Acceleration Of the CUDA Implementation of the Grayscale and Negative Filters

Dzigal Dzemil
dept. of Computing and Informatics
Fac. of Electrical Engineering Sarajevo
Sarajevo, Bosnia and Herzegovina
ddzigal1@etf.unsa.ba

Hodo Midhat
dept. of Computing and Informatics
Fac. of Electrical Engineering Sarajevo
Sarajevo, Bosnia and Herzegovina
mhodo1@etf.unsa.ba

Begic Lejla
dept. of Computing and Informatics
Fac. of Electrical Engineering Sarajevo
Sarajevo, Bosnia and Herzegovina
lbegic1@etf.unsa.ba

Borovac Selma
dept. of Computing and Informatics
Fac. of Electrical Engineering Sarajevo
Sarajevo, Bosnia and Herzegovina
sborovac1@etf.unsa.ba

Nosovic Novica
dept. of Computing and Informatics
Fac. of Electrical Engineering Sarajevo
Sarajevo, Bosnia and Herzegovina
nnosovic@etf.unsa.ba

***Abstract*—Real-Time Image Processing is a difficult, labor-expensive and challenging task using standard high-level Programming Languages such as C, C++, Java, C# and others. Compiler-level Optimisation ensures some acceleration of execution, but for the usual uses of Image Processing, such optimisations are not enough. Processors of the modern time assign the use of HyperThreading and similar technologies, to the extent that we can, without a reasonable doubt say that there are no truly sequential programs. Thus the parallel software is of the upmost importance to be written. After the optimal Work Arrangement for every worker in the system, code and compiler-level optimization should be implemented for the final goal of having an optimal Parallel Implementation, code and compiler-wise and work arrangement-wise. The methods presented in this paper are new methods for computing Grayscale and Negative filters which have been optimised in their code, using bit-wise operators and Compiler Restrictions on their assumptions, unlike other forms of optimization, and can be used for General Purpose Image processing.**

***Keywords*—Real-Time Image Processing, Compiler-level Optimisation, Work Arrangement, Parallel Cmplementation, Grayscale Filter, Negative Filter, Bit-Wise Operators, Compiler Restrictions General Purpose Image Processing**

I. Introduction

For the use in speed cameras [1] , the Grayscale filter is used as a preprocessing step in algorithms [2] in order to correctly identify vehicle licence plates. Because standard programming languages are sequential in nature, a parallel implementation of these preprocessing steps is needed to ensure that the feature of vehicle speed calculation of the speed camera is minimally effected. In this paper, a parallel implementation of The Grayscale filter using the CUDA environment is proposed, as well as the comparison between the sequential and parallel implementation in the sense of execution time, and the comparrison of execution time between the new proposed method over the existing implementations in the CUDA environment [2]. Code-level optimisation steps are also provided with the explanation why they are implemented. The Negative filter is used in standard image segmentation algorithms [3] for the simple reason that most image segmentation algorithms assume the background of an object it is trying to segment is black and the foreground (as too the said object) is white. So the Negative filter can be used to ease the segmentation of a certain darker object with a bright background. Altough rare, the Negative filter can be used for the purpose of enhancing low-brightness images to be processed more easily, and is generally used in this purpose in the field of Image Processing [4]. Thus, the new method is proposed with the final goal of having an open-source code usable by anyone that is optimized to the extent shown in this paper.

II. Related Work

A. Research in the field of Image Processing

Standardised filters such as Grayscale and Negative are studied througout the years by a myriad of researchers [2] [3] [4] [5] for their potential role in image processing. Some of the related work is based only on processing still images, but others on real-time video processing. Such is the purpose this paper is trying to strive to.

B. Cuda implementations

In the paper [2], implementation of the Grayscale filter is provided as is the implementation. What is important to keep in mind is that the latest implementations and advances in the optimisation of these two filters are not frequent nor up-to-date to the latest hardware standards and new optimisation techniques now available (not only for the parallel, but sequential programming too). All of the related work is made on small datasets, and the measurements that are proposed are used in comparrison to our method. All of the hardware differences are noted and is a thing to keep in mind when observing the given results of our method.

