2021 IEEE Western New York Image and Signal Processing Workshop (WNYISPW) | 978-1-6654-3630-4/21/$31.00 ©2021 IEEE | DOI: 10.1109/WNYISPW53194.2021.9661289

Accelerating 2-D Image Convolution Using a Graphics Processing Unit

Charles Yoo
Department of Mechanical Enginering
Villanova University
Villanova, PA, USA
cyoo@villanova.edu

Shadi Alawneh
Department of Electrical and Computer Engineering
Oakland University
Rochester, MI, USA
shadialawneh@oakland.edu

***Abstract*— Image processing is an important technique that is used in many fields, such as self-driving vehicles or facial recognition. One method is called image convolution, which involves many calculations that manipulate the pixels of an image to produce a new image with a desired effect. This is computation intensive and requires a significant amount of time when run on a traditional computer processing unit (CPU). Since image processing is used for real-time applications, such as those mentioned above, it is essential that convolution algorithms run as quickly as possible. A common way to speed up image convolution algorithms is to take advantage of the highly parallel structure of graphical processing units (GPU) to perform concurrent calculations. One problem with GPU applications is that they are often limited by the latency delays associated with transferring data between the CPU and the GPU. Previous works have looked into different ways to address this issue and optimize GPU programs. This research aims to explore different memory implementations and compare them to see which is best at optimizing data transfers.**

Keywords—GPU acceleration, Image convolution

I. Introduction

A. Image Convolution

Image convolution is a method for image processing where a small filter is moved along an image that needs to be processed. As the filter is moved over the image, it is centered over each individual pixel in the image and element wise multiplication is performed between the filter and the portion of the image that it covers. These products are then added together to get an output pixel that corresponds to the input pixel that you centered over. When creating an image convolution algorithm, the data in the input image and the filter being applied are stored in matrices.

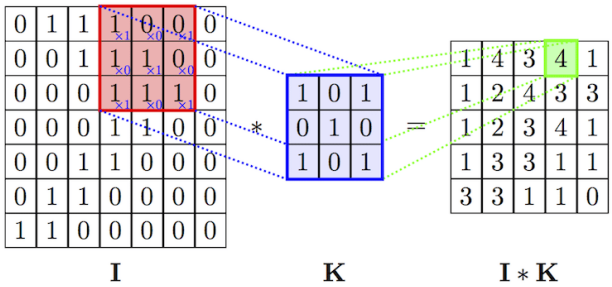

Fig 1. An example of Image Convolution [1]

In figure 1, there is an image matrix that contains the pixel values of the input image and a filter matrix that contains values which are determined based on how the image is processed. In this particular instance, the filter is centered over the pixel boxed in red and each element in the covered portion of the image matrix is multiplied by the corresponding element in the filter matrix. These products are summed up to produce a pixel value in the output image. Once this is done with every pixel in the input image, the output image is complete.

Using image convolution, images can be processed in a variety of different ways. For example, in the case of black and white images, image convolution can be used to color them, or to do the opposite and convert colored images into grayscale. Image convolution can also be used for noise reduction to get more accurate coloring in an image. Additionally, image convolution can be used to sharpen images and make fine details more pronounced. As mentioned before, the values of the filter matrix can be found based on how the input image is processed. For example, an edge detection filter matrix is generally [2]:

$$\begin{matrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{matrix}$$

However, these values are not a fixed rule and can be adjusted in order to achieve the desired effect and intensity.

There are many applications of image convolution. Certain applications, like self-driving cars, occur in real-time so the processes and mechanisms behind such applications must be capable of handling this [3]. While, image convolution is relatively fast, it should be made as fast as possible so that it can be performed in real-time. This research aims to utilize the thousands of cores in a graphics processing unit to see how much an image convolution program can be accelerated.

B. OpenMP

Many codes are commonly written for serial computing. This means that the lines are run sequentially on a single thread. Since image convolution involves many computations [4], performing these one at a time is very time consuming. These computations are very similar, which makes them ideal for parallel computing [5]. Unlike serial computing, parallel computing has lines of code that can be executed concurrently on multiple threads.

