# EE 382C: Multicore Computing

## Parallel GPU based Algorithms for Image Processing

Wenbo Xu, Wenwen Zhang, Yichi Zhang

## I. Introduction

Recently, the requirement for GPU (graphics processing unit) performance as well as the computation speed are increasing rapidly. As comparison, GPU computation speed can be several times faster than traditional CPU. Moreover, as the programmability and parallel processing emerge[1], people began using GPU in some non-graphics applications, such as general-purpose computing on the GPU (GPGPU). To be more user-friendly, CUDA brings the C-like development environment and some CUDA extended libraries to programmers, which are based on industry-standard C/C++ and have straightforward APIs to manage devices, memory etc.

As a general use of GPU, image processing algorithms are always computationally expensive, however, parallel image processing algorithms can enhance the speed to a greater extent, especially for large-scale images.

In this paper, we realized Gaussian Filter in both CPU sequential logic and GPU parallel logic. Then, we implemented three improvements for GPU. By using cuda command line tool nvprof and the NVIDIA Visual Profiler, these optimizations have been proved to improve performance a lot.

## II. Gaussian blur

### A. Definition and Usage

In image processing, a Gaussian blur (also known as Gaussian smoothing) is the result of blurring an image by a Gaussian function. It is a widely used effect in graphics software, typically to reduce image noise and detail.

Mathematically, the Gaussian blur is a type of image-blurring filter that uses a Gaussian function for calculating the transformation to apply to each pixel in the image. For our implementation, we use a two-dimension gaussian blur to filter each image pixel. The related two-dimension gaussian function is the product of two such one-dimension function above, on in each dimention.

$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

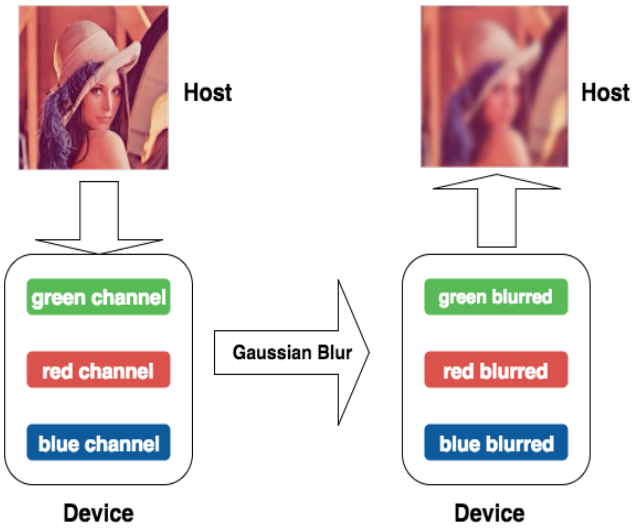


Fig. 1: Diagram for Parallel Image Processing

Variable $x$ is the distance from the origin in the horizontal axis, $y$ is the distance from the origin in the vertical axis, and $\sigma$ is the standard deviation of the Gaussian distribution. A convolution matrix is built with values generated by this distribution. To have a better view of the powerfulness of parallel computing, we did a comparison experiment, which comprised of two implementations of Gaussian blur. One is sequential implementation and the other is parallel implementation.

### *B. Sequential Implementation*

There are four steps of the sequential implementation. The first step is preprocess, which is responsible for reading image file and store it as Image class. The second step is to generate a gaussian filter, a matrix, based on parameters, like the standard deviation, passed from users. The next step is to traverse each pixel in the image object and compute the values of a given pixel in the output image by multiplying each kernel value by the corresponding input image pixel values. The final step is to generate an image file from those new pixel values.

### *C. Parallel Implementation*

In this section, we implemented a normal parallel version of gaussian blurring based on CUDA. In the following section, we optimized this implementation in several ways. First, we read the image file and generate filter matrix. Then we copy the image into device memory. Each pixel is combined with three channels, $r$, $g$ and $b$. They are separated and are gaussian blurred separately. Then we start the kernel three times and do gaussian blurring to each channel every time. At last, we copy three blurred channels from device to host and save them into an image file. This process is described with the Fig.1.

