GPU Acceleration of JPEG Encoding

Matthew French, Konrad Rauscher

EECS 598-003 Fall 2021

Table of Contents

Background and Motivation

The JPEG standard was established in 1986 and has since become one of the most widely used compressed image formats. As camera resolution has increased throughout time, the need for compression to store photos has also increased. JPEG provides an efficient method to do so. Its specification has grown and changed to leverage efficient algorithms, and its royalty-free baseline components have provided a solid foundation for many open source implementations [1]. JPEG has become the standard for images on the web, and more generally in photos everywhere.

As a base level, the process of JPEG compression contains several steps. Raw photos are typically not represented in the YCbCr colorspace that JPEG uses. As a result, the first step in the JPEG process is to convert the photo's colorspace to YCbCr which includes a luminance component (Y) and a chrominance component (Cb and Cr comprise a 2D chrominance space). We chose to compress .ppm raw images which use the RGB colorspace. The conversion we used is given below:

$$Y = 0 + (0.299 * R) + (0.587 * G) + (0.114 * B) - 128$$
$$C_B = 128 - (0.16874 * R) - (0.33126 * G) + (0.5 * B) - 128$$
$$C_R = 129 + (0.299 * R) - (0.587 * G) - (0.114 * B) - 128$$

Equation 1: RGB to YCbCr colorspace conversion

After converting the image to YCbCr, JPEG compression may perform downsampling on the chrominance portion of the colorspace. Humans can see more detail in the luminance component of the colorspace, so it is possible to reduce the information in the chrominance portion without changing how the image appears significantly. A more aggressive downsampling creates more compression in the resulting image. We chose to forego downsampling (a valid option in JPEG) to simplify our implementation.

The remainder of the compression process operates on 8x8 blocks of pixels. Our later parallel implementation launched a grid of 8x8x3 blocks (8x8 blocks for each colorspace channel). First, we perform a 2-dimensional discrete cosine transformation (DCT) of the 8x8 block. The formula for the DCT of the 8x8 block is given below:

$$G_{u,v} = \frac{1}{4}\alpha(u)\alpha(v) * \sum_{x=0}^{7}\sum_{y=0}^{7} g_{x,y} \cos\left[\frac{(2x+1)u\pi}{16}\right] \cos\left[\frac{(2y+1)v\pi}{16}\right]$$

Equation 2: DCT definition

x and y are the pixel row and column in the pre-transformed block; u and v are the row and column of the post-transformation block; $\alpha(z) = \frac{1}{\sqrt{2}}$ if $z = 0$ and 1 otherwise; $g_{x,y}$ is the pixel value in the pre-transformation block at index x, y; and $G_{x,y}$ is the DCT value in the

post-transformation block at index u, v. We perform a faster DCT by creating lookup tables for the cosine functions.

After transformation, we perform a quantization of the resulting block. The quantization is designed to get rid of the high-frequency components of the block while retaining the low-frequency and DC components of the block. It has been shown that human eyesight is much more sensitive to low-frequency information, so removing the high-frequency component offers an opportunity to significantly reduce size while retaining image quality. We use the following quantization tables in Figure 1.

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

17	18	24	47	99	99	99	99
18	21	26	66	99	99	99	99
24	26	56	99	99	99	99	99
47	66	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99

Table 1: Quantization tables for luminance (left) and chrominance (right)

After dividing the block by this quantization table, many of the values are close to zero. This is where the most significant reduction in information occurs. More specifically, most of the nonzero information is concentrated in the top-left corner of the block which corresponds to low-frequency information. By zigzagging diagonally through the block, we can linearize the block and keep a majority of the zero values at the end.

Finally, the linearized block is encoded. In order to minimally encode the nonzero values, a Huffman encoding is used. The encoding happens sequentially through the linearized block. The first component, the DC component, A new Huffman symbol is generated based on the run length of zeros that precede the next non-zero value, and the non-zero value itself. We use a lookup table for efficiency but they contain hundreds of values, so we don't include them here.

A .jpg file is generated with a header specifying the downsampling rate, image information, quantization and Huffman tables, and encoded image data. This implementation is based on information available in the official JPEG specification [2].

