PARALLEL SOBEL EDGE DETECTION

Pontus Wedén & Rickard Lundberg

{pwn15002, rlg15001}@student.mdh.se

Mälardalen University, School of Innovation, Design and Engineering.

1 ABSTRACT

This report describes the implementation and parallelization of the Sobel operator edge detection algorithm. We implemented a sequential solution to the problem, which we then use to compare execution times with a parallel solution, implemented with Nvidia CUDA. We found that for our CUDA implementation, memory allocation and transfer take up the largest fraction of the execution time. Although, our CUDA implementation is still up to 6.2 times faster than the sequential implementation.

2 BACKGROUND

Edge detection is an important part of computer vision and feature detection, and has been since the early 1970s [1]. It is widely used as a precursory step for extracting information in image processing and computer vision applications [2].

The Sobel operator was first described in a talk given by Irwin Sobel and Gary Feldman in 1968 [3]. We chose to parallelize the Sobel operator since when using it, each pixel is handled individually, with no dependencies on previous pixel calculations. This means that in theory, all race conditions are eliminated, which makes it a good candidate for parallelization. Adding to that, as image resolution and video frame rates increase the need for a high-performance edge detection solution is even greater.

3 PROBLEM DESCRIPTION

The main component of the algorithm is based on calculating the Hadamard product of two 3x3 matrices for each pixel and use the result as addends in Pythagoras's theorem to get the pixel's final value. This part of the algorithm is called Sobel Operator. To be able to utilize the Sobel operator, the image it is applied on must be grayscale, since the edge detection is based on finding the transitions between darker and lighter areas. In addition to that, a slightly blurred image often produces a cleaner result since it eliminates some noise which otherwise could be detected as edges. Therefore our final implementation consists of the following steps (visualized in Figure 1):

1. Image to Grayscale
2. Gaussian Blur
3. Sobel Operator (Vertical and horizontal derivate, and pytahgorean)

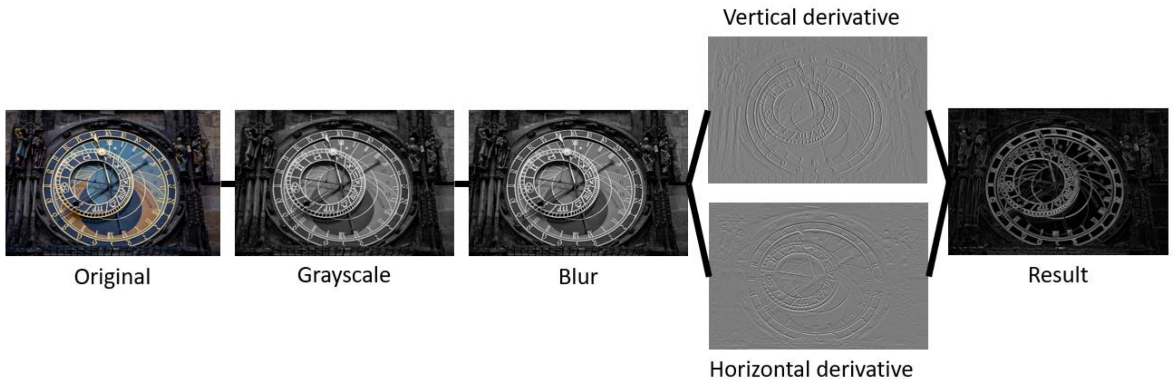


Figure 1. Graphical overview of the edge detection algorithm.

In addition to the aforementioned steps, we also chose to normalize the images pixel values. The resulting image from the algorithm can be a bit dark. Our normalization function remedies this by mapping the pixel values to range from 0 to 255.

3.1 REFERENCE IMPLEMENTATION

To have a reference for our solution, we implemented a sequential version. The code is optimized to maximize speed (using the /O2 flag) at compile time. To further optimize the sequential solution, the loops are ordered for optimal cache performance. Moreover, we have tried to reduce any unnecessary computations and function calls. Replacing unnecessary computations with constants and eliminating redundant computations improved the measured times considerably, although we could not measure any significant differences in the execution times when inlining function calls. The explanation for this, we assume, is that since the compiler already does lot of optimizations at compile time, the functions are already inlined at runtime. For the sequential version, we have also unrolled the loops in the blur, Sobel, Pythagorean and normalization functions, which yielded a considerable improvement to the execution time of the sequential solution. Our sequential reference solution is faster than the example implementation given by OpenCV, as can be seen in Figure 5, and we think it gives a fair comparison to the parallelized implementation.

