A Comparison of the Sobel Filter in C, OpenCV and CUDA

Daniele Gader
Department of Computer Science, University of Pisa
Programming Tools for Parallel and Distributed Systems course

2nd Semester of Academic Year 2018-2019

1 Algorithm Introduction

The Sobel Filter, also known as Sobel Operator, is an algorithm used in computer vision and image processing for identifying regions characterised by sharp changes in intensity (i.e., edges). It is a discrete differentiation operator, which computes an approximation of the gradient of the image intensity function by convolving the input image with a 3x3 kernel in the horizontal and vertical directions, respectively. As outlined in [2], such operator lies at the basis of several algorithms for pedestrian detection in autonomous driving systems.

Figure 1a shows an input image for the Sobel filter. Figure 1b shows the application of the Sobel filter algorithm outlined in Section 1.1 to the input image.

(a) Input RGB image I of the Sobel Filter (b) Application of the Sobel Filter onto I

Figure 1: Input RGB image and resulting image, following Sobel Filter's application

1.1 Algorithm Description

Given an input RGB image I, the Sobel Filter algorithm proceeds according to the following phases:

1. Convert I to a gray-scale image B according to Formula 1. Such formula is applied to every pixel p of I to produce a corresponding gray-scale pixel b of B. p_r, p_g, p_b represent the R,G,B intensity of pixel p, respectively, and each one lies in the range [0,255]. Figure 2 shows the resulting gray-scale image B.

$$b = 0.3 * p_r + 0.59 * p_g + 0.11 * p_b \tag{1}$$

2. Compute G_x, an approximation of the horizontal gradient of B, by convolving B with a 3x3 kernel according to Formula 2. $*$ denotes the 2-dimensional convolution operation. Figure 3a shows the resulting G_x.

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * B \tag{2}$$

Figure 2: Gray-scale image B resulting from phase (1) of Section 1.1

3. Compute G_y, an approximation of the vertical gradient of B, by convolving B with a 3x3 kernel according to Formula 3. Figure 3b shows the resulting G_y.

$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} * B \tag{3}$$

(a) Horizontal gradient G_x resulting from phase (2) of Section 1.1. G_x increases in the "right" direction.

(b) Vertical gradient G_y resulting from phase (3) of Section 1.1. G_y increases in the "down" direction.

Figure 3: Resulting G_x and G_y, following the respective application of Formulas 2 and 3 onto B.

4. Compute G, the final gradient magnitude, by combining the horizontal gradient G_x and the vertical gradient G_y according to Formula 4. Figure 4 shows the resulting G.

$$G = \sqrt{G_x^2 + G_y^2} \tag{4}$$

2 Implementation

Initially, I propose to implement a sequential version of the Sobel Filter in OpenCV and in the native C programming language. Afterwards, I aim to implement a parallel GPU version in CUDA. These versions will be tested on the hardware outlined in Table 1.

The GPU (CUDA) version of the program is just tested on the *Laptop* and *Bruciato* hardware because the *Gewurztraminer* and *Raspberry PI 3 B+* machines do not feature a CUDA-capable device.

The OpenCV version of the program is just tested on the *Laptop* and *Raspberry PI 3 B+* hardware because the report writer does not have super-user permissions on the *Gewurztraminer* and *Bruciato* machines to install OpenCV.

Figure 4: Gradient magnitude G resulting from phase (4) of Section 1.1

Nickname	CPU	GPU	RAM	Hard Disk	Native C	GPU	OpenCV
Laptop	Intel Core i7 8565U@1.8 GHz	Geforce MX 150 390.116	16 GBs	SSD - 512 GBs	x	x	x
Bruciato	Intel Core i7 7700@3.6 GHz	Geforce GTX 1080 384.81	63 GBs	Magnetic - 2 TBs	x	x	
Gewurztraminer	Intel Core i7 4770k @3.5 GHz	-	31 GBs	SSD - 120 GBs	x		
Raspberry PI 3 B+	ARM v8@1.4 GHz	Broadcom Videocore IV	1 GB	MicroSD - 32 GBs	x		x

Table 1: Technical characteristics of the hardware used for benchmarking the native C, GPU and OpenCV versions of the implemented Sobel Filter

2.1 Sequential Version - Native C

The sequential version of the Sobel Filter was implemented in native C programming language [1]. It operates according to the following steps:

1. **Step. Convert input JPG/PNG/GIF image to RGB format**: By leveraging the 'convert' utility of the 'imagemagick' command-line package, a format- and encoding-independent representation (I) of the input image is produced in RGB format. In such format, every pixel p of I has an R,G,B component: p_r, p_g, p_b.

2. **Step. Convert RGB image to Gray-Scale**: By applying Formula 1, we convert every single pixel p of the RGB image I in a sequential manner to product pixel b of the gray-scale image B.

3. **Step. Compute Horizontal Gradient**: By applying Formula 2, we compute the horizontal gradient G_x over all pixels b of B by taking a 3x3 region around b and convolving it with the 3x3 kernel of Formula 2.

4. **Step. Compute Vertical Gradient**: By applying Formula 3, we compute the vertical gradient G_y over all pixels b of B in a sequential manner by taking a 3x3 region around b and convolving it with the 3x3 kernel of Formula 3.

5. **Step. Combine Vertical Gradient and Horizontal Gradient**: We combine the horizontal and vertical gradient in a sequential manner over all the pixels g_x of G_x and g_y of G_y to yield pixel g of G according to Formula 5.
$$g = \sqrt{g_x^2 + g_y^2} \tag{5}$$

6. **Step. Convert gray file to PNG**: The resulting '.gray' file from step (5) is converted to a PNG image by using the 'convert' utility of the 'imagemagick' command-line package.

[1] The native C implementation is available at https://github.com/DanyEle/Sobel_Filter/tree/master/Native_Sobel

2.2 GPU Version - CUDA

CUDA (Compute Unified Device Architecture) is a C/C++ parallel computing platform that easens the process of writing GPU code. The parallel GPU version of the Sobel Filter was implemented leveraging the CUDA framework [2].

CUDA parallelisation: The CUDA implementation leverages the fact that the computation highlighted in steps (2), (3), (4), (5) of Section 2.1 occurs in parallel over every single pixel p of the input image I and B. Therefore, a **Map** parallelisation is applied over every single pixel b of the gray-scale image B and every single pixel p of the input image I.

Such parallel abstraction is implemented concretely in three CUDA kernels (phase (2) and (3) share the same code) having the following common characteristics:

```
kernel_name <<< width, height >>> (argument_1, argument_2, ..., argument_n)
```

The image `width` represents the amount of blocks in the CUDA grid, whereas the image `height` is the amount of threads per CUDA block. Such parallelisation is shown in Figure 2.2. In this manner, every single pixel is allocated one single thread, hence:

```
Amount of Pixels = width * height

width = blocks; height = threads

Amount of Threads = blocks * threads
```

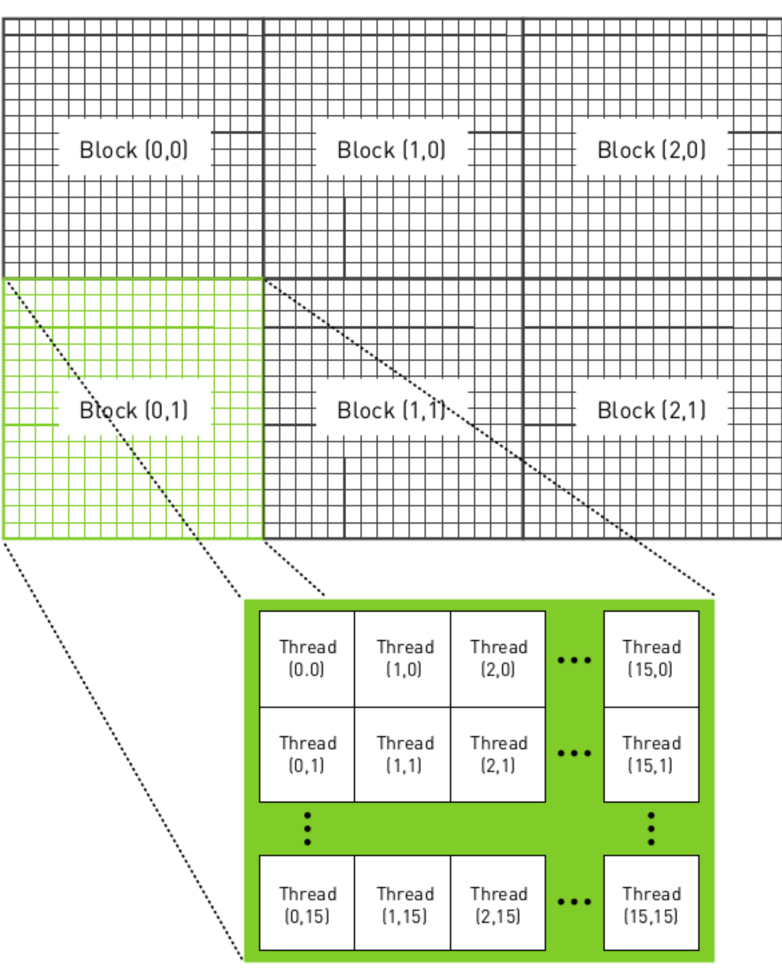


Figure 5: A 2D hierarchy of blocks and threads used to process an image using one thread per pixel [3]. Presently, it shows 6 blocks and 16x16 threads per block. In my CUDA parallelisation, the amount of threads and blocks is equal, instead.

