Accelerating the WebP Encoding/Decoding Pipeline

Kevin Geng and Emma Liu

Project Page: https://emmaloool.github.io/15418-Final-Project/

Summary

We implemented one of the stages of the WebP image encoding pipeline on NVIDIA GPUs, to take advantage of the parallelism offered by CUDA. To do so, we investigated the algorithms used by the reference implementation of WebP encoding and rewrote several versions of portions of the encoding pipeline to best take advantage of CUDA. We then compared the modified pipeline's performance on both GHC Cluster machines and the Pittsburgh Supercomputing Cluster Bridges machines, versus the sequential C implementation.

Background

Image compression involves the reduction of image data without degrading the quality of its visual perception. It is a canonical problem in the intersection of computational photography and high-performance computing as it relates to fast computation of large datasets because of the redundant nature of image representation (i.e., pixels) among colors and similarity among pixels. There are two main categories of compression algorithms for images: lossless compression, which is able to recover the original image data with no loss of quality, and lossy compression, which sacrifices the possibility to exactly reconstruct of the image for smaller file sizes.

Google's WebP image formats for ARGB images come in these flavors. We decided to focus on the lossless compression algorithm for the scope of our project. Specifically, our efforts will be to parallelize the encoding stage of the pipeline, which involves computing heuristics and testing multiple parameters to find the ones that will compress the best.

(We chose not to focus on the decoding pipeline, which would be relatively straightforward and therefore less interesting for parallelization.)

Representation

The input to the WebP encoding pipeline is an image, which can be thought of as a two-dimensional array of pixels visually, but interpreted as a one-dimensional stream of pixels. Each pixel is comprised of four component values: alpha, red, blue, and green. Components are represented by an 8-bit byte, defined as a `uint8_t`. Pixels are represented as a combination of these four components, and thus is represented by an unsigned 32-bit integer, defined as a `uint32_t`, where the alpha value occupies the upper 24-31 bits, descending in representation such that the lower 0-7 bits are occupied by the blue component. Thus, images are represented as an `uint32_t` array.

The transformations operates on 32 x 32 tiles of pixels, which is the granularity for determining different transform settings, and is also used when outputting data.

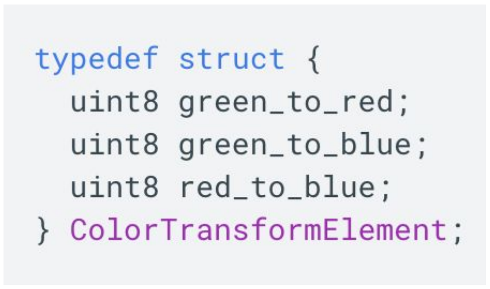
Operations

There are several stages the WebP encoding pipeline:

1) ***Reading and interpreting the input***. The input image is read as a stream of bytes in least-significant byte ordering, interpreted as a `uint32_t`, as mentioned above.

2) ***Undergoing transformations***. The lossless compression encoding pipeline makes use of the *entropy encoding scheme*, which compresses original image data to a smaller representation. Under this encoding scheme, images undergo transformations, or reversible manipulations of image data, which are aware of spatial/color correlations between the pixels in the image. These transformations attempt to reduce the Shannon entropy of the image data, in order to allow it to be better compressed by the encoding stage of the pipeline.

   There are four types of transformation used in the WebP pipeline:
   - ***Predictor Transform***: The predictor transform takes advantage of the correlation between neighboring pixels, where pixels are predicted based previously-considered pixels. The transformation data stored is the residual value between the actual pixel value and the predicted value.
   - ***Color Transform***: The color transform decorrelates the red, green, and blue values of each pixel by keeping the original green value, transforming the red value based on the green and blue values, and then similarly transforming the blue value based on the green and red values. These values are stored in a new structure, called a `ColorTransformElement`, as depicted in Figure 1.



```
typedef struct {
  uint8 green_to_red;
  uint8 green_to_blue;
  uint8 red_to_blue;
} ColorTransformElement;
```

Figure 1: `ColorTransformElement` struct

     In the color transform, a color transform delta value is found. To apply the transformation, the transform deltas between the fields are added to their corresponding original values.
   - ***Subtract Green***: The subtract green transform simply subtracts the green value from the blue and red values for each pixel, and does not return any transform data (it is directly applied to the image), unlike the other transformations.

- ***Color Indexing***: For images with an abundance of similar pixel values, the color indexing transform can be used to efficiently create a color index array (array of indices with similar pixel values) and substitute pixel values by the array of indices.

Each transform operates on the tile level, meaning that it divides the image into tiles (blocks) and performs the transformation on each block. Therefore, the block size can be viewed to be equivalent to the tile size. The final transform data for compression is chosen/generated based on the combination of transformation types that are the best to reduce the *Shannon entropy* for a particular image.

3) ***Encoding***. This stage of the pipeline involves encoding the raw image data losslessly using Huffman codes, and LZ77 prefix coding. Since these have both been attempted before on CUDA, we decided not to focus our efforts on this area.

Based on our analysis of each of these elements of the pipeline, we determined that the image transforms would provide the most interesting opportunity for parallelization, since the other parts of the pipeline are more focused on the specifics of the image format itself rather than the underlying computations.

There are significant dependencies between tiles. The most significant dependency is that the original code updates a histogram of pixel values after processing each tile. Since the sequential implementation processes tiles in row-major order, each tile has a dependency on the previous tile in that order. However, the histogram is only used as a heuristic to determine the parameters of the current tile that will produce the best compression; it is not crucial to correctness.

Additionally, another dependency between tiles is on the color transform parameters of the tile immediately above and to the left of the current tile (prev_y and prev_x respectively). These are used to bias the transform parameters to be able to take advantage of spatial locality later in the encoding process. Fortunately, this dependency only serves as a heuristic for better compression as well, and should not affect the correctness of the output.

