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.

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 $B$.

$$b = 0.3 \cdot p_r + 0.59 \cdot p_g + 0.11 \cdot p_b \quad (1)$$

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 $G_x$.

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} \ast B \quad (2)$$

Figure 1: Input RGB image and resulting image, following Sobel Filter’s application
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 $G_y$.

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

(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 $G$.

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

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

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

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.

$$g = \sqrt{g_x^2 + g_y^2} \quad (5)$$

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.

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:

\[
\text{Amount of Pixels} = \text{width} \times \text{height}
\]

\[
\text{width} = \text{blocks}; \text{height} = \text{threads}
\]

\[
\text{Amount of Threads} = \text{blocks} \times \text{threads}
\]

![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.](image)

The CUDA implementation proceeds according to the following steps:

---

2The 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.

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.

---

3.2.1 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+
<table>
<thead>
<tr>
<th>#Pixels</th>
<th>262144</th>
<th>524288</th>
<th>1048576</th>
<th>2097152</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image Resolution</td>
<td>512x512</td>
<td>1024x512</td>
<td>1024x1024</td>
<td>2048x1024</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>#Pixels</th>
<th>262144</th>
<th>524288</th>
<th>1048576</th>
<th>2097152</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image Resolution</td>
<td>512x512</td>
<td>1024x512</td>
<td>1024x1024</td>
<td>2048x1024</td>
</tr>
<tr>
<td>I/O</td>
<td>90.1806</td>
<td>150.111</td>
<td>281.3179</td>
<td>638.868</td>
</tr>
<tr>
<td>Computation</td>
<td>38.353</td>
<td>62.2</td>
<td>116.2214</td>
<td>265.642</td>
</tr>
<tr>
<td>Total</td>
<td>128.5336</td>
<td>212.311</td>
<td>397.5393</td>
<td>904.51</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>#Pixels</th>
<th>262144</th>
<th>524288</th>
<th>1048576</th>
<th>2097152</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image Resolution</td>
<td>512x512</td>
<td>1024x512</td>
<td>1024x1024</td>
<td>2048x1024</td>
</tr>
<tr>
<td>I/O</td>
<td>221.3504</td>
<td>402.2781</td>
<td>774.4789</td>
<td>1494.8355</td>
</tr>
<tr>
<td>Computation</td>
<td>37.063</td>
<td>64.7642</td>
<td>125.7405</td>
<td>240.2341</td>
</tr>
<tr>
<td>Total</td>
<td>258.4134</td>
<td>467.0423</td>
<td>900.2194</td>
<td>1735.0696</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>#Pixels</th>
<th>262144</th>
<th>524288</th>
<th>1048576</th>
<th>2097152</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image Resolution</td>
<td>512x512</td>
<td>1024x512</td>
<td>1024x1024</td>
<td>2048x1024</td>
</tr>
<tr>
<td>I/O</td>
<td>91.9068</td>
<td>167.17</td>
<td>325.4511</td>
<td>613.6729</td>
</tr>
<tr>
<td>Computation</td>
<td>40.4716</td>
<td>71.0934</td>
<td>138.9503</td>
<td>266.3467</td>
</tr>
<tr>
<td>Total</td>
<td>132.3784</td>
<td>238.2634</td>
<td>464.4014</td>
<td>880.0196</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>#Pixels</th>
<th>262144</th>
<th>524288</th>
<th>1048576</th>
<th>2097152</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image Resolution</td>
<td>512x512</td>
<td>1024x512</td>
<td>1024x1024</td>
<td>2048x1024</td>
</tr>
<tr>
<td>I/O</td>
<td>1313.0202</td>
<td>2193.6921</td>
<td>4520.5371</td>
<td>8426.9005</td>
</tr>
<tr>
<td>Computation</td>
<td>497.876</td>
<td>906.1778</td>
<td>1733.4482</td>
<td>3670.3458</td>
</tr>
<tr>
<td>Total</td>
<td>1810.8962</td>
<td>3099.8699</td>
<td>6253.9853</td>
<td>12097.2463</td>
</tr>
</tbody>
</table>

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

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

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

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

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

<table>
<thead>
<tr>
<th>#Pixels</th>
<th>262144</th>
<th>524288</th>
<th>1048576</th>
<th>2097152</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image Resolution</td>
<td>512x512</td>
<td>1024x512</td>
<td>1024x1024</td>
<td>2048x1024</td>
</tr>
<tr>
<td>I/O</td>
<td>19.4348</td>
<td>25.2596</td>
<td>45.5474</td>
<td>87.6996</td>
</tr>
<tr>
<td>Computation</td>
<td>73.6047</td>
<td>81.0963</td>
<td>85.2875</td>
<td>87.6996</td>
</tr>
<tr>
<td>Total</td>
<td>93.0395</td>
<td>106.3559</td>
<td>130.8349</td>
<td>156.4796</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>#Pixels</th>
<th>262144</th>
<th>524288</th>
<th>1048576</th>
<th>2097152</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image Resolution</td>
<td>512x512</td>
<td>1024x512</td>
<td>1024x1024</td>
<td>2048x1024</td>
</tr>
<tr>
<td>I/O</td>
<td>169.3188</td>
<td>384.5017</td>
<td>583.6068</td>
<td>1384.1298</td>
</tr>
<tr>
<td>Computation</td>
<td>60.5518</td>
<td>135.6102</td>
<td>248.2017</td>
<td>460.0023</td>
</tr>
<tr>
<td>Total</td>
<td>229.8706</td>
<td>520.1119</td>
<td>831.8085</td>
<td>1844.1321</td>
</tr>
</tbody>
</table>

