Parallel Style Transfer

Jason Israel (jaisrael), Michael Kamm (mkamm),

1 Summary

We implemented style transfer in CUDA on the GPU with the assistance of OpenCV. We compared it to a serial brute-force implementation and a OpenCV FLANN (Fast Library for Approximate Nearest Neighbor) implementation. We were able to achieve competitive runtimes with excellent accuracy using a brute-force best neighbor search on the GHC 5205 machines.

2 Background

Example-based style transfer is the process of transferring the "style" of one image to another. For example, if we have an image of a watercolor painting and an image of a real-world photograph, we'd like to transfer the "style" of the painting to the photograph, while preserving the photograph's structure. This can be imagined as an image analogy: $A : A' :: B : B'$. This notation can be understood as follows. Image A is the original source image. Image A' is the stylized source image. Image B is the original query image. Using these three images, the algorithm attempts to create image B', the stylized version of the original query image (figure 1).

This is a pixel-by-pixel algorithm that constructs B' in scan-line order. For each pixel q in B', the algorithm attempts to find the pixel p in A whose neighborhood feature is most similar to the neighborhood feature of q. To deal with textures of multiple scale in an image, B' is constructed at levels of the Gaussian Pyramids of A, A' and B. The feature of q, $F(q)$, is represented as the concatenation of its surrounding neighborhood in B_l, its surrounding neighborhood in B'_l, as well as its corresponding surrounding neighborhoods of B_{l-1} and B'_{l-1}. The new value of $B'_l(q)$ is $A'_l(p)$, where $F(p)$ of A is the most similar to $F(q)$. We construct B' from the coarsest level to the finest level.

2.1 Data Structures

The image-related data structures provided by OpenCV were extremely useful for writing simpler code. Our images are represented as 2-Dimensional Matrices of uchars. We calculated the We represent a set of features of an image as a 2D Matrix where each row represents a feature. Our matrix of reference features (features of A/A') is denoted $RMatrix$ and our matrix of query features (B/B') is denoted $QMatrix$. We defined the distance between two features as the sum of squared differences along each dimension.

The use of a Gaussian Pyramid helps with correctness while not significantly increasing the run-time. In this application, a Gaussian Pyramid is defined by a series of "levels" of coarseness. As the images get coarser, they get smaller as well. The pyramid is created by blurring and then sub-sampling an image on the previous level, and is computed for A, A', and B before the algorithm begins constructing B'.

The most important operation executed by this algorithm is bestMatch. For a given query point q, we must search through all $RMatrix$ to find the closest match, p. If there are Q query points (number of pixels in B) and P reference points, a brute force solution would require $O(PQ)$ comparisons. If we use an approximate nearest neighbor structure (such as a KDTree), we can reduce the number of comparisons to

$O(P*log(Q))$. However, it won't return the best theoretical answer.

The most significant challenge for the parallelization of this problem is that the intuitive algorithm is inherently sequential. To construct a pixel q in B', it is important to have knowledge of the surrounding neighborhood. Therefore, finding the best match with this algorithm in parallel would lead to more inaccurate results.

2.2 Parallelism Benefits

Parallelism can still be useful however. We attempt to circumvent the inherent dependency by using a different approach to the algorithm. This entails using non-scan-line brute-force. By not progressing in scan-line order, we essentially ignore dependencies at an initial cost of correctness. This is remedied through the use of several levels in the gaussian pyramid and a couple "correction passes". A correction pass is an additional execution of the algorithm given the results of the previous pass. Theoretically, even though there is incomplete neighborhood data, several passes allow for the result to "converge" on the best result.

3 Approach

We had three major solutions for this project. The first was a pure Brute-Force approach that exhaustively searched through every pixel in both images. This method took an extremely long time, so we moved on to a better approach. We next used OpenCV's FLANN library to implement Approximate Nearest Neighbors. This algorithm ran our largest images in around 15 seconds. However, the results were not accurate because of approximation and border cases around the image. Lastly, we implemented a brute-force best neighbor algorithm in CUDA to attempt to have a fast and accurate solution. We implemented this GPU-based algorithm with the help of C++, CUDA, and OpenCV. We targeted the GHC 5205 machines with the NVidia GTX 480 graphics card.

