Accelerating WebP Lossless Encoding

Kevin Geng and Emma Liu, *School of Computer Science*
15-418: Parallel Computer Architecture and Programming
Carnegie Mellon University

Abstract

We implemented one of the stages of the WebP image encoding pipeline on NVIDIA GPUs, to take advantage of the parallelism offered by CUDA. We then compared the modified pipeline's performance on GHC Cluster machines and the Pittsburgh Supercomputing Cluster Bridges machines, versus the sequential C implementation.

Background

The WebP image format was developed by Google as an alternative to the existing JPEG and PNG formats. For our project, we decided to focus on the encoding pipeline of the lossless compression algorithm. This uses several heuristics, and tests multiple parameter values to determine which will result in the best compression.

There are four transforms that the WebP lossless algorithm can apply to image data to reduce their entropy to allow for better compression:

- Predictor Transform
- Color Transform
- Subtract Green
- Color Indexing

We targeted the Color Transform, which tries to decorrelates the RGB values by shifting the red and blue values based on the green values of each pixel.

Approach

We began with the existing libwebp library, which already took advantage of vectorized operations and some multithreading. Thus, we decided to accelerate the color transform with CUDA.

To amortize the overhead of copying memory, we used a single kernel for the `ColorSpaceTransform` function. WebP processes an image in 32x32 tiles, so we used this as our CUDA block size.

To implement the transform operations, we mapped pixels to their transformed values using CUDA threads, used reduction operations from the CUB libraries to compute entropy, and performed atomic addition to update global histogram counts.

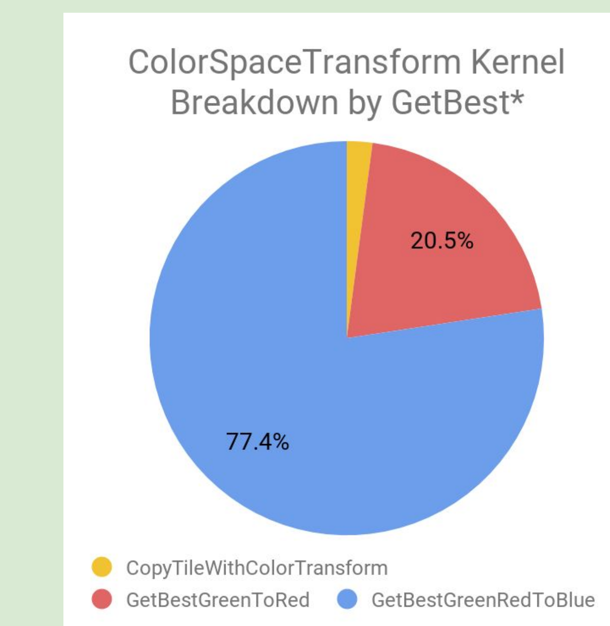
Results

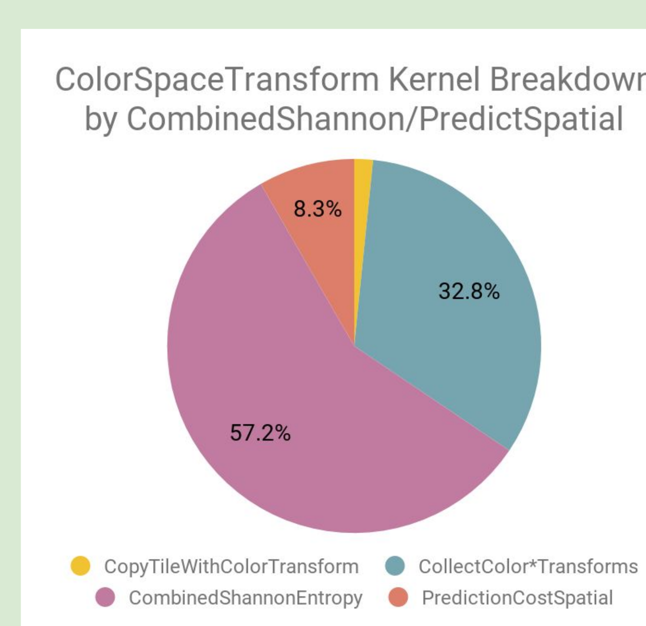
Experimental Setup

To test our program, we inserted timing code in `cwebp-cuda` to compare the execution times of the C and CUDA implementations, and wrote a script to run `cwebp` and `cwebp-cuda` to obtain timings. We also noted the compressed image sizes produced by both implementations.

Breakdown of Execution Timing

To determine which parts of the code merited further speedup, we estimated the amount of time each helper function in our kernel took by subtracting the time when each function was commented out.

