Saumya Jain and Srikar Pyda
Alexandros-Stavros Iliopoulos, Nikos Pitsianis, Xiaobai Sun
Computer Science 590: Parallel Programming
April 19, 2017

Final Project: Photomosaic Creation

Part 1: Introduction

Necessary background for an outsider to understand and appreciate your project

A photomosaic of a picture is an assemblage of many smaller photographs into an image which resembles the original "target." Photomosaic algorithms are interesting because they coherently aggregate a set of unrelated pictures into a recognizable image.

The idea of constructing a picture from a group of smaller objects can be dated back to the paintings of Giuseppe Arcimboldo in the 16th century. Ken Knowlton first introduced the idea of using a computer to decompose an image into building blocks in his software DominoPix, which constructed photographs from sets of dominoes. Rob Silvers invented photomosaics in 1955 as a graduate student at the MIT Media Lab while working on his master's thesis under Michael Hawley. Silver utilized graphics workstations and enormous databases of images resulting from advances in semiconductor and storage technology to construct photomosaics.

Photomosaics highlight a unique application of computer science to art: intricate pictures can be efficiently constructed from unrelated images. However, they also have potential application in digital copyrighting: proprietary images can be displayed as photomosaic previews for consumers before they purchase access. Furthermore, photomosaic-algorithms have an application in compression, database, and diagonalization technique in complexity theory because they prescribe methods of organizing information.

Related work (in algorithm, application, and tools) with references

In Generating Photomosaics: An Empirical Study, Nicholas Tran conducts an empirical study comparing two different photomosaic algorithms. He analyzes the algorithms by comparing their effectiveness and cost. He measured effectiveness through three metrics: similarity between the original and resulting image, granularity tile-size, and variety of tile-set. He analyzed cost through two metrics: the run-time of algorithm and library size of tiles. Although both algorithms aim at minimizing the minimizing the difference between the "target" tile and image from the database, the first algorithm models each comparison as an exact matching problem while the second algorithm models each comparison as an inexact matching problem. Our implementation is more like the first algorithm as opposed to the second algorithm because we calculate the average RGB values over the entire image. The second algorithm treats each row of pixels in a tile as a string and performs a string alignment algorithm. After comparing the algorithms, the author concluded that the first algorithm has a faster running time which is proportional to the product of the library size with the area of the original image. We found this paper interesting because it effectively dissected the photomosaic problem into its various logical steps and helped us elucidate the bottlenecks required for a parallelized implementation.

There have been a multitude of papers published on photomosaic applications: they focus on various parts of the photomosaic production algorithm: tile-matching, image segmentation, and tile-size. Most other papers also implement their algorithm in C to enable granular control of the manipulation of the image. Some authors focus on the detection of borders and objects in the production of photomosaics. Kim and Pellacini propose a method which poorly reproduces original photos from a composition of tiny arbitrarily sized images. Furthermore, Karner proposes an implementation of a segmentation process as an alternative to mono-sized tiles which aims at improving the sharpness of an image. Many papers focus on artistic manipulations in the creation of a photomosaic. Choi et al. uses masks on arbitrarily sized tiles to determine tile size, energy, and edge masks, preserving the edges and margins of the target image. Jing et al. implement tile blending techniques for artistic effect. Although some papers tangentially discuss the possibility of parallelizing various parts of the algorithm, there is sparse literature which focuses on an in-depth implementation of a parallelized photomosaic algorithm and the tradeoffs in parallelizing various steps in the algorithm.

Part 2: Problem Description

Application, algorithm, architecture – including problem size order and architecture-algorithm rationale

We implemented an application which produces a photomosaic of a user-selected image in MATLAB. MATLAB provides an ideal platform for image-processing because it provides developers with a multitude of built-in data-structures and methods for the indexing and processing of image. For example, it contains convenient methods to read in directories of images, iterate through multi-dimensional vectors, populate Map data-structures, and construct and edit matrices and images. Furthermore, it provides easy integration of CUDAKernel within MATLAB applications to enable developers to access the GPU and execute code in parallel. MATLAB allows the user to streamline transfer of data from CPU to GPU through built-in GPU array objects which can contain. The user must specify the path of a “target” image, the path of a directory containing the “database” of images utilized for the construction of the photomosaic, the type of images in the database, and the size of photomosaic tiles. Our application promotes user flexibility and modularity because users can specify tile-size: the program will output photomosaics with different granularity based on the segmentation of target image. The smaller the tile-size, the greater the granularity; the photomosaics better resembles the original “target” image when the tile-size is smaller. Our database of images was small: around 250 .png files and around 250 .jpg files. The application contains a couple of assumptions because the focus of our project was the parallelization of the photomosaic creation process. First, the input “target” image must be a square. Second, the tile-size must evenly divide the side-length of the image. If either of these requirements are not met, the program will crash from indexing issues. The program outputs the photomosaic as “mosaic.png” in the user’s working directory. The overall problem’s size is based on the size of the “target” image, the size of tile, and the number of photographs within the “database” of images. The problem size is of order $\theta(N*(T^2+D+R) + DC^2)$, where N is the number of tiles in the “target” image, R is the run-time of the built-in MATLAB function *imresize*, T is the tile-size, D is the size of the database, and C is the size of sample images within the database.