## III. Optimization

### A. Shared memory v.s. Global Memory

In the previous version implementation of Gaussian blur, we store three channels in global memory, and access same global memory several times when we read neighbor channels. Since reading shared memory is much faster (shared memory $1.7TB/s$, while global memory $150GB/s$), instead of pulling from global memory every time, we put channel values in the shared memory for other threads within the same block to see and reuse.

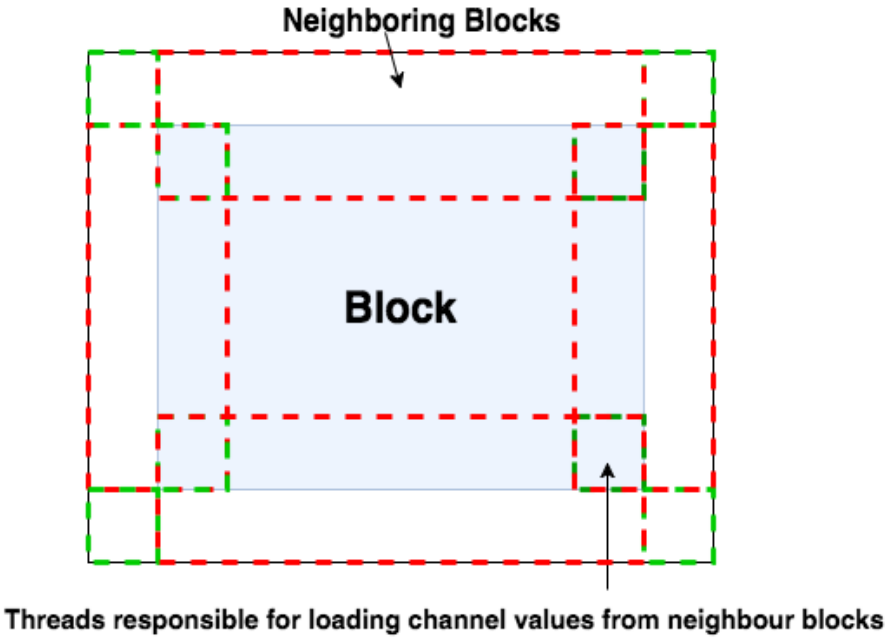


Fig. 2: Loading neighbour channel values for shared variable

In order to compute blurred channel values at the boundary of one block, we need to load channel values from neighboring blocks, so shared memory size should be a little larger than a block size and threads at the boundary area are responsible for loading channel values from neighboring blocks. This is displayed in Fig.2. Threads in blue area of dotted rectangles are responsible for loading channel values from corresponding white area.

### B. Pageable v.s. Pinned Memory

Host data allocations are pageable by default, However, GPU cannot access data directly from pageable memory, but from pinned memory. As shown in Figure [2], whenever a data transfer is invoked on

pageable memory, the CUDA driver has to allocate a temporary pinned memory array to copy host data and then transfer it to the device.