Other than these dependencies, however, the computation is more or less data-parallel between different image tiles, which suggests that the tiles might be able to be computed in parallel if those dependencies are resolved. In addition, inside of each block, many computations per-pixel are also data-parallel and exhibit locality in a way that may be amenable to SIMD or CUDA execution. As mentioned later, the existing code in libwebp already takes advantage of this property on several architectures, including SSE4.1 on the machines we tested.

Approach

Resources

Technologies

Our project is based on the open-source `libwebp` encoding and decoding library for the WebP image format. The library and format were developed by the WebM Project, sponsored by Google. The project homepage is located at https://developers.google.com/speed/webp. This library is implemented in ANSI C.

We were specifically interested in building and using the `cwebp` encoder application. Since `libwebp` already contains several working C implementations, we used one of them as our baseline implementation. Note that the hardware abstraction layer chooses a C version of the implementation at runtime; specifically, the one that was chosen to suit our machines' hardware was the SSE version, which makes use of multithreaded CPU code with vectorized subroutines.

In light of this, we chose not to reimplement some parts of the pipeline, as the working versions are already highly performant. Our efforts were focused on investigating potential opportunities for parallelism throughout the pipeline and implementing CUDA versions for some of these portions.

Images

Up to the project checkpoint, we primarily used two images for testing: mitski.jpg, a medium-sized image, and starry_night_cropped.jpg, a significantly larger image. After the checkpoint, we collected a large set of test images, with variation in dimensions (size), color representation, and compression quality (mix of lossless and lossy). Note that we chose roughly equivalent-sized images for the small/medium set of test images. We did not choose significantly smaller images, as we recognized that our parallelization efforts would perform sufficiently poorly on smaller images due to the lack of parallelization opportunities associated with smaller data samples.

Because `cwebp` chooses a subset of the four transformations to perform on an image at runtime, and because we were isolating our efforts to optimize the Color Space Transformation pipeline, we performed an initial batch run of cwebp on the larger set of images to identify those that actually use `ColorSpaceTransform`, tossing out the rest of the images that didn't apply.

Figure 2 depicts the subset of images we used for further analysis.

IMAGE NAME	DIMENSIONS
anil.jpg	2,135 x 2,135
apple_holiday.png	2,880 x 1,512
apples.jpg	1,600 x 1,066
bon_appetit.png	2,864 x 1,460
carbon_emissions.png	1,240 x 1,754
corgi.jpeg	2,967 x 1,978
crow.jpg	1,250 x 1,000
google_notes.png	1,520 x 760
iphone_11.png	2,876 x 1,550
layout.png	1,520 x 760
mitski.png	2,876 x 1,572

IMAGE NAME	DIMENSIONS
brisbane.jpg	15,104 x 3,328
commencement.jpg	15,104 x 3,328
inauguration.jpg	7,168 x 3,584
machu.jpg	6,656 x 2,560
paris.jpg	8,704 x 1,536
pittsburgh.jpg	8,960 x 2,304
yosemite.jpg	10,752 x 4,096
starry_night_cropped.jpg	15,000 x 11,878

noise.png	2,550 x 1,650
ohqueue.png	2,880 x 1,568
sethg.jpg	654×971
sf.jpg	1,224 x 918
siamese.jpg	3,840 x 2,400
twinpeaks.jpg	3,976 x 2,652
va_districting.tif	2,240 x 1,010
zip.png	2,064 x 1,528

Figure 2: Images used for testing. Larger images are highlighted in light blue.

All "large" images with the exception of `starry_night_crop` were GigaPan photos, downloaded using the Python script at https://github.com/DeniR/Gigapan-Downloader-and-stitcher at the largest resolution possible with the script, and were converted to the jpg format (to avoid exceeding AFS quota). Credit goes to photographers Chris Powell, Carnegie Mellon, David Bergman, Jeff Cremer, Ronnie Miranda, James Albright, and Grant Meyers.

The starry_night_crop image is a high-quality image of Vincent van Gogh's *The Starry Night* painting, currently located in the Museum of Modern Art, and captured by Google Cultural Institute. The image is in the public domain, since the work dates from 1889.

Machines

We believe that GPUs provide the best platform for studying image compression. Existing code for encoding and decoding images is written for and optimized for CPU workloads. However, the compression of a single image is usually not significant enough to benefit from parallelization over multiple machines. Using GPU technology strikes the perfect balance between efficiency in that it can speed up image encoding / decoding by taking advantage of parallelism, and practicality in that it doesn't require the use of a compute cluster for fast and efficient operations.

Therefore, we will be targeting both the GHC Cluster and Pittsburgh Supercomputing Center (PSC) Bridges machines, in order to include some evaluation about the efficacy of their GPUs on our workloads. The specifications for the machines are depicted in Figure 3.

	GHC Cluster Machines	**PSC Bridges (RSM-GPU / GPU-small) Machines**
CPU	Intel Xeon E5-1660 v4	Intel Broadwell E5-2683 v4 (x2)
RAM	32 GB	128 GB
GPU	NVIDIA GeForce GTX 1080 NVIDIA Quadro K620 (unused)	NVIDIA Tesla P100 NVIDIA Tesla P100 (unused)

Figure 3: Machine Specifications for GHC Cluster/PSC Bridges

The interconnect bandwidth of the Tesla P100 has a theoretical maximum of 32 GB/s, whereas the interconnect bandwidth of the GeForce GTX 1080 has a theoretical maximum of 16 GB/s; both are limited by the speed of their respective PCIe bus implementations.

Mapping the Problem

Because the image tile size is 32 x 32 pixels, which is a size supported by a CUDA block, we decided to fix this size as the CUDA block size. This would allow us to easily synchronize all threads in the block; make it possible advantage of block-level memory specific to a tile; and possibly use one warp for each row of the tile, which could be useful for warp-level operations in the future. However, this does somewhat constrain our implementation, since 32x32 is a relatively large CUDA block size, and there is therefore less shared memory available per block than there might otherwise be.

