SF2568 Project: GPU Accelerated Edge Detection

Daniel Persson Proos dproos@kth.se
Borja Rodríguez Gàlvez borjarg@kth.se

May 7, 2018

1 Project Description

In this project we aim to tackle the well known problem of edge detection. This problem has been extensively studied [1] and many famous algorithms such as Canny [2] or Perona [3] exist and give good results.

Edge detection *per se* aims to identify the points in a digital image at which the intensity changes sharply (which we perceive as edges). Hence, the majority of the techniques rely on a convolution kernel filter to obtain local estimates of the first or second order gradient of the image which are later used to find the edges either by thresholding or by the use of their angles.

Our particular approach will be a simple two-step convolution of a 5x5 Gaussian filter of 1.4 standard deviation (widely used in the literature), g, followed by a 3x3 Laplacian estimator filter, l as the one we studied in class.

$$g = \frac{1}{289}\begin{bmatrix} 4 & 8 & 10 & 8 & 4 \\ 8 & 16 & 20 & 16 & 20 \\ 10 & 20 & 25 & 20 & 10 \\ 8 & 16 & 20 & 16 & 20 \\ 4 & 8 & 10 & 8 & 4 \end{bmatrix}, \quad l = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} \tag{1}$$

Then the resulting image will be thresholded to obtain the edges.

Such a simple approach will be taken because our particular interest is in how to efficiently parallelize the convolution in a Graphics Processing Unit (GPU). For this reason, the majority of the work will be on studying different improvements to the parallelization algorithms for the convolutions.

Hence, the speed up and theoretical performance of the thresholding method will not be assessed for being considered trivial and embarrassingly parallel. Furthermore, the intention of this report is to state the knowledge obtained in GPU parallel programming rather than repeating known concepts from the course.

All the algorithms will be performed in C++ with the usage of CUDA [4] (9.1 version) for programming over NVIDIA graphics cards. Specifically, we are going to run our algorithms in the GTX 960M [5]. The relevant specs for this graphics card (obtained using *deviceQuery*) are displayed in Table 1:

The CPU used for the sequential algorithm is an Intel(R) Core(TM) i7-6700HQ CPU with 4.28 GFLOPS/s per core, around the double than the 2.19 GFLOPS/s per core of the GPU.

2 Theoretical Background

2.1 NVIDIA GPU Architecture

A Graphics Processing Unit, or GPU, is a massively parallel co-processor that is structured in a way that is different from CPUs. CPUs usually feature 2-16 cores with up to three layers of fast cache memory that is no more than 50MB in total. In contrast, the working memory on a GPU is normally in the order of gigabytes to accommodate workloads that require large amounts of data. This memory is not fast though, and is comparable to the DRAM on a computer in speed. Because many applications of a GPU require large amounts of data per calculation the processing time for

Specification	**Value**
Global memory:	4046 MBytes
Memory interface:	GDDR5
Memory bus width:	128-bits
Memory bandwidth:	80 GBytes/s
Floating-point performance:	1403 GFLOPS
CUDA cores:	640 (5 Multiprocessors)
GPU max clock rate:	1.18 GHz
Memory clock rate:	2505 MHz
Constant memory:	65536 bytes
Shared memory per block:	49152 bytes
Warp size:	32
Maximum number of threads per multiprocessor:	2048
Maximum number of threads per block:	1024

Table 1: Specifications of the GeForce GTX 960M NVIDIA GPU

many tasks is limited by the bandwidth of this memory. There are several layers of cache memory inside a GPU to combat this problem, however.

An NVIDIA GPU consists of a number of CUDA cores, which are individual processing units that execute one command per cycle. These cores are ordered in groups of 32, which are called warps. These warps are in turn grouped together within a Streaming Multiprocessor (SM), of which there are multiple on a GPU. The number of warps that are included inside an SM varies depending on the architecture that a specific GPU is based on and can even vary between models using the same architecture. [6]

2.2 CUDA

CUDA is described as an extension of the C programming language, adding some concepts and commands that make use of an NVIDIA GPU. The central concept is the kernel. A kernel is a function that is meant to be run in parallel on many CUDA cores. When running a kernel inside a C program, the programmer can specify a number of threads and blocks that the kernel will run on. A thread is a single instance of the code inside the kernel and a block is a group of threads that share memory. The number of threads that reside within a block can be set by the programmer. However, a block cannot execute on more than one SM so the number of threads within a block is upper bound by the number of cores within an SM on the GPU that the code is running on.

There are several layers of memory that a thread can use. Highest in the hierarchy of memory is global memory, which is normally the memory which is specified in the name of the GPU model, i.e. "GTX1060 6GB" for example. This memory is relatively large in size but not as fast as the memory that resides lower down in the hierarchy.

Next is shared memory, which is the memory shared within a block. The bandwidth of this memory is much higher and the latency much lower than that of global memory. Shared memory uses several memory banks to store the data that is put in, which enables multiple memory requests to be served simultaneously as long as the data requested resides in separate banks or is at the same address. If it does not, the conflicting requests are serialized. The correct method for avoiding these memory bank conflicts differ slightly between architectures of different compute capability level, but is generally very similar.

