# DESIGN SPACE EXPLORATION OF GPU IMAGE PROCESSING

REN CHEN, ANDREA SANNY, GEOFFREY TRAN

## 1. Background

Image processing is an extensive field that has been deeply studied in the community, providing a multitude of kernels which can be applied to a variety of applications such as aerospace, medical imaging and entertainment. Due to the need for real-time processing as well as requirements for quick processing of large input datasets, a great deal of research has been done to determine suitable platforms for implementing these image processing algorithms, including comparison between multi-cores, GPUs and FPGAs.

Prevalent in day-to-day life, image processing kernels pervade nearly every facet within today's technology. Image processing can be separated into three categories: low-level, mid-level and high-level image processing, separated based on the complexity of the kernel, from simpler, computationally-intensive algorithms to complex models of the human brain's ability of recognition and learning. Kernels from all three categories can be seen in numerous applications, and with the high demands placed on these applications, determining the appropriate platform becomes a vital step.

The 2D image gaussian blurring algorithm is one such algorithm which is commonly used as a filtering technique (i.e., Instagram filters or photo editing software). In general, it can be considered a standard representation of a filtering algorithm within the realm of image processing, which tend to have similar computational formats (i.e., Gaussian filtering, mean filtering, median filtering, convolution). As a window-based filtering solution, the 2D image gaussian blurring algorithm is separated into two kernels: image separation and filtering. The image separation kernel separates the image into the three standard components of colored images: Red (R), Green (G) and Blue (B). By separating the image into three separate images containing only the elements of the individual color, the filtering kernel can then be applied to properly blur the image using a window of pixels surrounding the selected pixels and a corresponding filter array of weight values to determine the new, blurred pixel value. The algorithm is discussed in further details below in Section 2.

## 2. Problem Description

In this project, we propose to realize 2D image gaussian blurring algorithm on GPU. As a commonly-used image processing algorithm, gaussian blurring can be implemented onto GPU in order to determine the suitability of the platform for low-level image processing kernels and explore the various optimizations available on GPU to achieve high throughput for these types of algorithms. To implement this algorithm, we need to first separate an

RGB image to different channels. Then each pixel in the image will be multiplied with a set of constants to blur the image. Accordingly, two kernels will be implemented and run on GPU. Finally, the separate channels will be merged to form the original image.

**Kernel 1: Image separation**

Each pixel in the RGB image will be divided into three different values. Each value presents the one of the color components of the pixel, i.e., Red (R), Green (G), and Blue (B). The Red component is the "x" value of the pixel. The Green component is the "y" value of the pixel. The Blue component is the "z" value of the pixel. After retrieving all the three color components, we will have three separate images, each representative of one of the three color components. These color components can be retrieved as follows:

Red=Image[Pixel_index].x;

Green=Image[Pixel_index].y;

Blue=Image[Pixel_index].z;

**Kernel 2: Filtering**

After separating the image into the color components, the next step is to blur the image by multiplying it by the corresponding elements in the filter array. A square array of weight values will be used. For each pixel in the image, we overlay this square array of weights on top of the image such that the center of the weight array is aligned with the current pixel, creating a window of pixels surrounding the current pixel to be used. To compute a blurred pixel value, we multiply each pair of corresponding numbers that line up from the input image and the filter array. Once these pairs have been multiplied together, Finally, we sum together the partial results, assigning the final value as the output for the current pixel. This process is repeated for all the pixels in the image.

Here is an example for a pixel of image Y, using filter array X as the weights:

Matrix X denotes an array of weights; Matrix Y denotes sample image:

$$X = \begin{bmatrix} 0.0 & 0.2 & 0.0 \\ 0.2 & 0.2 & 0.2 \\ 0.0 & 0.2 & 0.0 \end{bmatrix}, Y = \begin{bmatrix} 1 & 2 & 5 & 2 & 0 & 3 \\ 3 & 2 & 5 & 1 & 6 & 0 \\ 4 & 3 & 6 & 2 & 1 & 4 \\ 0 & 4 & 0 & 3 & 4 & 2 \\ 9 & 6 & 5 & 0 & 3 & 9 \end{bmatrix},$$

$$\begin{aligned} &0.0 * 2 + 0.2 * 5 + 0.0 * 1 + \\ &0.2 * 3 + 0.2 * 6 + 0.2 * 2 + \\ &0.0 * 4 + 0.2 * 0 + 0.0 * 3 \end{aligned} = 3.2$$

The following two images provide an example of the effects of applying gaussian filtering.