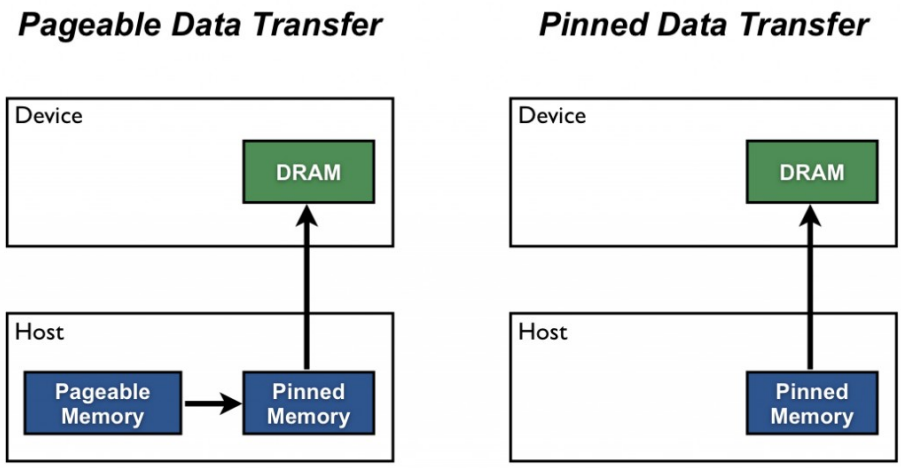


Fig. 3: CUDA data transfer.[2]

We can use *cudaMallocHost()* and *cudaFreeHost()* instead of *malloc()* and *free()*, to remove the overhead described above. However, *cudaMallocHost()* and *cudaFreeHost()* are more expensive with additional overheads. According to Boyer[3], pinned memory is faster when the size of data to be transfered is larger than 16MB.

### *C. Streams*

A stream is defined as a sequence of operations in that execute in issue-order on the GPU. A typical CUDA task consists of three operations, memcpy HToD, kernel, and memcpyDToH. Kernel operations in different streams may run concurrently as long as the hardware resources were available. For memory copies, one transfer per direction is allowed at any time. And memory copies in different streams are in different direction may run concurrently when hardware supports two copy engine.

Figure 8 illustrates the execution time line for three different scenarios. In this project, the pipelined fashion was implemented. In order to pipeline the memory copies, asynchronous memory copiy functions, *cudaMemcpyAsync()*, are used instead of synchronous functions.[4]

## IV. Results

Both CPU and GPU implementations was running on the TACC Stampede Supercomputer. The CPU is Intel Xeon E5-2680 2.7GHz Processors. And the GPU is NVIDIA K20 with 5120 MB GDDR5 memory and 2 copy engines.

The command line tool, nvprof, and the Nvidia Visual Profiler are used to profile the performance of our implementation. Profiling results are shown in Figure 4 and Figure 7 in Appendix. In Figure 4 whith logarithmic scaled y axis,

1. The CPU represents run time of a sequential CPU Gaussian filter as described in Section II-B.

2. The stage1 represents the preliminary GPU parallel Gaussian filter as described in Section II-C.

3. The stage2 represents the GPU parallel Gassian filter with pipelined optimization as described in Section III-B and III-C.

4. The stage3 represents the GPU with shared memory as described in Section III-A additional to the optimization of stage2.