The CUDA implementation proceeds according to the following steps:

[2]The CUDA implementation is available at `https://github.com/DanyEle/Sobel_Filter/tree/master/CUDA_Sobel`

1. **Step. Convert input JPG/PNG/GIF image to RGB format**: The 'convert' utility proceeds sequentially in the initial phase of the computation.
2. **Step. Convert RGB image to Gray-Scale**: We notice that Formula 1 can be applied in parallel over the p_r, p_g, p_b components of the p pixel of the RGB input image I. Therefore, a **Map** Parallelisation is applied over all pixels of the input image according to the schema in Figure 2.2 to produce pixel b of the gray-scale image B.
3. **Step. Compute Horizontal Gradient**: We notice that Formula 2 can be applied in parallel to every single pixel b of B. We hence compute the horizontal gradient G_x over all pixels b of B by taking a 3x3 region around b and convolving it with the 3x3 kernel of Formula 2 according to the Map parallelisation of Figure 2.2.
4. **Step. Compute Vertical Gradient**: We notice that Formula 3 can be applied in parallel to every single pixel b of B. We hence compute the vertical gradient G_y over all pixels b of B by taking a 3x3 region around b and convolving it with the 3x3 kernel of Formula 3 according to the Map parallelisation of Figure 2.2.
5. **Step. Combine Vertical Gradient and Horizontal Gradient**: We notice that formula 5 can be applied in parallel over all the pixels g_x of G_x and g_y of G_y to yield pixel g of G according to the Map parallelisation of Figure 2.2.
6. **Step. Convert gray file to PNG**: The 'convert' utility proceeds sequentially in the final phase of the computation to convert the '.gray'. file resulting from phase (5) into the final output PNG file.

2.3 Sequential Version - OpenCV

OpenCV is a library of programming functions aimed at real-time computer vision, available both in C++ and Python. The OpenCV 3.2.0 C++ library was picked as a comparison baseline for the native-C Sobel Filter implementation as its performance and behaviour resemble more closely the native C implementation rather than the OpenCV Python library[3].

The sequential OpenCV version of the Sobel filter proceeds analogously to the native C implementation of the sequential version outlined in Section 2.1, in which lengthy native C code is replaced by more simple calls to the OpenCV C++ API. For this reason, the amount of lines of code of the OpenCV version is very much reduced w.r.t. the native C version.

3 Experimental Results

In this section, the experimental results obtained from running the Native C, GPU CUDA and OpenCV versions of the Sobel Filter implementation are compared.

3.1 Anomalous cudaMalloc() behaviour

An anomalous behaviour was detected when running the CUDA version on the Geforce MX 150 on my laptop and on the Geforce GTX 1080 of the *bruciato* server. As reported in Figure 3.1, the very first cudaMalloc() executed in the CUDA program appears to take an extremely long time when compared to subsequent memory operations and makes up for over 98% of time of all GPU memory operations. In fact, the CUDA environment is initialized in conjunction with the first cudaMalloc() invoked.

Because of the anomalous behaviour reported, a 'dummy' cudaMalloc(), responsible for allocating one single byte, yet taking over 600 ms on *bruciato* and 100 ms on my laptop, was placed at the very beginning of the program. The anomalous computation of this initial cudaMalloc() was not considered as part of the runtimes reported in Section 3.2.

3.2 Experimental Data

In this section, the experimental data obtained from the experiments carried out on the hardware architectures outlined in Table 1 is shown. In every single table, the average of 10 runs is calculated for every single runtime reported. All times reported are in ms, milliseconds.

Input data: The same image was tiled repeatedly over the different resolutions considered during the experiments carried out, as shown in Table 2.

[3]The OpenCV implementation is available at `https://github.com/DanyEle/Sobel_Filter/tree/master/OpenCV_Sobel`

```
daniele@spaceship:~/cuda-workspace/CUDA_Sobel$ ./Debug/CUDA_Sobel imgs_in/512x512.png
Time spent on GPU memory operations: [103.305000] ms
Time spent on GPU computation: [9.704000] ms
Time spent on I/O operations from/to disk: [118.317000] ms
Overall time spent in program [231.326000] ms
First cuda malloc has taken [102.054000] ms
```

(a) Anomalous behaviour of the first cudaMalloc() on my local laptop

```
gadler@bruciato:~/CUDA_Sobel$ ./Debug/CUDA_Sobel imgs_in/512x512.png
Time spent on GPU memory operations: [603.994000] ms
Time spent on GPU computation: [6.927000] ms
Time spent on I/O operations from/to disk: [228.515000] ms
Overall time spent in program [839.436000] ms
First cuda malloc has taken [602.710000] ms
```

(b) Anomalous behaviour of the first cudaMalloc() on the *bruciato* server

Figure 6: Anomalous behaviour of the first cudaMalloc() on my local laptop and on the *bruciato* server.

3.2.1 Sequential Version - Native C

Tables 3, 4, 5 and 6 show the runtime obtained by running the native C implementation on the *laptop*, *bruciato*, *gewurztraminer* and *Raspberry PI 3 B+* hardware configuration, respectively, with the input image data of Table 2. Figure 3.2.1 shows a pictorial representation of such runtimes.

The fields in Tables 3, 4, 5 and 6 are the following:

- **I/O**: encompasses the time spent reading the input image from the disk and writing the output image to the disk
- **Computation**: encompasses the time spent during the rest of the program.
- **Total**: considers the sum of the two fields mentioned above.

3.2.2 GPU Version - CUDA

Tables 8 and 7 show the data obtained by running the CUDA implementation on the *laptop* and *bruciato* hardware configurations with the input data of Table 2 and a CUDA synchronize statement after every kernel. Figure 3.2.2 shows a pictorial representation of such runtimes.

The fields in Tables 8 and 7 are the following:

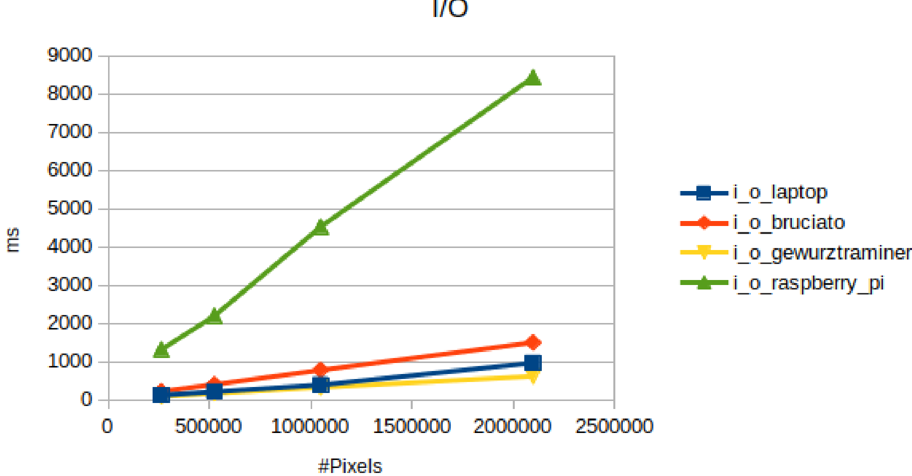
- **GPU memory movement**: encompasses the time spent running `cudaMalloc()` operations, `cudaMemCpy()` operations from device to host and from host to device and `cudaFree()` operations.
- **GPU computation**: encompasses the time spent in kernels on the GPU.
- **I/O**: encompasses the time spent reading the input image from the disk and writing the output image to the disk.
- **Total**: considers the sum of the three fields mentioned above.