FIGURE 1. Unfiltered Image

FIGURE 2. Gaussian Filtered Image

## 3. Methodology

### Kernel implementation

We implement two kernels in CUDA: the image separation kernel and the Gaussian blurring kernel. The definitions of the kernels can be found in Section 2. Because the focus of our work is on implementing parallel algorithms and analyze the performance based on implementation and optimizations, we will utilize a framework that has been provided by the Udacity course "Intro to Parallel Programming". The framework is available at https://github.com/udacity/cs344.

### Framework

In order to conduct the experiments, we required a framework that met the following requirements:

(1) Verify correctness of the given kernel implementation

(2) Provide timing analysis for performance evaluation

From previous experience, we found that the "Intro to Parallel Programming" course from Udacity provides such a framework. This framework is provided at: `https://github.com/udacity/cs344/tree/master/Problem%20Sets/Problem%20Set%202`.

The framework provides test cases and reports the execution time of the kernel used. All user code relating to the kernel is contained within a single file, which fits our needs well. To test different versions of the code, it is simply a matter of switching out a symbolic link to different versions. Through this, we were able to test many design points with a fair amount of work.

We would also like to stress that the kernel implementations were all written by our team, the framework is strictly used to test and benchmark our designs.

**Design space exploration**

There are many factors which can affect the final performance of these kernels, creating a large design space that we explore to determine the most suitable parameters for high performance. In the case of these low-level image processing kernels, there is a strong opportunity for parallelism, which is one of the strengths of the GPU architecture. When utilizing GPUs, both block and grid size greatly affect the performance of parallel code on GPUs. These knobs essentially define the amount of parallelism that is being targeted on the platform. However, this is restricted by many factors such as the size of the device and architectural limitations. At a lower level, threads are sorted into warps. If a thread isn't ready, a warp will have to context switch to another.

The way these warps are scheduled and created also greatly affects the memory access patterns. Some memory patterns will have incredibly poor performance due to a possibility of multiple threads accessing the same memory bank. Therefore, we will vary the dimensions of the sizes and attempt to identify preferred dimensions for these kernels, exploring available block and grid sizes.

**Optimizations**

To develop competitive kernels for high-speed image processing, we need

1) Shared memory

Shared memory is a fast memory available on GPU, which can significantly reduce latency in comparison to global memory. However, the trade-off when considering which memory to utilize is the limitation of available shared memory. Another potential issue is the requirement that data loaded into shared memory must be explicitly done by the programmer, thus adding another level of complexity during the development period to determine the most suitable data to be stored into the space-limited shared memory.

2) Data reuse and locality

Many image processing algorithms required memory accesses to consecutive pixels in memory, which allows the designer to take advantage of spatial locality. In particular, window-based operations often used a sliding window to minimize the number of accesses by reusing previously accessed data, stored safely into local memory for fast and minimal accesses, minimizing the need to access larger memories such as global memory.

3) Additional algorithmic and architectural optimizations

We will also considered other algorithmic-specific or architecture-specific optimizations that may be suitable for the selected kernels while utilizing the GPU platform.

## 4. Results

During the design space exploration of the 2D image gaussian blurring algorithm, a number of optimizations were utilized in order to observe the implications of each optimization individually as well as accumulated on the execution time. The individual optimizations as well as the combination of optimizations are listed below in Table 1. Our baseline is presented as the basic image separation and processing kernel implemented in CUDA excluding all possible optimizations such as different memory types or improved methods for processing. We explore each of these datapoints shown in Table 1 by changing the block size used for processing.

Table 1. Explored datapoints

| | |
|---|---|
| Baseline | Basic image separation and image processing kernel implemented in CUDA |
| Constant Memory (Filter) | The filter is stored into constant memory |
| Shared Memory (Filter and Data) | The image and filter data are both stored into shared memory, managed by the programmer explicitly |
| Constant Memory (Filter) + Shared (Data) | The filter is stored into constant memory. The image data is stored into shared memory |
| Single Kernel | The image separation kernel is omitted and each thread processes all three colors per pixel |
| Constant Memory (Filter) + Single Kernel | The filter is stored into constant memory. The image separation kernel is omitted and each thread processes all three colors per pixel |
| Shared Memory (Filter and Data) + Single Kernel | The data is stored in shared memory, but the image is processed all at once |
| Constant Memory (Filter) + Shared Memory + Single Kernel | All three optimizations are utilized |

The results of changing the block size and utilized optimizations is shown below as Figure 3. Some additional results for a wide variety of block sizes are shown in Figure 4, Figure 5, Figure 7 and Figure 9 for the baseline, constant memory filter, single kernel and constant memory filter + single kernel, respectively. These results are presented as a few examples for more detailed analysis when using our optimizations and varying the block size.