Another type of memory that can be used is constant memory. Constant memory resides within the constant cache inside each SM and functions somewhat differently from shared memory. Memory requests to constant memory are serialized only if there are requests for different memory addresses. This means that if multiple threads request the same memory address the requests can be served simultaneously, whereas if N different memory addresses are requested the throughput of the memory will decrease by a factor of N. [4]

2.3 Performance metrics

In this section a few performance metrics used in this project are described. Speed-Up will also be used but not defined because of recurrent usage during the course.

2.3.1 Arithmetic intensity

Two important features of computer graphics are data parallelism and independence (i.e., not only is the same or similar computation applied to streams of many vertexes and fragments, but also the computation on each element has little or no dependence on other elements).

A good way to combine these attributes is arithmetic intensity (AI) [7], which is the ratio of computation to bandwidth (i.e., work to traffic):

$$AI = \frac{W}{Q} \tag{2}$$

where the work, W is expressed in FLOPS and the traffic, Q, in bytes.

2.3.2 Roofline Model

The Roofline Model [8] provides a way to estimate the peak performance that a kernel in the GPU can achieve (in FLOPS). This estimate is computed by multiplying the AI of that kernel with the bandwidth of the device. If this estimation exceeds the theoretical performance of the device then the kernel is limited by computation, otherwise it is memory bandwidth limited [8, 9].

2.4 Image Filtering

Before starting to describe the algorithms we will quickly define an image convolution between two images in the context of image filtering.

The convolution operation computes a new value for every pixel of the original image, I, based on a weighted average of the original pixel and the pixels in its neighborhood. These weights are stored in what is commonly called the convolution filter, F, the size of which determines the size of the neighborhood.

$$I'(x, y) = \sum_{\forall r} \sum_{\forall s} I(x - r, y - s) F(r, s) \tag{3}$$

We will assume the dimensions of the original and resulting image to be $N_x \times N_y$ and the filter's $M_x \times M_y$.

3 Algorithms

3.1 Naive Algorithm

The straightforward algorithm for parallelizing the convolution in a GPU using CUDA is the following:

- The image, I, and the filter F, are stored in the global memory of the graphics card.
- The image is divided in blocks of threads of size B_x, B_y and a thread is assigned to each pixel as in Figure 1. (if B_x did not divide N_x or B_y did not divide N_y we could just pad the image to have the desired dimensions; for the time being, we will assume they are divisible)
- Each thread computes the weighted average of the pixel and its neighbors in the original image and writes it in the result image.

This first implementation takes no care on how CUDA handles the GPU memory and is therefore a simple embarrassingly parallelization task. Since it uses the highest memory in the hierarchy (see Section 2) the algorithm is bandwidth limited and is commonly known as the naive algorithm in the literature [9–11].

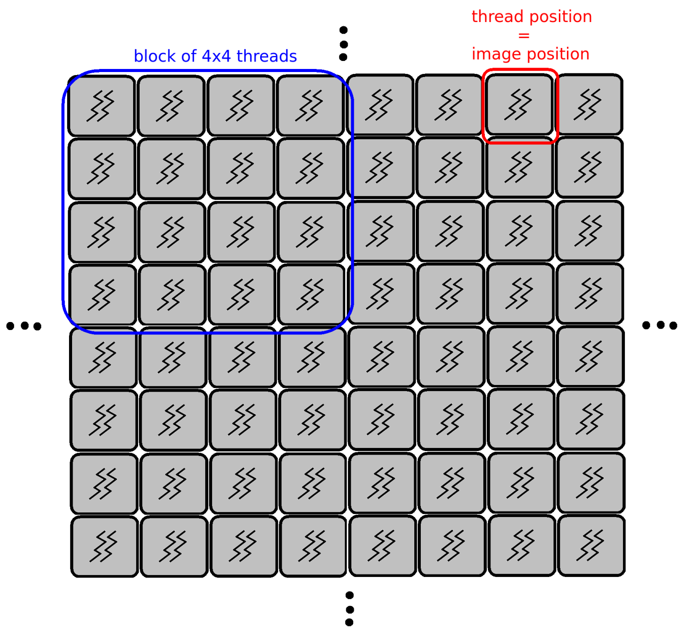

Figure 1: Example of naive algorithm thread distribution over the data with blocks of 4x4 threads.

3.2 Shared Memory Algorithm

The next well known optimization of this algorithm is to make use of the shared memory within the blocks of threads in order to avoid loading each pixel of data as many times as before.

Before explaining the common algorithm to deal with this advantage lets consider a block of threads $(B_x \times B_y)$ as the one shown in Figure 2 and consider also a kernel radius of K.

A common approach is to increase the number of threads in a block in order to load both the data in the positions for which that block will write the result [10], but it is not the only one. There are mainly two approaches:

(i) Having $((B_x + K) \times (B_y + K))$ threads loading each of them a pixel value. This approach is usually not very efficient since it has many idle threads, $2K(B_x + B_y + 2K)$, to be exact. Furthermore, with this approach the number or threads in a block might not be a multiple of the warp size, which would lead to much poorer performances.

(ii) As an attempt to solve the issue of idle threads, we can have some threads to load more than one pixel. An example of this solution is to use a column by column or row by row read of the needed pixels and then redistribute the threads into the grid of the output pixels positions and compute the convolution (see [10]). Even this approach might seem unreasonable for the amount of new computations needed, since the kernel is usually bandwidth limited it is a reasonable implementation.