3.2.3 Sequential Version - OpenCV

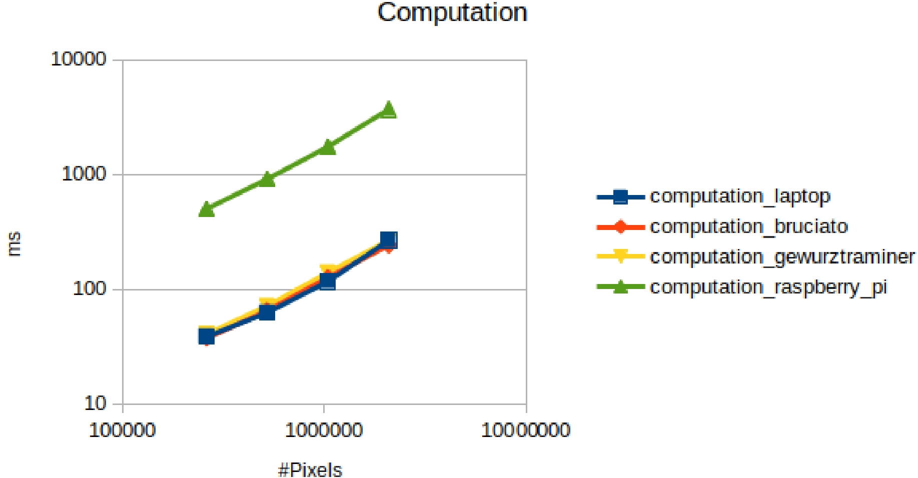
Tables 9 and 10 show the data obtained by running the CUDA implementation on the *laptop* and *raspberry* hardware configurations with the input data of Table 2. Figure 3.2.3 shows a pictorial representation of such runtimes.

The fields in Tables 9 and 10 are the following:

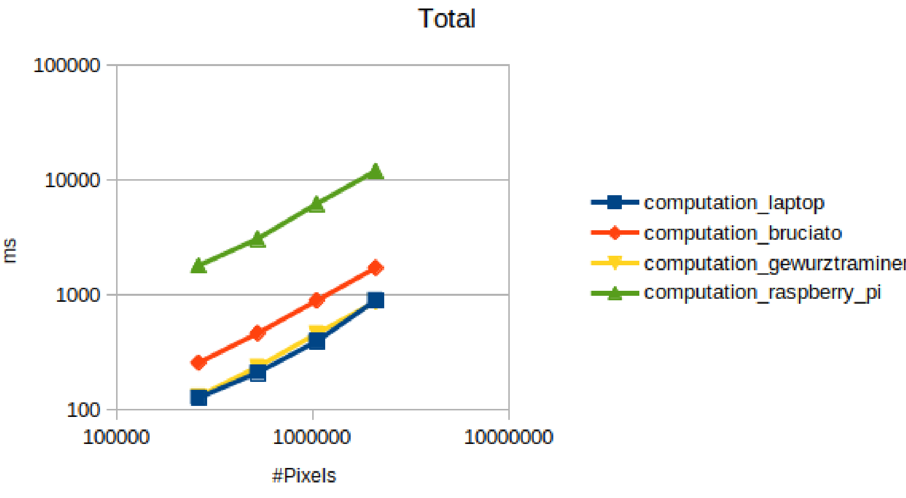
- **I/O**: encompasses the time spent reading the input image from the disk and writing the output image to the disk
- **Computation**: encompasses the time spent during the rest of the program.
- **Total**: considers the sum of the two fields mentioned above.



(a) I/O time of the native C program on my *laptop*, *bruciato*, *gewurztraminer*, *Raspberry PI 3 B+*



(b) Computation time of the native C program on my *laptop*, *bruciato*, *gewurztraminer*, *Raspberry PI 3 B+*



(c) Total time of the native C program on my *laptop*, *bruciato*, *gewurztraminer*, *Raspberry PI 3 B+* in log-log scale

Figure 7: I/O, Computation and total time of the native C program on my *laptop*, *bruciato*, *gewurztraminer*, *Raspberry PI 3 B+*

Image's #Pixels	Image Resolution	Image
262144	512x512	
524288	1024x512	
1048576	1024x1024	
2097152	2048x1024	

Table 2: Input image considered for the 512x512, 1024x512, 1024x1024 and 2048x1024 image resolutions

#Pixels	**262144**	**524288**	**1048576**	**2097152**
Image Resolution	**512x512**	**1024x512**	**1024x1024**	**2048x1024**
I/O	90.1806	150.111	281.3179	638.868
Computation	38.353	62.2	116.2214	265.642
Total	128.5336	212.311	397.5393	904.51

Table 3: Experimental data resulting from running the native C implementation on the *laptop* hardware configuration

#Pixels	**262144**	**524288**	**1048576**	**2097152**
Image Resolution	**512x512**	**1024x512**	**1024x1024**	**2048x1024**
I/O	221.3504	402.2781	774.4789	1494.8355
Computation	37.063	64.7642	125.7405	240.2341
Total	258.4134	467.0423	900.2194	1735.0696

Table 4: Experimental data resulting from running the native C implementation on the *bruciato* hardware configuration

#Pixels	**262144**	**524288**	**1048576**	**2097152**
Image Resolution	**512x512**	**1024x512**	**1024x1024**	**2048x1024**
I/O	91.9068	167.17	325.4511	613.6729
Computation	40.4716	71.0934	138.9503	266.3467
Total	132.3784	238.2634	464.4014	880.0196

Table 5: Experimental data resulting from running the native C implementation on the *gewurztraminer* hardware configuration

4 Discussion

In the present section, I am going to discuss the experimental results obtained in Section 3.2.

#Pixels	262144	524288	1048576	2097152
Image Resolution	**512x512**	**1024x512**	**1024x1024**	**2048x1024**
I/O	1313.0202	2193.6921	4520.5371	8426.9005
Computation	497.876	906.1778	1733.4482	3670.3458
Total	1810.8962	3099.8699	6253.9853	12097.2463

Table 6: Experimental data resulting from running the native C implementation on the *Raspberry PI 3 B+* hardware configuration

#Pixels	262144	524288	1048576	2097152
Image Resolution	**512x512**	**1024x512**	**1024x1024**	**2048x1024**
GPU memory movement	1.5413	2.4112	4.5904	9.8378
GPU Computation	15.8482	31.4309	63.2679	144.1928
I/O	92.9807	150.1936	290.4326	645.1516
Total	110.3702	184.0357	358.2909	799.1822

Table 7: Experimental data resulting from running the CUDA implementation on the *laptop* hardware configuration

#Pixels	262144	524288	1048576	2097152
Image Resolution	**512x512**	**1024x512**	**1024x1024**	**2048x1024**
GPU memory movement	0.9956	1.6498	3.7508	6.8657
GPU Computation	14.924	29.8788	58.8453	113.1021
I/O	231.4684	409.4673	782.6594	1515.7717
Total	247.388	440.9959	845.2555	1635.7395

Table 8: Experimental data resulting from running the CUDA implementation on the *bruciato* hardware configuration.

#Pixels	262144	524288	1048576	2097152
Image Resolution	**512x512**	**1024x512**	**1024x1024**	**2048x1024**
I/O	19.4348	25.2596	45.5474	68.78
Computation	73.6047	81.0963	85.2875	87.6996
Total	93.0395	106.3559	130.8349	156.4796

Table 9: Experimental data resulting from running the OpenCV implementation on the *laptop* hardware configuration

#Pixels	262144	524288	1048576	2097152
Image Resolution	**512x512**	**1024x512**	**1024x1024**	**2048x1024**
I/O	169.3188	384.5017	583.6068	1384.1298
Computation	60.5518	135.6102	248.2017	460.0023
Total	229.8706	520.1119	831.8085	1844.1321

