Parallel Image Processing

SUMMARY

We implemented a parallelized version of alpha blending on Google Cloud Platform using PyCUDA. Our optimized parallel implementation achieves 288x speedup on the colored test image of size 1024x1024 compared to the sequential version, while the original parallel version achieves 174x speedup.

BACKGROUND

We are trying to parallelize alpha blending, an algorithm that allows users to combine one foreground image with another background image with given transparency. Alpha value controls the transparency, and it takes a value between 0 and 1. Alpha blending processes the foreground image and the background image pixel by pixel. For each color of each pixel, it calculates a weighted sum of the color's value from the foreground image and that from the background image:

$$C_{new} = \alpha\ C_{foreground} + (1 - \alpha)\ C_{background}$$

In the following figure, taking an alpha value of 0.6, using the blue rectangle as foreground image and the orange circle as background image, we can obtain an output image where the blue rectangle and orange circle transparently overlap with each other.

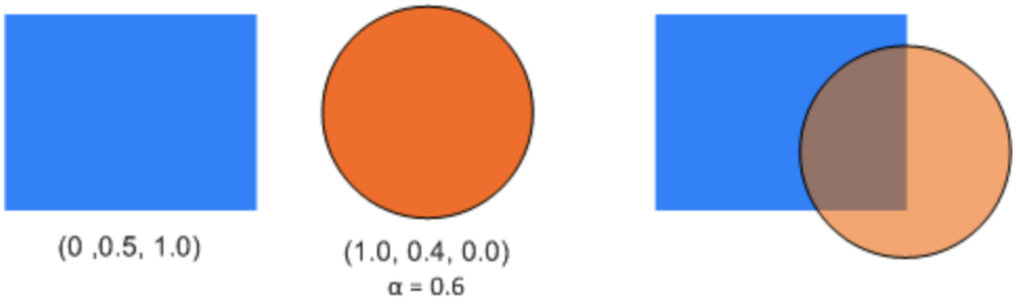


Figure 1. Combine two images using alpha blending

Our implementation receives two images as inputs, and combines them into a merged image as output. Each image is treated as a floating point array. The original method processes each pixel sequentially and that takes up most computation time. Since computation of each pixel doesn't rely on each other, we can divide pixels into blocks and parallelize them on different threads. Then we also optimized on the memory access pattern to further improve computation speed.

APPROACH

To implement our system, we initially tried to write the project in C/C++ with CUDA library on the PSC machines. However, due to a lack of available libraries for working with image files and our pivoted approach to explore a different domain-specific framework, we decided to implement our system using PyCUDA which enables us to program in the CUDA programming model using Pythonic syntax and CUDA's API. PyCUDA helps us develop high performance systems with great efficiency. We benchmarked the results on Google Cloud Platform with 2x vCPU, 24 GB of RAM, and Nvidia Tesla P100 with 16GB of GPU Memory.

The sequential version of alpha blending iterates over all the pixels of the pairs of images and combines the pixels' color. Our first naive parallel version aims to map each of those pair operations to a thread. We then break the image into blocks that map to CUDA blocks, which are groups of CUDA threads. Initially, we tested on greyscale images with single color channels. Hence for a 256x256 image, we launch 16x16 blocks, each block of size 16x16x1 (total of 256 threads per CUDA block). The mapping can be seen in the following figure.