We followed the second approach (ii) taking advantage of the problem domain. We are aware that all of our filters are small and will always be smaller than the block size, which will be 32×32 for taking advantage of the GTX 960M architecture. Hence, we make each thread load 1, 2 or 4 pixels values into the shared memory depending on its position:

1. Threads in positions (i, j) such that $K < i < B_x - K$ and $K < j < B_y - K$ will only load its own pixel.

2. Threads in positions (i, j) such that $(i \leq K)$, $(i \geq B_x - K)$, $(j \leq K)$ or $(j \geq B_y - K)$ will also load the the i_{th} pixel to left or right of the apron or the j_{th} pixel to the top or bottom or the apron respectively, as reflected in Figure 2.

3. Threads in positions (i, j) such that $(i \leq K$ and $j \leq K)$ or $(i \geq B_x - K$ and $j \leq K)$ or $(i \leq K$ and $j \geq B_y - K)$ or $(i \geq B_x - K$ and $j \geq B_y - K)$ will also load the apron pixel i positions to the left and j positions to the top, or right and top or left and bottom or right and bottom respectively. This is shown for the top-left pixel and the second bottom-left pixels in Figure 2.

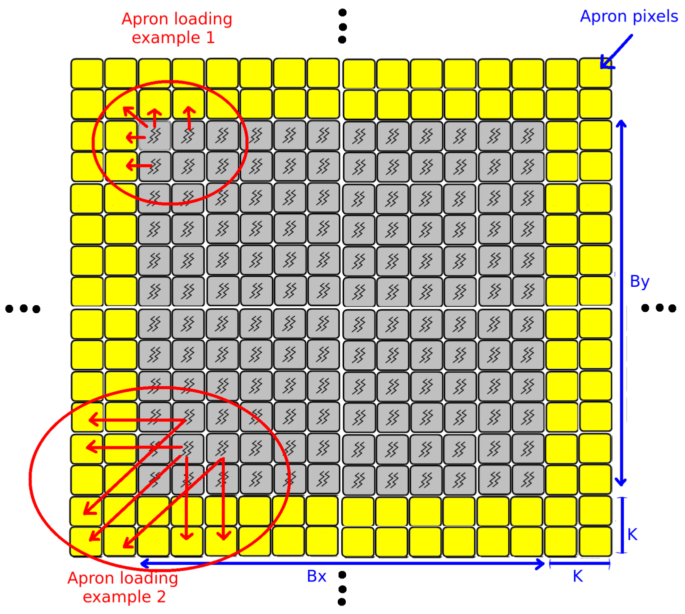

Figure 2: Example of the shared algorithm thread distribution and data loading with blocks of 12x12 threads and a kernel radius of 2.

3.3 Shared + Constant Memory Algorithm

??

The next step to take advantage of the way CUDA handles the GPU memory hierarchy is to make use of the constant memory. Since all threads in a warp load the same filter value at a time, as explained in Section 2, using constant memory to store these values will achieve the best theoretical GPU throughput in reading them.

This implementation can be easily added to the previous algorithms by pre-loading the filters to the NVIDIA chip.

3.4 Separable filters

As we know from signal processing courses, there exist some filters which are separable (i.e., the application of a row filter convolution after a column filter convolution is equivalent to a convolution of the 2D filter). These filters are particular since the matrix defining the filter has rank 1.

We see how the filter defining the Laplacian approximation has rank 2 and hence it is not separable. The Gaussian filter, however, is supposed to be separable, and computing the rank confirms it, it is rank 1. Furthermore, as we generated the Gaussian filter to be of 1.4 standard deviation in each direction we see how the 2D filter itself is the outer product of the 1D Gaussian filter obtained with the same standard deviation. I.e.,

$$g = \frac{1}{289}\begin{bmatrix} 4 & 8 & 10 & 8 & 4 \\ 8 & 16 & 20 & 16 & 20 \\ 10 & 20 & 25 & 20 & 10 \\ 8 & 16 & 20 & 16 & 20 \\ 4 & 8 & 10 & 8 & 4 \end{bmatrix} = \frac{1}{17}\begin{bmatrix} 2 \\ 4 \\ 5 \\ 4 \\ 2 \end{bmatrix} \frac{1}{17}\begin{bmatrix} 2 & 4 & 5 & 4 & 2 \end{bmatrix} = \frac{1}{289} g_l g_l^T \tag{4}$$

Then, when the filters are separable the operations required for computing them reduce too. It is sufficient to compute a row and a column convolution only, instead of a 2D convolution. So the convolution operation is reduced from $M_xM_yN_xN_y$ to $(M_x + M_y)N_xN_y$ in the sequential algorithms:

$$I'(x,y) = \sum_{\forall r} I(x-r,y)f(r) + \sum_{\forall s} I(x,y-s)f(s) \tag{5}$$

On the GPU parallelized algorithm this is also an advantage, since it would require to load less pixels per thread (each thread would only need to load at most one pixel from the apron) and also less operations would be carried out for the algorithm.

3.5 Tiling

As we have said previously, the performance of our algorithms is mainly memory bandwidth limited. Therefore, if a block of threads could compute a higher amount of result pixels without loading as many pixels as before, the performance of the algorithms would increase. And this is exactly what tiling does.