For our computation, block-level memory is particularly significant for the encoding pipeline, as it makes use of histograms (arrays of size 256, since each byte can take on values from 0 to 256) which compute data about the frequency of each pixel value in the block. Synchronizing an entire block is also particularly important, since the threads in a block need to cooperate to determine the optimal transform parameters for the entire block. If we were to choose a smaller CUDA block size, we would have significant difficulty in adapting our algorithm to perform this sort of computation for each WebP image tile.

Iterations

Part 1: Implementing helper functions in CUDA

At the beginning of the project, after an initial scan of the code, we realized that the decoding pipeline has less interesting/relatively non-trivial opportunities for parallelization compared to the encoding scheme, so we pivoted to start from the encoding side to start.

The existing libwebp library has an abstraction layer which allows for specialized implementations of certain functions to be used, depending on the hardware. For instance, the C code calls `VP8LSubtractGreenFromBlueAndRed` in order to perform a linear pass over image data that transforms the color stored in each pixel. However, `VP8LSubtractGreenFromBlueAndRed` is actually a function pointer. The value of the pointer can be changed from the default C implementation, `SubtractGreenFromBlueAndRed_C`, to run more specialized implementations that take advantage of the machine's hardware. For instance, on the GHC machines, it is changed to point to `SubtractGreenFromBlueAndRed_SSE41`, which performs the same operation while taking advantage of SSE4.1 vectorized instructions.

Our initial thought was that we could take advantage of this existing abstraction layer. As such, we decided to implement several of those helper functions in CUDA, including writing a `SubtractGreenFromBlueAndRed_CUDA` function. We chose to do this because this could be done using the existing interface, and without significantly restructuring the code. This would also provide a good opportunity to test adding CUDA to the build system and make sure it is able to run properly.

We were able to complete and verify the correctness of implementations for the following helper functions, which are all part of the lossless encoding scheme. (The VP8L prefix refers to lossless VP8 compression.)

- VP8LSubtractGreenFromBlueAndRed
- VP8LTransformColor
- VP8LCollectColorBlueTransforms
- VP8LCollectColorRedTransforms
- VP8LBundleColorMap

We chose to implement these functions since they were relatively straightforward to write, and so they provided a good start for our project. We then chose to focus on timing two representative functions. As previously described, the `SubtractGreenFromBlueAndRed` function performs a transform across the entire image, so its runtime scales with the size of the image chosen. On the other hand, the `CollectColorRedTransforms` function operates on a single 32x32 tile, so its performance is invariant to the image size. The `TransformColor` function is similar, though it operates only on a 1x32 block.

	Original	**Plain C**	**CUDA**	**CUDA kernel**
`SubtractGreenFromBlueAndRed` *Size 15000 x 11878 (starry_night)*	70 ms	83 ms	278 ms	0.043 ms
`SubtractGreenFromBlueAndRed` *Size 1277 x 1632 (mitski)*	1.3 ms	1.1ms	121 ms	0.032 ms
`TransformColor`	0.4 μs	0.5 μs	205 μs	6.9 μs
`CollectColorRedTransforms`	2.0 μs	2.5 μs	230 μs	8.2 μs

Figure 4: Initial timings for C vs. CUDA implementations for lower-level transformations

An explanation for the column titles in Figure 4:

- "Original" refers to the function that would be run if the codebase were unchanged. Since the GHC clusters support hardware vectorization, the SSE2 or SSE4.1 implementations are used in this column.
- "Plain C" refers to the C function that is used when no vectorization hardware is available.
- "CUDA kernel" refers to the time taken to execute the CUDA kernel only, excluding the cost of the cudaMalloc and cudaMemcpy operations.

For `SubtractGreenFromBlueAndRed`, if we examine the CUDA kernel alone, it runs orders of magnitude faster than the other implementations for all images. However, the function also incurs communication overhead in copying memory to and from the GPU. As a result, the overall time is orders of magnitude *slower*. As we learned in Assignment 2 in 15-418, this is inevitable, since the kernel only performs one linear pass over the data, resulting in **low arithmetic intensity**. Furthermore, since this function is only called once per program execution, there is no opportunity to amortize the overhead of copying memory across multiple computations.

For TransformColor and CollectColorRedTransforms, though overhead is still an issue, it is worth noting that these functions are called on the same block multiple times with **different parameters**, in order to find the best transform parameters. Thus, if a higher-level function is moved into CUDA, the overhead of copying memory (and of launching a kernel) can be amortized over multiple function calls, increasing **arithmetic intensity**.

At this point, we invested some time in more closely investigating the structure of the codebase. In the lossless encoding pipeline, we identified the following major components:

- `AnalyzeImage`: Analyze the input image to determine the best encoding plan
    - `AnalyzeAndCreatePalette`
    - `AnalyzeEntropy`
- `EncodeStreamHook`: Perform the transforms, and write the encoded image
    - `ApplySubtractGreen`
    - `ApplyPredictFilter`
    - `ApplyCrossColorFilter`
    - `EncodeImageInternal`

Based on this investigation, we decided to focus our efforts on optimizing the `VP8LColorSpaceTransform` function. This is the primary subroutine of `ApplyCrossColorFilter`, which is one of four transforms that can be performed while encoding the image. This is also the function that calls `TransformColor` and `CollectColorRedTransforms` as helper functions. We chose this since we had already written CUDA kernels for those helper functions, and since we had already determined that they would be amenable to a speedup if parallelized at a higher level on the call stack.

As such, we decided to add `VP8LColorSpaceTransform` function to the abstraction layer in order to support a CUDA implementation, even though no other high-level functions appeared in the abstraction layer. This is likely because the abstraction layer was designed to support vectorized operations on the CPU, with no need to consider communication overhead. However, in order to mitigate the overhead of copying to and from GPU memory repeatedly, we needed to do so in a higher level function, which is why we made this change.

Part 2: Parallelizing parameter search for GetBestGreenToRed

The previous approach for `SubtractGreenFromBlueAndRed` mapped one CUDA thread to each pixel to exploit pixel-level parallelism. However, because of the aforementioned dependencies between tile computations, we initially tried to process each tile in serial, but we soon realized that at this level, the level of parallelism we sought wasn't achievable. If we mapped each tile to one CUDA block, then we would only be using one CUDA block at a time on the GPU.

