Checkpoint Report:
Accelerating the WebP Encoding/Decoding Pipeline

Kevin Geng
Emma Liu

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

Update

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.

We found that the WebP library already has an interface to support helper functions that have vectorized versions, where the appropriate vectorized version of a helper function to run is selected based on the available hardware. As such, we decided to move several of those functions to CUDA, since 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.

At this point, we've been able to complete and verify the correctness of implementations the following image transformations for the encoding scheme. (The VP8L prefix refers to lossless compression.)

- `VP8LSubtractGreenFromBlueAndRed`
- `VP8LTransformColor`
- `VP8LCollectColorBlueTransforms`
- `VP8LCollectColorRedTransforms`
- `VP8LBundleColorMap`

We chose to implement these functions since they were relatively straightforward write, and so they provided a good start for our project. Furthermore, we've begun to analyze the performance of these functions by inserting timing code to compare them to the existing implementations (including the straight implementation in C, and vectorized versions).

We chose to focus on timing two representative functions. The `SubtractGreenFromBlueAndRed` function performs a transform across the entire image in one pass, so its runtime scales with the size of the image chosen. On the other hand, the `CollectColorRedTransforms` function operates only on a 32x32 block of the image, so its

performance is more or less 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

An explanation for the column titles:

- "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.

We discuss the performance of two different types of functions below:

- For `SubtractGreenFromBlueAndRed`, we see that the CUDA kernel alone runs orders of magnitude faster than the other implementations for both small and large images. However, if we measure the overall time including the overhead of copying memory, we see that it is orders of magnitude *slower* overall.

  It is almost inevitable that the runtime of this function will be dominated by communication overhead, since the function is linear-time in the amount of data processed, but copying memory is linear-time in the amount of data as well. Since this function only appears to be called once per compression, there is no opportunity to amortize the overhead of copying memory across multiple computations. **As such, this is unlikely to be a good candidate for implementation with CUDA.**

- 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 compression parameters. Thus, the overhead of copying memory can be amortized over multiple function calls if a higher-level function is moved into CUDA, which would increase arithmetic intensity overall. **As such, these functions will likely benefit from a CUDA implementation**.

Though the runtime for the CUDA kernels alone for these two functions remains slower than the corresponding C implementations, this is likely also due to overhead in the kernel launch itself, which can similarly be amortized.

These conclusions are interesting to note in light of the fact that `SubtractGreenFromBlueAndRed` is actually just a specialization of `TransformColor` with fixed parameters. However, the fact that the latter has tunable parameters that can be changed for each block, whereas the former does not, which accounts for the difference in suitability for CUDA implementation.

Having most of these transforms (and their supporting routines) completed, we can proceed to investigate whether focusing on higher-level functions in the pipeline will yield better speedup for the lossless encoding/decoding techniques.

Progress on Goals and Deliverables

At this point of the project, we recognize that we will have to explore better parallelization opportunities higher up the stack in order to achieve speedup better than the sequential implementations, which was our original goal. Our initial forays into introducing CUDA code have allowed us to determine which parts of the code we need to focus on for parallelization.

For instance, we have assessed that operations which perform only a single pass over the image data are likely not viable for a speedup with CUDA, because of the overhead of copying image data to and from the GPU. (That is, unless we are able to move the *entire* pipeline into the GPU, which would be more work than we are willing to take on.) For example, this would likely apply to most operations needed for image decoding. As such, we will focus on portions of the image encoding pipeline that require multiple passes over the same image data.

We still need to front the investigation into the codebase in order to determine which stages of the pipeline we end up targeting for parallelization. Through the work we've already done, we have become more familiar with the structure of the codebase, which allows us to update our goals to be more specific.

The major stages of the pipeline that we've identified are:

- `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`

For the rest of the project, we'd like to focus our efforts on the `ApplyCrossColorFilter` function (and in particular **`VP8LColorSpaceTransform`** which performs the computation in that function, rather than encoding), since we have already implemented the `TransformColor` and `CollectColorRedTransforms` kernels which it uses as helper functions.

This function shifts the RGB values relative to each other in each block to obtain better compression, and tries several shift values in order to determine which works the best. This is interesting because while a CPU can easily take advantage of the locality involved in such a computation through CPU caches, it is less obvious how to do so in GPU code.

Our goal will be to obtain a speedup when running this function, as compared to the reference C implementation. To do this, we will need to more carefully read the code to study the operations that it performs, and determine at what level it is feasible to introduce CUDA kernels while actually obtaining a speedup.