In previous approaches each block of threads calculated the values of a square, or tile, of output pixels, marked in red in Figure 3. Now, with tiling a block of $(B_x \times B_y)$ threads computes T tiles of $(B_x \times B_y)$ pixels. With this approach all the apron pixels that existed between tiles must not be loaded again and hence, the arithmetic intensity of the kernel increases. In Figure 3, for instance, tile 1 and 2 do not need a right and left apron, respectively, since they are handled by the same thread block. Without tiling, however, they would need these aprons and the total number of pixels that need to be loaded for each tile would look like in Figure 2.

Then, after loading the respective pixels, each thread (i, j) will compute the output values on position (i, j) of every tile.

We will refer to this to $1 \times T$ tiling.

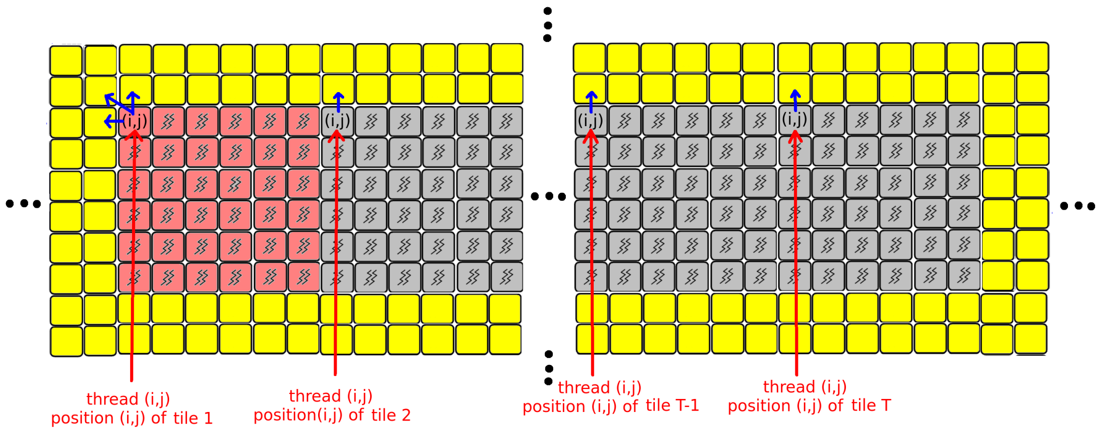

Figure 3: Example of the adaptive tiling algorithm thread distribution and data loading with blocks of 6x6 threads and a kernel radius of 2 with T tiles.

3.5.1 Adaptive tiling

While tiling can make convolutions more efficient on a GPU, there is no one tiling factor T that can rule them all. The optimal tiling factor for a convolution operation depends on the size of the tiles, the size of the filters and the amount of shared memory per SM on the GPU. Of these, the shared memory size is the most important, as the block that is executing the calculations needs to be able to fit inside a SM. An increased tile size would increase the amount of threads that execute in parallel but ultimately, it all needs to fit inside the same SM. Calculating this optimal tiling factor at runtime and using it for the convolution on the GPU is called Adaptive tiling [9, 11].

First a few variables will be defined to describe the elements in Figure 2 in terms of number of pixels. Let LA be the size of: lower left apron + upper left apron + center left apron, RA be: lower right apron + upper right apron + center right apron, LoA be the size of the center lower apron, UpA be the size of the center upper apron and TS to be the tile size.

Every tile will consist of numPixPerTile $= UpA + TS + LoA$ pixels, whereas only the first and last tile will need to store LA and RA extra pixels, respectively.

First we calculate the number of pixels that can fit into shared memory:

$$\text{numPixels} = \frac{\text{Shared memory size}}{\text{sizeOf(short)}} \tag{6}$$

Then we remove the left and right aprons:

$$\text{numPixAvailForTiling} = numPixels - LA - RA \tag{7}$$

Lastly we take the ceiling of the number of pixels available for tiling divided by the number of pixels per tile to get the tiling factor T:

$$\text{Tiling factor} = \left\lceil \frac{\text{numPixAvailForTiling}}{\text{numPixPerTile}} \right\rceil \tag{8}$$

In our particular scenario, the best tiling factor is 21 for the Laplacian filter and 22 for the Gaussian one. However, we will use 21 tiling in both filters for convenience.

3.5.2 Separate filters in tiling

One other advantage of using tiling is the possibility to again take advantage of separate filters. If the filters are separable then we can apply $1 \times T$ tiling for the row convolution and then analogously apply $T \times 1$ tiling for the column convolution. An easy trick for performing $T \times 1$ tiling is to transpose the shared memory data inside the convolution kernel and then applying the standard $1 \times T$ tiling explained above. This trick might seem irrelevant but can save us some time and avoid memory bank conflicts, since now we will access consecutive memory positions (which are allocated in separate banks) within a warp rather than trying to access the different memory positions within a bank (see Figure 4 for a visual understanding).