C. Note on the performance figures

The authors urge the reader to keep in mind that the difference between the performance of the GPU's of 10 years ago and a CPU's of 10 years ago in respect to performance is much more than the difference of performance of the average CPU's of today (2020) and the average GPU's of today. The results obtained in this paper are sufficient and to the extent of exellent for real-time processing of images in the use of Optical-Character-Recognition as the proposed method as a preprocessing step.

III. Our Method

Our methods are consisted of simple procedures both for Grayscale and Negative filters. The Grayscale filter is achieved by a pixel-wise operation described in equation 1.

$$P_{Grayscale}(x,y) = \frac{P_{Red}(x,y) + P_{Green}(x,y) + P_{blue}(x,y)}{3} \tag{1}$$

And so, the Negative filter is also achieved by a pixel-wise operation described in eq. 2.

$$P_{Negative,n}(x,y) = 255 - P_n(x,y) \tag{2}$$

where n stands for the channel in question that can be Red,Green or Blue. The Negative filter is achieved when eq. 2. is performed upon all channels for each pixel in the image.

A pseudocode that illustrates the functionallity of the two proposed filters is given in the sequential implementation. The pseudocode for the Grayscale and Negative filter are given in a C-style pseudocode for the purpose of ease of reading, and for the reader that wishes to implement our method to have an easy way to do so. The pseudocode is given as two functions implemented and whos mechanism of operation is as follows. The Grayscale filter is calculated as an average value of the R, G and B value of every pixel in the selected picture, saved to a single value in an array that is to be converted back to a single-channel image. The negative filter is calculated as the value of R G and B subtracted from the number 255 and saved to the corresponding R,G or B channels of the output image. Pseudocode is as follows in the following programms for their respective sequential implementation.

Listing 1. Pseudocode of the Grayscale filter

```
void convertToGrayscale
(unsigned char*input,unsigned char*output,
unsigned int img_size){

for(unsigned char*p = input,*pg = output;
p!=input+img_size;
p+=channels, pg += gray_channels) {
*pg=(uint8_t)((*p+*(p+1)+*(p+2))*constant);
if (channels == 4)
        *(pg + 1) = *(p + 3);
    }
}
```

Listing 2. Pseudocode of the Negative filter

```
void convertToNegative
(unsigned char*input,unsigned char*output,
unsigned int img_size){

for(unsigned char*p = input,*pn = output;
p!=input+img_size;
p+=3, pn+=3) {
*pn=(uint8_t)(255-*p);
*(pn+1)=(uint8_t)(255-*(p+1));
*(pn+2)=(uint8_t)(255-*(p+2));
}
```

Both of the procedures are as described in the beginning of the section. The next step is to analyse the granulation of the problem and divide the problem into smaller problems applicable to high level of parallelism. Note that in Image Processing, the ideal of a parallel Image Processing algorithm is of one that is in the form of "1 Pixel, 1 Processing-Element". For small images, the ideal is obtainable. But, given an image more than the capability of the given hardware, the ideal is no longer valid. Thus, the algorithm must resort to Thread-Switching. The CUDA environment is does not carry the overhead of manual Thread-Switching, and the NVCC compiler, as-is has a good mechanism of obtaining relatively good figures of Thread-Switching, Context-Switching and such technology exploitation to the level sufficient for this paper.

The main kernel is optimised not only by the compiler optimisation techniques, but unorthodox, cunning optimisation techniques using bit-wise shifting operators $>>$ and $<<$ for division and multiplication. The use of constants is exploited in the kernel also, as is the $\_\_restrict\_\_$ command for the use of forcing the compiler to not assume aliasing of the pointers in the kernel. Also, exploitation of the local constant cache is made in the code by declaring constant values for every value in the kernel. This is due to the fact that only the input_gray image will be altered by the algorithm. The kernel for the Grayscale and Negative filter are given in the next code segments.