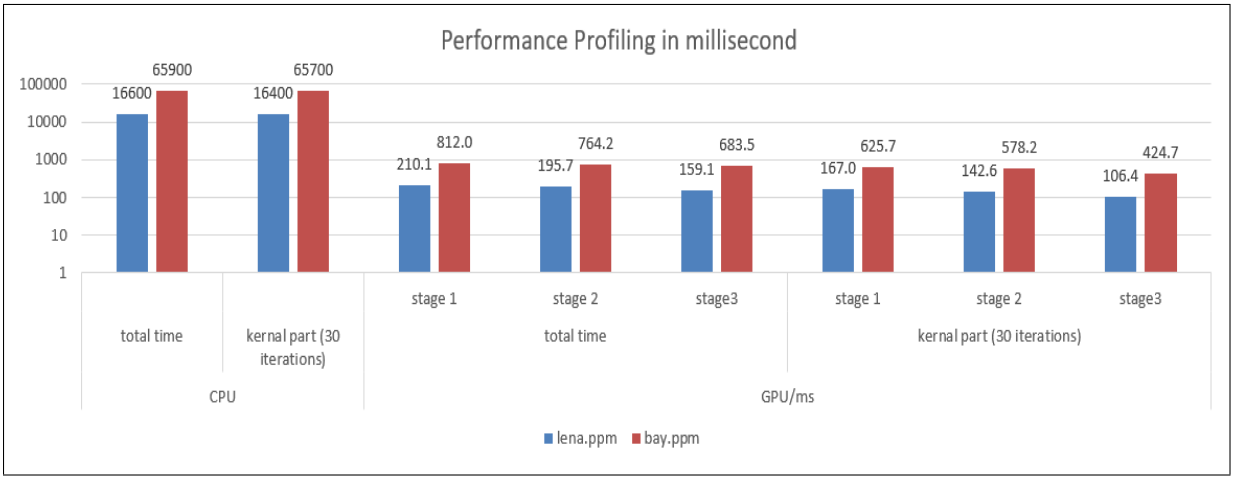


Fig. 4: Performance Improvement in GPU.

Each of the algorithm listed above will be running with two images: lena.ppm with $(512 * 512)$ pixels, and bay.ppm with $(1024 * 1024)$ pixels. For each algorithm, the kernel section, which contains memcpy HToD, actual Gaussian kernel, and memcpy DToH for GPU based algorithms, will be executed 30 consecutive times for better comparison between different stages. The duration of overall execution and Gaussian blur are both profiled.

The visual profiler timeline of the stage1 and stage3 are shown in Figure 5. The timescale of the two timelines are the same. The length difference of the green bars demonstrates the improvement for the kernel executions. There is approximated of 40% improvement when the shared memory for GPU is implemented. And the stage3 with three streams, which are run in a pipelined fashion, and overlapping of kernel operations can be observed. However, the overlapping of kernels doesn't indicate of ture concurrency as described in Figure 8. This is due to the limitation of the hardware resource. The