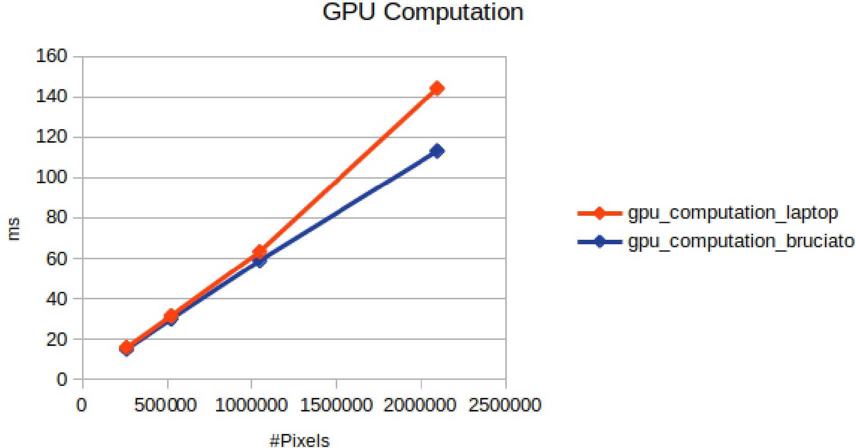
Table 10: Experimental data resulting from running the OpenCV implementation on the *raspberry* hardware configuration

4.1 CUDA vs OpenCV vs Native C

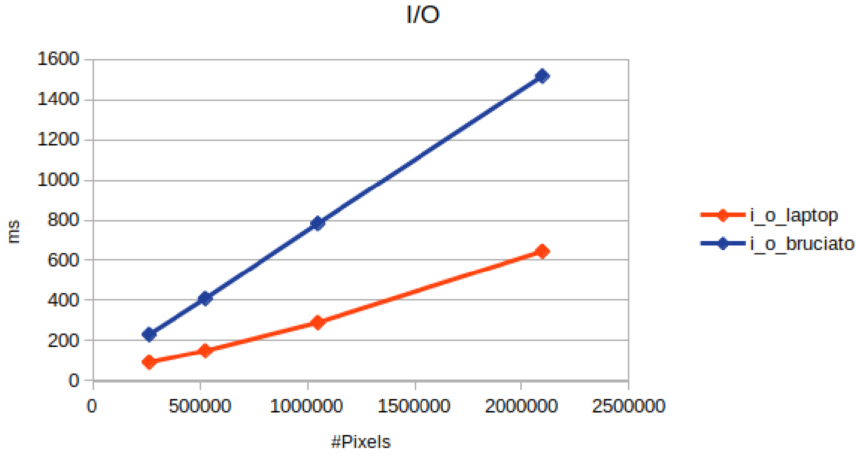
After removing the anomalous behaviour outlined in Section 3.1, we notice from Figures 3.2.2, 3.2.1 that the CUDA and native C's computation time scales linearly with N_{pixels}, the amount of pixels of the input image considered. This behaviour applies to the I/O operations, the GPU memory movement, the GPU computation, the CPU computation and the Total computation of the CUDA and Native C implementations. The behaviour of these four metrics is hence modelled according to Formula 6.

Instead, from Figure 3.2.3 we notice that the `computation` time and the `total` of the OpenCV implementation do not scale linearly with N_{pixels} on the *laptop* configuration. Instead, a quasi-linear scaling behaviour for the `computation` and `total` time is present for the *Raspberry PI* configuration. Such poor scaling on the *laptop* configuration is due to the underlying optimizations present in OpenCV algorithms being compiled for general-purpose commodity hardware architectures (such as my laptop). As shown experimentally in Figure 3.2.3, no such optimizations seem to be present for ARM architecture (such as the Raspberry PI).

(a) GPU memory movement comparison of the CUDA program on my *laptop* vs *bruciato* server



(b) GPU computation comparison of the CUDA program on my *laptop* vs *bruciato* server

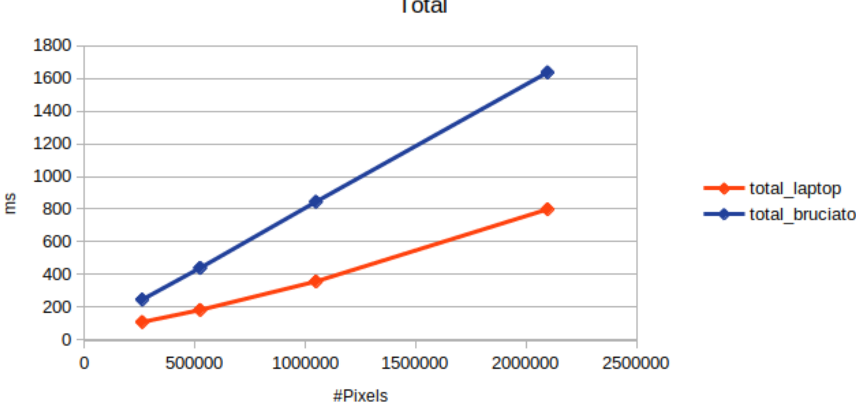


(c) I/O comparison of the CUDA program on my *laptop* vs *bruciato* server

$$T_{Image} = N_{pixels} * T_{Pixel} \tag{6}$$

where T_{Image} is the time to process an input image (in ms), N_{pixels} is the amount of pixels in the input image and T_{Pixel} is the time to process one single pixel of the input image.

Worth discussing is also the I/O behaviour of *bruciato* in Figures 3.2.1 and 3.2.2, where *bruciato* is outper-



(d) Total time elapsed by the CUDA program on my *laptop* vs *bruciato* server

Figure 8: The GPU memory movement, GPU computation and I/O time and total time of the CUDA program on my *laptop* vs *bruciato* server

formed by the *laptop* and *gewurztraminer* hardware configurations. In fach the *bruciato* hardware configuration is equipped with a magnetic hard disk, whereas the *laptop* and the *gewurztraminer* hardware configurations are both equipped with an SSD.

From Figure 3.2.3, we notice that the OpenCV version obtains an excellent `Total` performance on the *laptop* configuration and even beats the CUDA implementation for the 2048x1024 resolution. Furthermore, the OpenCV's `I/O` runtime is significantly lower than the CUDA `I/O` for all input images considered. This behaviour is due to the massive optimizations present in the OpenCV version, which make OpenCV a framework suitable for real-time image processing, requiring a very low computational latency.

Finally, the *Raspberry PI*, unsurprisingly, is outperformed by the *bruciato*, *gewurztraminer* and *laptop* hardware configurations for all metrics considered because of its low computational power, as outlined in Table 1.

4.1.1 Speedup Comparison and Limitations

By comparing the CUDA implementation with the native C implementation on the *laptop* hardware configuration, we notice that the CUDA computation runtime on the *laptop* has a speedup of approximately 1.9x w.r.t. the native C computation runtime.

Analogously, the CUDA GPU computation vs native C computation runtime on the *bruciato* server obtains a speedup of approximately 2.3x. Therefore, the CUDA implementation does benefit from a faster GPU: this fact shows experimentally that the CUDA implementation's performance scales when running on faster hardware.

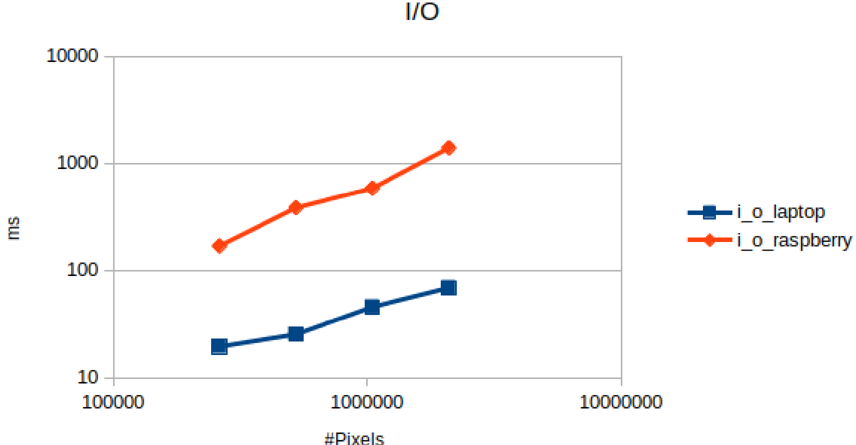
Such results show that the CUDA implementation was successfully parallelised. The report writer suggests that, if the arithmetic intensity of the CUDA kernels were higher, the speedup obtained would be higher as well.

Finally, if all N_{pixels} of the input image could be fed immediately to $N_{threads}$ CUDA threads (assuming $N_{pixels} = N_{threads}$) of the GPU and could be processed in parallel, we would obtain a speedup equal to N_{pixels}. This would only be possible if $N_{threads}$ threads operated in parallel over N_{pixels} and no time were spent in the kernel for scheduling and moving data over the CPU-GPU bus. However, this $N_{threads}$ speedup scenario is not realistic in a real-world GPU computation scenario, because of the hardware characteristics of the GPUs considered in Table 1. Namely, the speedup is limited by the fact that the GPU architecture of general purpose GPUs (such as the MX150 and GTX1080 GPUs) are heavily optimized for streaming operations, thanks to their small caches and their high on-chip ALU/memory ratio [1].