One way to create parallel computing codes is to use OpenMP. OpenMP is a library that can be used in C, C++, and

National Science Foundation Grant No EEC-1659650.

978-1-6654-3630-4/21/$31.00 ©2021 IEEE

Fortran by including omp.h. OpenMP enables the user to utilize multiple CPU threads, which allows for parallel programming [6]. With parallelism, the workload can be split between the different threads for sections of the code where many similar operations are being performed. By using multiple threads, these operations can be performed much more efficiently than when using a single thread [7]. However, it is important to note that there will be sections of the code that cannot be executed in parallel. For instance, memory allocation and variable initialization must be performed sequentially on the main thread. To differentiate between parallel and sequential sections, OpenMP has special compiler directives [8]. OpenMP also has other compiler directives that can be used to control the threading behavior. Among these are directives that can specify how the workload is split up between the threads and when the threads should be synchronized.

C. *Cuda*

Cuda is a parallel computing platform that is an extension of C, C++, and Fortran. It is similar to OpenMP in that it also enables the use of multiple threads. However, the key difference is that Cuda uses threads that are run on a graphics processing unit, which have many more cores than a computer processing unit.

This allows for the use of many threads for massively parallel computations. The main part of the code is still run sequentially and in the main code on the CPU. These parts generally contain setup code such as variable initialization and memory allocation. On the other hand, the parallel portions of the code are written as functions, called kernels, that are called in the main code to be run exclusively on the graphics processing unit. Kernels generally contain similar and repetitive operations that can be performed in parallel [9].

When calling these kernels, the number of thread blocks and threads in each block are specified. Each of these blocks and individual threads have their own id, which can be used to split up the work between the threads. Generally, memory must be allocated on both the CPU and the GPU for entities that are used in the parallel sections. As such, data must be first be loaded in the main code on the CPU. It is then transferred to the GPU before the computations have occurred and then back to the CPU once the computations have occurred. Additionally, Cuda adds a variety of functions, including one that specifies when threads are to be synchronized [10].

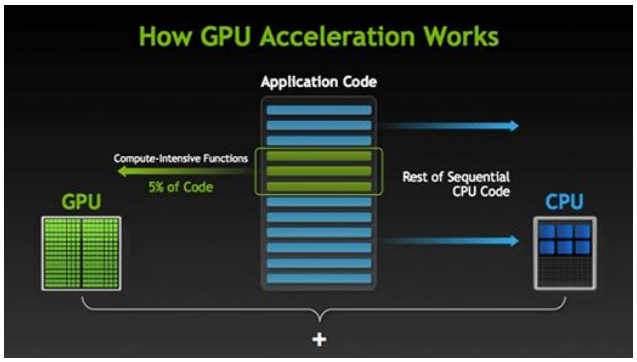

Fig 2. The distribution of work for a code using Cuda [11]

There are a variety of different memory types that are available when using Cuda. One of them being global memory, which is system memory that can be used by the GPU. Variables using this memory can be designated using the __global__ keyword. The global memory is the total amount of DRAM of the GPU and is accessible by all the threads on the GPU. It is also the largest, but slowest, memory type that can be used for the GPU. Another type of memory that can be used by the GPU is the shared memory. This type of memory is also system memory and can be designated using the __shared__ keyword. The shared memory is smaller than global memory but offers faster access. Shared memory is allocated per thread block and is shared between threads that are in the same thread block. This facilitates memory coalescing and eliminates redundant transferring of data between the individual threads. However, since the threads within each block are sharing memory, it is important to synchronize the threads to prevent memory consistency errors [12]. One other type of memory accessible by the GPU is the constant memory. This memory is dedicated memory located on the GPU and can be designated with the __constant__ keyword. The constant memory is much smaller than the global memory and is read-only. However, the constant memory offers aggressive caching, which allows for much faster data transfers than the global memory. It is also accessible to all the threads on the GPU [13].