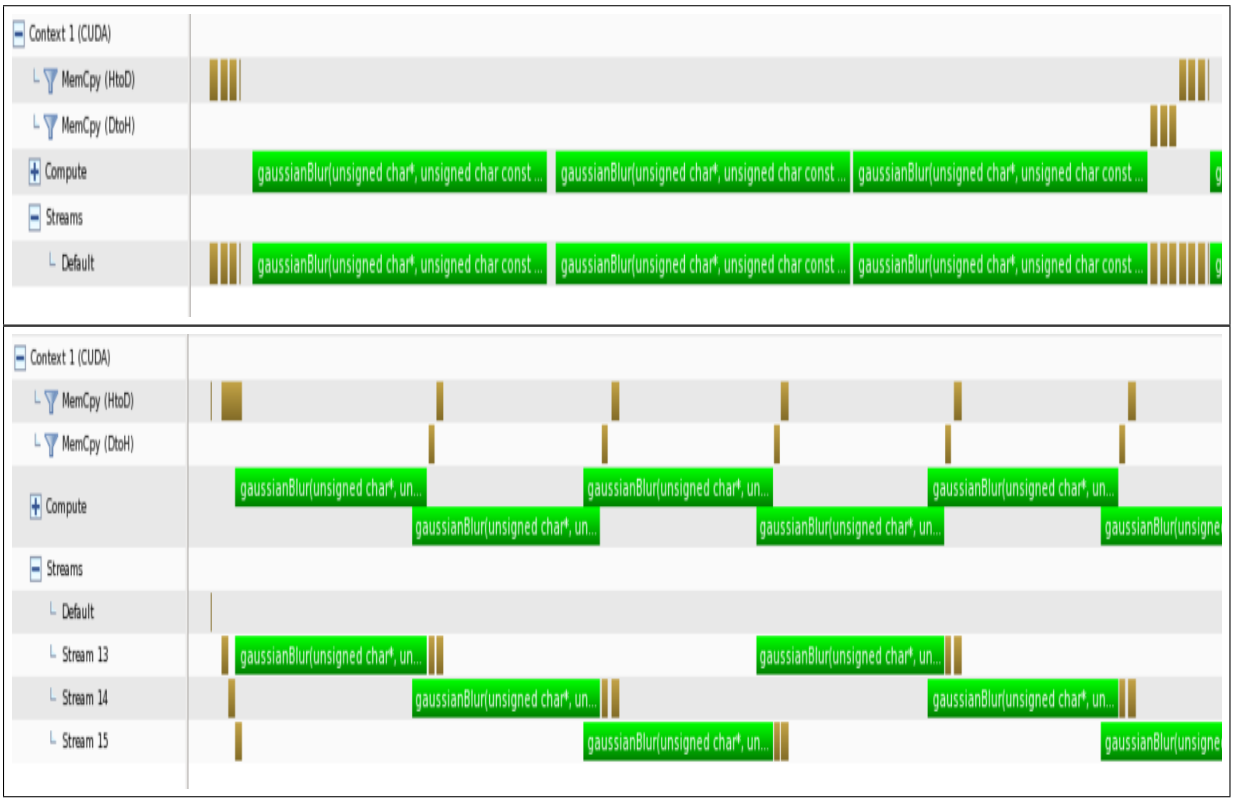


Fig. 5: Time profiler viewer without using streams(top) verses using streams(bottom)

number of blocks can be schedule on GPU is limited. For NVIDIA K20, this limitation is 180 blocks. Whenever this limit is reached, kernels has to wait for next free block.

## V. Conclusion

In this project, we implemented a parallel GPU based Gaussian filter. The GPU based algorithm is benchmarked with two images, $(512 * 512)$ and $(1024 * 1024)$ pixels, and compared with performance of a sequential CPU based algorithm. The GPU based algorithm is optimized with three methods:

1. Memory allocation changed from pageable memory to pinned memory to reduce overhead
2. Stream sequences to allow operations run in pipelined fashion
3. Shared memory inside kernels to improve kernel memory access bandwidth

As shown above, the kernel part time with image lena.ppm for CPU sequential algorithm is 16.4s. This time is improved to 167.0ms with the preliminarily GPU based algorithm, which yields to a speed up of 98.2x. Ultimately, the stage3 achieved a speed up of 156.0x.

## APPENDIX

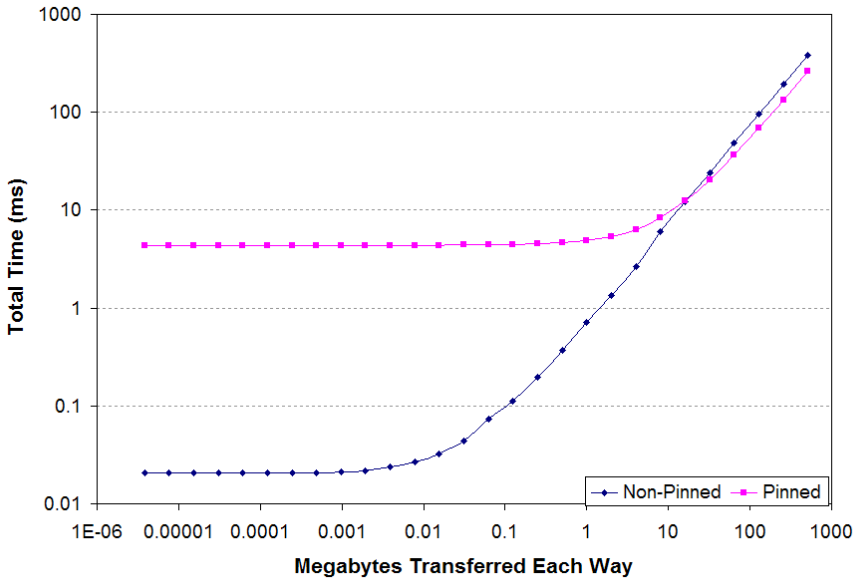


Fig. 6: Time required to allocate, transfer to the GPU, transfer back to the CPU, and deallocate pinned and non-pinned memory.[3]

| image | CPU | | GPU/ms | | | | | |
|---|---|---|---|---|---|---|---|---|
| | total time | kernal part (30 iterations) | total time | | | kernal part (30 iterations) | | |
| | | | stage 1 | stage 2 | stage3 | stage 1 | stage 2 | stage3 |
| lena.ppm | 16.6s | 16.4s | 210.1 | 195.7 | 159.1 | 167.0 | 142.6 | 106.4 |
| bay.ppm | 65.9s | 65.7s | 812.0 | 764.2 | 683.5 | 625.7 | 578.2 | 424.7 |