Code Information

An initial sequential C++ implementation was written using descriptions of the JPEG algorithm [2] [3]. Because we only planned to implement a portion of the algorithm on GPU, we separated the algorithm into two stages: the portion to implement on GPU and the portion which would only be implemented on CPU. We define the first stage (RGB-> YCbCr through zigzag) as the

transformation stage, and the remainder as the encoding stage. The encoding part of the code was derived from the open-source toojpeg library [4], and toojpeg was studied in the process of writing the sequential transform code.

A naive parallel implementation was written by rewriting the sequential transform code into CUDA kernels. The naive implementation is split into kernels for colorspace conversion, DCT, quantization, and zigzagging. Data is transferred to/from the host at the beginning and end of the transform process, and stays in GPU memory between kernel launches. Specifically, the functions `kernel_rgb_to_ycbcr`, `kernel_block_dct`, `kernel_quantize_dct_output`, and `kernel_zigzag` comprise the naive GPU implementation. They are called from the `parallel_compress_slice()` function.

Profiling Results

Initial profiling of the naive host code, shown in Figure 1, showed that the colorspace transformation, DCT, quantization, and zigzagging steps took the vast majority of the time. Only 1.3% was spent in the write_file function, which contained the encoding portion of the compression algorithm. Because of this, we decided to focus our efforts on the pre-encoding portions of the algorithm.

Overhead	Command	Shared Object	Symbol
76.50%	parallel.out	parallel.out	[.] Compressor::sequential_dct
9.41%	parallel.out	parallel.out	[.] Compressor::sequential_compress
3.97%	parallel.out	libm-2.17.so	[.] __roundf
3.61%	parallel.out	libwb.so	[.] wbPPM_import
1.80%	parallel.out	[unknown]	[k] 0xffffffff9018c4ef
1.32%	parallel.out	parallel.out	[.] Compressor::write_file

Figure 1: Time distribution of host code, profiled using perf.

The other profile we produced, shown in Figure 2, was of the naive GPU version. The information here was used to guide our optimizations. The most notable takeaway from this was that the vast majority of device time (approximately 95%) was spent on memory transfers between host and device. Because of this, we primarily focused our optimizations on reducing the amount of memory transferred between host and device, and on overlapping memory transfers to ensure that memory bandwidth resources are fully utilized.

Note that the large amount of time spent on cudaMalloc is primarily due to implicit initialization of the CUDA runtime API that occurs on the first call of some CUDA functions. This initialization overhead is inevitable when using CUDA and can be amortized for long-running applications.

Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:	46.95%	14.685ms	3	4.8951ms	4.5594ms	5.1283ms	[CUDA memcpy DtoH]
	46.78%	14.629ms	5	2.9259ms	1.1200us	14.624ms	[CUDA memcpy HtoD]
	2.50%	780.41us	3	260.14us	259.58us	261.05us	kernel_block_dct(float cons
	1.60%	498.88us	3	166.29us	166.02us	166.46us	kernel_quantize_dct_output(
	1.59%	498.11us	3	166.04us	165.86us	166.33us	kernel_zigzag(int const *,
	0.59%	183.87us	1	183.87us	183.87us	183.87us	kernel_rgb_to_ycbcr(float c
API calls:	89.49%	322.27ms	9	35.808ms	2.8920us	321.04ms	cudaMalloc
	8.48%	30.553ms	8	3.8192ms	6.2810us	14.794ms	cudaMemcpy
	0.72%	2.6104ms	10	261.04us	3.6710us	2.4986ms	cudaLaunchKernel
	0.70%	2.5253ms	4	631.32us	493.92us	770.25us	cudaDeviceSynchronize
	0.40%	1.4477ms	9	160.85us	3.1520us	433.49us	cudaFree
	0.10%	375.18us	1	375.18us	375.18us	375.18us	cuDeviceTotalMem
	0.07%	261.58us	101	2.5890us	120ns	111.26us	cuDeviceGetAttribute
	0.02%	55.558us	1	55.558us	55.558us	55.558us	cuDeviceGetName
	0.00%	9.9430us	1	9.9430us	9.9430us	9.9430us	cuDeviceGetPCIBusId
	0.00%	1.2930us	3	431ns	184ns	907ns	cuDeviceGetCount
	0.00%	701ns	2	350ns	159ns	542ns	cuDeviceGet
	0.00%	317ns	1	317ns	317ns	317ns	cuDeviceGetUuid

Figure 2: Time distribution of naive version of device code, profiled using the nvprof tool.