By default, memory allocated on the CPU is pageable memory. This type of memory is only accessible on the CPU and is allocated using malloc(). Pageable memory can be paged in and out to be used for other processes. In order to transfer data from the CPU to the GPU, the memory must be copied from the pageable memory to a temporary, unpageable memory and then transferred to the GPU. To eliminate this additional memory transfer, memory on the host can be allocated to pinned memory. This memory is also only accessible on the CPU and is allocated using cudaMallocHost() [14]. The pinned memory is page locked and can't be used for other processes. With this memory, data can be directly transferred to the GPU. Another method for allocating memory on the CPU is unified memory. This managed memory is actually accessible from both the CPU and the GPU and is allocated using cudaMallocManaged(). Since the unified memory is accessible from both devices, no data transfers need to occur [15].

II. Related Work

There have been various other studies exploring the use of parallel computing. Tousimojarad et al. [16] explored the speedup of an image convolution algorithm when using 3 different CPU parallel programming models. The three models that were tested were: OpenMP, OpenCL, and GPRM. In this work, it was observed that OpenMP had the best overall performance. Afif et al. [17] sought to improve convolution algorithm execution times by using a GPU implementation. In this work, a GPU implementation was compared to a CPU implementation for a Sobel filter and a Gaussian blur convolution. Payne et al. [18] also investigated the use of GPUs to accelerate image convolution algorithms. In this work, they compared a GPU implementation to a CPU implementation that was optimized using the Fast Fourier Transform. Karas and Svoboda [19] looked into the use of a Cuda parallel computing model to accelerate a 3-D image convolution algorithm. They

compared GPU and CPU implementations as well as multi-GPU and multi-CPU implementations. One important thing mentioned in this work is that the image can be divided into smaller parts to circumvent limitations of the GPU's memory size. Lu et al. [20] aimed to improve GPU image convolution by optimizing memory access. They did this by avoiding redundant loading of the same image elements and improving data locality of image elements within the same row. Iandola et al. [21] explored ways to get around the memory limitation of GPU accelerated image convolution algorithms. They did this by prefetching image regions directly to registers and using fewer threads to maximize the work done per thread.

III. Setup

For this research, the software that used were Nvidia Cuda tool kit 11.3 and Microsoft Visual Studio. The research was conducted on a machine containing an AMD Ryzen R5 3600, with 6 cores, and an Nvidia RTX 2060, with 1920 Cuda cores.

For the image convolution, a 7x7 mask was used on images of varying sizes. These image sizes were: 256x256, 512x512, 1000x1000, 2000x2000, 4000x4000, 8000x8000, 16000x16000, and 20000x20000. Since the contents of these images does not have any significant effect on the collection of data, these images were randomly initialized in the program before performing the convolution. For each image size, the image convolution was performed 20 times and the measured times were averaged.

When collecting data, 2 CPU implementations and 8 GPU implementations were tested. The two CPU implementations were a single thread and a 12-thread version, with the 12 threads corresponding to 2 threads per core. The first GPU code was a base code that performed the image convolution without any optimizations. From there, a second version was created that uses shared memory to preload the image data before performing the calculations. Finally, more implementations were created to observe the effect of using constant memory, unified memory, and pinned memory on the overall run time. These implementations were tested once without preloading data and again with preloading the data. For all of these implementations, the convolution calculation, setup, and total times were measured for the different image sizes.

