

Contents

Abstract.....	2
Introduction.....	2
Process of Canny Edge Detection algorithm	3
Gaussian filter	3
Finding the intensity gradient of the image	4
Non-maximum suppression	4
Double threshold.....	5
Edge tracking by hysteresis	5
Result	7
Serial Implementation.....	7
Execution time for serial implementation.....	7
OpenMP Implementation.....	8
Comparison of the execution time from the number of CPU threads.....	8
OpenCL Implementation	10
Comparison of the execution time from different block sizes	10
CUDA Implementation.....	12
Comparison of the execution time from different block sizes	12
Discussion.....	14
Comparison of execution time from different implementations	14
Current limitation and future works.....	16
References.....	17

Abstract

Edge Detection is a topic of importance for computer vision and in general, image processing. Detection of edges allows distinct characteristics of 3-D space to be identified with reduced storage and processing overhead. Processing speed is critical in popular technical disciplines which require real-time image analysis, such as automotive driving assistance systems and facial recognition. The Canny Edge Detection algorithm is unique within the edge detection field because it drastically reduces the amount of data that needs to be processed relative to other first-order image processing algorithms. This report gives an overview of the Canny Edge Detection algorithm, as well as the CUDA, OpenCL, OpenMP programming model. Results will be visualised and discussed thoroughly. Current limitation and future works to improve the performance will be addressed as well.

Introduction

Edge detection is a mathematical method of determining the “edges” within an image. An “edge” is defined as a point which the brightness of the image has discontinuities. Sharp changes in pixel intensity allow the edge detection algorithm to map an outline of the image being observed, and these edges can correspond to changes in depth, orientation, materials, or lighting. Mapping the edges of an image is a nontrivial task, as many false edges can be mapped, or segments of a real edge can be left off if the change is not drastic enough for the algorithm to pick up. The algorithm implemented in this paper is the Canny Edge Detection algorithm. This algorithm is a first-order image processing algorithm that was developed by John F. Canny in 1986 (Canny, 1986) and follows a 5-step detection algorithm:

1. Apply a Gaussian filter
2. Find the intensity gradient
3. Apply non-maximum suppression
4. Apply a double threshold
5. Track the edges using hysteresis

The size of the Gaussian filter and the double threshold can be manipulated in order to affect both the computation time and the effectiveness of the algorithm. Based on the image being processed, different combinations of the two may yield more “accurate” results. Edge detection using a program that runs serially causes the program to run very slow, especially as the number of pixels increases because the amount of work is directly proportional to the number of pixels in the image. In real-time image processing analysis, a serial implementation will be much too slow. Edge detection can be parallelised to drastically speed up computation time, so this report details how the algorithm can be

parallelised using the parallel computing platform CUDA, OpenCL and OpenMP. CUDA was chosen as the parallelisation platform because it has been designed/optimised for graphics and image processing falls into that category.

Process of Canny Edge Detection algorithm

The Process of Canny edge detection algorithm can be broken down to 5 different steps: (Canny, 1986)

1. Apply Gaussian filter to smooth the image to remove the noise
2. Find the intensity gradients of the image
3. Apply non-maximum suppression to get rid of spurious response to edge detection
4. Apply double threshold to determine potential edges
5. Track edge by hysteresis: Finalise the detection of edges by suppressing all the other edges that are weak and not connected to strong edges.

Gaussian filter

Since all edge detection results are easily affected by the noise in the image, it is essential to filter out the noise to prevent false detection caused by it. To smooth the image, a Gaussian filter kernel is convolved with the image. This step will slightly smooth the image to reduce the effects of obvious noise on the edge detector. The equation for a Gaussian filter kernel of size $(2k+1) \times (2k+1)$ is given by:

$$H_{ij} = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{(i - (k+1))^2 + (j - (k+1))^2}{2\sigma^2}\right); 1 \leq i, j \leq (2k+1)$$

Here is an example of a 5×5 Gaussian filter, used to create the adjacent image, with $\sigma = 1$. (The asterisk denotes a convolution operation.)

$$\mathbf{B} = \frac{1}{159} \begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix} * \mathbf{A}.$$

It is important to understand that the selection of the size of the Gaussian kernel will affect the performance of the detector. The larger the size is, the lower the detector's sensitivity to noise.

Additionally, the localisation error to detect the edge will slightly increase with the increase of the Gaussian filter kernel size. A 5×5 is a good size for most cases, but this will also vary depending on specific situations.

Finding the intensity gradient of the image

An edge in an image may point in a variety of directions, so the Canny algorithm uses four filters to detect horizontal, vertical and diagonal edges in the blurred image. The edge detection operator (such as Roberts, Prewitt, or Sobel) returns a value for the first derivative in the horizontal direction (G_x) and the vertical direction (G_y). From this, the edge gradient and direction can be determined:

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

$$\Theta = \text{atan2}(G_y, G_x),$$

where G can be computed using the hypot function and atan2 is the arctangent function with two arguments. The edge direction angle is rounded to one of four angles representing vertical, horizontal and the two diagonals (0° , 45° , 90° and 135°). An edge direction falling in each colour region will be set to a specific angle value, for instance, θ in $[0^\circ, 22.5^\circ]$ or $[157.5^\circ, 180^\circ]$ maps to 0° .