3.1 Brute Force

The first approach was pure brute-force. This entails going through each pixel in B and looking through each pixel in A to find the pairing with the smallest difference. Since this is an exhaustive search, it scaled up quickly with image size. While an image of size 120x160 took about 15-20 minutes, larger pictures took hours. Due to the rather long completion time, we decided not to take precise measurements.

3.2 FLANN

Our second approach was to speedup the bestMatch process of our serial algorithm. To do this, we passed in $RMatrix$ to an approximate nearest neighborhood search structure (from OpenCV's FLANN). Using this structure allows a query point to find an approximate best match in $O(log(P))$ time. However, there is a problem with edge cases (border pixels of the image have different neighborhood sizes, so they cannot search through our ANN structure). We first tried using a brute force search for these queries, but we noticed that the computation time of these edge points overshadowed the speedup gained from ANN. After realizing that the few brute-force pixels were the major bottleneck, we came up with a solution commonly used in image algorithms. We artificially extended the boundaries of the image by reflecting a few rows of border pixels over the edge. This allowed us to use the ANN search for each point, but it reduced the accuracy of the image. We speculate that the false neighborhoods had too much of an effect on the output image because the results from the coarse levels of the pyramid would influence to a large area of the fine image.

3.3 CUDA

In order to try to get a result that was both fast and accurate, our third attempt was to exploit the parallel structure of images using CUDA. We notice that there is a lot of potential for SPMD execution going on when trying to find the best match for all the query points of the target image. In order to make use of this potential, we decided to relax the dependency

constraints of the algorithm. From [2], we speculated that running bestMatch on each query point in parallel would give an approximate solution. If we would run the same process on the image a couple of times, we would eventually converge to an accurate solution.

3.3.1 Upsampling and Correction

Our algorithm is composed of two steps, upsampling and correction. To construct B'_l at each level, we would initialize B'_l by upsampling from B'_{l-1}. The initial value at $B'_l(r, c)$ is $B'_{l-1}(r/2, c/2)$. This compensates for some of the accuracy lost from the sequential algorithm by initializing with an estimated guess of what the neighborhood around $B'_l(r, c)$ would be.

Our second step improves upon the guess made by the upsample by running the bestMatch algorithm for each pixel in parallel with the approximated neighborhood. After a user defined number of iterations of this step, the algorithm moves on to the next level. We found that this correctness step was the true bottleneck of the algorithm.