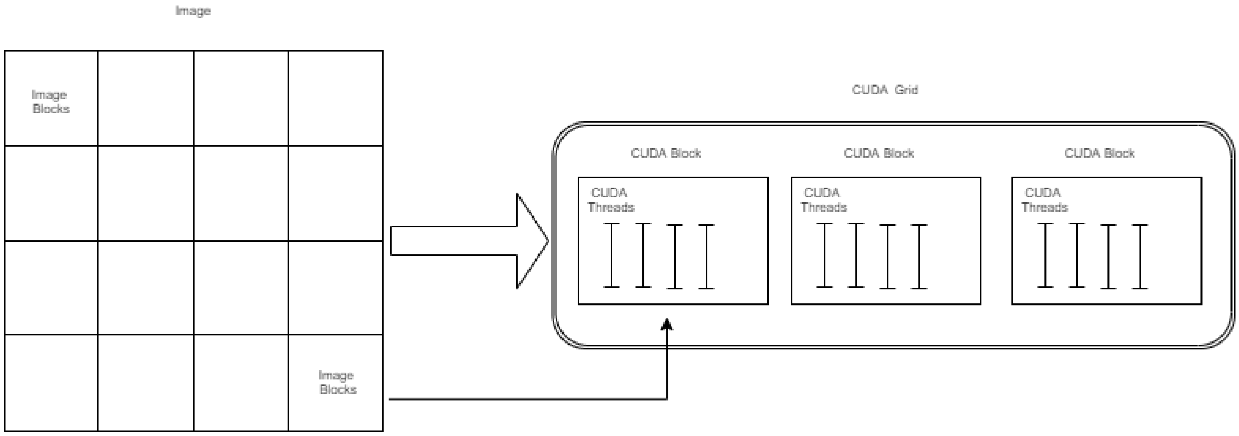


Figure 2. Image block to CUDA block mapping

Since the CUDA architecture limits the number of threads per block to be a total of 1024 threads, the largest image block size for greyscale images are 32x32. And the block dimensions can be assigned arbitrarily by the user with an upper limit of 65535 for both x and y dimension due to memory limit.

The results of our first parallel adaptation of the alpha blending algorithm can be seen in the first part of the results section. On average, our first try of parallel alpha blending achieves a speedup of 705.19 on a 256x256 pairs of greyscale images.

Next we adapt our image block approach for colored image which typically consist of 3 to 4 channels, red, green, blue, and alpha which handles transparency. Our mapping for the colored images become 16x16x4 (as each block's dimension) for each CUDA block, with 16 in height and width, 4 for the number of channels. We also see a similar speedup results for our parallel RGB alpha blending method.

Our initial approach in parallelizing the alpha blending algorithm did not take into consideration of the memory hierarchy of GPUs. Hence we decide to improve on that by optimizing memory access patterns as well as caching pixels instead of constantly accessing global memories.

First, we tried to analyze the memory access pattern of our image blocks approach. Since this approach divides the image as blocks and iterate through pixels in row-major order, we first changed the iteration to column-major order so that GPUs can coalesce reads and writes to memory blocks, resulting in 1 read and 1 write per warp instead of N reads and writes, where N is the width of the image block. However, we did not see a boost in terms of speedup. The column-major approach is still memory-bounded. After some analysis, we discovered that this is due to the image containing multiple channels and how 3d numpy arrays are stored and transferred to GPU device memory. The image is stored in channel-major order, as seen in the following figure.

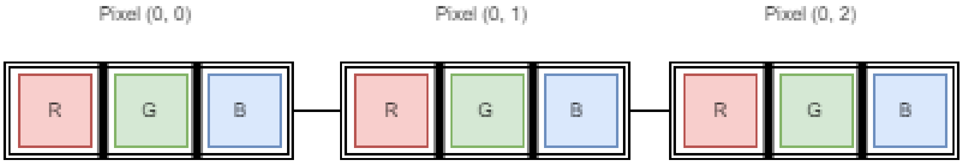


Figure 3. Memory Location of Image Data

Therefore, changing the access pattern from row-major order to column major order cannot reduce the number of reads / writes.

Instead, what we can do is to iterate through the image data array stored in the memory and assign each subpart of the array to the streaming multiprocessors (SM) on GPUs for better spatial locality. In addition, we cached those subparts in the shared memory for each SM so the thread blocks does not have to repeatedly access the slower global memory.

RESULTS

We measured on performance in terms of raw speedup. We initially tried to measure performance through PyCUDA's event, but the results are off potentially due to a bug in the PyCUDA library. Therefore, we decided to measure the speedup between the time it takes transfering the data to the GPU, running the image blending program, and transferring the data back to the host. As the speedup provides better relative comparison between the proposed optimization methods.

We randomly generate test images of different sizes using Python's numpy library. For all the experiments we decide to illustrate with the following four sizes: 4x4, 16x16, 256x256, and 1024x1024. The 4x4 image fit inside one warp operation. The 16x16 image can fit inside one GPU thread block. The 256x256 needs to be parallelized across GPU SMs. And we want to see the performance on a large 1024x1024 image. The images are initialized or read as numpy float matrices, and then transferred to the GPU device through PyCUDA's wrapper for transferring from host to device. The CUDA code are compiled as SourceModules and called as python functions with pointers to data on the GPU device. We call each implementation from 100 to 10000 times to average out possible random variations.