However, as we learned in 15-418, taking full advantage of CUDA requires scheduling multiple blocks to run simultaneously on the GPU. If we serialized processing each tile, then we would only be using one of the GPU's SMM cores at a time.

So we decided to try to look for other avenues of parallelization. We noticed that the `GetBestGreenToRed` function attempts to use a basic iterative refinement / hill-climbing approach to determine the best value for the `green_to_red` transform parameter. Each possible parameter value was evaluated with the helper function `GetPredictionCostCrossColorRed`. (The `GetBestGreenRedToBlue` function does the same for the `green_to_blue` and `red_to_blue` parameters.)

The number of steps taken by this process depended on the quality desired, but we will assume the maximum of 6 for simplicity. It used a delta parameter at each iteration to determine which direction to move in, which started at 64 and went down to 1, for a final parameter value ranging from -127 to 127. Instead of using an iterative process, we hoped that we would be able to use CUDA to try all 255 parameter values simultaneously, and thereby achieve a speedup. To do this, we would launch 255 simultaneous CUDA blocks, each testing a 32x32 tile with a specific value of `green_to_red`.

Unfortunately, when we tested the performance of this approach, we measured a 0.15x speedup relative to the original implementation, i.e. a **more than 6x slowdown**. We suspect this is because the increased amount of work that was performed negated the improved parallelism. Though there are other aspects we could have optimized further, we did not believe that this would be a reasonable avenue to pursue further work, given the large slowdown.

Part 3: Creating one kernel for ColorSpaceTransform

As a result, we decided to revisit the dependencies between tiles that we described earlier. We knew at this point that the linear dependencies between the tiles (dependency on the previous histogram value in row-major order) were merely used as a heuristic. As such, we decided that it would be possible to ignore this dependency while still accumulating these global histogram values in an approximate fashion, e.g. by using atomic addition without any serialization between tiles. Even with the change, we were able to maintain a comparable compression ratio. The downside of this was that the output produced by our program would no longer exactly match the output of the baseline implementation. (Later on, we also realized it might be possible to precompute these histogram values efficiently with a 2D prefix sum / scan, but we did not have time to pursue this.)

The other dependency that we discussed was on the transform parameters of the tiles adjacent to the top and left of the current tile. This dependency is less strict than the other one, since each diagonal (from bottom-left to top-right) of image tiles can be processed in parallel, only depending on the results from the previous diagonal (to the top-left). As such, we might have been to continue without violating this dependency by performing (W + H) / 32 kernel launches, one for each diagonal. However, the first and last few kernel launches would only have a small number of blocks, limiting parallelism. As a result, we decided to simply ignore this dependency for the time being by inputting 0 as the previous tiles' transform values.

We realized at this point that we needed to make the computation as data-parallel as possible in order to obtain a speedup, which is why we decided to essentially ignore the two previously mentioned dependencies between tiles. In order to obtain the best speedup, we also decided to move the entire `VP8LColorSpaceTransform` function inside a CUDA kernel. The end result was that we simply performed one kernel launch for all tiles in an image, with each CUDA block mapping to one 32x32 tile.

While we did not yet have a speedup at this point, strictly speaking, the results were much more promising than they had previously been, with a slowdown of only around 40% compared to the baseline implementation. This was particularly notable since at this point, **the majority of the CUDA kernel was still serialized within each block**, with the exception of the `CollectColorRedTransforms` and `CollectColorBlueTransforms` functions.

Next, we decided to parallelize the `CombinedShannonEntropy` function This function estimates the Shannon entropy of image data in the current block, taking into account the accumulated global histogram (which we had set to 0). Since it looped through 256 array values sequentially, and performed a logarithm operation on each value, it was relatively expensive. This was the most challenging computation to parallelize, since it required summing values across the CUDA block.

We spent some time researching different strategies for block-level reduction in CUDA, and attempted to use some of the techniques we learned in 15-210 for reductions, but had difficulty in implementing them efficiently. (Warp-level reductions in CUDA can be very fast with specialized warp-level primitives such as `__shfl_down_sync()`, but we had difficulty in figuring out how to use them correctly.) Eventually, we decided to instead use primitives from the CUB library from NVIDIA Research to efficiently perform summations across the entire block. This allowed our code to finally achieve a speedup compared to the baseline implementation.

Finally, we proceeded to parallelize more parts of our CUDA kernel for `ColorSpaceTransform`. For instance, we parallelized the `CopyTileWithColorTransform` helper function by assigning each thread in the block to perform a computation for each pixel. Additionally, we similarly parallelized global histogram updates in this fashion, with each CUDA thread performing atomic additions in parallel instead of sequentially.

For testing purposes, however, we disabled the nondeterministic global histogram updates across tiles, since the compression parameters would change between runs. This made the compression ratio and performance difficult to measure, since the size of the output image would change each time! Removing this source of nondeterminism resulted in a **slightly worse compression ratio**, but no significant change in the program runtime.

Results

Experimental Setup

As previously mentioned (and as detailed by size in Figure []) , we chose a set of test images that could inform us of the quality of our parallelization attempts. After implementing `ColorSpaceTransform` and inserted timing code in `cwebp-cuda` to compare the execution times of the C and CUDA implementations, we wrote a script to run `cwebp` and `cwebp-cuda` to obtain timings. For reference, we ran `cwebp` (the full SSE-vectorized C implementation) ***once*** to obtain a reference compressed image (and corresponding size). Then we ran `cwebp-cuda` for a variable number of iterations. (For our purposes, we ran it 5 times for each image; the timings reported used the average of these timings.) Each time we ran `cwebp-cuda`, we also collected the compressed image it produced (and its size). We replicated this process for both sets of small/medium and large images, and then on both the GHC and PSC machines.