4 METHOD

For our parallel solution, we chose to use GPU parallelization, more particularly NVIDIA CUDA. The implementation was done with native CUDA programming in C. Our tests was conducted on a computer with an Intel Core i7-4770K 3.5 GHz CPU, 16 GB RAM, and an NVIDIA GTX 780Ti GPU.

5 SOLUTION

The problem involves several distinct steps, which must be done in series because the correctness of the value of a pixel is dependent on pixel values calculated by a previous step. This means that we could not fully exploit the benefit of parallelizing the algorithm since each step must be executed in a sequence. The parallelization is therefore on a pixel level, wherein each kernel one thread is assigned for each pixel to perform the calculation.

We divided the steps of the algorithm so that each step is executed by a specific kernel, where each kernel waits for the previous kernel to complete. The blur step is dependent on grayscale, the Sobel derivation is dependent on blur, and the Pythagorean gradient magnitude approximation is dependent on the Sobel derivation. The only two kernels that are not dependent on each other are the Sobel calculations since they use the same source image and produce two different results. Although, we compute the Sobel derivations in series in our implementation, since experiments with running them in parallel did not yield any significant time improvements.

In addition to the previous steps, we also implemented a normalize kernel. This step is not required for edge detection, but it makes the edges in the final image clearer by remapping the images pixel values to the range 0-255. The previous steps already guarantee that no values will be greater than 255, but the remapping intensifies the colors if the max pixel value is a lot less than 255.

5.1 MEASUREMENTS

We conducted our tests on five images with the longest side being 500, 1000, 1600, 3200 and 4100 pixels, and an aspect ratio of 3:2. Our gathered results show an advantage for the parallelized computation even for small images, with the achieved speedup increasing as the total amount of pixels grow. The reported times, seen in Figure 2, are the average of five runs for each width with the sequential and CUDA implementation.

500 px		1000 px		1600 px		3200 px		4100 px	
CUDA	Seq	CUDA	Seq	CUDA	Seq	CUDA	Seq	CUDA	Seq
2.59 ms	4.14 ms	4.71 ms	15.44 ms	8.31 ms	39.31 ms	24.47 ms	149.33 ms	38.02 ms	237.79 ms

Figure 2. Comparing execution times for different images sizes.

The required time to compute the edges increase as the images get larger, both for CUDA and sequential, although, the achieved speedup also increases as the images get larger. The speedup of the CUDA implementation over the sequential one can be seen in Figure 3. For a detailed comparison of the implementations, we also measured the execution time of each individual step (function) in the algorithm.

As can be seen Figure 4, the memory allocation and transfer time for the CUDA implementation contribute to 85,1% of the total execution time, while the actual computation time only contributes to 11,7% of the entire execution time. For the sequential implementation, however, there is no significant execution time for memory or data transfer. Instead, almost the entire execution time comes from calculating the different edge detection steps.

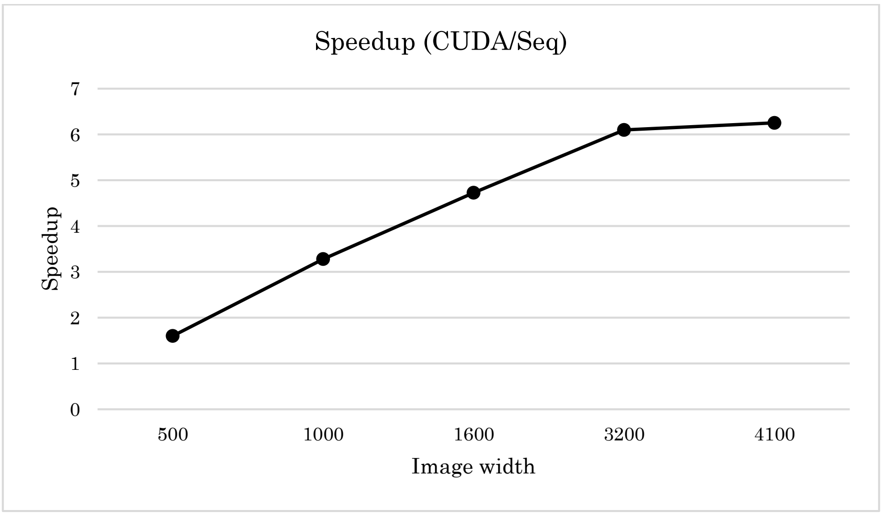


Figure 3. CUDA speedup compared to the reference sequential implementation.