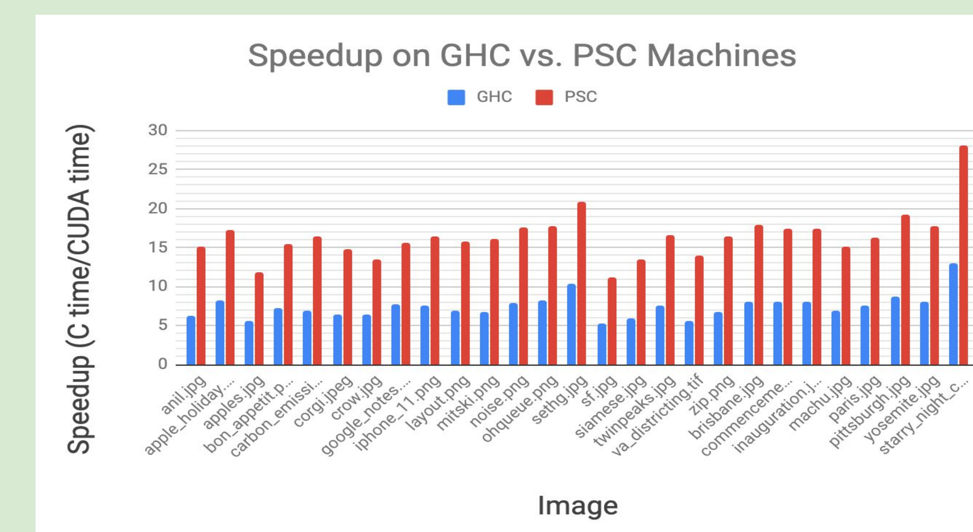




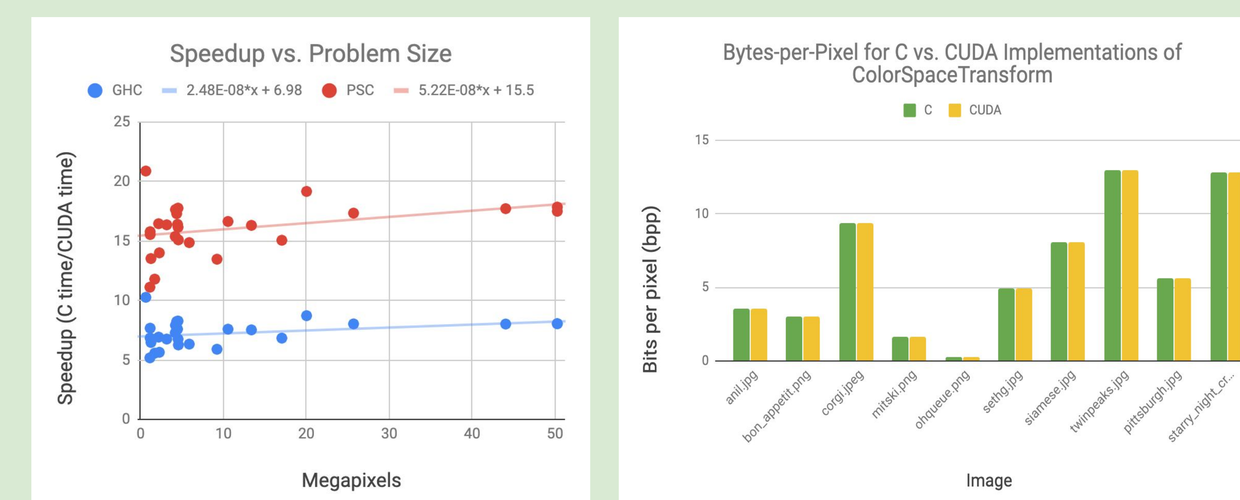
Analysis

Throughout our project, we measured the performance of various portions of the encoding pipeline in wall-clock time (milliseconds). We compared the performance of the original C implementation, which were used both multithreading and vectorized subroutines (SSE on the machines we used) to speed up its execution.

The following graph depicts the speedup experienced by the CUDA implementation of `ColorSpaceTransform` compared to the reference C implementation. 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). Notably, starry_night_crop experienced nearly 30x speedup on PSC!



We observed only a weak correlation between the speedup and the size of the data (the image size, in megapixels). This is seen below.



Another metric we measured was the compression ratio (compression power), the ratio between the original uncompressed size of an image and its compressed size. This can be alternatively be studied via a comparison between the ***bytes-per-pixel*** values, the number of bits needed to encode each pixel. We saw that the bpp of the sequential and CUDA implementations was nearly equivalent, which indicates that the compression ratio between the sizes of the images produced by the CUDA and C implementations was nearly one-to-one: the CUDA implementation is comparable with the fully-C implementation.

Discussion

Both medium and large sizes of images experienced comparable speedups. However, we notice only a weak correlation between the speedup and the size of the data. Further testing could determine the efficacy of our algorithms on smaller-sized images, to determine the image sizes for which we can yield a noticeable speedup.

It's worth exploring what factors limited the speedup we could obtain. Though the overhead of copying memory was a concern early in the project, that cost has been amortized due to the use of only one large kernel.

In our kernel, `CombinedShannonEntropy` still takes a large proportion of execution time, even after parallelizing the reduction step, perhaps due to `log2`. This could be improved further: for instance, libwebp uses a lookup table for small integer arguments.

Beyond this, the main limitation for the speedup is likely the miscellaneous code that needs to be executed sequentially, requiring synchronization between threads. For example, earlier in our project, we explored parallelizing the hill-climbing search performed by `GetBestGreenToRed` to determine the best value of the `green_to_red` parameter.

Conclusion

We analyzed the performance of the encoding pipeline in the WebP lossless compression algorithm and then implemented several parallelized image transformation routines to be done on NVIDIA GPUs in CUDA. By addressing previously serialized portions of the helper routines, offloading control flow to the GPU, and reducing the amount of data transfer overhead by targeting higher-level functions, our approach allowed us to gain great speedup (on order of 5x-25x speedup depending on machine) with nearly equivalent compression efficacy.

References

"Compression Techniques | Webp". 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/.