Figure 4, Figure 5, Figure 7 and Figure 9 present a wide range of available block sizes for different optimizations or combinations of the optimizations. As the block size increases, the execution time significantly reduces in an exponential-fashion, in particular when shifting the block size to 4 or 8, however as the block size continues to increase, the improvement in execution time becomes minimal, remaining around similar execution time despite attempts for further improvements. When the block size is changed to 32 from 16, the execution time actually increases rather than continuing the downward trend of previous block size adjustments. This indicates that there is a limitation on the available

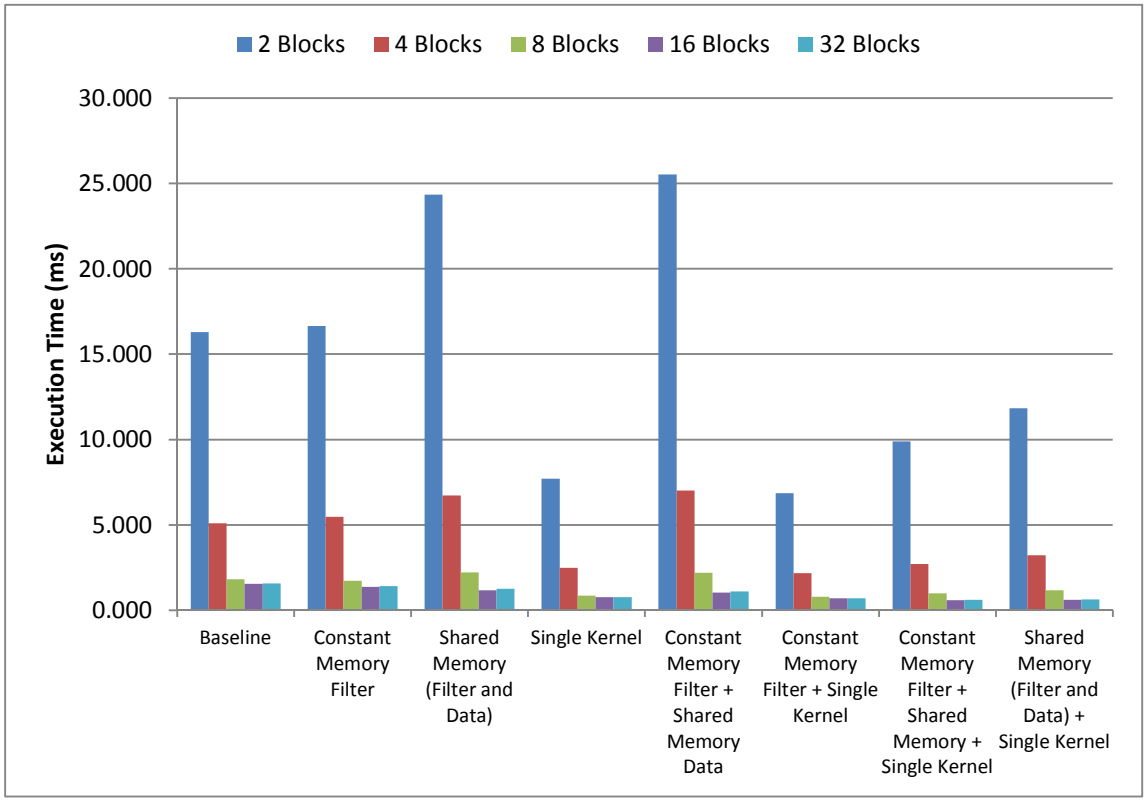


FIGURE 3. Execution time of various datapoints

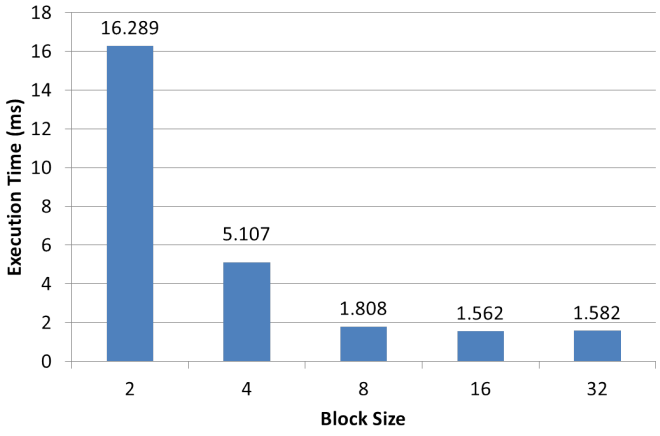


FIGURE 4. Execution time of the baseline

improvement due to block size increase, and any further attempts to increase the block size will result in unnecessary work with minimal improvements.