IV. Results

A. Comparing CPU and GPU Implementations

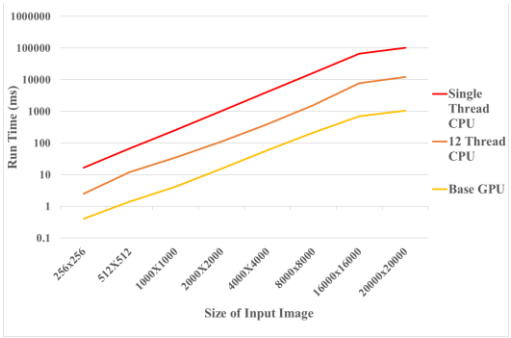

Fig 3. Calculation Time for CPU and GPU Implementations

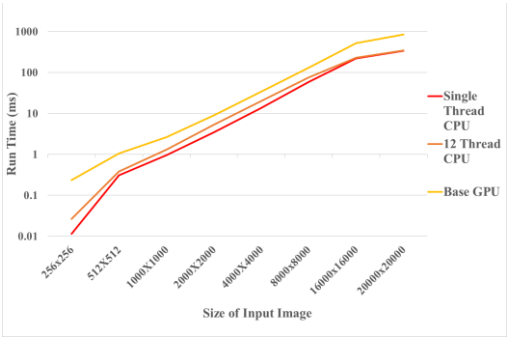

Fig 4. Setup Time for CPU and GPU Implementations

Figures 2 and 3 compare the convolution and setup times for the single thread CPU, multi-thread CPU, and base GPU implementations. In figure 2, the GPU code performs calculations the fastest for all image sizes. The GPU code is then followed by the CPU multi-thread code, which is faster than the CPU single thread code. On the other hand, in figure 3, the single thread and multi-thread CPU codes have similar setup times and are both faster than the base GPU. This makes sense because memory must be allocated twice and data needs to be transferred between the CPU and GPU for the base GPU implementation. However, it is important to notice that even though the GPU has the longest setup time, it is still not nearly as long as the calculation times for the CPU codes. For instance, for the 20000x20000 image, the setup time for the GPU is a little less than 1000 milliseconds whereas the convolution times for the single thread and multithread CPU are around 100,000 and 10,0000 milliseconds, respectively. Therefore, the base GPU implementation is still much faster than either CPU implementation.

B. Comparing Different GPU Implementations

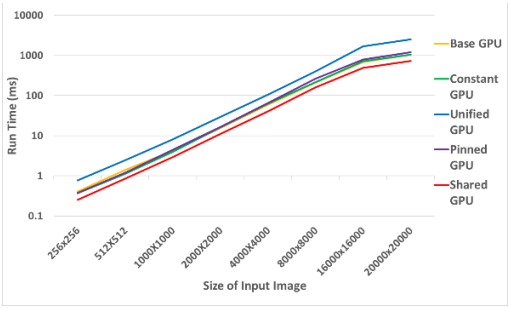

Fig 5. Calculation Time for Different GPU Implementations

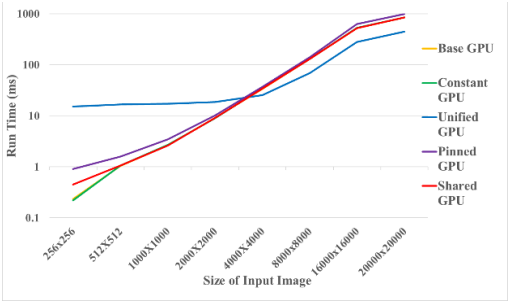

Fig 6. Setup Time for Different GPU Implementations

In this section, the calculation and setup times for the different GPU implementations will be compared. In figures 4 and 5, the times for the base GPU are compared with the times for each of the individual memories. In figure 4, the calculation times are very similar between the different codes. The pinned memory is slightly faster than the base GPU implementation, the constant memory is even faster, and the fastest calculation time comes with the use of shared memory. On the other hand, unified memory takes the longest amount of time and is even slower than the base GPU implementation. Figure 5 shows that the shared and constant memories both have nearly the same setup times as the base GPU implementation, hence why the 3 lines largely overlap. The code that used pinned memory is slower than these 3 implementations. Meanwhile, the code that used unified memory is initially slower than all of the other ones. However, as the image size increases, unified memory becomes more worthwhile to use because it has the fastest setup time. Since the shared memory code has the fastest calculation and setup time, it is the fastest out of these implementations.

The use of shared memory to preload data is not mutually exclusive with the use of the other memory types. Therefore, more research was done to compare code that just used shared memory with codes that used shared memory and another memory type. Figures 6 and 7 compare the calculation and setup times for these codes. As seen in both figures, the behavior of the codes using unified memory and pinned memory remain unchanged even when using shared memory to preload the data. On the other hand, the behavior code that uses constant does change when using shared memory. Without the use of shared memory, the calculation time for the constant memory code was slower than the shared memory code (as seen in figure 4). However, when the data are preloaded, the constant memory code actually has a faster calculation time than the code that just used shared memory. Since the code using both constant and shared memory has effectively the same setup time as the shared memory code, it is the fastest implementation.