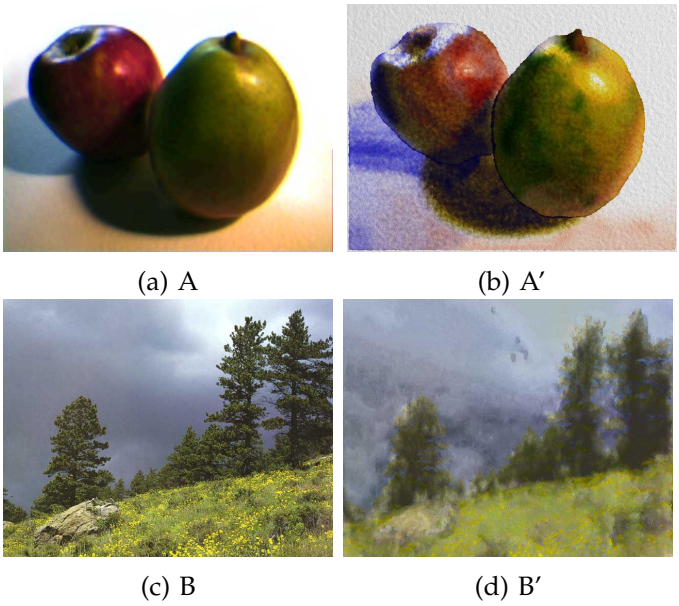
3.3.2 Optimization

In the interest of ensuring a "functioning" CUDA solution, we started off by doing a brute-force matching solution. We copied $QMatrix$ and $RMatrix$ from host to device, and utilized these values in global memory. Each thread represented a query point in $QMatrix$ and would iterate through each feature of $RMatrix$ and record the closest reference point. We quickly found out that this was not at all reasonable once memory access became a major bottleneck. This is because of the latency of global memory as well as the poor use of locality since the rows of $RMatrix$ were accessed at differing times (non-coalesced access) by each thread. We confirmed this suspicion by having each thread perform the same number of computations but on the same set of local data and noticed an embarrassingly large difference in run-time. The algorithms in [2] and [3] suggest an approach using texture Memory for the reference set. Texture memory is read-only, but is optimized for the locality present in a 2D access situation. Another important reason that texture memory is used that there are less penalties for non-coalesced accesses. This makes sense for the reference set, since we are only using it for distance calculation, and we have no need to modify A and A' after the pre-computation. However, texture memory cannot be utilized for images with more than 256^2 pixels. This is because CUDA did not allow us to have a texture with dimensions greater than $65536x65536$. Although using texture memory significantly decreases our run-time for coarser levels of the algorithm, we do not have enough space to utilize texture memory for finer levels. In order to deal with poor memory access from global memory, we decided to use shared memory within each block to speedup our bestMatch kernel [2].

In order to reduce the number of global accesses, we decided to organize the computation in a way such that each block only does computation on points in its own shared memory. We store a matrix bestDist, where element (r,c) would store the distance from $QMatrix(r)$ to $RMatrix(c)$. Therefore, the size of best dist is (Q, P) where Q is the number of query points and P is the number of reference points. We partitioned the calculation of bestDist such that each block handles a disjoint submatrix of bestDist. Each thread is indexed by a threadIdx.x and threadIdx.y and will load their feature into shared memory. After all threads have synchronized, each thread will update the position(Idx.x, Idx.y) of the submatrix. Because the number of query points and reference points can be large, we only feed in a certain number of rows from $QMatrix$ at a time to make sure that there is enough memory to allocate bestDist. We found that using texture memory for coarser levels and shared memory for finer levels gave us very efficient run-time and accurate results.

4 Results

In Figure 4, we can see a comparison of the run-times of our various approaches. Our first serial brute-force solution took so long to run we were unable to reasonably represent it here.



(a) A (b) A′

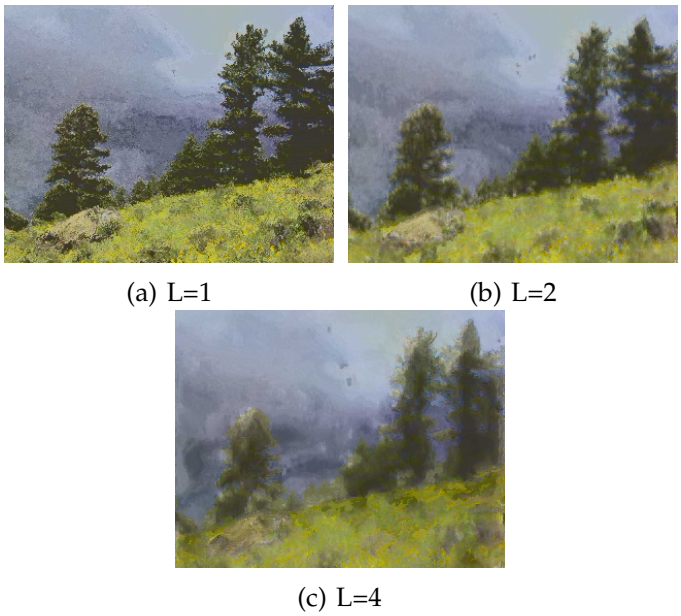
(c) B (d) B′

Fig. 1: An example of an image analogy. We learn the "style" from A to A′ and then transfer this style to B to create B′

For the Global approach, we can see that it is already an order of magnitude slower even on the smallest images. For the larger images, we were not even able to run the code to completion. The FLANN solution was by far the fastest on large images. However, the sacrifice of accuracy is apparent in Fig. 5. The shared, 2-correction pass solution, was our first result that completed in a reasonable amount of time with good accuracy. Our best approach used constant texture memory where possible. When the images were small enough that they could be stored in texture memory, the run-times were competitive with FLANN without sacrificing accuracy.