CUDA Operation	Time	% of total
Memory allocation	13.872 ms	31.1%
Data transfer to device	15.821 ms	35.5%
Grayscale	0.851 ms	1.9%
Blur	1.098 ms	2.5%
Horizontal derivative	1.011 ms	2.3%
Vertical derivative	0.989 ms	2.2%
Pythagorean	0.679 ms	1.5%
Normalize	0.580 ms	1.3%
Data transfer to host	8.229 ms	18.5%
Free memory	0.996 ms	2.2%
CUDA total time	44.549 ms	100.0%

Sequential Operation	Time	% of total
Memory Allocation	0.166 ms	0.1%
Grayscale	19.614 ms	7.7%
Blur	61.598 ms	24.3%
Horizontal derivative	44.373 ms	17.5%
Vertical derivative	43.773 ms	17.3%
Pythagorean	66.986 ms	26.4%
Normalize	9.385 ms	3.7%
Free memory	7.177 ms	2.8%
Seq total time	253.488 ms	100.0%

CUDA	Time	% of total
Memory and transfer	37.922 ms	85.1%
Kernels total	5.208 ms	11.7%
Overhead	1.419 ms	3.2%

Seq	Time	% of total
Memory and transfer	7.343 ms	2.9%
Functions total	245.729 ms	96.9%
Overhead	0.416 ms	0.2%

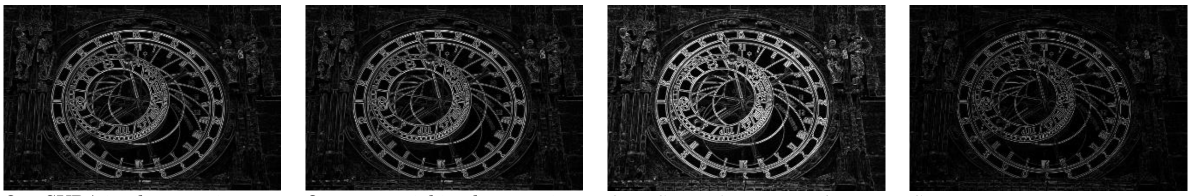
Figure 4. Per function time measuring for both CUDA and sequential implementation.

5.2 OPTIMIZATIONS

We have experimented with different optimizations for the CUDA solution. Like the sequential, we moved to using constants instead of calculated values where applicable, which yielded a small execution time improvement. Also, akin to the sequential implementation, we tried to implement loop unrolling in the CUDA kernels, although this did not improve our execution times significantly as it did in the sequential implementation. To eliminate kernel call overhead, we also experimented with having fewer, but larger, kernels, although this resulted in longer execution times. In general, since the actual computations only contribute to ~12% of the total execution time, we found that any optimization done to them only yielded small improvements at best.

5.3 CORRECTNESS

To verify our results, we mainly had to rely on visual inspection of the results and comparing them to reference material of other Sobel edge computations. We also compare each pixel of the images produced by the sequential and CUDA implementation to check for possible deviances between the two. Our resulting images corresponds with the result from Sobel edge detection done by other image processing applications. However, as presented in Figure 5, all results differ to some degree. One reason for the differences could be that the algorithm does not specify if or how to map the pixel values to produce the final image. Additional images from our implementation are shown in Figure 6.



Our CUDA implementation
Execution time: 44.5ms

Our sequential implementation
Execution time: 253.5ms

OpenCV implementation[1]
Execution time: 490.9ms

GIMP
Execution time: n/a

Figure 5. A comparison of the result from running Sobel edge detection with different implementations on the same image.

6 RUNNING THE PROGRAM

The program was built using Microsoft Visual Studio 2017 and OpenCV 3.3.1. The project files for Visual Studio are included together with this report, and requires an installation of either Visual Studio 2017 or 2015.

To run the program from Visual Studio, OpenCV must be installed, and the Visual Studio project must be configured, below are the steps for doing this on a 64-bit installation of Windows 10. We suggest configuring for release since this is the configuration we used when measuring times.