If time allows, our stretch goals will be to pursue implementing one of the other parts of the pipeline, such as `AnalyzeEntropy`. This part of the pipeline performs more mathematical computations, namely computing logarithms to estimate entropy; the C implementation uses a lookup table to speed this up. It will be interesting to look at which strategies for this perform well on the GPU, and whether they differ from the strategies that work well on the CPU.

Anticipated Poster Session Artifacts

At the poster session, we plan to show graphs to depict the speedup for the final versions of a select few notable functions we have/will end up implementing in CUDA in the next couple of weeks, compared to their sequential versions. Since there exist several variants of the baseline (sequential) C implementation in the repository, we also think it would be valuable to perform timing on the pertinent functions on these variants as well, and provide a holistic comparison between these and our parallel implementation(s).

If possible (meaning if we can figure out how to reproduce intermediate representations of images as they undergo the encoding pipelines), we'd also like to display a demonstration of an image undergoing the encoding pipeline with each stage completed in representative time, side-by-side to the same image undergoing the sequential version of the pipeline. However, this would require a little more work on our end to determine how to capture these intermediate representations (if they exist in image forms).

Issues / Concerns

Initially, we pursued porting low-level operations for the encoding scheme because we thought they would provide more intuitive opportunities for parallelization, as there exists a file containing primitives for the SSE (streaming SIMD) implementation version of the baseline C code. We intuited that we could pick similar functions from ones implemented here to be viable opportunities for vector-wide instruction speedup.

But after observing the subpar (critically worse, honestly) speedup of the CUDA versions compared to the sequential ones, we recognized that we need to rethink our approach of migrating only lower-level processing functions to the GPU. However, it's apparent that much of the overhead occured results from the overwhelming communication costs to send intermediate buffers back from the GPU. This was not an issue for the original WebP developers, since SIMD instructions do not incur communication overhead, but it is an issue for us.

Our new approach will involve analyzing the performance of higher-level functions that invoke these transforms. Other functions that may be interesting are even higher up on the stack (we'll have to inspect the repository to determine how much higher they appear). But in its essence, we've essentially reached a largely investigate portion of our project. To reach our goals, this will require more dedicated time on our part to carefully determine viable higher-level functions via analysis and to crunch on implementing their CUDA versions.

Schedule (Revised)

Monday, November 4, 2019
Read description of WebP image encoding / decoding algorithm in detail.

Monday, November 11, 2019
Set up project space (migrate source code from libwebp to our repository).
Investigate initial viable opportunities to express parallelism in encoding pipeline.
Decide on which aspect(s) of the pipeline we should study for parallelization.

- Vp8l_enc.c: ApplySubtractGreen
- The rest of the transforms
- Compare different CUDA implementations of the same transforms?
- AnalyzeHistogram

Wednesday, Nov 13, 2019

- **Reconfigure build system to support CUDA**
- **Implement a basic direct CUDA translation of the pipeline stage that compiles**
- **Revisit & refine list of opportunities for parallelization in this stage**

Monday, November 18, 2019

- Submit the **checkpoint report** by midnight
- Compile performance measurements for different parallelization opportunities
- **Consolidate test cases (what are good images to compress?)**
- **Determine how to display analysis (speedup, computation time)**
- **Integrate timing code before/after functions kernels for 4 encoding transformations, compared to sequential implementation**

Checkpoint Deadline

Thursday, November 21, 2019

- Determining higher-level functions to parallel (both)
- Complete timing analysis on the sequential implementations, producing graphs/tables of speedup (Emma)
- Complete timing analysis on parallel implementations (Kevin)

Monday, November 25, 2019

- Analyze higher-level function `VP8LColorSpaceTransform` and how much control flow it is possible to move into the GPU / decide on function boundaries (Kevin)
- Begin writing kernel for `GetBestGreenToRed` (Emma)
- Test implementation (both)

Thursday, November 28, 2019

- Begin writing kernel for `GetBestGreenRedToBlue` (Kevin)
- Instrument and collect performance timing data to compare `GetBestGreenToRed` and `GetBestGreenRedToBlue` functions (Emma)

Monday, December 2, 2019

- Stretch goal: Analyze of feasibility of parallelizing `AnalyzeEntropy` (Kevin)
- Complete implementation of `VP8LColorSpaceTransform` (Emma)

Thursday, December 5, 2019

- Complete deliverable (both)
- Create performance tables (Kevin)
- Create performance graphs (Emma)
- Prepare for poster session and documentation (both)

Sunday, December 8, 2019

- Complete writing and submit the **final report** by midnight. (both)

Final Deadline