The code is written for CUDA 10.1 and/or 10.2 environment and is given in the next listings in the form of their implementation. The code is also available at the author's GitHub account [6], as well as the sequential implementation. Keep in mind that the global preprocessing directive for the constant "constant" is given as **#define constant 0.3333f;**

Listing 3. Optimized CUDA kernel for the Grayscale filter

```
__global__
grayscale(const unsigned char*
__restrict__ input_rgb,
unsigned char*
__restrict__ input_gray,
const unsigned int resolution) {



const unsigned int
offset = (blockIdx.x*blockDim.x + threadIdx.x);


const unsigned int a = (offset << 1) + (offset);
const unsigned int b = a + 1;
const unsigned int c = a + 2;
const unsigned char a_a = input_rgb[a];
const unsigned char b_b = input_rgb[b];
const unsigned char c_c = input_rgb[c];
const unsigned int val = a_a + b_b + c_c;

input_gray[offset] = val * constant;

}
```

Listing 4. Optimized CUDA kernel for the Negative filter

```
__global__
void negative(const unsigned char*
__restrict__ input_rgb,
unsigned char*
__restrict__ output,
const int resolution) {

const unsigned int
offset=blockIdx.x*blockDim.x + threadIdx.x;
const unsigned int
a = (offset << 1) + (offset);

const unsigned int b = a + 1;
const unsigned int c = a + 2;
const unsigned int val_1 = 255 - input_rgb[a];
const unsigned int val_2 = 255 - input_rgb[b];
const unsigned int val_3 = 255 - input_rgb[c];
        output[a] = val_1;
        output[b] = val_2;
        output[c] = val_3;
}
```

IV. Results

Implementation and recording of the given algorithms and their results is done on a Lenovo Legion, Y520 laptop with a Core i5-7300HQ @2.5GHz, 8GB of DDR4 2400MHz RAM and an NVidia GTX 1050M GPU. The computer is running Windows 10 / Kali Linux, v. 2020.4 and is running the code given in the repository [6]

The dataset that the performance of the proposed methods is measured upon is the Corisian Fire Database [7] consisted of a 1135 images at the time of writing this paper in color of varying dimensions. For that reason, the Corisian Fire Database is a very good dataset to test the performance of the proposed methods. The time elapsed between the call to the function of the filter to its end in the sequential implementation, as well as the elapsed time from the call to the kernel on the device to its terminating is used as a valid time given in milliseconds to calculate and compare the performance gain on the parallel implementation of the algorithm. The performance measurement of the kernel is implemented with respect to NVidia's documentation concerning the said subject [8] . Starting with the Grayscale filter, being followed by the Negative filter, a chart of computation time is given as well as the chart of achieved speedup of parallel implementation over sequential with respect to the number of pixels in a particular image processed by the algorithms. A chart of execution time for the kernel with respect to the number of assigned blocks to the kernel over the whole dataset is given, implying the optimal number of blocks to be assigned in a general use-case, where the resolution of the given images is unknown. A conclusion is given on the use of such a parallel implementation with respect to the achieved performance gain and execution time in general-use Image Processing algorithms.

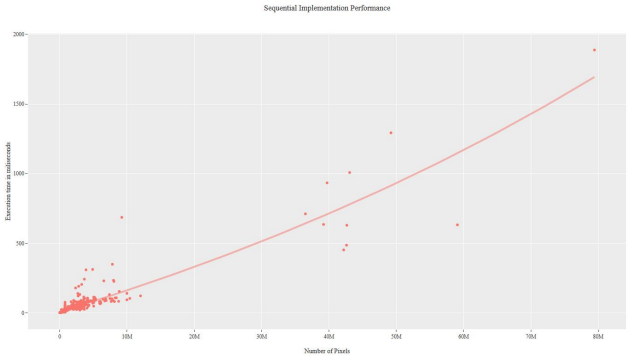

Fig. 1. Execution time of Sequential Grayscale

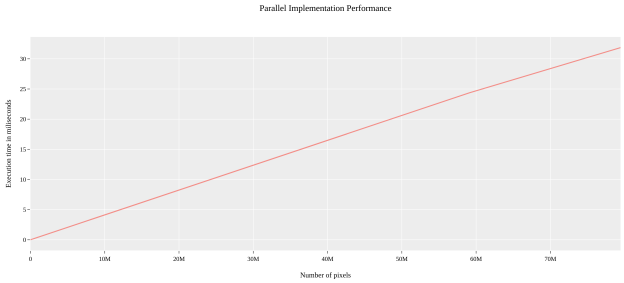

Fig. 2. Execution time of Parallel Grayscale

Thus we conclude that the speedup plot of the Parallel over Sequential implementation of the Grayscale filter of the proposed method in this paper is given in figure 3. Note that in fig. 3 and 1, fitted functions are provided that give a more