Fig. 7: Performance Profiling Chart

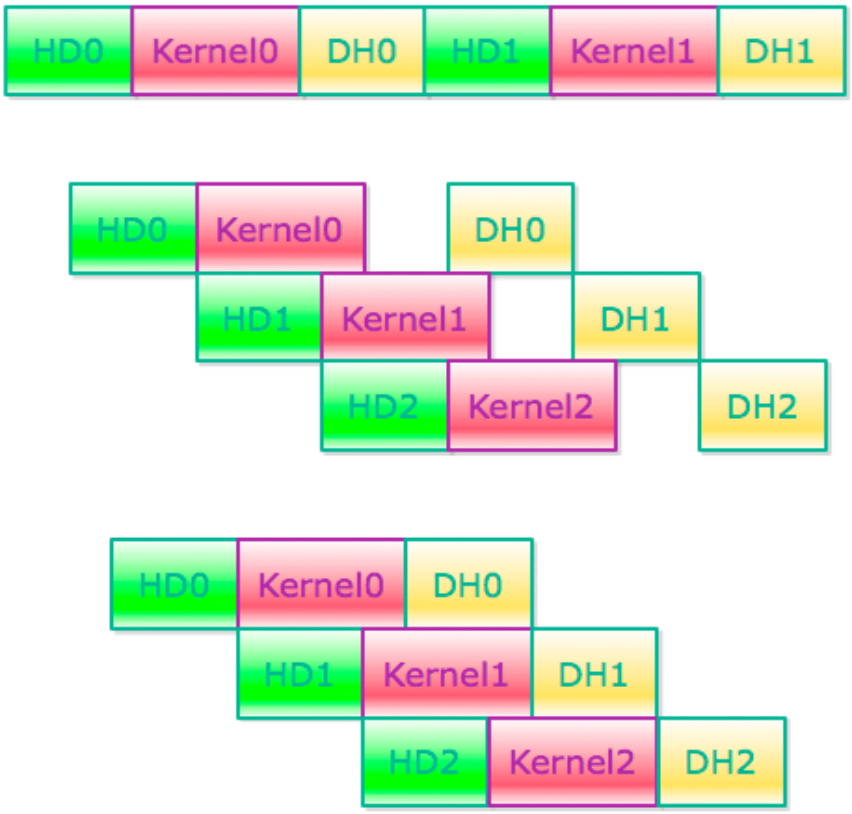


Fig. 8: Top: all operation in default stream. Mid: concurrent streams with one copy engine. Bottom: concurrent streams with two copy engine.

## REFERENCES


[1] WU En Hua , ?State of the Art and Future Challenge on General Purpose Computation by Graphics Processing Unit?, Journal of Software, vol. 15, no. 10, 2004,pp.1493 1504.

[2] Harries, M. (2012, December). How to Optimize Data Transfers in CUDA C/C++. Retrieved from https://devblogs.nvidia.com/parallelforall/how-optimize-data-transfers-cuda-cc/

[3] Boyer, M. Choosing Between Pinned and Non-Pinned Memory. Retrieved from https://www.cs.virginia.edu/ mwb7w/cuda˙support/pinned˙tradeoff.html

[4] Harries, M. (2012, December). How to Overlap Data Transfers in CUDA C/C++. Retrieved from https://devblogs.nvidia.com/parallelforall/how-overlap-data-transfers-cuda-cc/