Non-maximum suppression

Non-maximum suppression is an edge thinning technique. It is applied to find the locations with the sharpest change of intensity value. The algorithm for each pixel in the gradient image is:

1. Compare the edge strength of the current pixel with the edge strength of the pixel in the positive and negative gradient directions.
2. If the edge strength of the current pixel is the largest compared to the other pixels in the mask with the same direction (e.g., a pixel that is pointing in the y-direction will be compared to the pixel above and below it in the vertical axis), the value will be preserved. Otherwise, the value will be suppressed.

In some implementations, the algorithm categorises the continuous gradient directions into a small set of discrete directions, and then moves a 3×3 filter over the output of the previous step (that is, the edge strength and gradient directions). At every pixel, it suppresses the edge strength of the centre pixel (by setting its value to 0) if its magnitude is not greater than the magnitude of the two neighbours in the gradient direction. For example:

- if the rounded gradient angle is 0° (i.e. the edge is in the north-south direction) the point will be considered to be on the edge if its gradient magnitude is greater than the magnitudes at pixels in the **east and west** directions,
- if the rounded gradient angle is 90° (i.e. the edge is in the east-west direction) the point will be considered to be on the edge if its gradient magnitude is greater than the magnitudes at pixels in the **north and south** directions,
- if the rounded gradient angle is 135° (i.e. the edge is in the northeast-southwest direction) the point will be considered to be on the edge if its gradient magnitude is greater than the magnitudes at pixels in the **north west and south-east** directions,
- if the rounded gradient angle is 45° (i.e. the edge is in the north west–south east direction) the point will be considered to be on the edge if its gradient magnitude is greater than the magnitudes at pixels in the **north east and south west** directions.

In more accurate implementations, linear interpolation is used between the two neighbouring pixels that straddle the gradient direction. For example, if the gradient angle is between 89° and 180° , interpolation between gradients at the **north** and **north east** pixels will give one interpolated value, and interpolation between the **south** and **south west** pixels will give the other (using the conventions of the last paragraph). The gradient magnitude at the central pixel must be greater than both of these for it to be marked as an edge.

Note that the sign of the direction is irrelevant, for example north–south is the same as south–north and so on.

Double threshold

The previous three filters were all applied to the image to detect the true edges, but along the edges, there still may be noise or spurious edges. The double threshold filter applies a high and a low threshold across the image to determine the strong and weak edges. If a pixel's value is greater than the high threshold, it is marked as a strong edge, if a pixel's value is less than the high threshold but greater than the low threshold, it is marked as a weak edge, and lastly, if a pixel's value is less than the low threshold, the pixel is suppressed completely. This filter is used to eliminate the last bit of noise.

Edge tracking by hysteresis

In addition to detecting the edges, all four of the previous filters have filtered noise that is present in the image. Because there may be some edges that were filtered too severely (for example, edge information could have been blurred away in the image during the Gaussian blur), the edge tracking

hysteresis filter makes one final pass over the image and connects edges that should have been connected. Using the strong edges that were determined by the double threshold filter, the edge tracking hysteresis filter passes over the image using a 3x3 matrix that connects weak edges adjacent to strong edges. Doing so re-establishes edges that should have been detected but were dropped by aggressive filtering. After this filter finishes, the edges in the image have been properly detected.



Example of photos applied with canny edge detection

Result

All the following metrics are the average of 5-time record to ensure accuracy of the data.

Hardware used

- NVIDIA Geforce GTX 1070 8GB VRAM
- 16GB RAM 2666Mhz
- AMD Ryzen 5 2600 3.4 GHz
- SATA SSD

Serial Implementation

Execution time for serial implementation

Image Dimension	Serial Execution Time(ms)	Increase in pixel count	Increase in execution time
640x480 (307200px)	82.01	1x	1x
1280x720 (921600px)	231.23	3x	2.81x
1920x1080 (2073600px)	530.56	6.75x	6.46x
2560x1440 (3686400px)	1100.21	12x	13.41x
3840x2160 (8294400px)	2007.23	27x	24.48x



As we can see, when the pixel count (problem size) increases, the execution time increases as well.

When the pixel count increase, the CPU has to iterate over more number of pixels to do computation.

OpenMP Implementation

Comparison of the execution time from the number of CPU threads

Image Dimension	Serial time(ms)	Execution time (ms) for different number of threads				
		2	4	6	12	24
640x480 (307200px)	82.01	44.39	29.22	24.32	23.47	28.98
1280x720 (921600px)	231.23	129.94	77.56	73.31	71.87	76.98
1920x1080 (2073600px)	530.56	302.28	200.3	172.15	143.09	167.37
2560x1440 (3686400px)	1100.21	609.99	405.8	356.68	325.52	354.08
3840x2160 (8294400px)	2007.23	1113.2	842	690.68	594.92	641.89
Average	790.248	439.96	311	263.43	231.77	253.86