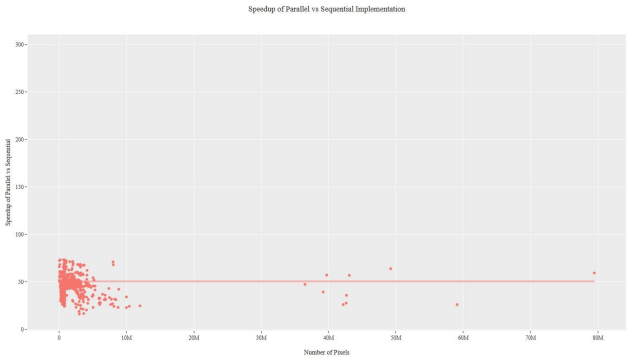

Fig. 3. Speedup of Parallel over Sequential Grayscale

intuitive insight into the meaning of the results. The fitted functions for the figures 3 and 1 are provided in the forms presented in the next equations.

$$f_{fitseq} = -1717.2 + 1724.39 \cdot e^{8.58 \cdot 10^{-9} \cdot x} \tag{3}$$

$$f_{fitspeedup} = 50.43 - 70.47 \cdot e^{-\frac{(x-42392613.71)^2}{2 \cdot 147849.62^2}} \tag{4}$$

Note that the fitted functions are both exponential in nature, but the parallel implementation function is approximately linear in nature. Note that the fitted function in the fig. 3 gives a good approximation of the adopted average speedup of 50.12. Next, the graph of performance of the parallel implementation of the algorithm in respect to the block dimensions is shown Note that the figures 1 2 3 are made with the blockDim.x

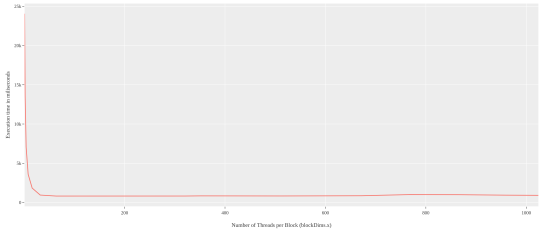

Fig. 4. Execution time in respect to blockDims.x

parameter set at 256 which is one of the local minima of the function displayed in figure 4. Thus concludes the analysis of the Parallel Grayscale Filter.

The negative filter has many uses in Image Processing so a new parallel method is proposed in this paper. Like the Grayscale filter, for the Negative filter, a chart of computation time for the sequential, parallel and the speedup between the two, where the average speedup is 25.34.

Thus we conclude that the speedup plot of the Parallel over Sequential implementation of the proposed method for the Negative filter in this paper is given in fig. 7 As before, the figure presented in fig. 4 holds true for the Negative Filter as well, so the parameter blockDims.x is set to 256, as well.

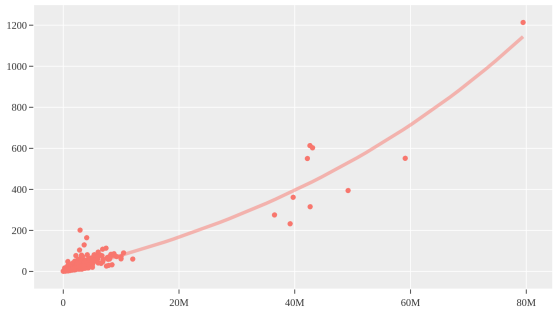

Fig. 5. Execution time of Sequential Negative

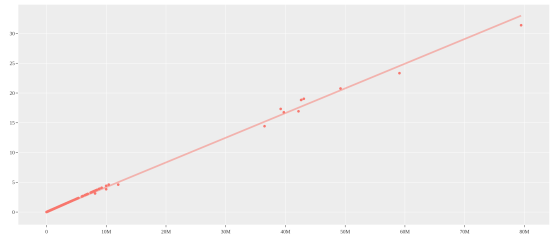

Fig. 6. Execution time of Parallel Negative

Note that the figures 5 6 7 are made with the blockDim.x parameter set at 256 which is one of the local minimi of the function displayed in figure 4. Thus concludes the analysis of the Parallel Negative Filter.

Some of the results of both filters are presented.