Optimizations

This section lists the optimizations performed and our rationale for how they could decrease the execution time of our application.

Fusing Kernels

When first generating our parallel implementation, we split the different portions of the JPEG compression algorithm into individual kernels. This allowed us to verify correctness at each step by substituting each sequential function with a parallel one. However, this resulted in the large runtime spent on memory transfer and a large memory usage in general since we needed intermediate data storage between each step. To reduce this overhead and improve runtime, we combined all parallel steps into a single kernel. We expected that this would provide several benefits. First, we used significantly less memory because we reduced allocated memory to just an input and output array instead of also including temporary storage. Second, we expected to reduce runtime overall by performing less memory transfer and reducing data scope. We used registers between steps where possible, and shared memory where impossible. Specifically, we used shared memory after the colorspace conversion, performed the DCT and quantization and held a temporary value in a register, and finally wrote back to global memory in zigzag order.

Constant Memory

The portion of JPEG compression we implemented on GPU requires several tables of constants: the DCT coefficients, the quantization coefficients, and the zigzag indices. The sizes and values of these are fixed, so they are prime candidates for constant memory. For this optimization, we created device-side constant arrays for these coefficients, and copied the host arrays into them using cudaMemcpyToSymbol, whereas previously they resided in global memory. When these variables are put in constant memory, they are cached on-chip in the constant cache, decreasing the latency and increasing the throughput of accesses. On modern

hardware, constant caching may not have very much benefit due to intelligent caching of global memory accesses.

Page-Locking Host Buffers

Page-locking host buffers is one of the primary ways to speed up memory copies between host and device. When a segment of virtual memory is page-locked, the OS ensures that all pages stay in physical memory and are not swapped to disk. When all pages in a memory segment are known to be in physical memory, the CUDA driver can copy the memory directly from physical RAM into device memory. If the pages have a chance of being swapped to disk, though, CUDA must copy to intermediate buffers before copying to device memory, meaning the memory transfer takes more time. Additionally, host-to-device and device-to-host memory transfers can only be overlapped with each other if the host buffers for both are page-locked. To summarize, page-locking host buffers decreases the time required to perform memory copies and allows full overlap of host-to-device and device-to-host memory copies.

Coalescing Global Memory Accesses

The combined kernel touches global memory twice: once when reading the input RGB data, and once when writing the zigzagged data. Neither of these accesses were coalesced initially. Reading the (band-interleaved by pixel) input was not coalesced initially because the RGB-indexing thread block dimension was the z dimension (z=0 reads R, z=1 reads G, z=2 reads B), so adjacent elements were read by threads separated by (blockDim.x * blockDim.y = 64). These accesses were made to be coalesced by treating the input block as flat and reading into shared memory based on flattened thread index. Once in shared memory, the access of adjacent elements by separate z-indices was no longer a problem.

The zigzag output was not coalesced either. Initially, global memory was directly written to in a zigzag pattern, which was not coalesced because of the irregularity of zigzagging. Coalescing was achieved by writing the zigzagged output into shared memory, then copying the shared memory to global memory, which is trivial to coalesce.

Ensuring that these memory accesses are coalesced improves global memory bandwidth by a factor of up to 32 - instead of using a full memory transaction for a single byte, a single transaction is used for 32 bytes. For reading the input, this factor is only 24 because blocks are non-contiguous and each row of a block is only 24 bytes. For a memory-bound kernel, this could translate to drastic speedups.

Combining Memory Allocations

A technique often used in high-performance computing is allocating the total amount of memory required for an application in a single buffer rather than making multiple calls to a memory allocation function. Because calls to memory allocation functions can often have high overhead, this reduces the time to set up the required memory for an application. It does not generally

have an impact on kernel execution time, though. We primarily implemented this optimization because we thought that cudaMalloc was taking a large amount of time. Once we realized that it only took so long because it was the first CUDA API call, we had already implemented this and saw no reason to remove it.