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 $N_{\text{pixels}}$, 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 $N_{\text{pixels}}$ 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

\[ T_{\text{Image}} = N_{\text{pixels}} \times T_{\text{Pixel}} \]  \hspace{1cm} (6)

where \( T_{\text{Image}} \) is the time to process an input image (in ms), \( N_{\text{pixels}} \) is the amount of pixels in the input image and \( T_{\text{Pixel}} \) 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-
formed by the laptop and gewurztraminer hardware configurations. In fact 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 \( N_{\text{pixels}} \) of the input image could be fed immediately to \( N_{\text{threads}} \) CUDA threads (assuming \( N_{\text{pixels}} = N_{\text{threads}} \)) of the GPU and could be processed in parallel, we would obtain a speedup equal to \( N_{\text{pixels}} \). This would only be possible if \( N_{\text{threads}} \) threads operated in parallel over \( N_{\text{pixels}} \) and no time were spent in the kernel for scheduling and moving data over the CPU-GPU bus. However, this \( N_{\text{threads}} \) 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
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 $N_{\text{pixels}} \in [512 \times 512, 1024 \times 512, 1024 \times 1024, 2048 \times 1024]$ (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 $N_{\text{pixels}}$ 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 $n \geq 8$.

$$T_{\text{GPU}} = n \times T_{\text{pixel}} \times N_{\text{pixels}}$$

where $n \in \{8, 16, 32, 64\}$ is the amount of pixels per thread, $T_{\text{pixel}}$ is the time required to perform the GPU computation for one pixel of the input image, $N_{\text{pixels}} \in [512 \times 512, 1024 \times 512, 1024 \times 1024, 2048 \times 1024]$ is the amount of pixels in the input images considered and $T_{\text{GPU}}$ 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 $T_{\text{GPU}}$. Analogously, by increasing $N_{\text{pixels}}$ we obtain a directly proportional linear increase in $T_{\text{GPU}}$.

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 $N_{\text{pixels}} \in [512 \times 512, 1024 \times 512, 1024 \times 1024, 2048 \times 1024]$ 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: $T_{\text{GPU}}^{1:1}(n)$, where $T_{\text{GPU}}^{1:1}(1)$ is the GPU computation time of the $n : 1$ implementation with 1 pixel per thread and $T_{\text{GPU}}^{1:1}(n)$ 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 $N_{\text{pixels}} = 1024 \times 512$ and $n = 2$ for $N_{\text{pixels}} = 2048 \times 1024$. Such exceptions are most likely justified by an efficient usage of GPU caches for the values of $n$ considered.
### 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

<table>
<thead>
<tr>
<th>#Pixels per Thread</th>
<th>1 p.t.</th>
<th>2 p.t.</th>
<th>4 p.t.</th>
<th>8 p.t.</th>
<th>16 p.t.</th>
<th>32 p.t.</th>
<th>64 p.t.</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU memory movement</td>
<td>1.5552</td>
<td>1.5412</td>
<td>1.6248</td>
<td>1.8299</td>
<td>1.5576</td>
<td>1.5215</td>
<td>2.4168</td>
</tr>
<tr>
<td>GPU Computation</td>
<td>91.4494</td>
<td>93.4583</td>
<td>100.5723</td>
<td>110.7912</td>
<td>136.6889</td>
<td>163.7567</td>
<td>264.4777</td>
</tr>
<tr>
<td>I/O</td>
<td>188.1964</td>
<td>195.0423</td>
<td>195.5758</td>
<td>213.836</td>
<td>234.663</td>
<td>257.8374</td>
<td>383.1248</td>
</tr>
<tr>
<td>Total</td>
<td>512x512</td>
<td>512x512</td>
<td>512x512</td>
<td>512x512</td>
<td>512x512</td>
<td>512x512</td>
<td>512x512</td>
</tr>
<tr>
<td>Relative Speedup</td>
<td>1</td>
<td>0.98</td>
<td>0.93</td>
<td>0.91</td>
<td>0.81</td>
<td>0.83</td>
<td>0.62</td>
</tr>
</tbody>
</table>