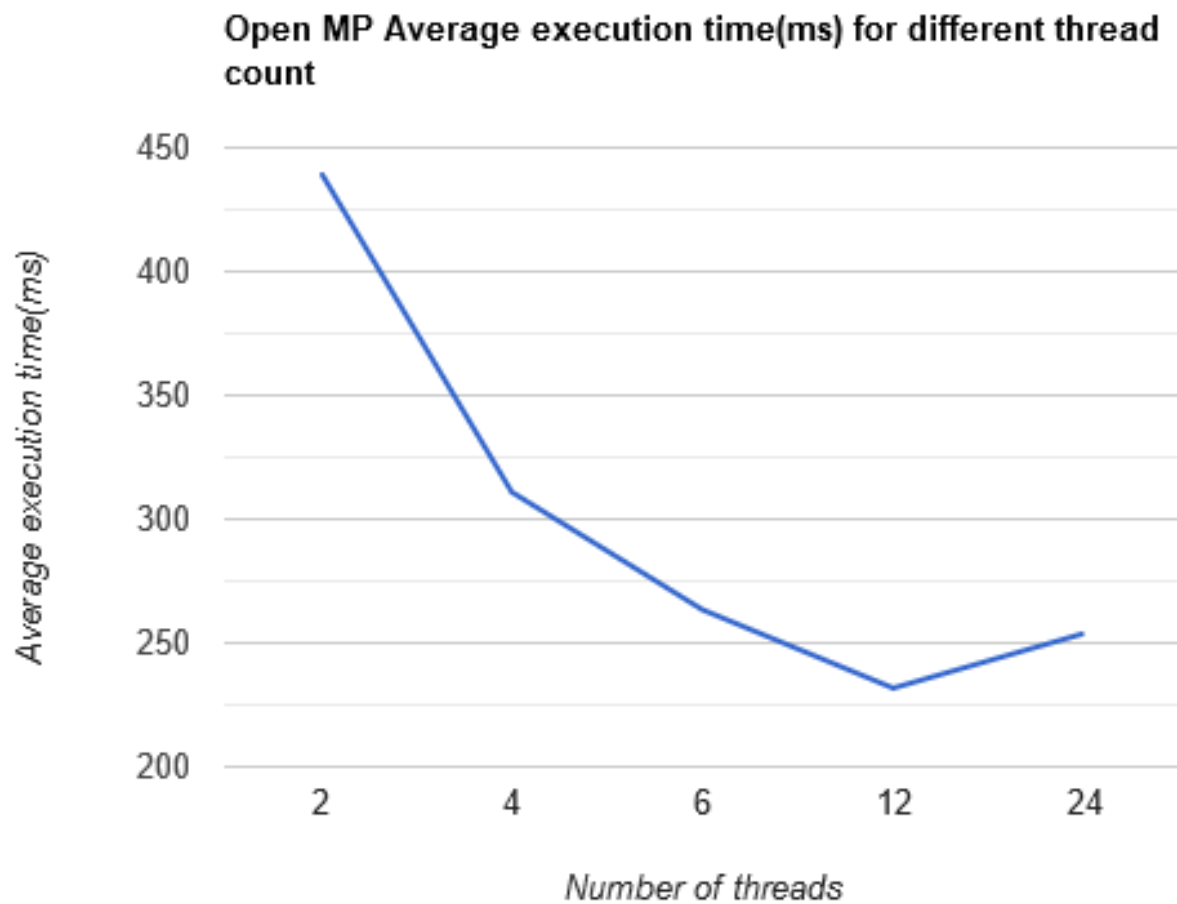
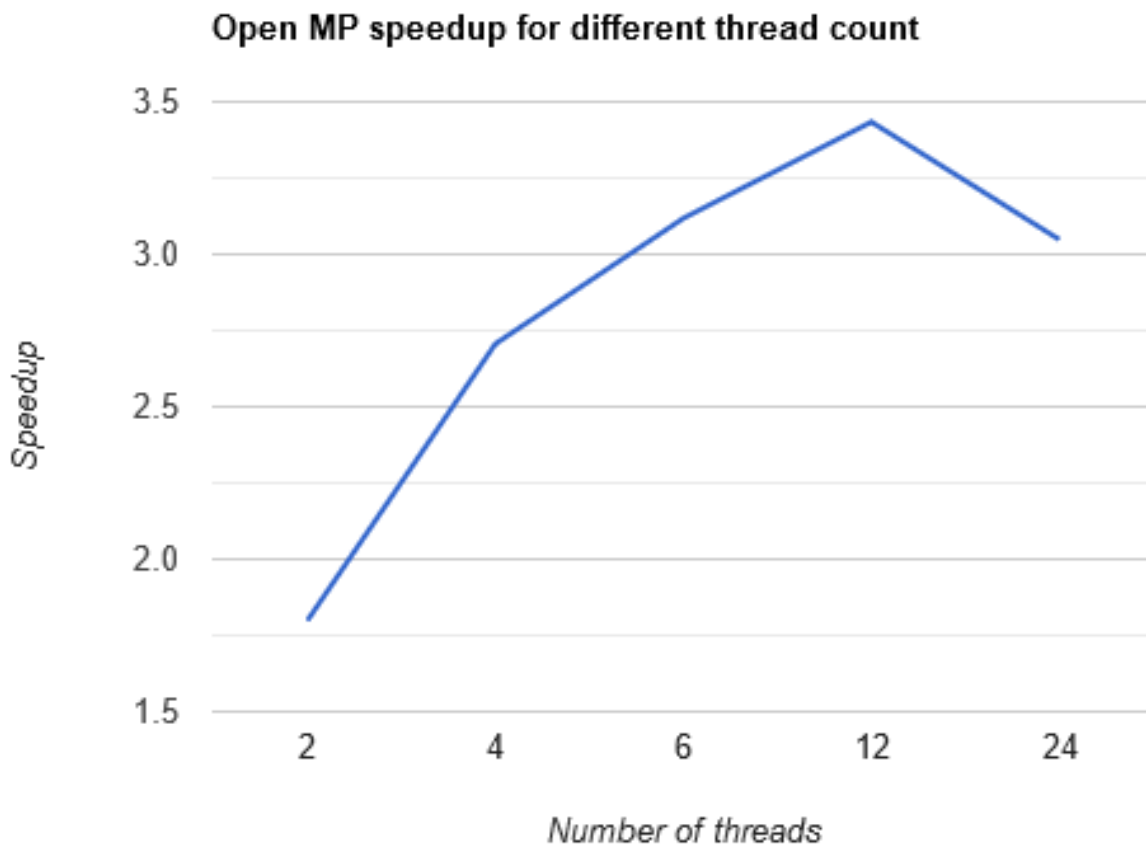