For all the experiments below, the baseline is a sequential version of alpha blending implemented in Python using Numpy with vectorized operations. Initially we assumed that the optimal speedup can scale with number of pixels linearly up to certain level due to diminishing returns as well as the limitation of the number of SMs on the GPU device. However, we also discovered some other interesting facts.

Table 1. Image blocks Approach (Greyscale)				
Image Size	4x4	16x16	256x256	1024x1024
Speedup	1.14108621185	16.6461363212	705.189021624	609.772172653

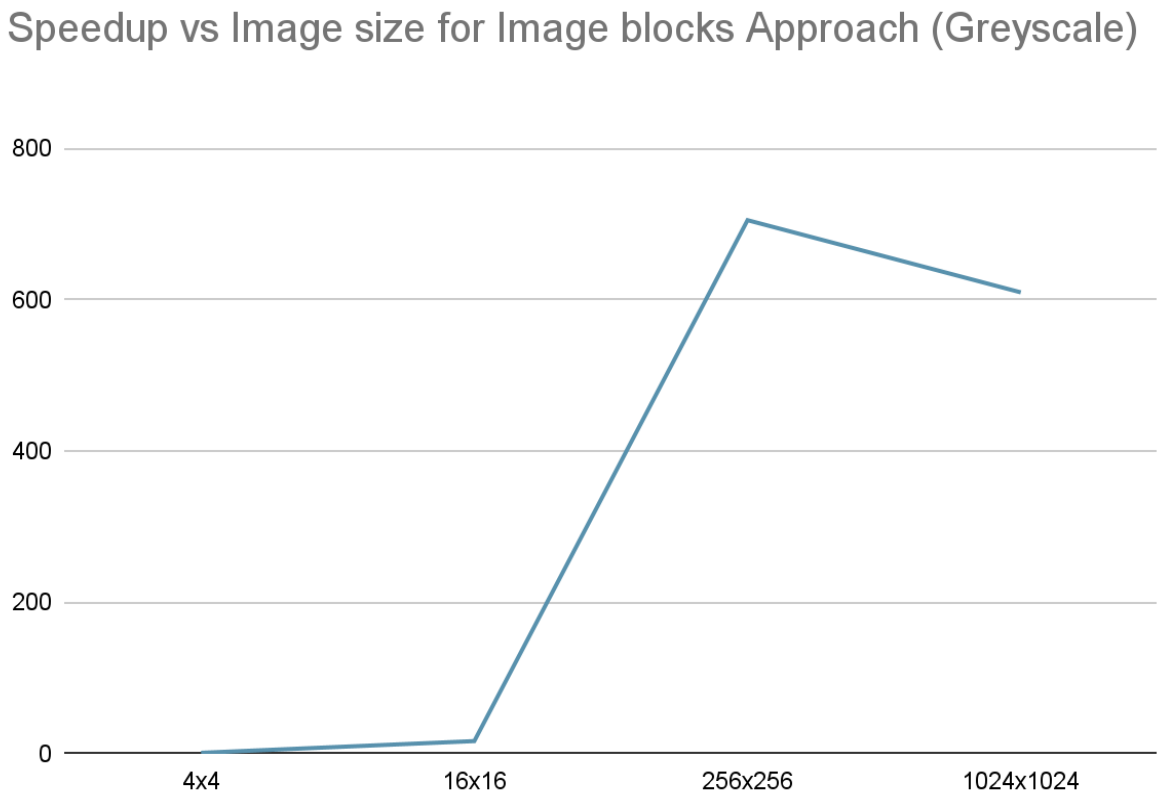


Table 2. Image blocks Approach (Colored)				
Image Size	4x4	16x16	256x256	1024x1024