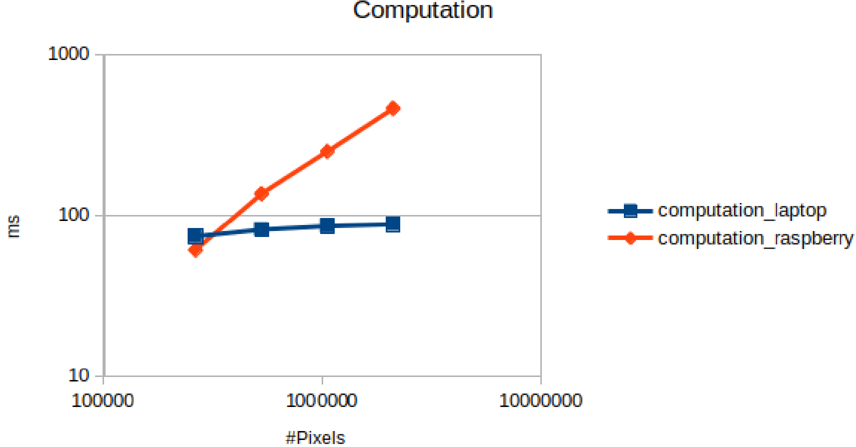
4.2 n:1 pixels per thread Version

In the CUDA implementation described in Section 2.2, I assigned one CUDA thread to each pixel of the input image. From now onwards, such implementation will be called the "1:1" implementation. In the present section, I would like to validate such design decision by comparing the performance of the 1:1 implementation with an implementation where n pixels of the input image are assigned to one thread ($n : 1$ implementation from now onwards).

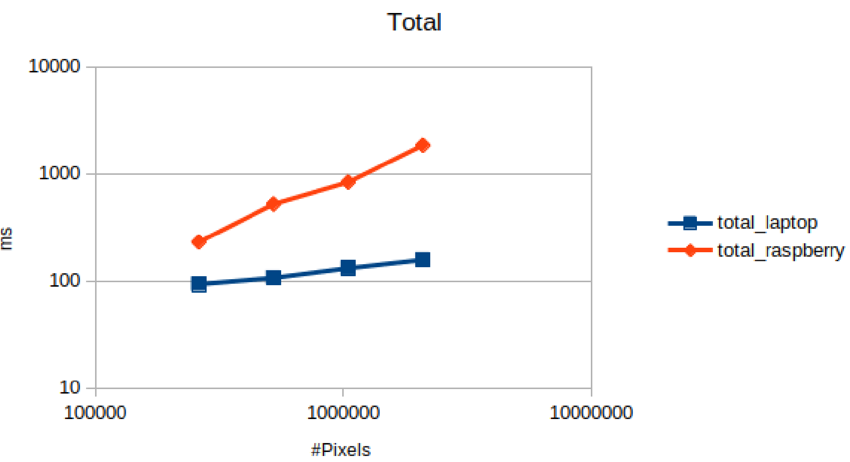
The $n : 1$ version was implemented by passing multiple pixels to one single thread and iterating over these pixels within one CUDA kernel. Alternatively, the report writer also considered an implementation where



(a) I/O time of the OpenCV program on my *laptop* vs the *Raspberry PI 3 B+* in log-log scale.



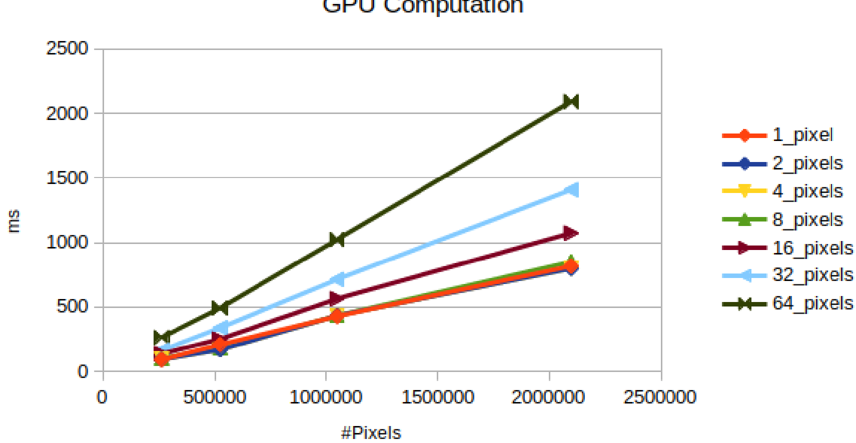
(b) Computation time of the OpenCV program on my *laptop* vs the *Raspberry PI 3 B+* in log-log scale.



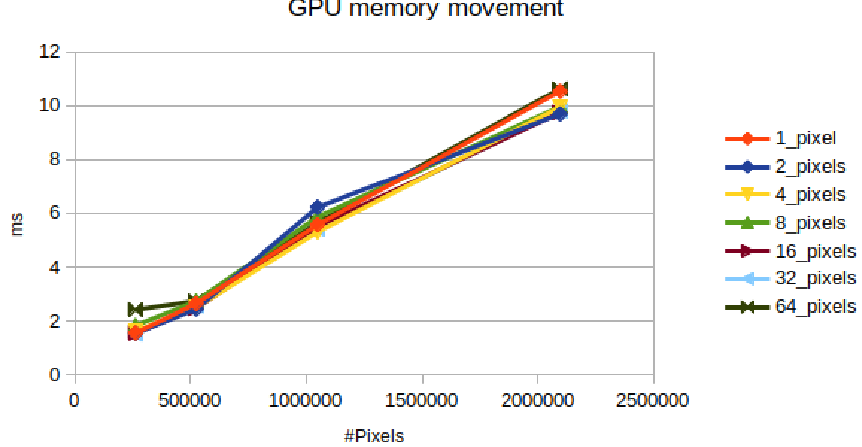
(c) Total time of the OpenCV program on my *laptop* vs the *Raspberry PI 3 B+* in log-log scale.

Figure 9: I/O, Computation and total time of the OpenCV program on my *laptop* and the *Raspberry PI 3 B+*

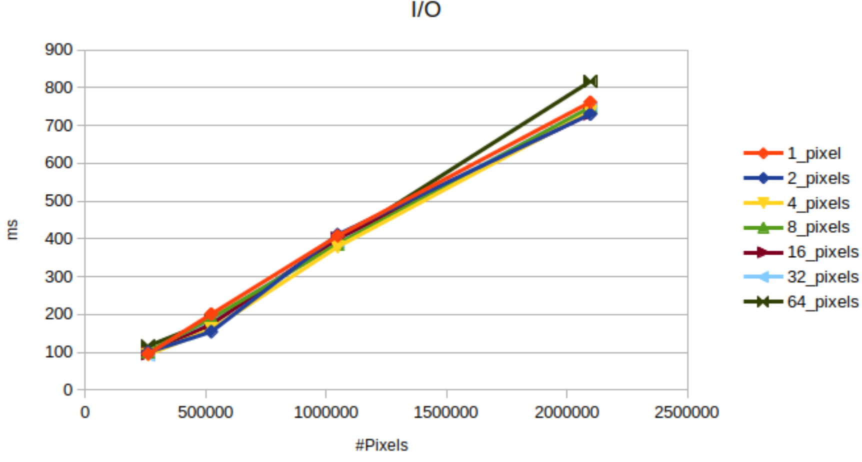
multiple statements were laid out over n pixels in a sequential manner, with no for loops. However, both



(a) GPU computation time of the CUDA implementation on my *laptop* for 1, 2, 4, 8, 16, 32, 64 pixels per thread.

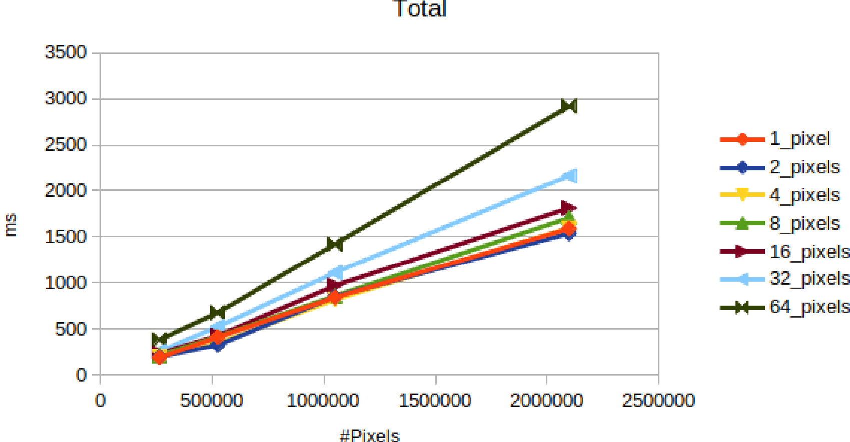


(b) GPU memory movement time of the CUDA implementation on my *laptop* for 1, 2, 4, 8, 16, 32, 64 pixels per thread.