Casting Input to 8-bit

To read images, we utilized the PPM reader provided in the WB library used in the class. When WB reads images, it casts the data into 32-bit floating point numbers with a range from 0 to 1. Originally, we directly copied the 32-bit input data to the GPU and performed the appropriate scaling within the kernels. This optimization involved scaling the data from 0-1 to 0-255 and casting it to unsigned 8-bit integers *before* copying to the device. Because we initially found when profiling that a large amount of time was spent on host-to-device memory copy operations, we knew that this optimization would be important. This optimization reduced the usage of host-to-device memory bandwidth by a factor of 4. Because memory transfers between host and device are very slow, this was a significant factor in improving performance.

Overlapping Computation and Memory Movement

Any implementation which simply copies data to the device, runs kernels, and copies the data back is underutilizing GPU resources. Overlapping kernel execution, host-to-device memory copies, and device-to-host memory copies saves time because each of those can be performed simultaneously with no impact on the speed of the others.

To implement this overlapping, we used a pipeline-based approach. The image was split into chunks of 256 lines, and each chunk was passed to our compression algorithm separately. Host-to-device memory copies, device computation, and then device-to-host memory copies were performed separately for each chunk. Device scratch buffers were used to ensure that memory would not have to be re-allocated for each chunk. Chunks were distributed among 6 streams, and all streams were allowed to execute simultaneously. Instead of manually pipelining, we allowed the CUDA schedulers to determine the optimal arrangement of the memory transfers and execution.

Evaluation Methodology

The correctness of the implementation was simple to verify. We verified correctness on both small images and large images. For small images, we printed out the data after all GPU operations were complete and compared that to the CPU output at the same point. The data at that point is 16-bit quantized and mostly zeroes, so it isn't very difficult to ensure that all values are the same for small images (approximately 16 x 16). For larger images, we simply used visual inspection of the output and ensured that the JPEG produced by the GPU version was visually indistinguishable from that produced by the CPU version.

A note on comparisons: we did not enforce that the quantized values or the decoded images had the exact same values. This falls within the spirit of the JPEG specification; it does not specify a specific DCT algorithm to be used, to ensure that it can be programmed optimally on a wide variety of devices [2]. Thus some small numerical differences are allowed and expected between implementations. Sometimes the values we compared were off by small values, and we assumed that meant our GPU version was still accurate.

The following figures represent the previously-stated concepts visually. Figure 3 shows a visual representation of the differences between intermediate coefficients for a small test image. Figure 4 shows the differences in the final output for the same image. Figure 5 shows a side-by-side comparison for a much larger test image.

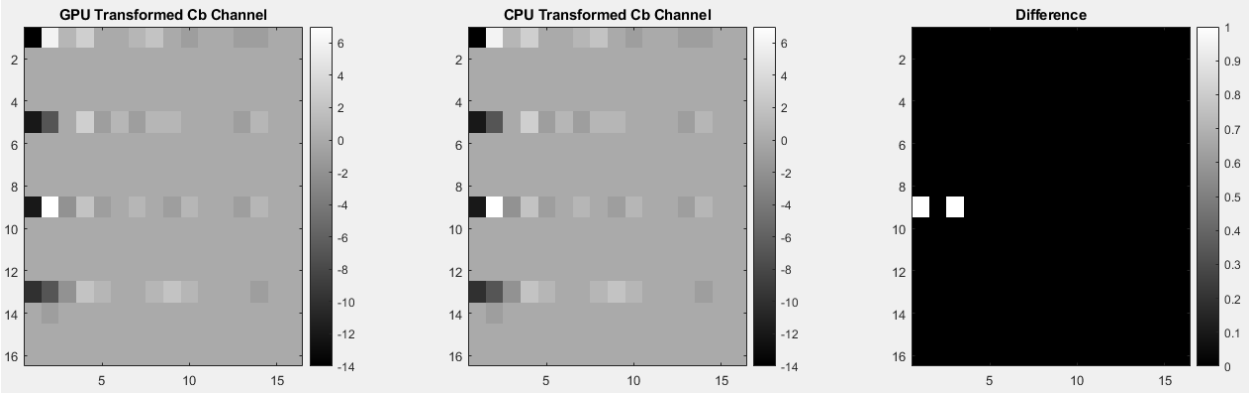

Figure 3: Difference in zigzag values between CPU and GPU for simple test image. Differences are only present in the Cb channel.