Metrics and Data

Throughout our project, we measured the performance of various portions of the encoding pipeline in wall-clock time (milliseconds). We added wrapper code to time the launch and execution of kernels and/or helper device routines in order to compare the performance of the original version of the functions, which were written primarily in C and used both multithreading and vectorized subroutines (SSE on the machines we used) to speed up its execution.

After we moved on to iteration 2 and then nearly immediately to iteration 3, we finally arrived at the main portion of our project: our implementation of `ColorSpaceTransform,` which launches a single CUDA kernel utilizing several device helper routines. We performed timing analysis on `ColorSpaceTransform` for this final iteration, measuring its total execution We also included all time spent copying data to and from the GPU device in the total execution time.

Figures 5 and 6 portrays the timing of the execution of the C and CUDA versions of `ColorSpaceTransform` (`ColorSpaceTransform_C` and `ColorSpaceTransform_CUDA`) on the GHC and PCS machines. We decided to partition the image set into two separate graphs for visual clarity, since the execution times of the larger images was much longer than the smaller image.

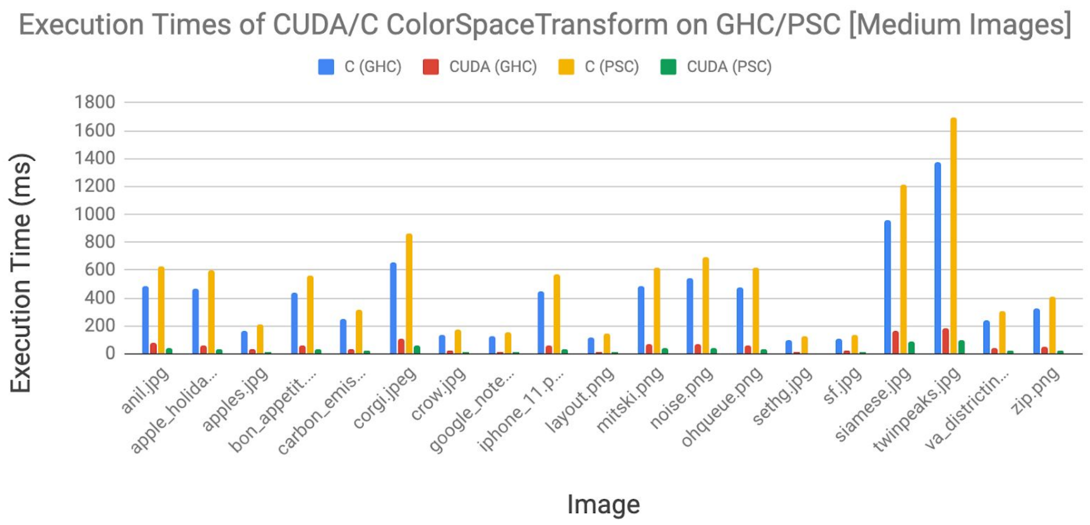


Figure 5: Execution times of CUDA/C `ColorSpaceTransform` on GHC/PSC (small/medium images)

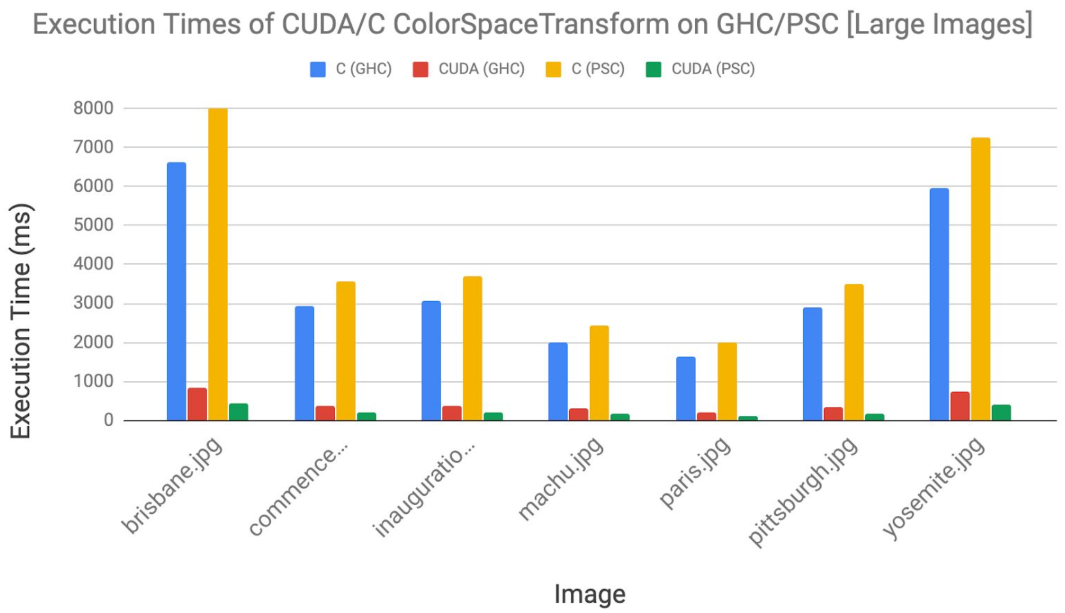


Figure 6: Execution times of CUDA/C `ColorSpaceTransform` on GHC/PSC (large images)

Based on these execution times for `ColorSpaceTransform`, Figures 7 and 8 depicts the table of speedup (C implementation time over CUDA implementation time) and a corresponding graph.