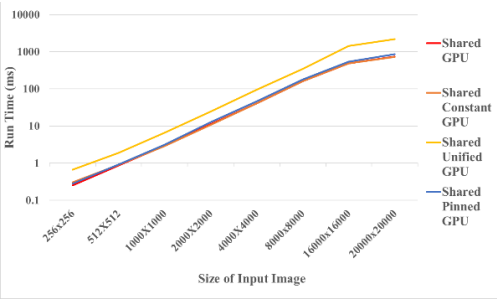

Fig 7. Calculation Time for Different Preloaded GPU Implementations

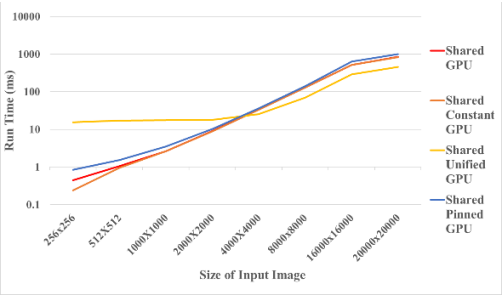

Fig 8. Setup Time for Different Preloaded GPU Implementations

C. *Comparing Total Run Times*

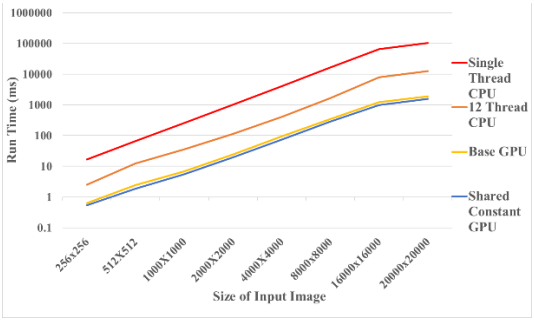

Fig 9. Total Run Time for CPU and GPU Implementations

When looking at the total run times it becomes definitively clear that the GPU implementations are better than the CPU ones. It can also be observed that the difference between the base GPU and the fastest GPU code is negligible compared to the difference between the GPU and CPU codes. This suggests that optimizations made to the GPU codes, while increasing performance, only offer marginal improvements compared to the overall improvement that comes from using a GPU implementation over a CPU one.

V. Conclusion and Future Work

From this research, the main conclusion is that GPU image convolution implementations are significantly faster than CPU ones, especially when processing larger images. One important thing to note is that the GPU used in this research only had 1920 cuda cores. There are currently GPUs that exist with over 10,000 cuda cores, which offer even more potential for massively parallel computations and bigger speed ups. However, the speed of GPU codes is limited by memory setup so it is important to optimize these to increase performance. The tested optimizations only offered marginal improvements over the overall performance boost from simply using a GPU over a CPU to perform the calculations.

In the future, more work can be done to test other optimizations in order to determine whether significant improvements can be made over an unoptimized GPU implementation. This is especially important for memory-related optimizations that may ease the limitations of memory allocation and data transfers. Additionally, research can be done using GPU acceleration in real-world applications. Such research can offer a wider range of experimentation and lead to a better understanding of the extent to which GPU acceleration is effective.

Acknowledgment

This research was supported by the National Science Foundation Grant No EEC-1659650. Additional support was provided by the School of Engineering and Computer Science at Oakland University.

REFERENCES

[1] I. S. Mohamed, "Detection and tracking of pallets using a laser rangefinder and machine learning techniques," M.S. thesis, Comp. Sci., Univ. degli studi di Genova, Genoa, Italy, Sept. 2017.

[2] Y. Kirani, "Multi-level edge detectors based on convolution matrices of base-lengths 2 and 3," *aprn Journal of Engineering and Applied Sciences*, vol. 5, no. 1, Jan. 2011.

[3] D. Hernández, G. Olague, B. Hernández, and E. Clemente, "CUDA-based parallelization of a bio-inspired model for fast object classification," *Neural Computing and Applications*, vol. 30, no . 10, pp. 3007–3018, Nov. 2018.