### 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

<table>
<thead>
<tr>
<th>#Pixels per Thread</th>
<th>1 p.t.</th>
<th>2 p.t.</th>
<th>4 p.t.</th>
<th>8 p.t.</th>
<th>16 p.t.</th>
<th>32 p.t.</th>
<th>64 p.t.</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU memory movement</td>
<td>2.6459</td>
<td>2.4389</td>
<td>2.4953</td>
<td>2.7343</td>
<td>2.4689</td>
<td>2.563</td>
<td>2.7396</td>
</tr>
<tr>
<td>GPU Computation</td>
<td>208.9538</td>
<td>169.1638</td>
<td>183.7097</td>
<td>225.108</td>
<td>251.4933</td>
<td>338.2722</td>
<td>491.5858</td>
</tr>
<tr>
<td>I/O</td>
<td>412.3497</td>
<td>325.6946</td>
<td>347.1873</td>
<td>417.8709</td>
<td>427.6576</td>
<td>526.065</td>
<td>677.2343</td>
</tr>
<tr>
<td>Total</td>
<td>1024x1024</td>
<td>1024x1024</td>
<td>1024x1024</td>
<td>1024x1024</td>
<td>1024x1024</td>
<td>1024x1024</td>
<td>1024x1024</td>
</tr>
<tr>
<td>Relative Speedup</td>
<td>1</td>
<td>1.24</td>
<td>1.14</td>
<td>0.93</td>
<td>0.83</td>
<td>0.62</td>
<td>0.43</td>
</tr>
</tbody>
</table>

### 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

<table>
<thead>
<tr>
<th>#Pixels per Thread</th>
<th>1 p.t.</th>
<th>2 p.t.</th>
<th>4 p.t.</th>
<th>8 p.t.</th>
<th>16 p.t.</th>
<th>32 p.t.</th>
<th>64 p.t.</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU memory movement</td>
<td>5.5768</td>
<td>6.2221</td>
<td>5.311</td>
<td>5.8691</td>
<td>5.5045</td>
<td>5.4143</td>
<td>5.6543</td>
</tr>
<tr>
<td>GPU Computation</td>
<td>427.8063</td>
<td>433.0888</td>
<td>433.1126</td>
<td>466.0709</td>
<td>563.8854</td>
<td>713.2571</td>
<td>1022.0434</td>
</tr>
<tr>
<td>I/O</td>
<td>850.8332</td>
<td>850.1319</td>
<td>817.2534</td>
<td>857.8179</td>
<td>969.9416</td>
<td>1111.8683</td>
<td>1420.5088</td>
</tr>
<tr>
<td>Total</td>
<td>2048x1024</td>
<td>2048x1024</td>
<td>2048x1024</td>
<td>2048x1024</td>
<td>2048x1024</td>
<td>2048x1024</td>
<td>2048x1024</td>
</tr>
<tr>
<td>Relative Speedup</td>
<td>1</td>
<td>0.99</td>
<td>0.99</td>
<td>0.93</td>
<td>0.77</td>
<td>0.65</td>
<td>0.55</td>
</tr>
</tbody>
</table>

### 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

<table>
<thead>
<tr>
<th>#Pixels per Thread</th>
<th>1 p.t.</th>
<th>2 p.t.</th>
<th>4 p.t.</th>
<th>8 p.t.</th>
<th>16 p.t.</th>
<th>32 p.t.</th>
<th>64 p.t.</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU Computation</td>
<td>817.3744</td>
<td>798.4245</td>
<td>848.9382</td>
<td>949.0839</td>
<td>1071.9428</td>
<td>1407.4057</td>
<td>2089.8141</td>
</tr>
<tr>
<td>I/O</td>
<td>1589.3778</td>
<td>1537.9058</td>
<td>1595.3008</td>
<td>1706.9155</td>
<td>1813.3124</td>
<td>2163.9425</td>
<td>2916.8232</td>
</tr>
<tr>
<td>Total</td>
<td>2048x1024</td>
<td>2048x1024</td>
<td>2048x1024</td>
<td>2048x1024</td>
<td>2048x1024</td>
<td>2048x1024</td>
<td>2048x1024</td>
</tr>
<tr>
<td>Relative Speedup</td>
<td>1</td>
<td>1.02</td>
<td>0.96</td>
<td>0.86</td>
<td>0.76</td>
<td>0.57</td>
<td>0.41</td>
</tr>
</tbody>
</table>

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

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: 
\[
\frac{T_{\text{GPU}}^{1:1}}{T_{\text{GPU}}^{n:1}}, \quad \text{where } T_{\text{GPU}}^{n:1} \text{ is the GPU computation time of the } n : 1 \text{ pixels’ mapping to threads version and } T_{\text{GPU}}^{1:1} \text{ is the GPU computation time of the 1 : 1 pixels’ mapping to threads version.}
\]

Table 15 reports a sub-linear speedup trend for \( n \), as the speedup increases substantially when considering \( n \geq 4 \).

- **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 \( N_{\text{pixels}} \) 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 \( N_{\text{pixels}} \) considered.

### 4.4 Map Fusion Optimization

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

![GPU Map parallelization](image)

**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 \( N_{\text{pixels}} \) 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.
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>
```
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