IMAGE NAME	PIXELS	SPEEDUP (pixels/ms)	
		GHC	PSC
anil.jpg	4558225	6.271	15.095
apple_holiday.png	4354560	8.237	17.297
apples.jpg	1705600	5.578	11.796
bon_appetit.png	4181440	7.289	15.38
carbon_emissions.png	2174960	6.926	16.453
corgi.jpeg	5868726	6.339	14.865
crow.jpg	1250000	6.478	13.524
google_notes.png	1155200	7.676	15.542
iphone_11.png	4457800	7.598	16.397
layout.png	1155200	6.851	15.782
mitski.png	4521072	6.76	16.158
noise.png	4207500	7.926	17.637
ohqueue.png	4515840	8.278	17.76
sethg.jpg	635034	10.27	20.877
sf.jpg	1123632	5.185	11.119
siamese.jpg	9216000	5.907	13.469
twinpeaks.jpg	10544352	7.593	16.639
va_districting.tif	2262400	5.652	14.004
zip.png	3153792	6.762	16.359

IMAGE NAME	PIXELS	SPEEDUP (pixels/ms)	
		GHC	PSC
brisbane.jpg	50266112	8.058	17.849
commencement.jpg	50266112	8.049	17.496
inauguration.jpg	25690112	8.036	17.34
machu.jpg	17039360	6.841	15.061
paris.jpg	13369344	7.534	16.312
pittsburgh.jpg	20021760	8.727	19.163
yosemite.jpg	44040192	8.024	17.717
starry_night_cropped.jpg	178170000	12.917	28.043

Figure 7: Speedup of `ColorSpaceTransform` on GHC vs. PSC machines

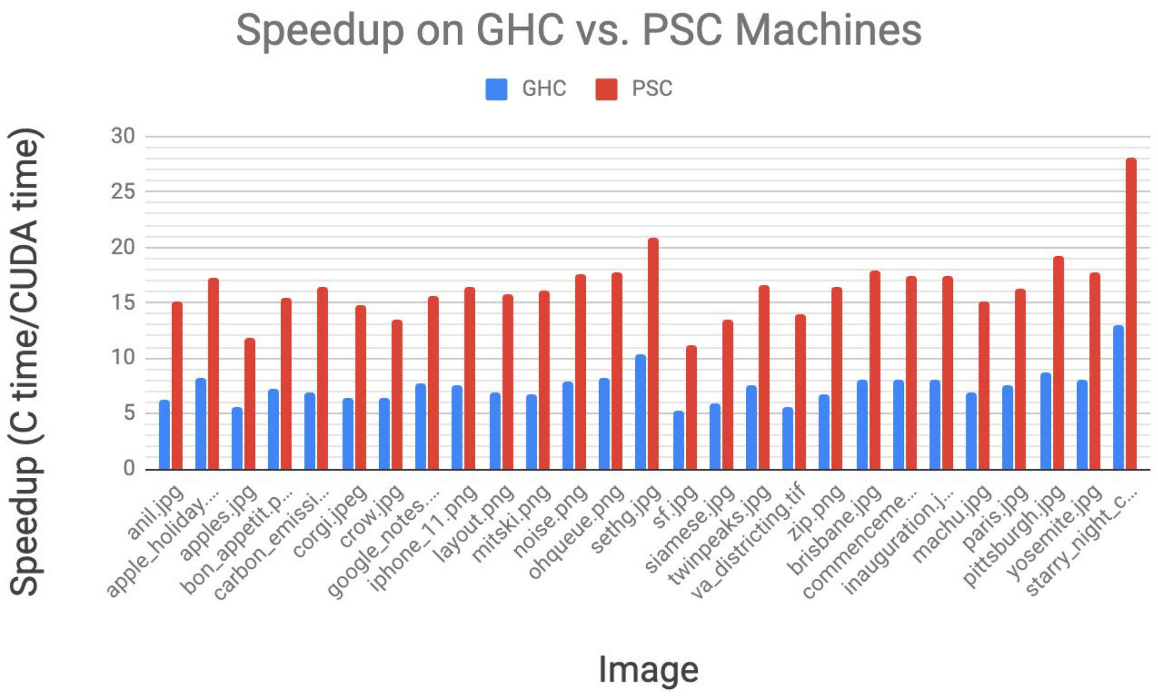


Figure 8: Speedup graph for `ColorSpaceTransform`, measured as C execution time over CUDA.

In Figures 7 and 8, we see that the speedup observed was on magnitude of 5-10x for the GHC machines, while speedup was on magnitude of 10-20x for the PSC machines (twice that of GHC). Among both medium-sized and larger images alike, these results were roughly consistent, with no apparent trend in regards to input size. The exception to this, of course, is starry_night_crop, which had over 10x speedup on the GHC machines originally and then nearly 30x speedup on the PSC machines.

Another metric we measured was the ***relative data compression ratio*** *(compression power)*, the ratio between the original uncompressed size of an image and its compressed size. A sizable portion of our test data consists of images that have already undergone some form of lossless/lossy compression (ex., JPG is a form of lossy compression), and we unfortunately didn't have access to the uncompressed size of original images that would be necessary to make an accurate raw comparison between the C and CUDA implementations. Nevertheless, we measured *relative* data compression between `cwebp-c` and `cwebp-cuda`, to compare the relative efficacy of our implementation versus the reference.

As depicted in Figure 9 below, we saw that the relative compression ratio between the sizes of the images produced by the CUDA and C implementations was nearly one-to-one. We found that the sizes of the CUDA-compressed images were slightly larger the C-compressed images, on average magnitude of approximately 1,000 bytes out of total image sizes of 1 million or more bytes. Since compression algorithms aim to reduce images as much as possible, this indicates that the CUDA implementation is comparable (barely worst) to the fully-C implementation of the encoding pipeline to produce lossless-compressed images.