Image Dimension	Speedup for different number of threads				
	2	4	6	12	24
640x480 (307200px)	1.85x	2.81x	3.37x	3.49x	2.83x
1280x720 (921600px)	1.78x	2.98x	3.15x	3.22x	3.00x
1920x1080 (2073600px)	1.76x	2.65x	3.08x	3.71x	3.17x
2560x1440 (3686400px)	1.8x	2.71x	3.08x	3.38x	3.11x
3840x2160 (8294400px)	1.8x	2.38x	2.91x	3.37x	3.13x
Average	1.798x	2.706x	3.118x	3.434x	3.048x

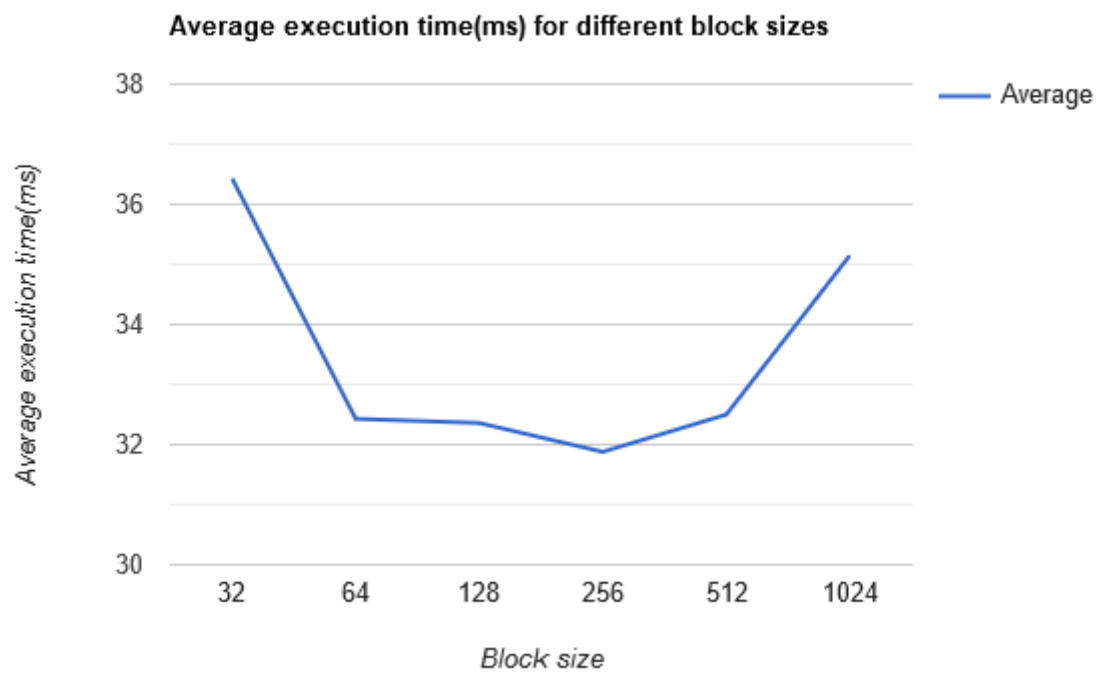


As we can see from the result above, the optimal number of CPU threads is 12 threads because it has the best performance (231.77ms and 3.434x speedup on average). This is very logical and sensical since our running CPU is AMD Ryzen 5 2600 which have 6 cores 12 threads. The performance will be at its best when we utilise all the available threads. Performance is worse at 24 threads because context switching will be needed, hence decreases the performance. For evaluation at later part, we will be using 12 threads for comparison since it offers the best performance. The shape of the line graph does not follow either Amdahl's law or Gustafson's law.