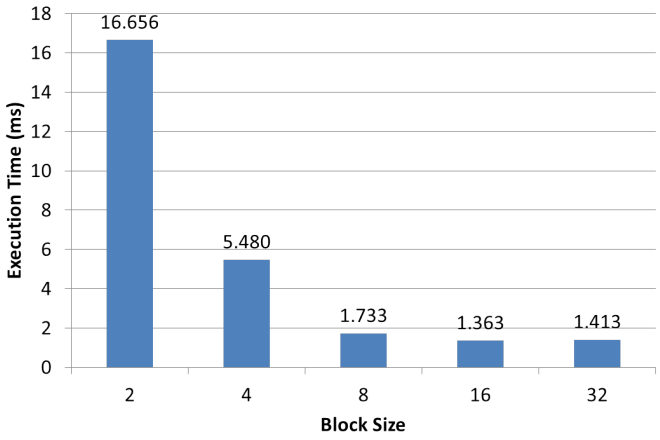


FIGURE 5. Execution time using constant memory (filter)

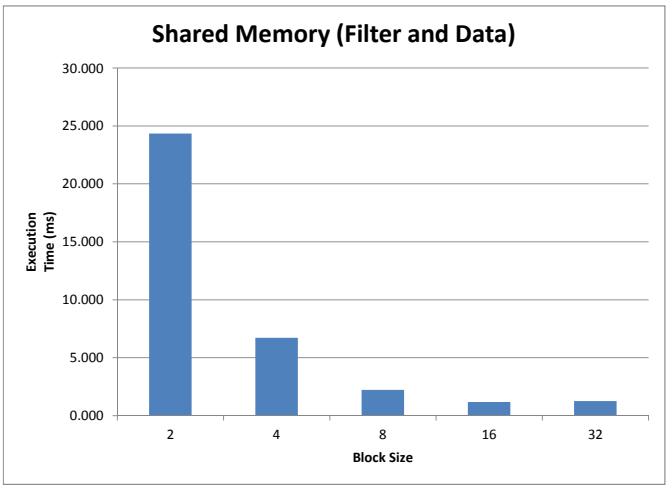


FIGURE 6. Execution time using shared memory filter

Figure 3 compares all the available optimizations and combinations of the optimizations against the baseline. It is clear that using any of the optimizations can improve the execution time, however while each optimization can individually improve the execution time, the lowest time is produced by using a combination of the optimizations for the maximum effect. The usage of available memory types on GPU also affect the execution time, as shown by the use of constant memory vs shared memory or a combination of the two. Using these special, faster kinds of memory on the GPU for the filter and data memory, execution time can also be reduced. The most significant contributed to reduced execution time however from the optimizations is the use of the single kernel, halving the execution time when compared to the baseline. Using these optimizations, we can achieve up to 2.65× improvement on the execution time of the algorithm, as shown in Figure 3 when comparing