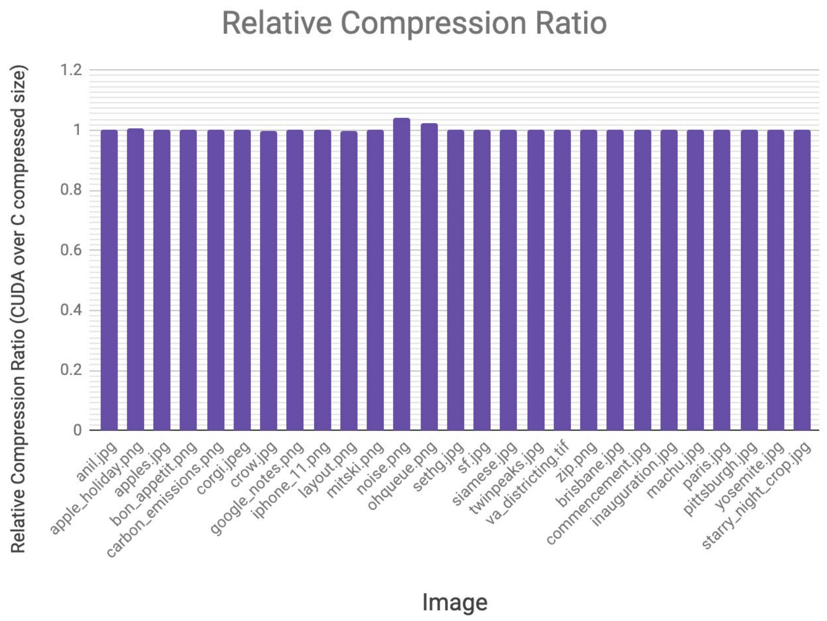


Figure 9: Relative Compression Ratio (CUDA compressed size over C compressed size)

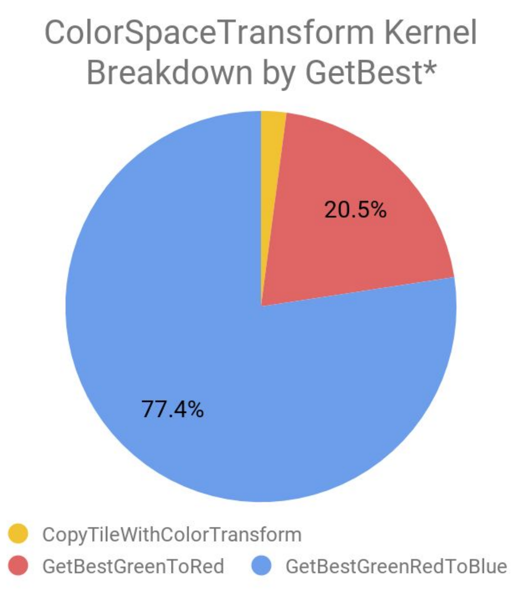
Further Analysis

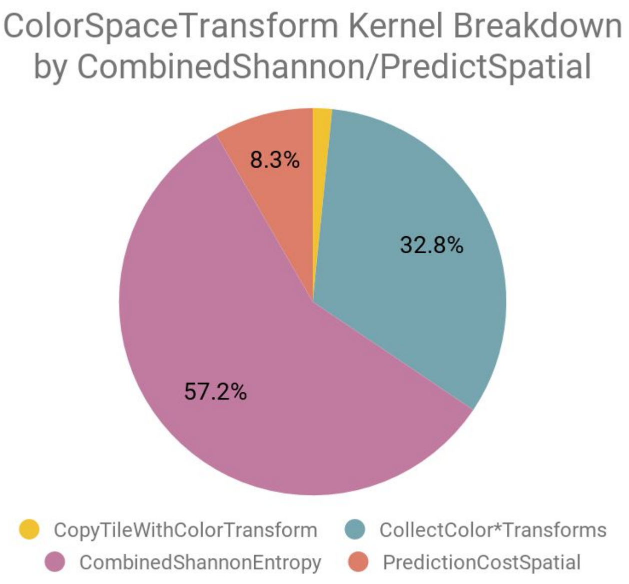
Breakdown of Execution Components

First of all, since our code is all inside one `ColorSpaceTransform` kernel launch, it is not easy for us to break down the timing data within the kernel. There isn't any straightforward way to do so from what we found. As such, we determined a way to estimate the runtime of different components: by subtraction. For instance, to measure the runtime of `GetBestGreenToRed`, we would subtract the runtime of the kernel with `GetBestGreenToRed` commented out from the original runtime. Since the runtimes cannot be measured directly, these measurements are **very approximate**.

For the original runtime, we averaged 6 runtime measurements to reduce accumulated error from repeated subtraction. For each of the other runtimes, we averaged 3 measurements.

One thing to note is that there are two axes upon which we can measure runtime. The `GetBestGreenToRed` and `GetBestGreenRedToBlue` functions determine the `green_to_red` and `green_to_blue` / `red_to_blue` parameters respectively. They each call respective `CollectColor*Transforms` functions (with * replaced by Red and Blue respectively), as well as the `CombinedShannonEntropy` and `PredictionCostSpatial`. This is depicted in Figure 10:





Function	**Time (ms)**
`CopyTileWithColorTransform`	8
`GetBestGreenToRed`	78
`GetBestGreenRedToBlue`	294

Function	**Time (ms)**
`CopyTileWithColorTransform`	8
`CollectColor*Transforms`	162
`CombinedShannonEntropy`	282
`PredictionCostSpatial`	41

Figure 10: Breakdown of kernel execution time, based on different helper functions, and their corresponding graphs

One of the most notable observations is that even after parallelizing the reduction operation with CUB, which resulted in a noticeable speedup, the `CombinedShannonEntropy` function continues to take a significant proportion of the runtime.

Overhead, Data Transfers, and Speedup Limitations

One timing issue we observed is that the first CUDA kernel launch performed by our program had a noticeable overhead of around 70 ms, which was perhaps needed to create an initial CUDA context. We know that JIT compilation was not the source of this overhead, since we made sure to pre-compile cubin files for each of the target GPU architectures in use, and since when we did force JIT compilation to be used with the CUDA_FORCE_PTX_JIT environment variable, we observed an additional overhead of around 150 ms. To mitigate this problem, we called the cub::PtxVersion() function on initialization, which launches an empty kernel. This eliminated the observed overhead.

As mentioned previously, the biggest overhead incurred in creating the CUDA ColorSpaceTransform was from copying the image data to and from the GPU. We found that by implementing a higher-level function in CUDA reduces the overhead of copying memory since more operations (i.e., more helper routines) is offloaded to the GPU.

