src.data).

```
uchar *gpu_orig, *gpu_sobel;
cudaMalloc((void**)&gpu_orig, width*height*sizeof(uchar));
cudaMalloc((void**)&gpu_sobel, width*height*sizeof(uchar));

cudaMemcpy(gpu_orig, src.data, width*height*sizeof(uchar), cudaMemcpyHostToDevice);
```

Figura 7: *cudaMalloc*

Siendo *cpu_orig* la que tendrá los datos de la imagen a tratar y *gpu_sobel* donde se tendrá la imagen final.

```
sobel_gpu<<<numBlocks, threadsPerBlock>>>(gpu_orig, gpu_sobel, width, height);
```

Figura 8: Kernel Call

5. Tiempo de ejecución y ganancia

Para obtener los tiempos de ejecución se han hecho distintas mediciones y se ha obtenido: la media de cada caso de las configuraciones y la dispersión estándar de estas.

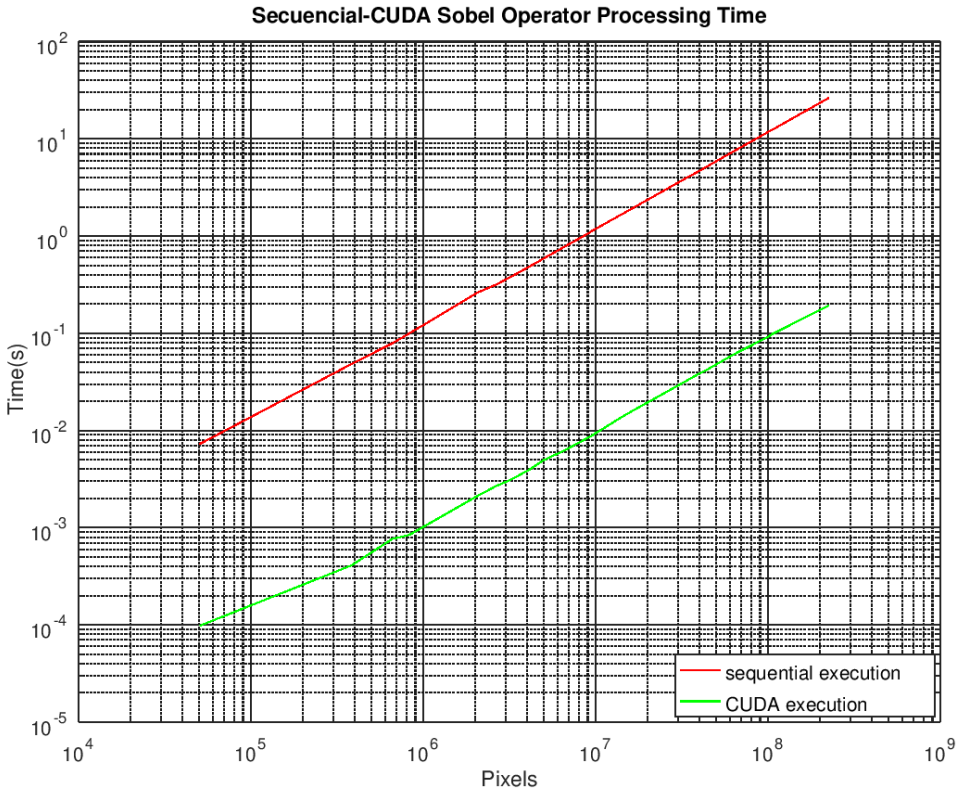

Figura 9: Tiempos de ejecución

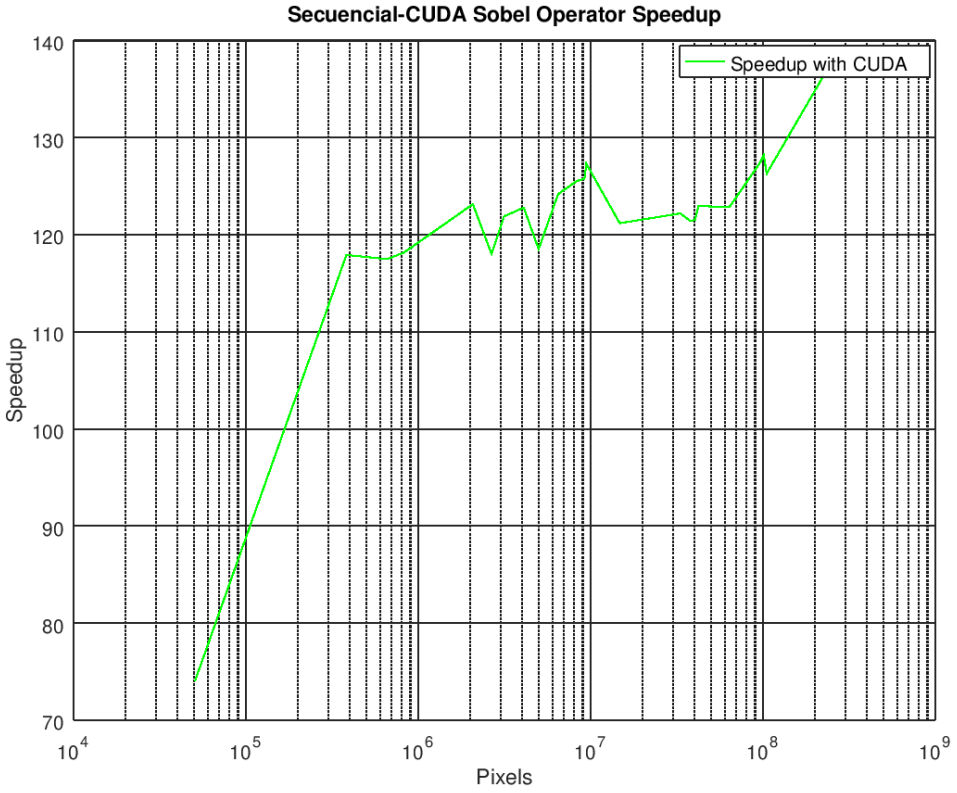

Figura 10: Ganancia con *CUDA*

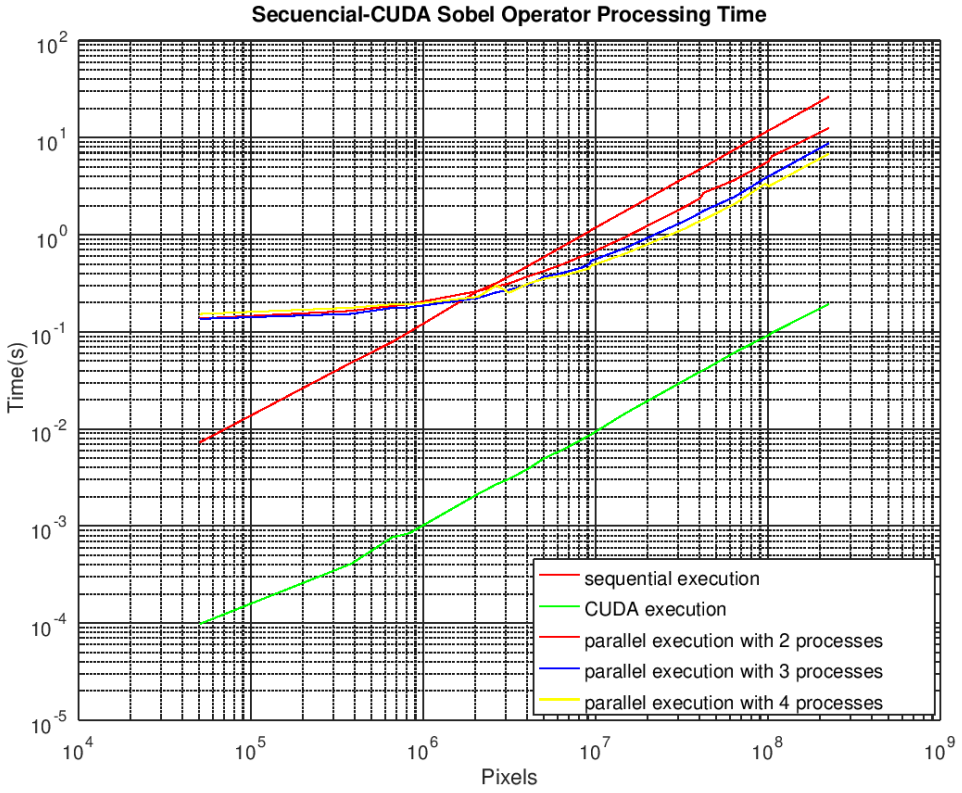

Figura 11: Comparación tiempos de ejecución seq-par-cuda

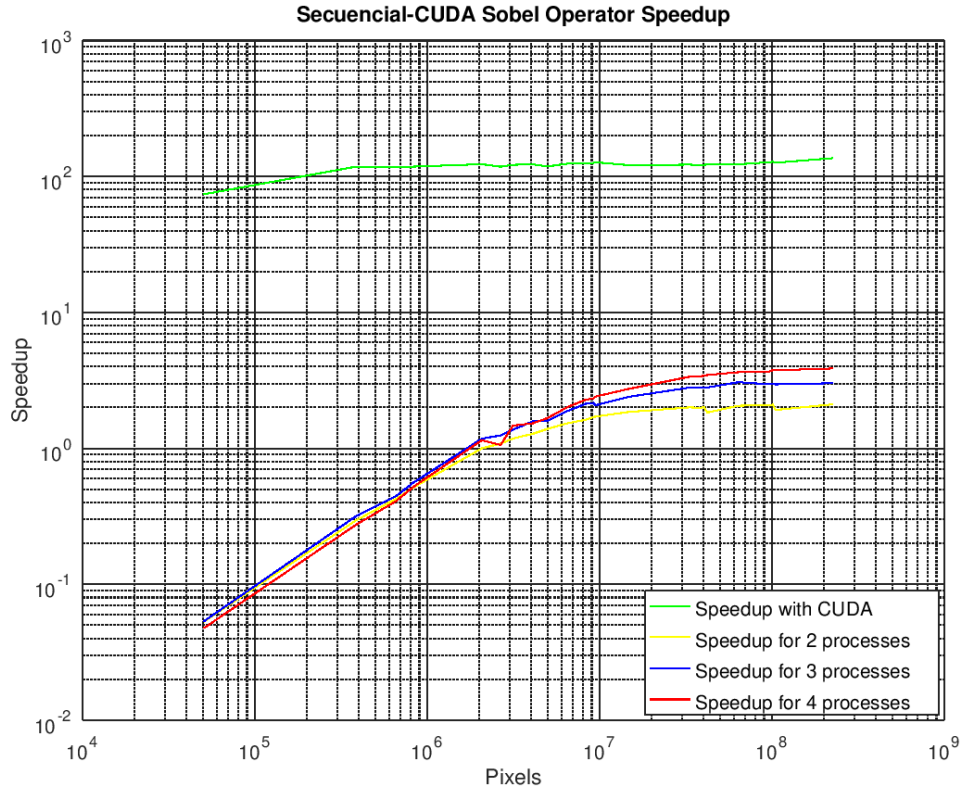

Figura 12: Comparación de ganancia seq-par-cuda

La media y desviación de los tiempos obtenidos están en los ficheros *calculos-estadisticos-seq.txt* y *calculos-estadisticos-cuda.txt*.

6. Conclusión

La conclusión es similar a la de la *práctica 4*, pero aún más grotesco. En vez de trabajar con vectores, se ha trabajado con imágenes de más de 10^8 píxeles, consiguiéndose más de 130 de ganancia. El tiempo de cómputo es ridículo comparado con el secuencial, incluso con el paralela obtenida con *OpenMPI*.