Our run-time is somewhat comparable to the results in [2]. Their results were faster than ours, but this is because they were able to more fully utilize the strengths of texture memory in two ways. First, they tailored the sizes of their images to use as much of the texture memory as possible. Second, they used a method called synthesis magnification to quickly complete the last level of the Gaussian Pyramid. Their run-time environment was also better than the environment provided by the Gates machines.

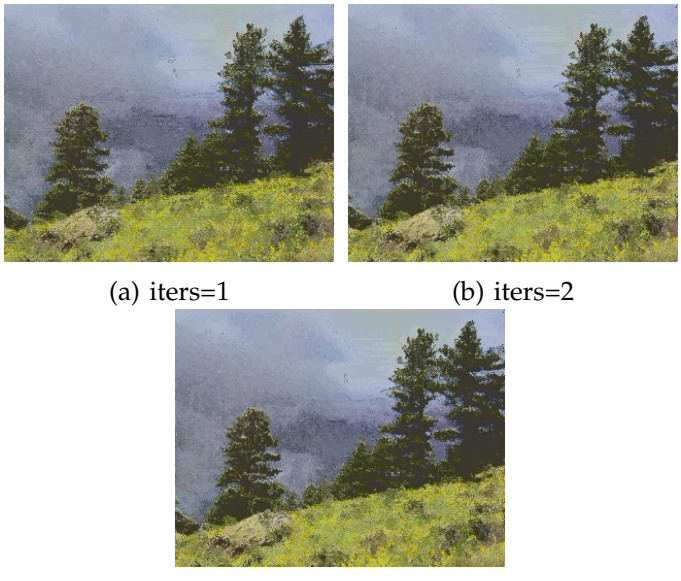
Equal work was performed by both project members.



(a) L=1 (b) L=2

(c) L=4

Fig. 2: Effects of multiple levels on B′. Using a multiscale representation allows for the capture of low frequency features of the watercolor filter. We used L=4 for our results.



(a) iters=1 (b) iters=2

(c) iters=4

Fig. 3: Effects of multiple correction passes on B′. Although there is a clear noise reduction between 1 and 2 correction passes, using more than two yielded negligable effects. We used iters=2 for our results.

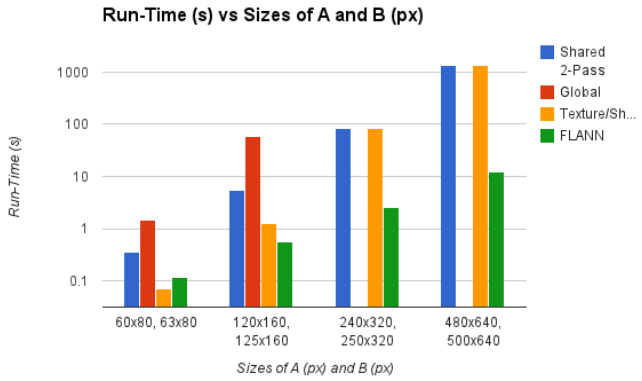


Fig. 4: Run-time Analysis of four of our different algorithms as a function of image size

(a) Pure FLANN result (b) CUDA result

Fig. 5: Comparison between Pure FLANN result and CUDA result. The CUDA result preserves the structure of the scene better than FLANN

REFERENCES

[1] A. Hertzmann, C. E. Jacobs, N. Oliver, B. Curless, and D. H. Salesin *Image Analogies*, SIGGRAPH Conference Proceedings, 2001.

[2] Y. Tang, X. Shi, T. Xiao, J. Fan *An improved image analogy method based on adaptive CUDA-accelerated neighborhood matching framework*, Published Online: Springer-Verlag, 2012.

[3] V. Garcia, E. Debreuve, M. Barlaud, *Fast k nearest neighbor search using GPU* IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops pp. 1-6, 2008

[4] S. Lefebvre, H. Hoppe, *Parallel controllable texture synthesis*, ACM Trans. Graph. 24, 777-786, 2005.