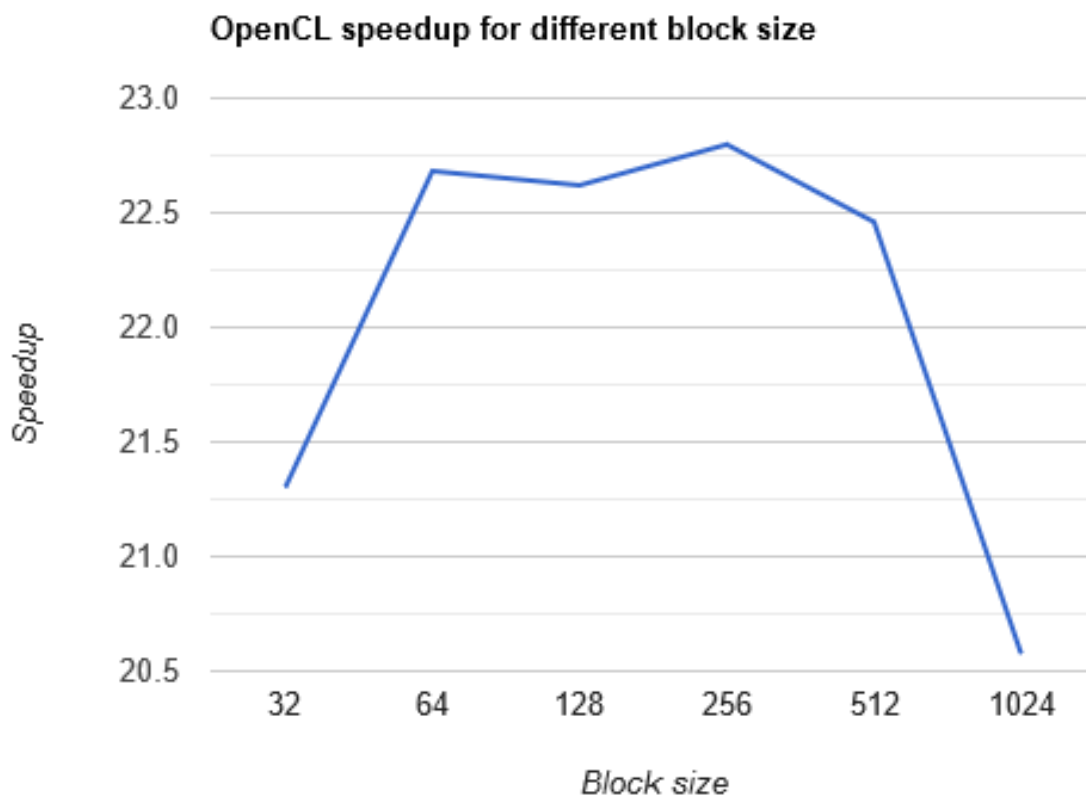
OpenCL Implementation

Comparison of the execution time from different block sizes

Image Dimension	Execution time (ms) for different block size					
	32	64	128	256	512	1024
640x480 (307200px)	5.99	4.62	4.77	4.83	4.91	6.28
1280x720 (921600px)	11.73	10.76	10.84	10.91	10.78	12.15
1920x1080 (2073600px)	23.42	22.62	23.21	23.12	23.56	24.5
2560x1440 (3686400px)	40.96	40.13	39.92	38.22	39.9	42.47
3840x2160 (8294400px)	85.08	84.02	83.04	82.32	83.34	86.39
Average	36.436	32.43	32.36	31.88	32.5	35.158



	Speed up for different block size					
Image Dimension	32	64	128	256	512	1024
640x480 (307200px)	13.69x	17.75x	17.19x	16.7x	16.7x	13.06x
1280x720 (921600px)	19.71x	21.49x	21.33x	21.19x	21.44x	19.03x
1920x1080 (2073600px)	22.65x	22.86x	22.85x	22.94x	22.51x	21.65x
2560x1440 (3686400px)	26.86x	27.42x	27.56x	28.78x	27.57x	25.90x
3840x2160 (8294400px)	23.59x	23.89x	24.17x	24.38x	24.08x	23.23x
Average	21.3x	22.682x	22.62x	22.798x	22.46x	20.574x