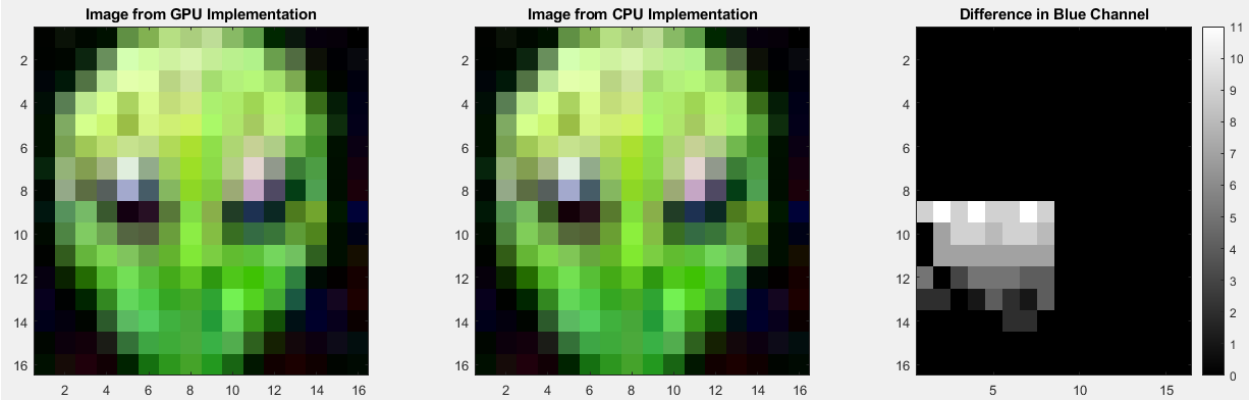

Figure 4: CPU and GPU versions for simple 16x16 test image, as well as differences in blue channel (the channel with the most differences). Note that the images are visually indistinguishable.

Figure 5: CPU version (left) and GPU version (right) of large test image [5]. They are indistinguishable.

Speed-up Analysis

All timing is performed on a 5488 x 5432 optical satellite image, provided by effigis and shown above in the speedup section [5]. The code is run exclusively on the Great Lakes cluster. Timing is performed using the C++ std::chrono library.

Timing Optimizations

The timing for evaluating the optimizations is performed only on the transformation portion of the code. The initialization, which took approximately 500-1000 ms and includes reading data and allocating memory, is completely disregarded. The encoding stage took approximately 325 ms, and we did not attempt to optimize it, but it is still important to know its execution time when comparing with state-of-the-art implementations.

We implemented these optimizations on top of each other and in order. For example, when reading the chart, the “page-locking” row represents timing including page-locking, constant memory, and combining the kernel. Timing information is shown in Table 2. The table gives the incremental speedup for each optimization as well as the total speedup. The table lists the runtime of the code with each set of optimizations in the first data column, and Figures 6 and 7 present this data visually.

Additional Optimization	Runtime (ms)	Incremental Speedup (x)	Total Speedup Over CPU (x)
Baseline CPU	7222.7	-	1
Naive GPU	239.9	30.10713	30.10713
Combined Kernel	240.6	0.997091	30.01953
Constant Memory	241.8	0.995037	29.87055
Page-Locking	62.1	3.89372	116.3076
Coalesce Global Accesses	61.9	1.003231	116.6834
Single Allocation	61.9	1	116.6834
8-bit input	33.9	1.825959	213.059
Pipelining	19.8	1.712121	364.7828

Table 2: Timing for various optimizations

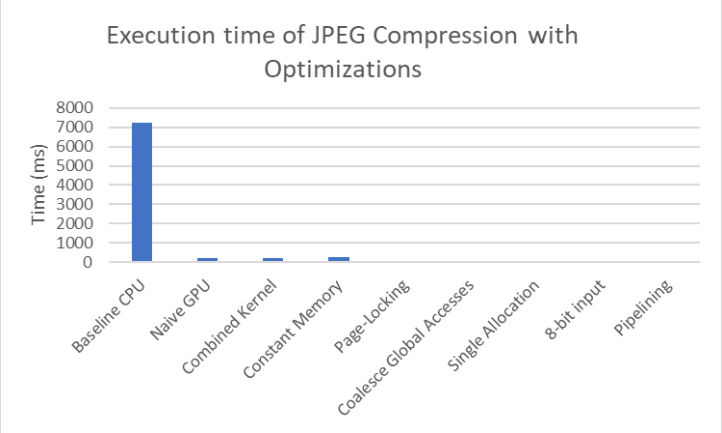

Figure 6: Execution time chart showing time taken for implementation with each optimization