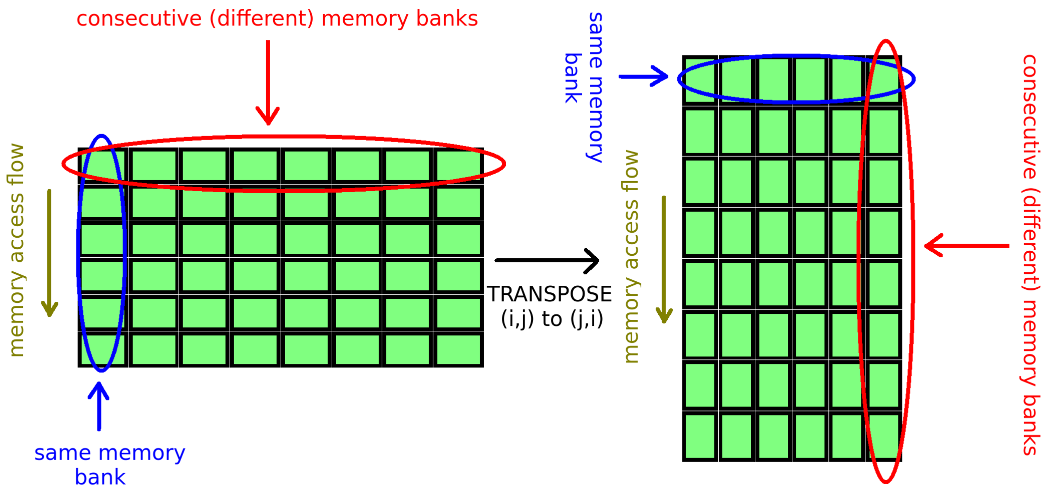

Figure 4: Exemplification of bank conflict avoidance after transposing the shared memory block of data.

4 Theoretical Performance

4.1 Disclaimer

Theoretical Speed Up will not be computed since the algorithms we are dealing with are bandwidth limited. Furthermore, in the literature, the usual way to tackle the performance of a GPU algorithm is to assess its performance in FLOPS

4.2 Naive Algorithm

With this implementation each thread will load, multiply and add the data and also filter values $M_x M_y$ times. Therefore, there are 2 operations being done and 8 bytes (we are using floats) used in the computations.

Hence, the arithmetic intensity of this naive kernel is:

$$AI_{naive} = \frac{2M_xM_y}{8M_xM_y} = 0.25FLOPS/byte \tag{9}$$

Which produces a performance of 20 GFLOPS. This performance is clearly inferior than 1403 GFLOPS achievable, which means this implementation is hugely bandwidth limited.

4.3 Shared Memory Algorithm

Now, the number of multiplications and additions is still the same as before. However, the number of global memory image loads is only 4 per thread and still M_xM_y loads for the filters.

This last statement is only true in the case that $N_x > M_x$ and $N_y > M_y$, which will always be the case. Furthermore, most of the threads will only load one pixel but since they will wait for the threads which load more pixels before starting computation we will count 4 loads per thread.

Hence, the arithmetic intensity of this shared memory kernel is:

$$AI_{shared} = \frac{2M_xM_y}{4 \times 4 + 4M_xM_y} \tag{10}$$

We can see how this implementation obtains higher performance for 3x3 (27.692 GFLOPS) filters and 5x5 filters (34.483 GFLOPS). A graphic of this function for varying sizes of the filter's width and height can be found in Figure 5.

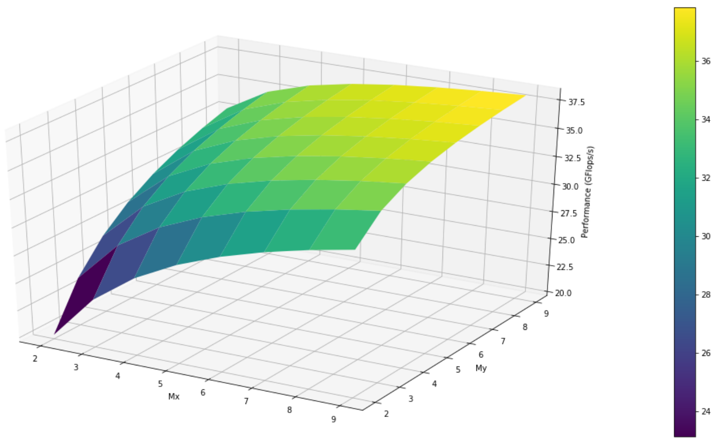

Figure 5: Convolution with shared memory algorithm for varying filter's width and height.

4.4 Shared + Constant Memory Algorithm

Similar to before, the number of multiplications and additions remain unchanged. Now, however, the global memory image loads are only 4 per thread and there is no load for the filters, since they are in the constant memory cache of the device.

Hence, the arithmetic intensity is:

$$AI_{constant} = \frac{2M_xM_y}{4 \times 4} \tag{11}$$

We can see a higher increase in performance than before (see Figure 6). Now the performance for a 3x3 filter is 90 GFLOPS and 250 GFLOPS for a 5x5 one.

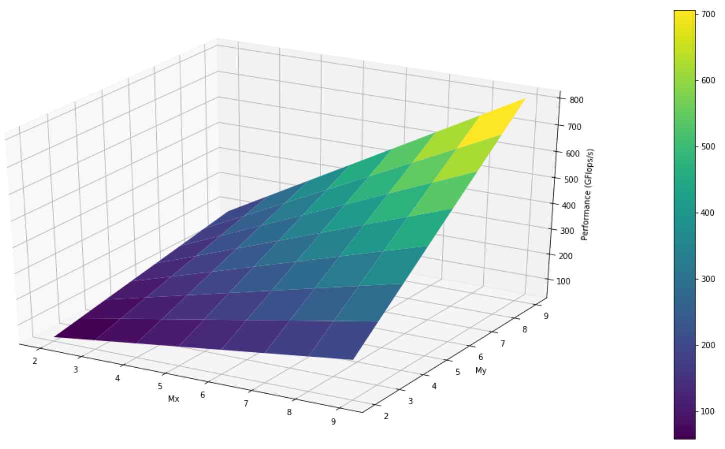

Figure 6: Convolution with constant + shared memory algorithm for varying filter's width and height.