Speedup	1.14382820814	14.8453458257	109.151839442	174.750616351

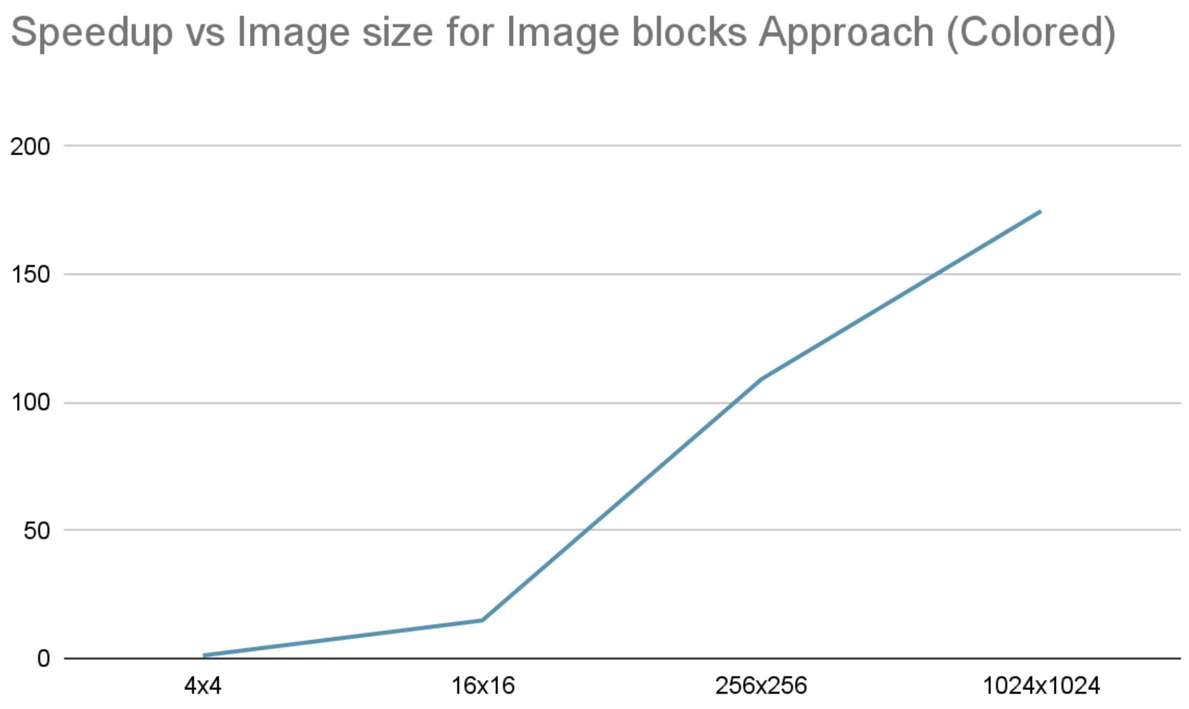
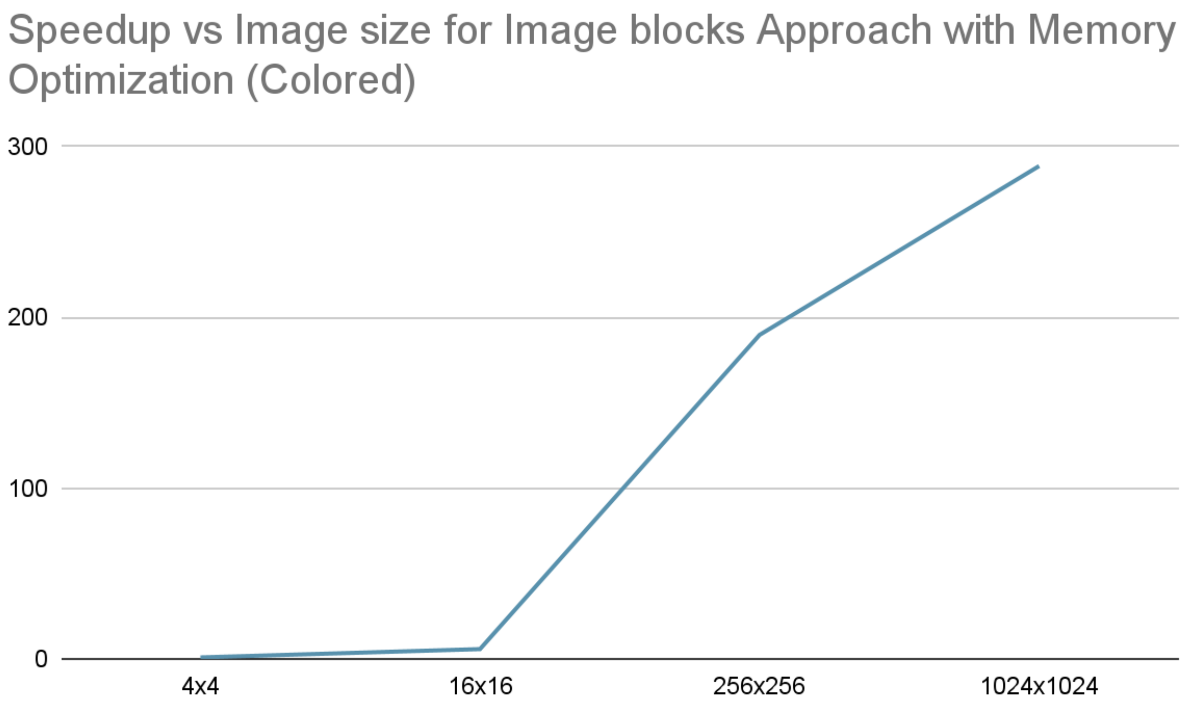