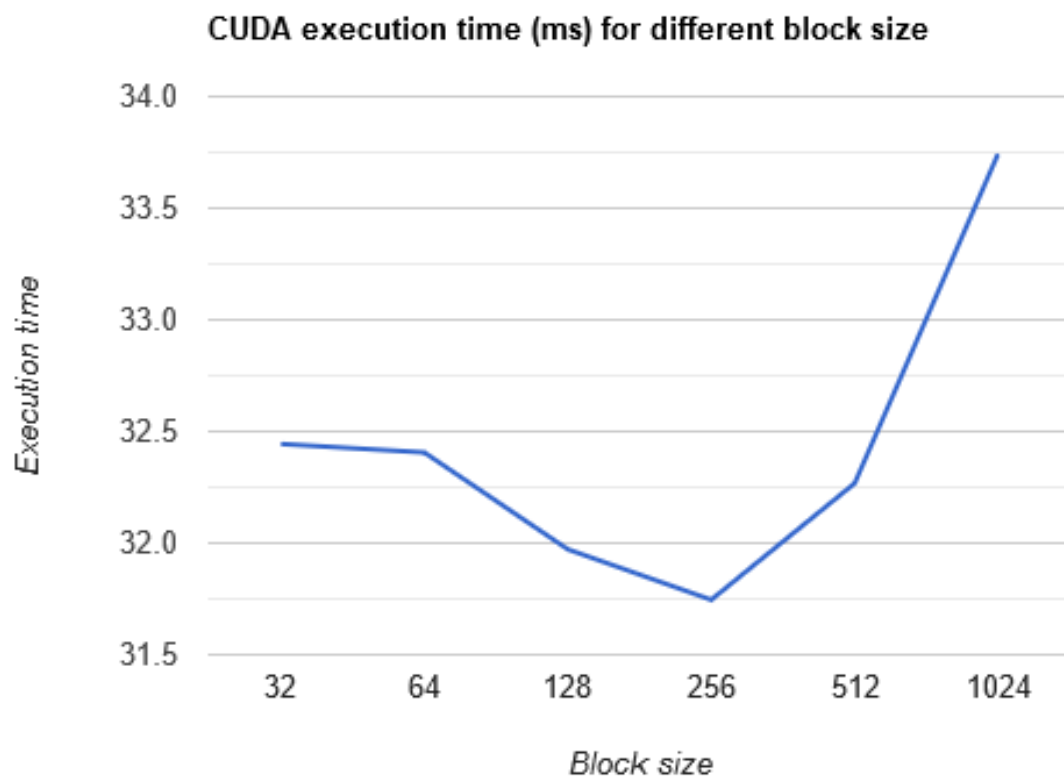


Due to warp granularity, it is always recommended to choose a size that is a multiple of 32. So we have tried experimenting with 32, 64, 128, 256, 512 and 1024 threads per block. The result shows that 256 threads per block are performing the best. Block size of 32 is performing poorly is probably due to low occupancy, while the block size of 1024 is performing poorly is probably because of context switching. We will use block size of 256 for further evaluation and discussion at later part since it is the optimal block size. The shape of the line graph does not follow either Amdahl's law or Gustafson's law.

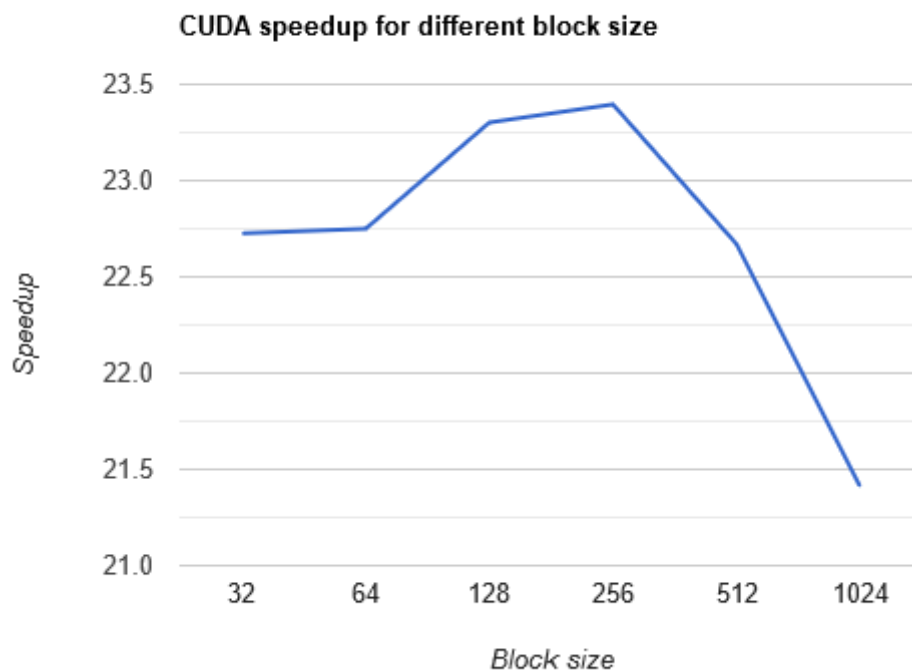
CUDA Implementation

Comparison of the execution time from different block sizes

Image Dimension	Execution time (ms) for different block size					
	32	64	128	256	512	1024
640x480 (307200px)	4.89	4.82	4.17	4.27	4.72	5.11
1280x720 (921600px)	10.51	11.02	10.74	11.13	11.02	12.01
1920x1080 (2073600px)	22.32	21.62	23.11	23.26	23.4	24.02
2560x1440 (3686400px)	40.42	40.01	39.12	39.18	38.99	41.89
3840x2160 (8294400px)	84.08	84.56	82.72	80.89	83.21	85.69
Average	32.444	32.406	31.972	31.746	32.268	33.744



	Speed up for different block size					
Image Dimension	32	64	128	256	512	1024
640x480 (307200px)	16.77x	17.01x	19.66x	19.21x	17.37x	16.04x
1280x720 (921600px)	22.01x	20.98x	21.52x	20.78x	20.98x	19.25x
1920x1080 (2073600px)	23.77x	24.54x	22.95x	24.10x	22.67x	22.08x
2560x1440 (3686400px)	27.21x	27.49x	28.12x	28.08x	28.21x	26.26x
3840x2160 (8294400px)	23.87x	23.73x	24.26x	24.81x	24.12x	23.42x
Average	22.726x	22.75x	23.302x	23.396x	22.67x	21.41x



Due to warp granularity, it is always recommended to choose a size that is a multiple of 32. So we have tried experimenting with 32, 64, 128, 256, 512 and 1024 threads per block. The result shows that 256 threads per block are performing the best. Block size of 32 is performing poorly is probably due to low occupancy, while the block size of 1024 is performing poorly is probably because of context switching. We will use block size of 256 for further evaluation and discussion at later part since it is the optimal block size. The shape of the line graph does not follow either Amdahl's law or Gustafson's law.