[4] D. Akgün and P. Erdoğmuş, "GPU accelerated training of image convolution filter weights using genetic algorithms," *Applied Soft Computing*, vol. 30, May 2015.

[5] N. Zhang, Y. Chen, and J. Wang, "Image parallel processing based on GPU," 2010 2nd International Conference on Advanced Computer Control, pp. 367-370, Mar. 2010.

[6] H. Jin, M. Frumkin, and J. Yan, "The OpenMP implementation of NAS parallel benchmarks and its performance," *NAS Technical Report*, Oct. 1999.

[7] S. Yu, M. Clement, Q. Snell, and B. Morse, "Parallel algorithms for image convolution," in *Proceedings of the International Conference on Parallel and Distributed Techniques and Applications*, Las Vegas, NV, 1998.

[8] H. Jang, A. Park, and K. Jung, "Neural network implementation using CUDA and OpenMP," *2008 Digital Image Computing: Techniques and Applications*, pp. 155-161, Dec. 2008.

[9] L. M. Russo, E. C. Pedrino, E. Kato, and V. O. Roda, "Image convolution processing: A GPU versus FPGA comparison," *2012 VIII Southern Conference on Programmable Logic*, pp. 1-6, Mar. 2012.

[10] S. I. Park, S. P. Ponce, J. Huang, Y. Cao, and F. Quek, "Low-cost, high-speed computer vision using NVIDIA's CUDA architecture," *2008 37th IEEE Applied Imagery Pattern Recognition Workshop*, pp. 1-7, Oct. 2008.

[11] S. Goyat and A. Sahoo, "Scheduling algorithm for CPU-GPU based heterogeneous clustered environment using map-reduce data processing," *ARPN Journal of Engineering and Applied Sciences*, vol. 14, no. 1, Jan. 2019.

[12] Z. Zhao, L. Song, R. Xie, and X. Yang, "GPU accelerated high-quality video/image super-resolution," *2016 IEEE International Symposium on Broadband Multimedia Systems and Broadcasting (BMSB)*, pp. 1-4, 2016.

[13] P. Chen, M. Wahib, S. Takizawa, and S. Matsuoka, "Pushing the liits for 2D convolution computation on CUDA-enabled GPUs," *IPSJ SIG Technical Report*, vol. HPC-163, no. 22, Mar. 2018.

[14] M. Beheshti Roui, K. Shekofteh, H. Noori, and A. Harati, "Efficient scheduling of streams on GPGPUs," *The Journal of Supercomputing*, vol. 76, Nov. 2020.

[15] R. Landaverde, Tiansheng Zhang, A. K. Coskun, and M. Herbordt, "An investigation of unified memory access performance in CUDA," *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1-6, Sept. 2014.

[16] A. Tousimojarad, W. Vanderbauwhede, and W. P. Cockshott, "2D image convolution using three parallel programming models on the Xeon Phi," unpublished.

[17] B. Payne, S. O. Belkasim, M. C. Weeks, and Y. Zhu, "Accelerated 2D image processing on GPUs," *Lecture Notes in Computer Science*, vol. 3515, 2005.

[18] F. N. Iandola, D. Sheffield, M. J. Anderson, P. M. Phothilimthana, and K. Keutzer, "Communication-minimizing 2D convolution in GPU registers," 2013 IEEE International Conference on Image Processing, 2013, pp. 2116-2120.

[19] P. Karas and D. Svoboda, "Convolution of large 3D images on GPU and its decomposition," *EURASIP Journal on Advances in Signal Processing*, 2011.

[20] M. Afif, Y. Said, and M. Atri, "Efficient 2D convolution filters implementations on graphics processing unit using Nvidia Cuda," *International Journal of Image, Graphics and Signal Processing*, vol. 10, no. 8, 2018.

[21] G. Lu, W. Zhang, and Z. Wang, "Optimizing GPU memory transactions for convolution operations," 2020 IEEE International Conference on Cluster Computing (CLUSTER), 2020, pp. 399-403, doi: 10.1109/CLUSTER49012.2020.00050.