(c) I/O time of the CUDA implementation on my *laptop* for 1, 2, 4, 8, 16, 32, 64 pixels per thread.

implementations resulted equivalent from a functional and a performance point of view. The report writer hence suggests that most likely a loop unrolling optimization is carried out by the `nvcc` compiler for CUDA kernels' code and that the penalty due to `jump` instructions in the loop is minimal. This hypothesis is backed experimentally by the increased GPU computation time shown in Tables 11, 12, 13, 13, 14 and theoretically by



(d) Total time of the CUDA implementation on my *laptop* for 1, 2, 4, 8, 16, 32, 64 pixels per thread.

Figure 10: I/O, Computation and total time of the OpenCV program on my *laptop* and the *Raspberry PI 3 B+*

the fact that loop branches are very inefficient in GPU threads, as they break a processor's fetch-decode-execute pipeline [1].

Figures 10b, 10c, 10a and 10d plot the GPU memory movement, I/O time, GPU computation time and Total time, respectively for $n \in [1, 2, 4, 8, 16, 32, 64]$ (the amount of pixels per thread) and $N_{pixels} \in [512 * 512, 1024 * 512, 1024 * 1024, 2048 * 1024]$ (the amount of pixels in the input images).

4.2.1 Performance Model

Figures 10b and 10c suggest that the amount of pixels per thread (n) has little to no impact as far as the GPU memory movement time and the I/O time are concerned. Namely, no matter which value of n is picked, the GPU memory movement and the I/O time will change by a minimum amount.

Figure 10a shows that by increasing n, the amount of pixels per thread, the GPU computation time increases quasi-linearly for $n = 8, 16, 32, 64$, whereas the computation time is very similar for $n = 1, 2, 4$. By increasing the resolution of the input images, and hence the amount of pixels N_{pixels} contained therein, the GPU computation time also appears to be scaling quasi-linearly.

Based on these observations, I propose the model of Formula 7 for the GPU computation time in the $n : 1$ case. Formula 7 appears to be pretty well fitting the GPU computation time's behaviour shown in Figure 10a for $n >= 8$.

$$T_{GPU} = n * T_{pixel} * N_{pixels} \tag{7}$$

where $n \in [8, 16, 32, 64]$ is the amount of pixels per thread, T_{pixel} is the time required to perform the GPU computation for one pixel of the input image, $N_{pixels} \in [512 * 512, 1024 * 512, 1024 * 1024, 2048 * 1024]$ is the amount of pixels in the input images considered and T_{GPU} is the computation time of the GPU during the processing of each input image.

Formula 7 implies that by increasing n, we obtain a directly proportional linear increase in T_{GPU}. Analogously, by increasing N_{pixels} we obtain a directly proportional linear increase in T_{GPU}.

The linear scaling behaviour highlighted in the Total computation time of Figure 10d is a direct consequence of the linear scaling behaviour of the GPU computation time shown in Figure 10a.

4.3 Performance Comparison of n:1 pixels per thread vs 1:1 pixels per thread

Tables 11, 12 13, 14 show the GPU memory movement, GPU computation, I/O time and Total time respectively for $N_{pixels} \in [512 * 512, 1024 * 512, 1024 * 1024, 2048 * 1024]$ and $n \in [1, 2, 4, 8, 16, 32, 64]$ for the $n : 1$ pixels per thread (p.t.) implementation. The **Relative Speedup** is defined as: $\frac{T^{n:1}_{GPU}(1)}{T^{n:1}_{GPU}(n)}$, where $T^{n:1}_{GPU}(1)$ is the GPU computation time of the $n : 1$ implementation with 1 pixel per thread and $T^{n:1}_{GPU}(n)$ is the GPU computation time of the $n : 1$ implementation with n pixels per thread.

From Tables 11, 12 13, 14, it is remarkable to notice that the relative speedup decreases by increasing n, with the exception of the $n = 2, n = 4$ case for $N_{pixels} = 1024 * 512$ and $n = 2$ for $N_{pixels} = 2048 * 1024$. Such exceptions are most likely justified by an efficient usage of GPU caches for the values of n considered.

512x512							
#Pixels per Thread	**1 p.t.**	**2 p.t.**	**4 p.t.**	**8 p.t.**	**16 p.t.**	**32 p.t.**	**64 p.t.**
GPU memory movement	1.5552	1.5412	1.6248	1.8299	1.5576	1.5215	2.4168
GPU Computation	91.4494	93.4583	100.5723	110.7912	136.6889	163.7567	264.4777
I/O	95.1918	100.0423	93.3787	101.2149	96.4165	92.5592	116.2303
Total	188.1964	195.0418	195.5758	213.836	234.663	257.8374	383.1248
Relative Speedup	1	0.98	0.93	0.91	0.81	0.83	0.62

Table 11: GPU memory movement, GPU computation, I/O, Total runtimes and Relative Speed for the 512x512 input image on the *laptop* hardware configuration for the $n:1$ CUDA implementation

1024x512							
#Pixels per Thread	**1 p.t.**	**2 p.t.**	**4 p.t.**	**8 p.t.**	**16 p.t.**	**32 p.t.**	**64 p.t.**
GPU memory movement	2.6459	2.4389	2.4953	2.7343	2.4689	2.563	2.7396
GPU Computation	208.9538	169.1638	183.7097	225.108	251.4933	338.2722	491.5858
I/O	200.75	154.0919	160.9823	190.0286	173.6954	185.2298	182.9089
Total	412.3497	325.6946	347.1873	417.8709	427.6576	526.065	677.2343
Relative Speedup	1	1.24	1.14	0.93	0.83	0.62	0.43

Table 12: GPU memory movement, GPU computation, I/O, Total runtimes and Relative Speed for the 1024x512 input image on the *laptop* hardware configuration for the $n:1$ CUDA implementation

1024x1024							
#Pixels per Thread	**1 p.t.**	**2 p.t.**	**4 p.t.**	**8 p.t.**	**16 p.t.**	**32 p.t.**	**64 p.t.**
GPU memory movement	5.5768	6.2221	5.311	5.8691	5.5045	5.4143	5.6543
GPU Computation	427.8063	433.0888	433.1126	466.0709	563.8854	713.2571	1022.0434
I/O	407.4501	410.821	378.8298	385.8779	400.5517	393.1969	392.8111
Total	850.8332	850.1319	817.2534	857.8179	969.9416	1111.8683	1420.5088
Relative Speedup	1	0.99	0.99	0.93	0.77	0.65	0.55

Table 13: GPU memory movement, GPU computation, I/O, Total runtimes and Relative Speed for the 1024x1024 input image on the *laptop* hardware configuration for the $n:1$ CUDA implementation

2048x1024							
#Pixels per Thread	**1 p.t.**	**2 p.t.**	**4 p.t.**	**8 p.t.**	**16 p.t.**	**32 p.t.**	**64 p.t.**
GPU memory movement	10.5396	9.6879	9.9469	9.965	9.7461	9.8072	10.6299
GPU Computation	817.3744	798.4245	848.9382	949.0839	1071.9428	1407.4057	2089.8141
I/O	761.4638	729.7934	736.4157	747.8666	731.9428	746.7296	816.379
Total	1589.3778	1537.9058	1595.3008	1706.9155	1813.3124	2163.9425	2916.823
Relative Speedup	1	1.02	0.96	0.86	0.76	0.57	0.41

Table 14: GPU memory movement, GPU computation, I/O, Total runtimes and Relative Speed for the 2048x1024 input image on the *laptop* hardware configuration for the $n:1$ CUDA implementation

- **GPU Memory movement**: Comparing the GPU memory movement of the $1:1$ implementation with the GPU memory movement of the $n:1$ implementation, we do not notice substantial differences among the runtimes obtained for these two implementations. In fact, the GPU memory movement does not vary proportionally over the n considered and is simply directly proportional to N_{pixels}, the amount of pixels moved over the bus (which stays the same both for the $1:1$ and $n:1$ implementation.