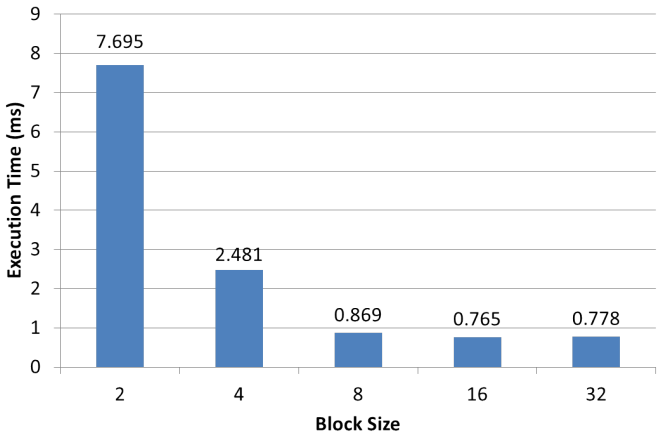


FIGURE 7. Execution time using single kernel

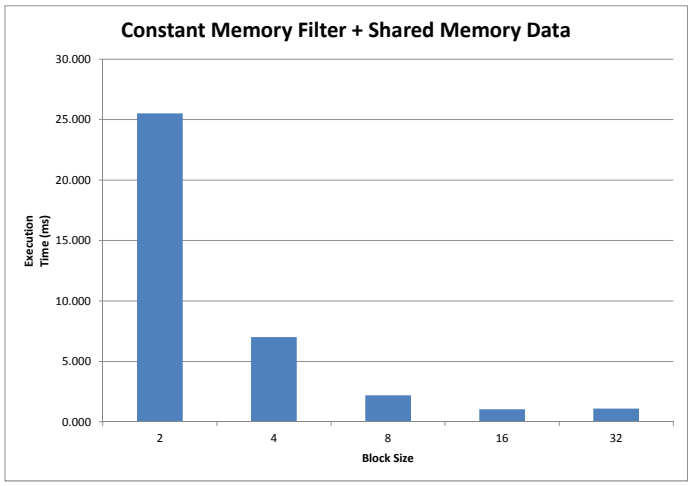


FIGURE 8. Execution time using constant memory (filter) + shared memory

the baseline against the combination of three optimizations (using constant memory for the filter, shared memory and the single kernel).

## 5. CONCLUSION

By identifying and utilizing suitable optimizations on GPUs as well as algorithm-specific optimizations, we can significantly reduce the amount of execution time required to perform the 2D image gaussian blurring algorithm. Using optimizations including memory types and block size, we can achieve up to 2.65× improvement on the execution time of the algorithm. In the future, we can consider additional suitable optimizations and further

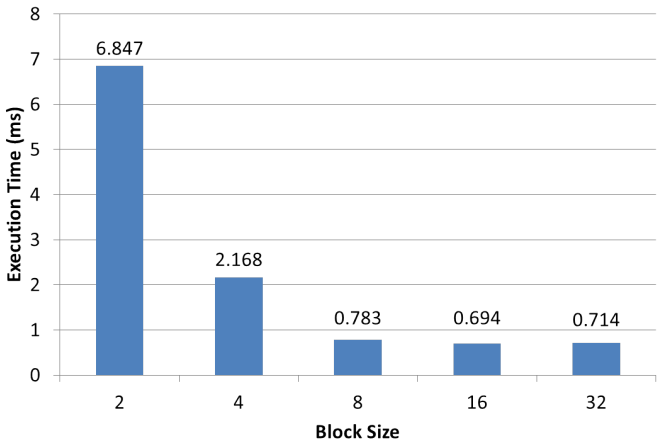


FIGURE 9. Execution time using constant memory (filter) + single kernel

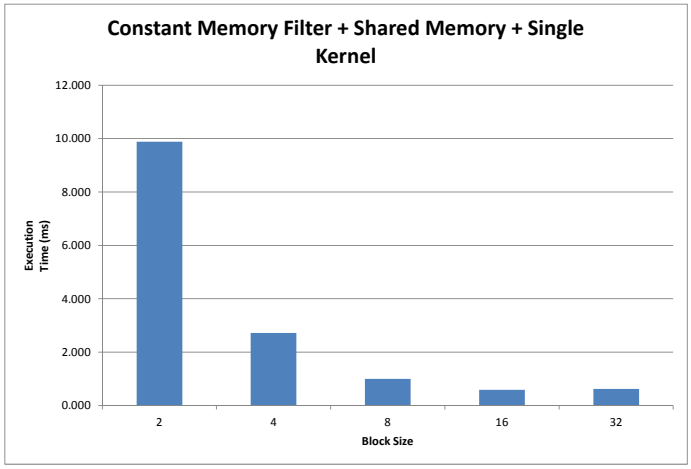


FIGURE 10. Execution time using constant memory (filter) + shared memory + single kernel

change the parameters, as well as compare the results against other available platforms such as FPGA or heterogeneous architectures.

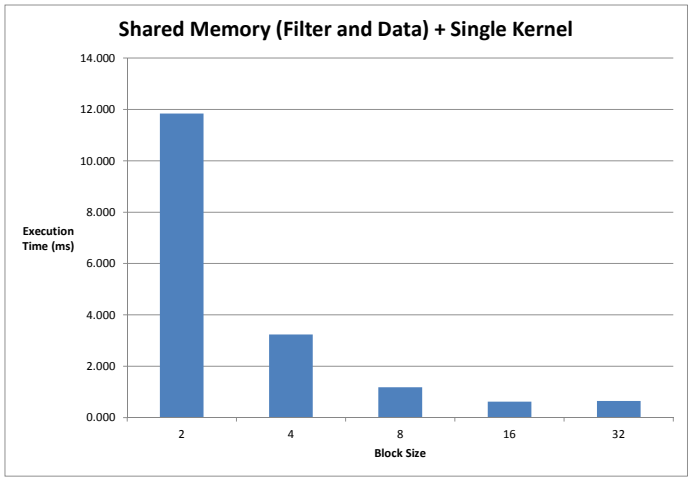


FIGURE 11. Execution time using shared memory + single kernel