Discussion

Comparison of execution time from different implementations

Image Dimension	Execution time (ms) for different implementation			
	Serial	OpenMP- 12 threads	OpenCL – Blocksize 256	CUDA – Blocksize 256
15x11 (165px)	0.05	0.5	1.41	0.82
640x480 (307200px)	82.01	23.47	4.83	4.27
1280x720 (921600px)	231.23	71.87	10.91	11.13
1920x1080 (2073600px)	530.56	143.09	23.12	23.26
2560x1440 (3686400px)	1100.21	325.52	38.22	39.18
3840x2160 (8294400px)	2007.23	594.92	82.32	80.89
Average	658.5483333	193.2283333	26.80166667	26.59166667

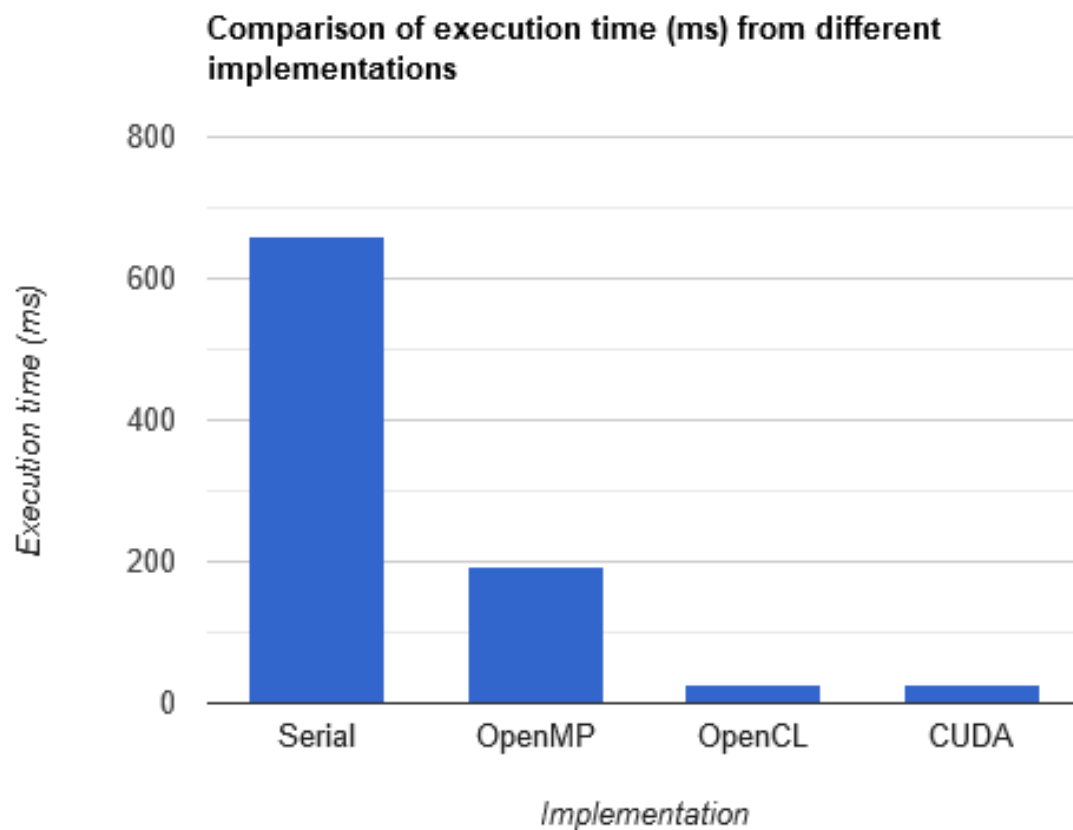
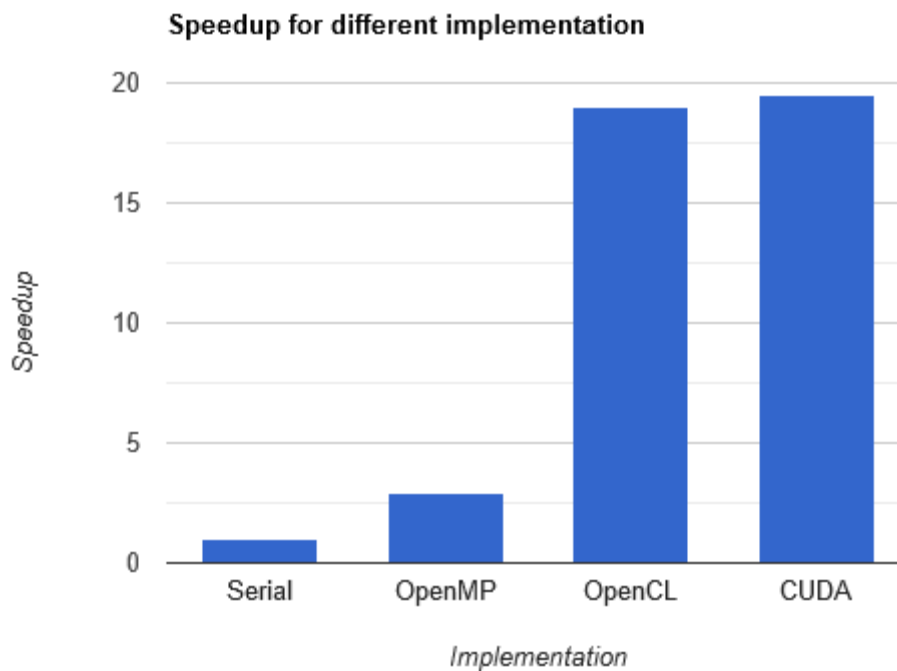