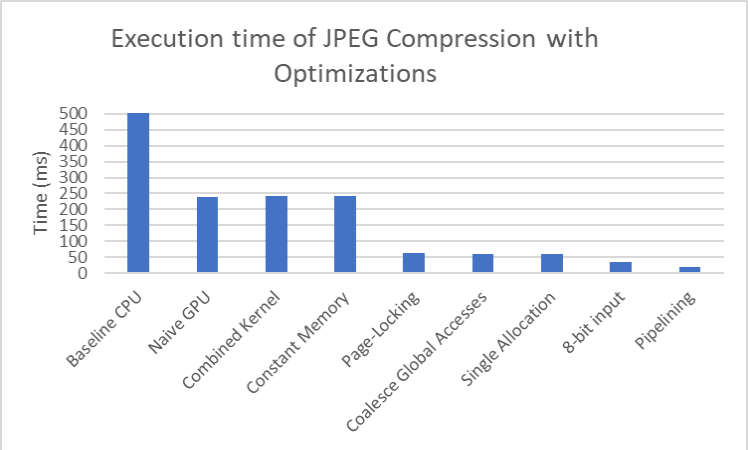

Figure 7: Execution time chart, zoomed in to see time variations between GPU implementations

Some optimizations performed as expected, while others did not. Notably, switching to the combined kernel had no effect on runtime. The primary change in the combined kernel is decreasing the number of global memory accesses, so the lack of speedup suggests that the transformation portion is compute-bound.

The optimized kernel performs approximately 192 single precision floating point operations, reads 1 byte from global memory, and writes 2 bytes to global memory. The Great Lakes V100 GPU is listed as having 14 single precision TFLOPS and 900 GB/s of memory bandwidth, so any kernel using more than 15.5 floating point operations per byte of memory access is compute-bound. Thus the optimized kernel is compute-bound, which explains why the GPU memory-related optimizations, constant memory and coalescing, do not provide very much speedup.

Successful optimizations included page-locking, 8-bit input, and pipelining. All three of these related to decreasing the contribution of host-device and device-host transfers to execution time. Page-locking increased the bandwidth of the memory transfers and avoided additional host-side page faults when the output array is copied to the host. Casting the input to 8-bit decreased the size of the host-to-device transfers, making them take less time with the same memory bandwidth. Lastly, using streams to overlap work allowed full utilization of both directions of memory bandwidth between host and device, because memory transfers could be running constantly, not only while the kernel was not running. These various methods of better utilizing memory transfer resources successfully decreased execution time.

Timing Against Reference Implementations

We are comparing our implementation to two reference implementations of JPEG compression: libjpeg (open-source CPU compression routines already built on Great Lakes)[6], and nvJPEG (the proprietary CUDA implementation of JPEG decoding by NVIDIA)[7]. The nvJPEG timing includes a cudaMemcpy to bring uint8_t RGB host data to the device and the nvJPEG API calls which write the encoded output into an in-memory host buffer. The libjpeg timing includes libjpeg

API calls with output written into a temporary file created using the C function tmpfile, which will likely not be written to disk and will act the same as an in-memory buffer. The state-of-the-art implementations are compared to our fully-optimized implementation. Table 3 shows the sum of times taken for the transformation and encoding stages for each implementation.

Implementation	Total Time (ms)
nvJPEG	28.5
libjpeg	207.3
Our fully-optimized GPU implementation	358.3

Table 3: Comparison of timing to state-of-the-art implementations.

The overall timing of our implementation is worse than either nvJPEG or libjpeg. However, recall that 325 ms of the timing is the encoding portion, which we did not optimize. The transformation time of 19.8 ms for our fully-optimized implementation is less time than the total for nvJPEG, so if we implemented the encoding portion on GPU it is possible that we could come close to the performance of nvJPEG.