4.5 Separable filters

If a filter is separable, then its performance changes a bit. For each row or column convolution only $2M_x$ or $2M_y$ operations are undertaken and also the pixels must be loaded only 2 times at maximum.

Therefore, the arithmetic intensity is the mean of the row and column filter arithmetic intensities:

$$AI_{separable} = \frac{1}{2}\left(\frac{2M_x}{4 \times 2} + \frac{2M_y}{4 \times 2}\right) \tag{12}$$

We see, however, that this metric does not help us to really compare the performance of the algorithm with the previous GPU algorithms since the computations performed are not the same.

Therefore, a good way to asses its viability is to observe how this algorithm is clearly memory bandwidth limited and how the improvement of reducing the number of operations does not matter. Then we observe how reducing from 4 to 2 pixel loads is still irrelevant since we need 2 applications of the filter.

For taking advantage of the filter separability the filter size should be higher and also the implementation of the previous algorithms should be done accordingly with possible larger filter sizes (e.g., the implementations provided by NVIDIA [10]).

4.6 Adaptive Tiling

When we use adaptive tiling we compute $2M_xM_yT$ operations, an addition and a multiplication for each pixel in each tile. In contrast, we only load at maximum 4 pixel values in the 1st and last tile and 2 pixel values in the intermediate tiles.

When we put these two things together we obtain the following arithmetic intensity:

$$AI_{tiling} = \frac{2M_xM_yT}{4(2 \times 4 + (T-2) \times 2)} \tag{13}$$

We can observe how this computation obtains more arithmetic intensity than before, obtaining a performance of 164.35 GFLOPS for a 3x3 convolution and 546.52 GFLOPS for a 5x5 one with 21 tiles.

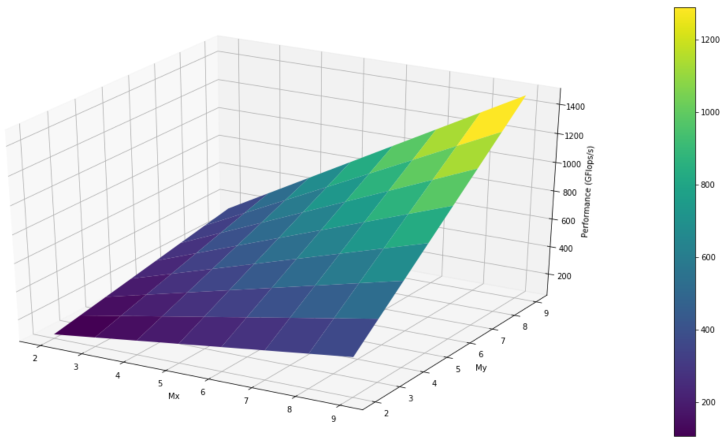

Figure 7: Convolution with an adaptive tiling algorithm for varying filter's width and height.

4.6.1 Separate filters in tiling

When the filter is separable, the arithmetic intensity changes. Now only $2(Mx+My)T$ operations are performed and also only 2 pixel loads are undertaken in the 1st and last tile and 1 pixel value in the intermediate tiles.

Hence, the resulting arithmetic intensity is:

$$AI_{tiling,shared} = \frac{2(M_x + M_y)T}{4(2 \times 2 + (T-2))} \tag{14}$$

5 Experimental Speed UP

In order to test the speed up of our algorithms we tested the speed up and running time performance of the convolutions on random images of sizes 256, 512, 1024, 2048 and 4096 for our two filter sizes (see Figures 9 and 8).

In the results we can appreciate how adaptive tiling is consistently the best implementation approach and it becomes better as the size of the image and the filter increases (is the one which scales better). Furthermore, we see how the successive improvements of the naive algorithm shine when the filter size increases (they are consistently better in 5x5 convolutions than in 3x3 ones), which agrees with the theoretical performance estimations and previously performed studies in the literature [9, 11].

Finally, for a very small filter size (3x3 filter) we observe how the improvements to the naive algorithm do not scale as well with the image size as they do for the 5x5 scenario, which also agrees with the literature results [9, 11]. In this scenario, the constant and shared memory approach with the loading aprons algorithm shown in Subsection **??** shows the best results.

It is important to mention that when computing or assessing the speed up of a GPU accelerate program it usually is compared with a multi-core implementation of the same algorithm in a CPU. However, the speed ups shown here are done against a single core serial implementation of the convolutions; which was the intention since the beginning.