1. OpenCV can be found at the address: www.opencv.org/releases.html, for this project we used version 3.3.1.
2. Once downloaded, OpenCV can be extracted to a directory of choice.
3. Setup the environment variables by pressing Win + R and running sysdm.cpl.
   a. Go to advanced, and then environment variables.
   b. Under System Variables, find Path and press edit.
   c. Press new and add the complete path to the 64-bit OpenCV binaries, for example C:\opencv\build\x64\vc14\bin.
4. Setup the dependencies in visual studio, start by right-clicking the project in the solution explorer and select Properties.
   a. Under C/C++ > General > Additional Include Directories, add the include folder from OpenCV. For example: C:\opencv\build\include.
   b. Under Linker > General > Additional Library Directories, add the lib folder from OpenCV. For example C:\opencv\build\x64\vc14\lib.
   c. Under Linker > Input > Additional Dependencies, add either opencv_world331d.lib if you are configuring for debug, or opencv_world331.lib if you're configuring for release.
5. Build and run the program from Visual Studio.

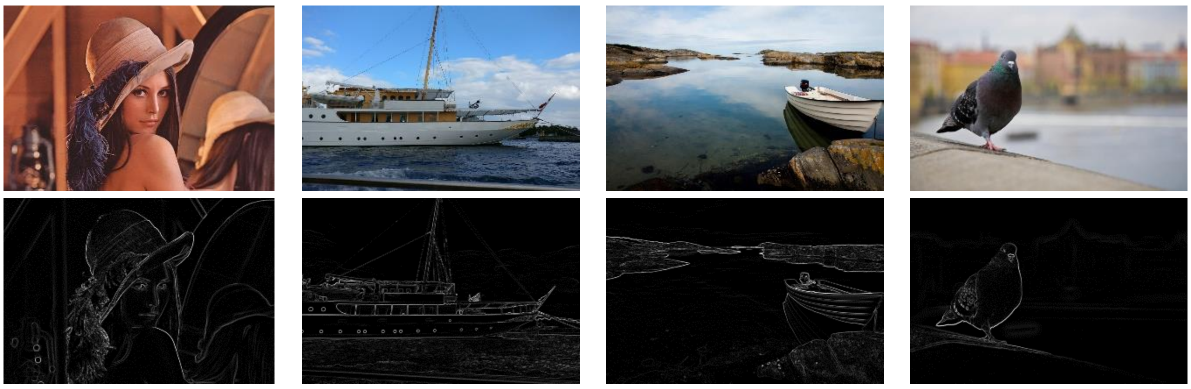


Figure 6. Results from our parallel Sobel edge detection implementation

7 CONCLUSIONS

Before we started to design our parallel solution, our expectation was that it should be possible to calculate each pixel independently. As the work progressed we realized that some calculations are dependent on adjacent pixels being done with their previous calculation. This led to us splitting up the problem into steps in a pipes-and-filters fashion. The parallelization was then local within each step. Our tests show that the CUDA implementation outperforms the sequential implementation, with a speedup ranging from 1.6 to 6.2.

[1] Code example from OpenCV documentation, https://docs.opencv.org/2.4/doc/tutorials/imgproc/imgtrans/sobel_derivatives/sobel_derivatives.html

As we expected, the parallel solution executes faster than the sequential. Although we expected the parallel solution to run faster, the speed of which the computations were done on the GPU was somewhat surprising. The final values are calculated in just above 5.2 ms for an 11.2 Megapixel image. However, the memory allocation on the GPU and the data transfer to and from the device added another 37.9 ms on the total execution time.

The limitations of our parallel solution of Sobel edge detection is that it had to be broken down into sequential steps. Another factor that limits our results are the overhead from memory management and data transfer which comes from the choice of implementing the algorithm on a GPU.

8 REFERENCES

[1] P. Dollár och L. C. Zitnick, ”Fast Edge Detection Using Structured Forests, ”*IEEE Transactions on Pattern Analysis and Machine Intelligence,* vol. 37, nr 8, pp. 1558-1569, 2015.

[2] H. S. Neoh och A. Hazanchuk, ”Adaptive Edge Detection for Real-Time Video Processing using FPGAs,” *Global Signal Processing,* vol. 7, nr 3, pp. 2-3, 2004.

[3] I. Sobel, An Isotropic 3x3 Image Gradient Operator. Presentation at Stanford A.I. Project 1968, 2014.