Table 3. Image blocks Approach with Memory Optimization (Colored)				
Image Size	4x4	16x16	256x256	1024x1024
Speedup	0.86243393084	5.73950005057	189.692084788	288.570069903



As we can see from the above results, smaller images such as 4x4 and 16x16 does not benefit as much from parallelization no matter if we distribute the pixels across the SM or fit all the pixels inside one warp-level operation similar to a SIMD approach. For those smaller images, the time it takes to transfer the image data from the host to the device and from the device back to the host outweighs the actual time it takes to perform the operation. Transferring the data takes from 50% to 80% total time for the entire pipeline. And for those smaller image, it would be better to parallelize on CPUs using SIMD instructions rather than GPUs, which is an overkill. And the downside of data transferring latency outweighs the benefit of parallelization.

Interestingly, for the smaller images. If we choose to spread the pixels across the SMs, it usually perform worse than fitting the entire image inside one SM and operate in the SIMT (Single Instruction, Multiple Thread) fashion. The CUDA SIMT execution model is an extension of the SIMD (Single Instruction, Multiple Data) mode. The one important difference between

SIMD and SIMT is that for SIMT, rather than a single thread issuing vector operation on vector data, SIMT issues 32 parallel threads that can use each of its own registers, which greatly benefit divergent workflows. If we were to distribute the smaller images across the SMs as in approach 3, the communication cost across SMs will once again outweigh the benefits of parallelization. Optimizing for best warp level performance also let us use the faster L1 cache as well as warp level primitives.

In addition, we see that our proposed list subdivision technique as well as caching the thread block inputs inside the local SM's shared memory gives us a 1.8x speedup compared to the original image block subdivision approach. Using the list subdivision approach, we can successfully avoid banking conflicts for accessing the same part of data across different SMs. The work done on each SM is independent to the data on other SMs, and caching the inputs gives us more benefits for large images as we successfully reduce the number of cache misses.

In conclusion, using CUDA and distribute work across multiple SMs is best for large images with independent data on each SMs. If the image sizes are small, the better approach would be using SIMD operations to avoid expensive transfer latency between host and device.

REFERENCES

Cuda C++ Programming Guide. NVIDIA Documentation Center. (n.d.). Retrieved May 5, 2022, from https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

pycuda 2021.1 documentation. (2022). PyCUDA Documentation. https://documen.tician.de/pycuda/

Bican, J., Janeba, D., Táborská, K., & Vesely, J. (2002). Image overlay using alpha-blending technique. *Nuclear Medicine Review*, *5*(1), 53-53.

Bialas, P., & Strzelecki, A. (2015, September). Benchmarking the cost of thread divergence in CUDA. In *International Conference on Parallel Processing and Applied Mathematics* (pp. 570-579). Springer, Cham.

LIST OF WORK

Ziming He:

- implemented serial and parallel version of alpha blending
- wrote the approach and results section of the report