Size	Naive	Shared	Constant	Adaptive	CPU
256x256	0.03028	0.02935	0.01154	0.02147	0.57747
512x512	0.06081	0.05731	0.03830	0.07054	2.21353
1024x1024	0.22893	0.22312	0.18631	0.26275	8.9972
2048x2048	0.74210	0.68913	0.62886	0.84656	35.89697
4096x4096	1.39531	1.40037	1.40050	1.41074	141.20673

Table 2: 3x3 convolution running time (seconds) for 100 images

Size	Naive	Shared	Constant	Separable	Adaptive	CPU
256x256	0.07643	0.05121	0.34102	0.02047	0.12751	1.53194
512x512	0.16982	0.10807	0.05403	0.059440	0.04246	5.94404
1024x1024	0.37363	0.30825	0.20551	0.22412	0.20722	24.66439
2048x2048	0.96227	0.78883	0.74028	0.80197	0.76989	96.23732
4096x4096	5.17129	5.31297	5.1702	2.42404	1.46357	387.84728

Table 3: 5x5 convolution running time (seconds) for 100 images

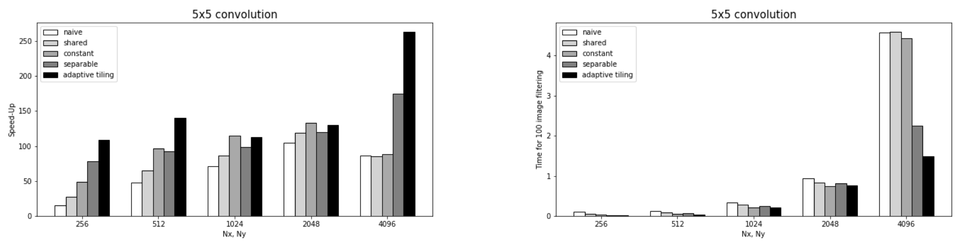

Figure 8: 5x5 filter convolution experimental Speed Up against a serial implementation on a CPU (left) and experimental running time over 100 images (right) with different image sizes.

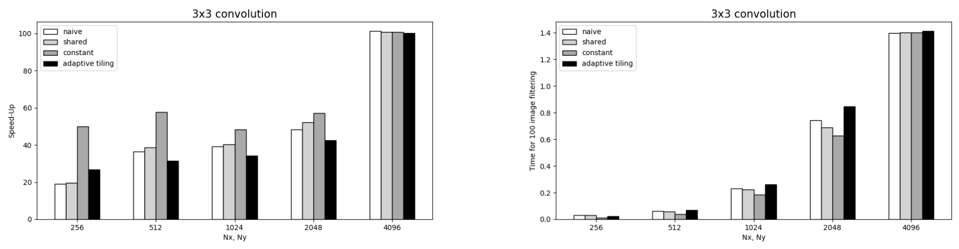

Figure 9: 3x3 filter convolution experimental Speed Up against a serial implementation on a CPU (left) and experimental running time over 100 images (right) with different image sizes.

The raw data for the 3x3 convolution are shown in Table 2 and the raw data for the 5x5 convolution in Table 3.

6 Results

After applying the smoothing filter, g, and thresholding the result of the Laplacian estimation (with a threshold of 5) we obtained the expected edges of the images. As an example you can see Figures 10 to 12. Where both the original image and the edges are shown for images of 512, 1024 and 2048 pixels of width and height.

These results that can easily be obtained by the usage of the sequential algorithm on the CPU were used for assessing the correctness of our implementations of the GPU acceleration algorithms.

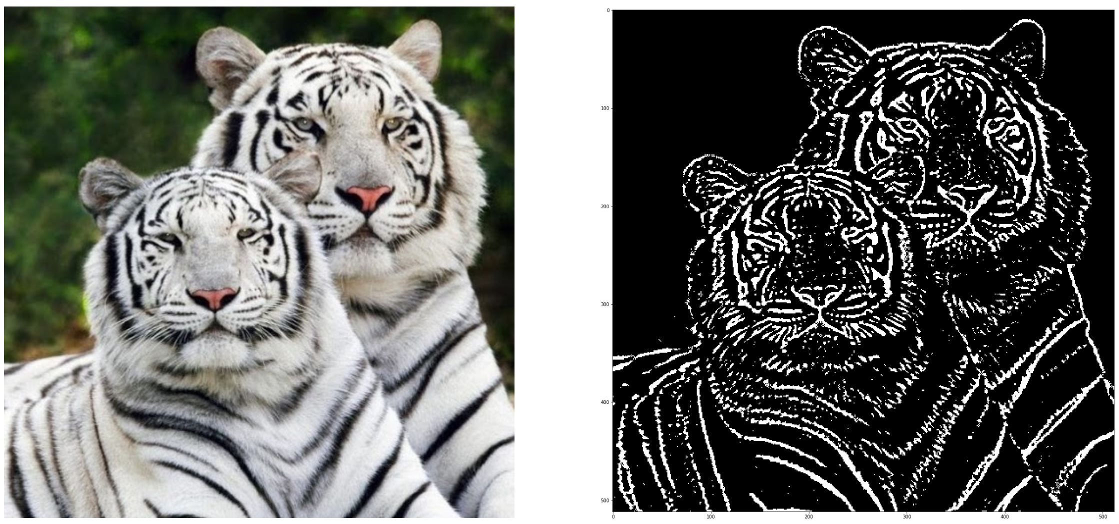

Figure 10: Edge detection on a 512x512 image.