- **I/O**: Similarly to what occurs in the GPU memory movement, we do not notice substantial differences in I/O time. In fact, the I/O time is directly proportional to N_{pixels} and does not change as a function of n for the $n:1$ implementation. Instead, the amount of I/O operations performed is just a function of N_{pixels}.

- **GPU Computation**: If we compare the GPU computation time of the $n:1$ implementation with the GPU computation time of the $1:1$ implementation, we do notice some differences. In Section 4.2.1, we noticed that T_{GPU}, the GPU computation time varied as a function of n, scaled by a constant S.

  For this reason, the more n is increased, the slower the GPU computation with n pixels and the higher the performance speedup of the $1:1$ version vs the $n:1$ version.

1:1 vs n:1 Absolutel Speedup							
Resolution	**1 p.t.**	**2 p.t.**	**4 p.t.**	**8 p.t.**	**16 p.t.**	**32 p.t.**	**64 p.t.**
512x512	0.173	0.169	0.158	0.143	0.116	0.097	**0.06**
1024x512	0.15	**0.186**	0.171	0.14	0.125	0.093	0.063
1024x1024	0.147	0.146	0.146	0.136	0.112	0.089	0.061
2048x1024	0.176	0.18	0.17	0.152	0.135	0.1	0.69

Table 15: 1:1 vs n:1 Absolute Speedup for $n \in [1, 2, 4, 8, 16, 32, 64]$ and $N_{pixels} \in [512 * 512, 1024 * 512, 1024 * 1024, 2048 * 1024]$ for the GPU computation time T_{GPU}.

Table 15 shows how many times the $1:1$ version computation is faster than the $n:1$ version computation time. The **Absolute Speedup** is computed as: $\frac{T^{1:1}_{GPU}}{T^{n:1}_{GPU}}$, where $T^{n:1}_{GPU}$ is the GPU computation time of the $n:1$ pixels' mapping to threads version and $T^{1:1}_{GPU}$ is the GPU computation time of the $1:1$ pixels' mapping to threads version. Table 15 reports a sub-linear speeup trend for n, as the speedup increases substantially when considering $n >= 4$.

- **Total**: The `Total` time is influenced by the GPU computation time, which is proportional to n for the $n:1$ Threads per Pixel version. Hence, the same conclusions as those drawn for the GPU computation time can be drawn.

The experimental results of Table 15 show that assigning one pixel per thread is experimentally more efficient than assigning n pixels per thread and that increasing the n values leads to a further decrease in the performance of the $n:1$ version. This trend repeats for all N_{pixels} considered.

Given the observations brought forward in the present section, we can conclude that the $n:1$ Threads per Pixel version is from 0.06x up to 0.186x times slower than the $1:1$ Threads per Pixel version, which outperforms the $n:1$ implementation for all values of n and N_{pixels} considered.

4.4 Map Fusion Optimization

Figure shows the architecture of the $1:1$ GPU Map parallelization, as described in 4.2.1.

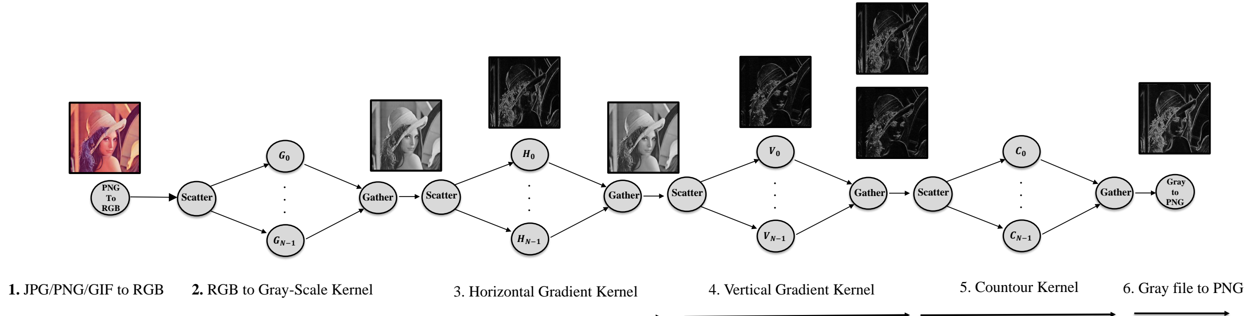


Figure 11: GPU Map parallelization adopted for the $1:1$ version, as described in 4.2.1

Based on Figure 4.4 and inspecting the kernel code of the horizontal and vertical gradient computation code, we noticed that the horizontal and vertical gradient kernels accessed the same memory areas, whereas the countour kernel just accessed the resulting pixels of the horizontal and vertical gradient computations.

Consequently, a **Map Fusion** optimization was carried out and implemented, in which the three separate kernels of the horizontal, vertical gradient computation, as well as the countour image computation were merged into one single kernel, as shown in Figure 4.4.

4.4.1 1:1 vs Map Fusion Experimental Comparison

Figures 13a, 13c, 13b, 13d respectively show a comparison of the $1:1$ vs the Map Fusion implementation for the I/O time, the GPU memory movement, the GPU computation time and the total time.

From Figure 13c, we notice that the Map Fusion optimization carried out has a strong impact on the GPU memory movement: as N_{pixels} increases, so does the impact of this optimization increase. However, because the GPU memory movement time's magnitude is small w.r.t. the GPU computation time, this reduction in GPU memory movement time does not reflect on the `total` computation time, which still appears to be similar to the $1:1$ `total` computation time. On the other hand, the GPU computation time is almost unchanged

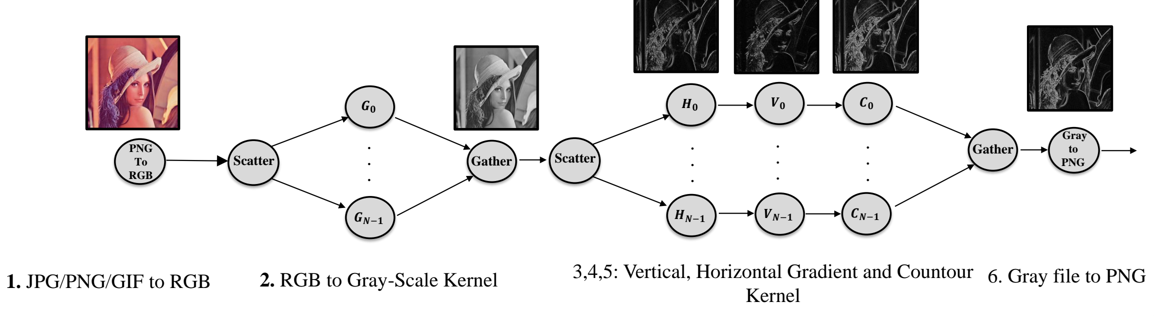


Figure 12: Map Fusion Optimization adopted for the 1 : 1 version

because the arithmetic operations carried out in the kernels are still the same, even following the Map Fusion optimization. The same reasoning applies to I/O operations, which are still the same following the Map Fusion optimization.

5 Instructions

This section contains instructions to install, compile and run the native C, OpenCV, CUDA 1 : 1 pixels' mapping to threads and $n:1$ pixels' mapping to threads for the Sobel Filter.

5.1 Dependencies

The 'convert' utility from the 'imagemagick' package is required for converting input PNG/JPG/GIF images to the RGB format, an encoding-independent format taken as input by all three implementations. Accepted input image formats are PNG, JPG and GIF .