Contributions

Matt researched and specified the baseline algorithm to be implemented. Given JPEG's wide support for a variety of formats and its highly configurable nature, there was significant effort spent to understand the algorithm in the first place. Matt wrote a sequential algorithm which could read in a .ppm image and output a .jpeg file by performing all of the encoding steps from scratch including colorspace conversion, DCT, quantization, zigzag rearrange, and file generation. We built upon these implementations when optimizing with parallel execution. Matt also implemented the fused kernel optimization and helped write this report.

Konrad implemented and verified the naive parallel version using Matt's sequential code as reference. He profiled the sequential and naive parallel implementations and used the profiling results to plan which optimizations to implement. Once Matt implemented the fused kernel, Konrad implemented the remainder of the optimizations. Konrad also performed the timing for the various optimizations and compared with the reference implementations, as well as writing part of this report.

Conclusion and Future Work

For this project, we implemented a portion of the classic JPEG compression algorithm in CUDA and performed a variety of optimizations to increase the performance. We achieved a 364x speedup over a naive CPU implementation of the transformation portion, and the performance of this part seems to be close to that of the state-of-the-art nvJPEG GPU library.

Many more optimizations could be performed on this code. The one with the highest benefit would be implementing the remainder of the algorithm on GPU. This would be very difficult because the encoding portion of the algorithm is not inherently parallel, so large amounts of serialization and data movement could be required to implement it. Another possible optimization is changing the DCT algorithm used. The current algorithm uses the definition of the 2D DCT, which runs in $O(N^4)$ time (where N is the size of a single dimension) as opposed to a separable implementation which runs in $O(N^3)$ time or a separable fast DCT which runs in $O(N^2 \log N)$ time. Because we determined that the kernel is computation-bound, this would almost definitely decrease kernel execution time. These are just some of nearly limitless optimizations possible.

References

[1] G. Hudson, A. Léger, B. Niss, I. Sebestyén, and J. Vaaben, “JPEG-1 standard 25 years: past, present, and future reasons for a success,” Journal of Electronic Imaging, vol. 27, no. 04, p. 1, 2018. doi: 10.1117/1.JEI.27.4.040901.

[2] "T.81 – DIGITAL COMPRESSION AND CODING OF CONTINUOUS-TONE STILL IMAGES – REQUIREMENTS AND GUIDELINES" (PDF). CCITT. September 1992.

[3] Information technology — Digital compression and coding of continuous-tone still images: Requirements and guidelines, ISO/IEC 10918-1:1994, 1994

[4] S. Brumme, “A JPEG encoder in a single C file ...,” toojpeg - a JPEG encoder in a single C file, 19-Jun-2019. [Online]. Available: https://create.stephan-brumme.com/toojpeg/. [Accessed: 07-Dec-2021].

[5] Effigis, “Free High-Resolution Satellite Images Samples,” Effigis, 07-Nov-2019. [Online]. Available: https://effigis.com/en/solutions/satellite-images/satellite-image-samples/. [Accessed: 07-Dec-2021].

[6] libjpeg. [Online]. Available: http://libjpeg.sourceforge.net/. [Accessed: 07-Dec-2021].

[7] “nvJPEG,” NVIDIA Developer, 10-Aug-2021. [Online]. Available: https://developer.nvidia.com/nvjpeg. [Accessed: 07-Dec-2021].

Appendix

Code can be found at https://github.com/konradrauscher/eecs598-project. The file compress.cu contains all relevant code, and the related scripts illustrate how we performed the timing.

On Great Lakes, code is located at `/scratch/eecs498f21_class_root/eecs498f21_class/krausch/proj/eecs598-project`. Code may need to be run with `module load gcc/4.8.5`. The script `run_tests {program} {test_number}` can be invoked with `sbatch` using the program `parallel.out` with an image in `test/{test_number}/input.ppm`. Sequential and parallel outputs will be produced in `test/{test_number}/attempt_{par/seq}.jpg`, and timing will be printed in the slurm output file. Test 1 is the smaller image shown in this paper, and test 3 is the larger image.