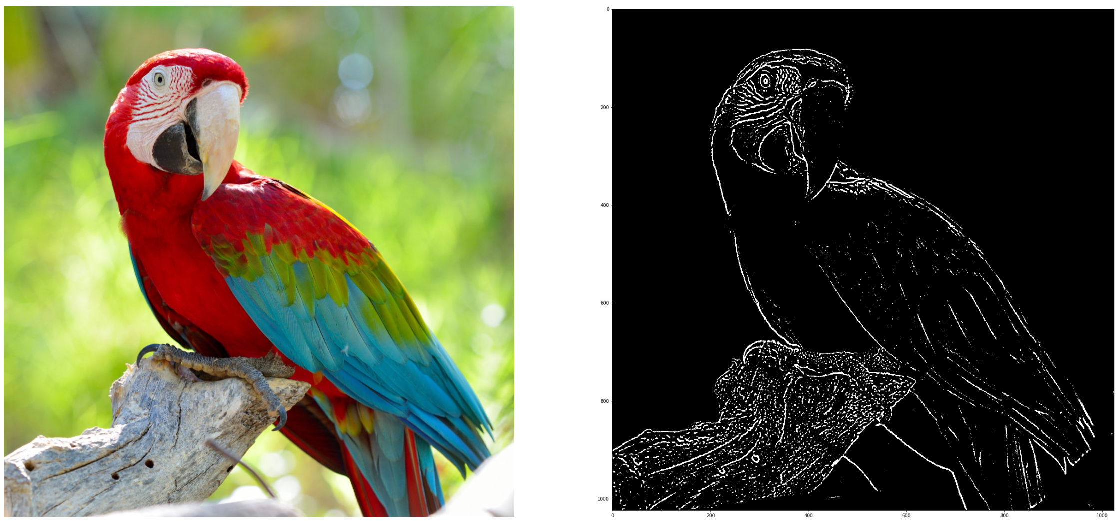

Figure 11: Edge detection on a 1024x1024 image.

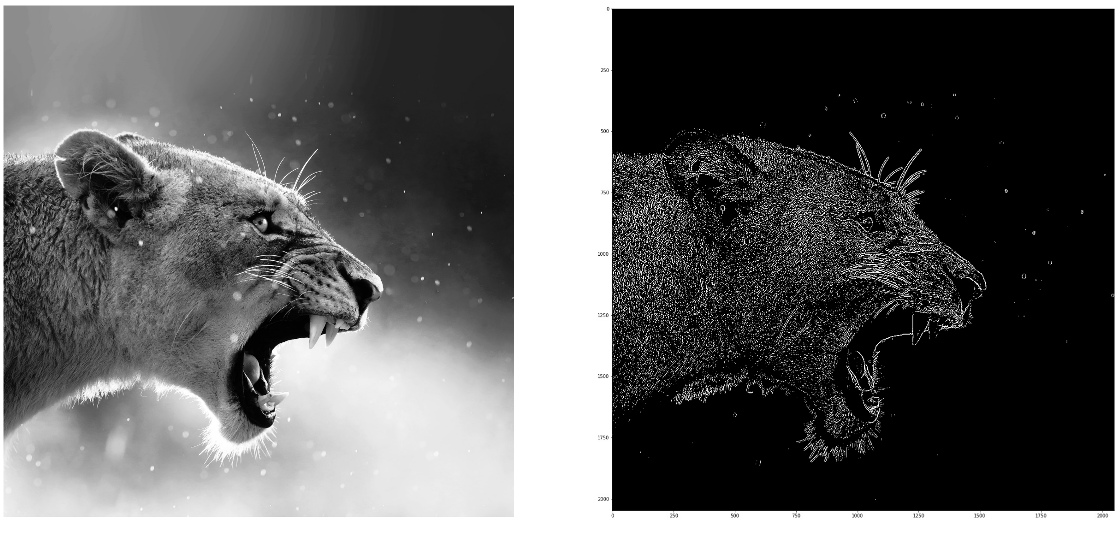

Figure 12: Edge detection on a 2048x2048 image.

References

[1] M. Sharifi, M. Fathy, and M. T. Mahmoudi, "A classified and comparative study of edge detection algorithms," in *Information Technology: Coding and Computing, 2002. Proceedings. International Conference on*, pp. 117–120, IEEE, 2002.

[2] J. Canny, "A computational approach to edge detection," in *Readings in Computer Vision*, pp. 184–203, Elsevier, 1987.

[3] P. Perona and J. Malik, "Scale-space and edge detection using anisotropic diffusion," *IEEE Transactions on pattern analysis and machine intelligence*, vol. 12, no. 7, pp. 629–639, 1990.

[4] C. Nvidia, "Programming guide," 2010.

[5] N. Corporation, "GeForce GTX 960M specifications," 2018.

[6] M. K. A. B. Jayshree Ghorpade, Jitendra Parande, "Gpgpu processing in cuda architecture," *Advanced Computing: An International Journal (ACIJ)*, vol. 3, no. 1, 2012.

[7] M. Harris, "Mapping computational concepts to gpus," in *ACM SIGGRAPH 2005 Courses*, p. 50, ACM, 2005.

[8] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[9] B. Van Werkhoven, J. Maassen, H. E. Bal, and F. J. Seinstra, "Optimizing convolution operations on gpus using adaptive tiling," *Future Generation Computer Systems*, vol. 30, pp. 14–26, 2014.

[10] V. Podlozhnyuk, "Image convolution with cuda," *NVIDIA Corporation white paper, June*, vol. 2097, no. 3, 2007.

[11] B. Van Werkhoven, J. Maassen, and F. Seinstra, "Optimizing convolution operations in cuda with adaptive tiling," in *A4MMC'11: Proc. Workshop on Applications for Multi and Many Core Processors*, 2011.