To install the 'imagemagick' package in Linux, run:

```
sudo apt-get install imagemagick
```

To install the 'imagemagick' package on a MAC, run:

```
brew install imagemagick
```

5.2 Clone the repository

Just run the following commands to clone the repository

```
cd $HOME
git clone https://github.com/DanyEle/Sobel_Filter.git
```

5.3 Native C Version

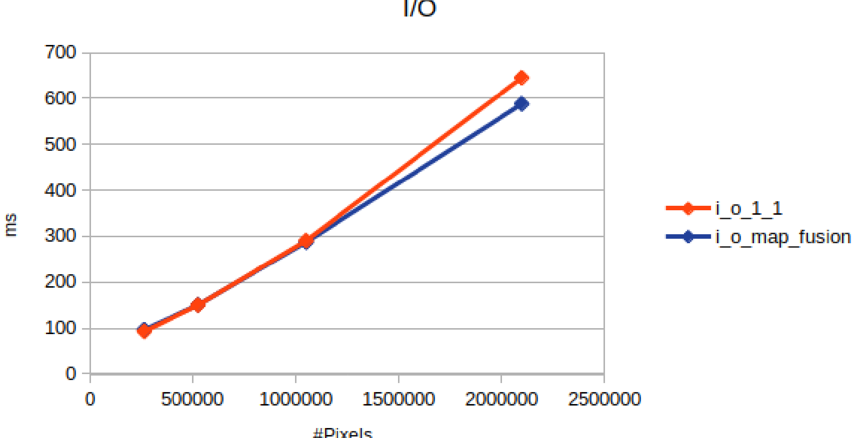
In order to run the native C version, make sure you have the gcc compiler installed. We will now proceed to compile the project files and perform one run of the Sobel Filter with an input image.

```
cd $HOME
cd Sobel_Filter/Native_Sobel
./compile.sh
./Debug/Native_Sobel imgs_in/512x512.png
```

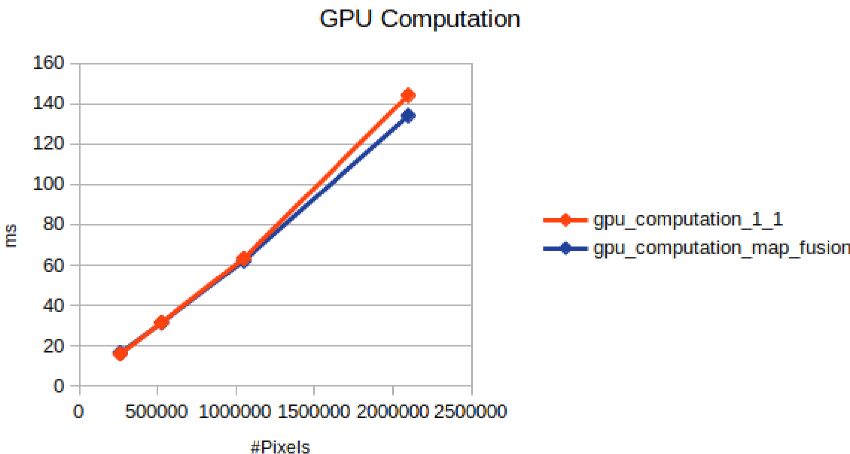
The 512x512.png image can be replaced by any other image (e.g.: 1024x512.png, 1024x1024.png, 2048x1024.png). The resulting output lies in the Sobel_Filter/Native_Sobel/imgs_out/ folder.

If you would like to execute 10 runs of the native C Sobel Filter with a certain input image, then just run:

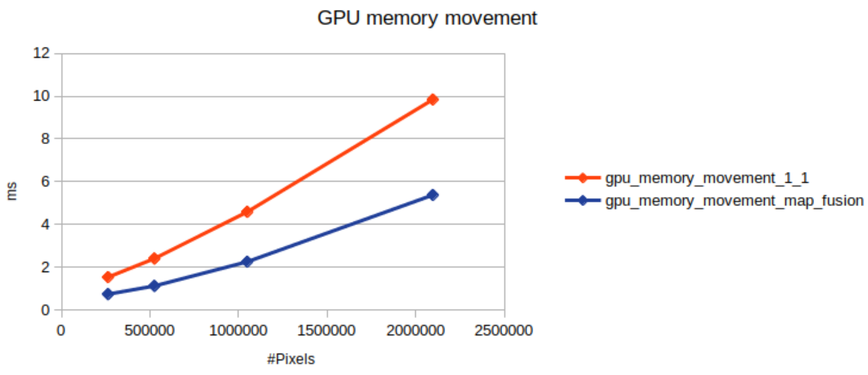
```
./run_experiments.sh <input_image>
```



(a) I/O time of the 1 : 1 version vs the map-fusion version on the *laptop* hardware configuration



(b) GPU computation time of the 1 : 1 version vs the map-fusion version on the *laptop* hardware configuration



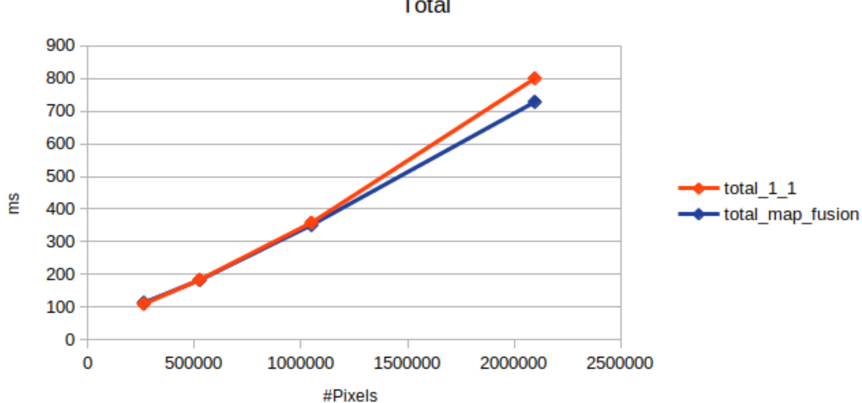
(c) GPU memory movement time of the 1 : 1 version vs the map-fusion version on the *laptop* hardware configuration

5.4 OpenCV Version

In order to run the OpenCV version, make sure you have the g++ compiler and OpenCV installed. To quickly install OpenCV in Linux:

```
sudo apt-get install libopencv-dev
```

Now, to compile and run the OpenCV version, execute the following commands:



(d) Total time of the 1 : 1 version vs the map-fusion version on the *laptop* hardware configuration

```
cd $HOME
cd Sobel\_Filter/OpenCV_Sobel
./compile.sh
./Debug/OpenCV\_Sobel imgs_in/512x512.png
```

The output lies in the Sobel_Filter/OpenCV_Sobel/imgs_out/ folder.
If you would like to execute 10 runs of the OpenCV Sobel Filter with a certain input image, then just run:

```
./run_experiments.sh <input_image>
```

5.5 CUDA Version - 1:1 pixels' mapping to threads

In order to run the CUDA version, make sure you have a CUDA-compatible GPU and its corresponding CUDA drivers, as well as the nvidia-cuda-toolkit to compile CUDA programs. To quickly install the NVIDIA-CUDA-Toolkit on Linux, just run:

```
sudo apt-get install nvidia-cuda-toolkit
```

Now, to compile and run the CUDA implementation, execute the following commands:

```
cd $HOME
cd Sobel_Filter/CUDA_Sobel
./compile.sh
./Debug/CUDA_Sobel imgs_in/512x512.png
```

The output lies in the Sobel_Filter/CUDA_Sobel/imgs_out/ folder.
If you would like to execute 10 runs of the CUDA Sobel Filter with a certain input image, then just run:

```
./run_experiments.sh <input_image>
```

5.6 CUDA version - n:1 pixels' mapping to threads

In order to run the CUDA version using multiple pixels per thread, just follow the instructions for setting up the plain CUDA version as by Section 5.5 and run the following commands:

```
cd $HOME
cd Sobel_Filter/CUDA_Sobel_Block
./compile.sh
./Debug/CUDA_Sobel imgs_in/512x512.png 2
```

The output lies in the Sobel_Filter/CUDA_Sobel_Block/imgs_out/ folder. If you would like to execute 10 runs of the CUDA Sobel Filter with a certain input image and a certain amount of pixels per thread, then just run:

```
./run_experiments.sh <input_image> <amount_of_pixels_per_thread>
```

5.7 CUDA version - Map Fusion

In order to run the CUDA version optimized by Kernels' map fusion, just follow the instructions for setting up the plain CUDA version as by Section 5.5 and run the following commands:

```
cd $HOME
cd Sobel_Filter/CUDA_Map_Fusion
./compile.sh
./Debug/CUDA_Sobel imgs_in/512x512.png 2
```

The output lies in the Sobel_Filter/CUDA_Sobel_Fusion/imgs_out/ folder. If you would like to execute 10 runs of the CUDA Sobel Filter with a certain input image and a certain amount of pixels per thread, then just run:

```
./run_experiments.sh <input_image>
```

References

[1] Massimo Coppola. Intro to GPGPU - General Purpose GPU programming. Programming Tools for Parallel and Distributed Systems, University of Pisa, 2019.

[2] Daniele Gadler. A Review of Object Classification Methods for Pedestrian Detection in Autonomous Vehicles. Seminar for Computer Graphics and Embedded Systems. Technische Universitaet Kaiserslautern, Germany, 2017.

[3] Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming, Portable Documents*. Addison-Wesley Professional, 2010.