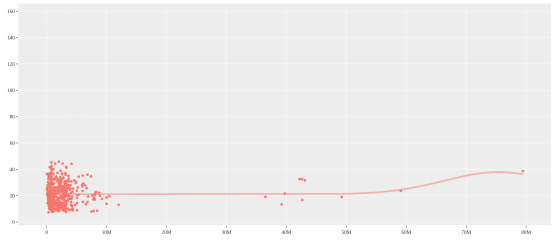

Fig. 7. Speedup of Parallel over Sequential Negative

V. Conclusion

In this paper, an optimisation for computing two basic filters of Image processing are proposed. In the form of a parallel implementation, using NVidia Graphics Processing Units, in this paper, it is shown that such an implementation can lead to a substantial amount of reduction of execution time of an algorithm, thus ensuring a considerable amount of speedup. A considerable amount of optimisation is needed to further reduce the runtime of the parallel and likewise the sequential

execution of the algorithms. bit-wise shifting, constant variable cache exploitation, compiler directives, Instruction-Level Parallelism exploiting, constant exploiting, avoiding branching and a myriad of similar Code-Level optimisation techniques were needed in order to speed up the code even further than the initial implementation. Why they are needed is because of a tendency for a compiler to not assume anything, even tough a problem that it assumes is going to happen has a very low probability of actually happening. Thus the techniques proposed in this paper are needed to evade imperatives of unoptimized and assumptive compilers. The use of a such parallel implementation is obvious in Computer Vision and Image Processing. The proposed method in this paper is consisted of parallel software, written for parallel hardware which in and of itself allows for streaming of FHD (and more) resolution in a frame rate akin to a minimum of 60 Frames Per Second. This ensures maximum throughput and speed of processing. The existing methods presented in [2] are identical to the implementation in this paper and the results can be reviewed in [6].

Fig. 8. Results of both filters 1

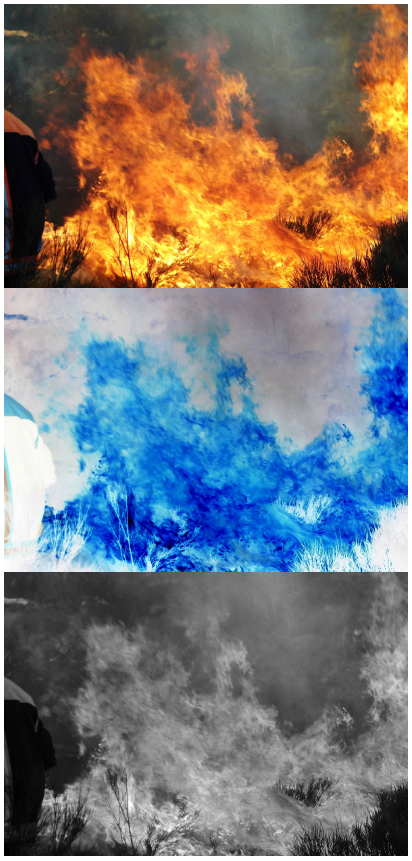

Fig. 9. Results of both filters 2

REFERENCES

[1] "Gatso speed cameras explained," https://www.speedcamerasuk.com/gatso.htm, 2020.

[2] W. Diniz, A. Horta, E. Nobrega, and L. Ferreira, "Parallel implementation of grayscale conversion in graphics hardware," 11 2011.

[3] M. Saha, M. Darji, N. Patel, and D. Thakore, "Implementation of image enhancement algorithms and recursive ray tracing using cuda," *Procedia Computer Science*, vol. 79, pp. 516–524, 12 2016.

[4] J. J. Tse, "Image processing with cuda," pp. UNLV Theses, Dissertations, Professional Papers, and Capstones. 1699., 2012.

[5] C. Ameli, "Parallel computing with cuda in image processing," October 2017.

[6] H. B. B. Dzigal, "Cuda grayscale and negative implementation," https://github.com/ddzigal1/CUDA-Grayscale-and-Negative-Implementation, 2019.

[7] T. Toulouse, L. Rossi, A. Campana, T. Celik, and M. Akhloufi, "Computer vision for wildfire research: An evolving image dataset for processing and analysis," *Fire Safety Journal*, vol. 92, pp. 188–194, 07 2017.

[8] "Cuda toolkit documentation," https://docs.nvidia.com/cuda/index.html, November 2019.