One factor that might currently limit our speedup is the fact that the `PredictionCostSpatial` helper function is still run in serial. This is because parallelizing it would require using a scan operation (in particular, prefix product), which we have not yet figured out how to do. On the other hand, since the function runs a loop for only 16 iterations, the benefit of parallelization is likely to be relatively small.

On the other hand, our measurements show that the `CombinedShannonEntropy` function continues to take a relatively large proportion of execution time, even after parallelizing the reduction with CUB. We speculate the reason the main reason for this is to its use of `log2` (base-2 logarithm), which is the only major computation that it performs other than the reductions.

The performance of `log2` was important enough that the baseline implementation included in libwebp uses several tricks to speed up its execution, including a lookup table for small integer arguments. Although we simply used the `log2` function provided by CUDA for reasons of simplicity, the first thing we would do to speed up our code if we had more time would be to implement some of the same techniques used by the baseline implementation.

Beyond this, the main limitation for the speedup is likely the miscellaneous code that needs to be executed sequentially, and requires synchronization between threads. For example, we previously described how `GetBestGreenToRed` performs a hill-climbing search to determine the best value of the `green_to_red` parameter; to avoid the problem with performing extraneous work that we described previously, all 1024 threads in the block need to perform this search at the same time.

Problem Size Effect

“Different workloads” in regards to our compression project refer to the sizes of the images in our data set, which we believed to be important. As previously mentioned, we chose specifically not to test our CUDA implementation with smaller image sizes, as the C implementation `cwebp` canonically performs very poorly with smaller images (useless image compression). As previously observed (Figure 8, both medium and large-sized sets of images experienced comparable amounts of speedup. Depicted in Figure 11 below, we notice only a weak correlation between the speedup and the size of the data (in megapixels). We can perform further testing to determine the efficacy of our algorithms on even smaller-sized images, to determine the “sweet spot” of where our algorithm can start yielding notable speedup gains.

The exception to all of this is starry_night_crop is an interesting data point. Its size is ten times the size of the second biggest image, in terms of file size in bytes. Moving forward, if we were to test similar-sized images utilizing `ColorSpaceTransform`, we postulate that we would observe a similar boom in speedup. However, the limitation of this effort is the availability of our file storage quotas under AFS, so we could not possibly use as many images as large as starry_night_crop.

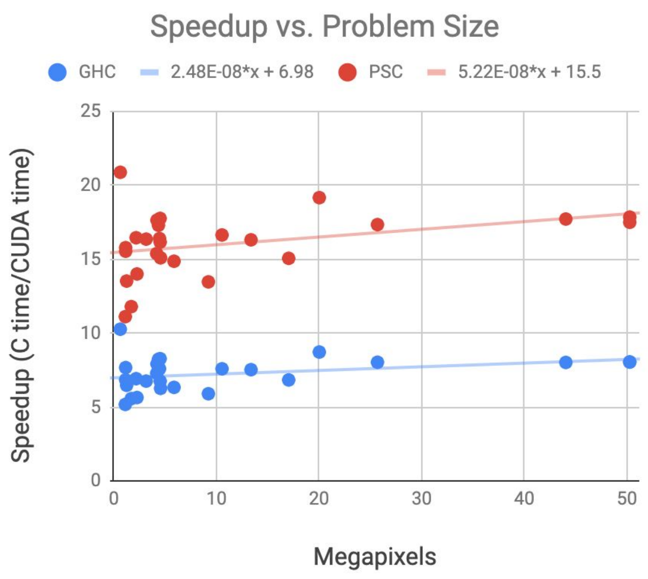


Figure 11: Relationship between speedup and problem size (megapixels)

<u>*Choice of Machine*</u>

Our choice of target was reasonable, given that the existing implementation written by `libwebp` already took advantage of parallelization techniques for CPUs, such as multithreading and vectorized instructions; in this way, the reference implementation is already impressively optimized for sequentially running code. This made it a challenge for us to determine viable areas for parallelization in order to obtain notable speedup on the GPU.

Nevertheless, by progressively offloading more and more of the transformation stage of the encoding onto the GPU, we were able to obtain impressive speedup on both machines.

Conclusion

In the scope of this project, we analyzed the performance of the encoding pipeline in the WebP lossless compression algorithm and then investigated several avenues (opportunities) for parallelization on the GPU using CUDA. After traversing the encoding pipeline's call stack, implementing both lower and higher-level transformation functions, and benchmarking all of the results we found several opportunities for parallelism in some of the helper routines they used to transform the original input data, which allowed us to successfully produce great speedup for one of the four major transformations.

References

"Compression Techniques | Webp | Google Developers". Google Developers, 2019, https://developers.google.com/speed/webp/docs/compression.

"RFC 6386 - VP8 Data Format And Decoding Guide". Datatracker.Ietf.Org, 2019, https://datatracker.ietf.org/doc/rfc6386/.

Images Used

For brevity, we've linked the source of our test image files in Figure 2 directly.

Work Breakdown

The total work distribution is approximately 50-50. The table describes each of the major efforts we initiated, but we also pair-programmed. Both of us equally contributed to final report and the poster.

Emma	Kevin
Study WebP pipeline (libwebp documentation, Google Developer guide)	
Set-up project page	Added CUDA to CMake build file
Determine feasibility of lower-level functions to parallelize	
Implemented the following functions/kernels: ● SubtractGreenFromRedAndBlue ● TransformColor ● CollectColorRedTransforms ● CollectColorBlueTransforms	Implemented BundleColorMap function/kernel
	Added wrapper for timing code in lossless_enc_cuda
	Studied compilation system and installed CUDA implementation
Implemented ColorSpaceTransform function/kernel	
Implemented ***parallel*** CopyTileWithColorTransform	Implemented ColorSpaceTransform wrapper
Wrote script to run cwebp in batch mode	Implemented CUB version of CombinedShannonEntropy
Performed timing analysis on GHC, Bridges machines	Brokedown timing of ColorSpaceTransform
Create performance tables and graphs	Added timing code in predictor_enc_cuda
Wrote final project write-up	
Create final project poster	