Image Dimension	Speedup for different implementation			
	Serial	OpenMP- 12 threads	OpenCL – Blocksize 256	CUDA – Blocksize 256
15x11 (165px)	1x	0.1x	0.04x	0.06x
640x480 (307200px)	1x	3.49x	16.7x	19.21x
1280x720 (921600px)	1x	3.22x	21.19x	20.78x
1920x1080 (2073600px)	1x	3.71x	22.94x	24.10x
2560x1440 (3686400px)	1x	3.38x	28.78x	28.08x
3840x2160 (8294400px)	1x	3.37x	24.38x	24.81x
Average	1.0x	2.8783x	19.005x	19.5067x




From the results, we can see that overall CUDA is performing the best. The second best is OpenCL which is very close to CUDA. Then follows by OpenMP and Serial implementation. The result shown here is very logical and sensical. CUDA and OpenCL are utilising GPU, so they have the advantages of performing better in image and matrix related computation. In contrast, OpenMP and Serial implementation is utilising CPU, which have less number of threads.

By looking at speedup result for 15x11 image, we can see that both OpenMP, OpenCL and CUDA perform much worse than serial implementation. This is because the problem size is too small, the image was small enough that the serial implementation was faster due to the overhead involved with device memory allocation, data transfer, and PCIe bus transmission between the CPU and GPU.

Current limitation and future works

59.1% apply_gaussian_filter(unsigned char*, unsigned char const *, int, int, double*)	apply_gaussian_filter(unsigned char*, unsigned char co...
33.1% apply_sobel_filter(double*, unsigned char*, unsigned char const *, int, int, char*, char*)	apply_sobel_filter(double*, un...
4.3% apply_non_max_suppression(double*, double*, unsigned char*, int, int)	
1.9% apply_edge_hysteresis(unsigned char*, unsigned char*, int, int)	
1.6% apply_double_threshold(unsigned char*, double*, int, int, int, int)	

 **Low Kernel Concurrency** [0 ns / 38.97132 ms = 0%]

The percentage of time when two kernels are being executed in parallel is low.

 **Low Memcpy/Kernel Overlap** [0 ns / 1.23622 ms = 0%]

The percentage of time when memcpy is being performed in parallel with kernel is low.

By using Nvidia Visual Profiler, we can see the `apply_gaussian_filter` kernel takes up 59.1%, and `apply_sobel_filter` kernel takes up 33.1% of the computation time. Besides, there is no kernel concurrency. We can introduce kernel concurrency by separating the `apply_sobel_filter` kernel into two kernels, which can be `sobel_seperable_pass_x` and `sobel_seperable_pass_y`. These two kernels use two different directions of the Sobel filter. They will have no dependency so they can be executed concurrently.

In addition, we also found out that Sobel and Gaussian filter is separable functions. In the current implementation, we have not utilised separable filter; therefore, a filter of window size $M \times M$ computes M^2 operations per pixel. If we utilised separable functions correctly, the cost would be reduced to computing $M + M = 2M$ operations. This is a two step process where the intermediate results from the first separable convolution is stored and then convolved with the second separable filter to produce the output. We believed the performance would be significantly improved by utilising separable functions.

We can also use multiple streams to parallelise the process of memcpy and kernel execution, although the improvement may not be too significant, it is one thing that can be done in order to push performance to its limit.

Conclusion

We have done a version of the complete Canny Edge detector using OpenMP, OpenCL, CUDA, including all stages of the algorithms. We had achieved significant speedup from the parallelised implementation. The conclusion is that CUDA had achieved the best performance, among others. The computation is dominated by the Gaussian filter and the Sobel filter. Further improvement can be made to improve these two stages of the algorithms.

References

Canny, J., 1986. A Computational Approach to Edge Detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-8(6), pp.679-698.

Chapman, B. and Massaioli, F., 2005. OpenMP. *Parallel Computing*, 31(10-12), pp.957-959.

GitHub. 2020. *Canny-Edge-Detector*. [online] Available at: <<https://github.com/sorazy/canny>> [Accessed 26 July 2020].

Jin, H., Jespersen, D., Mehrotra, P., Biswas, R., Huang, L. and Chapman, B., 2011. High performance computing using MPI and OpenMP on multi-core parallel systems. *Parallel Computing*, 37(9), pp.562-575.

Keymolen, B., 2020. *Canny*. [online] GitHub. Available at: <<https://github.com/brunokeymolen/canny>> [Accessed 26 July 2020].

Web.stanford.edu. 2020. *Stanford University Assignment: Edge Detection*. [online] Available at: <<https://web.stanford.edu/class/cs315b/assignment1.html>> [Accessed 26 July 2020].

Web.stanford.edu. 2020. *Stanford University Assignment: Parallel Edge Detection*. [online] Available at: <<https://web.stanford.edu/class/cs315b/assignment2.html>> [Accessed 26 July 2020].

Zeller, C., 2020. *Nvidia CUDA C/C++ Basics*. [online] Nvidia.com. Available at: <<https://www.nvidia.com/docs/IO/116711/sc11-cuda-c-basics.pdf>> [Accessed 26 July 2020]