The following is a high-level view of our algorithm for generating photomosaics. First, map each image-name within the "database" of images as a *String* to its average RGB value as a 3-tuple. We chose to process the "database" of images into a Map of image-names to average RGB values because the Map Object enables flexible indexing within its individual elements (both keys and values). We chose to process the directory into a map prior to iteration to avoid having to iterate through the directory of images and calculate the average RGB values for every tile within the image. Then, iterate through all the tiles within the "target" image as specified by the user-specified tile-size. For each tile, find the nearest image within the database in terms of its Euclidean distance. Then, resize the selected "nearest" image to the inputted tile-size and place it in the relevant tile within a cell matrix. We utilized a cell-array for the initial aggregation of images within the photomosaic matrix because it is a data type within MATLAB with indexed containers called cells, where each cell can contain any type of data. As a result, it can store the double arrays of individual images as cells within the array. Furthermore, the built-in MATLAB function *cell2mat* enables an easy conversion from cell arrays to matrix-arrays if all the cells contain the same data type and the contents of the array must support concatenation into an N-dimensional rectangle. After iterating through all the tiles of the "target" image, the cell-matrix is finished. Finally, convert the cell-matrix into an ordinary array to write the photomosaic as "mosaic.png" in the user working directory.

The focus of our project was the parallelized implementation of the photomosaic algorithm utilizing CUDA kernel integration within our application. Initially, we considered parallelizing three part of our algorithm: mapping the database, finding the average RGB value for the tile, and finding the nearest image from the "database" to the tile. However, we only implemented the latter two steps in parallel. We decided to implement the mapping logic sequentially because the GPU does not have access to CPU memory. As a result, if we had parallelized the mapping logic, we would have had to transfer all the images in the "database" to the GPU utilizing a GPU-object such as a GPUarray. We decided that the transfer of images to the GPU would be more time-costly than the time saved through parallelization. The problem size of the parallelized algorithm is based on the tile-size and size of database. The problem size is of order $\theta(T^2+D)$, where T is the size of the tile and D is the size of the database. We hope that the gap in performance between GPU and sequential code will grow with the size of the input.

We parallelized the segmentation of the "target" image and selection of nearest image from "database" for each tile. We created an executable CUDAKernel object using compiled PTX code which contains the GPU executable code. Before calling *feval* on the CUDAKernel object, we needed to pass in all the required parameters as GPU-arrays: an integer array containing the indices of nearest images for all of the tiles in the outputted photomosaic matrix, a double array of the image, a double array of the average Red values for the database images, a double array of the average Green values for the database images, a double array of the average Blue values for the database images, the number of samples within the "database" of images, the number of tiles per image side, and the number of threads per block. We were careful about passing in the parameters with pixel values as double GPU-arrays to ensure consistency in data-structure while doing computations. In C++, the image array is a linear column-major indexed array while the nearest tiles array is a row-major indexed linear array. We were required to

pass in the average RGB values of all the images within the database as three separate GPU-arrays, because unlike MATLAB, C++ does easily allow for an easy manipulation of multi-dimensional vectors. After the CUDAKernel finishes evaluating, it returns the nearest tiles GPU-array with the indices of the nearest images for each tile index within the photomosaic matrix. After the CUDAKernel returns, the program constructs a cell matrix with the relevant resized image from the map of the "database" of photographs from the indices in the returned array

Main program specification (input, output, properties/assumptions)

The main program takes four inputs: the directory path to the "target" image, the directory path to the tile database, the image-type of tile, and the size of tile. The output is an image saved as "mosaic.png" in the user's working directory.

Our program has a couple of assumptions. First, the input image must currently be a square. If the image isn't a square, the program will fail from indexing issues because the tiles in which the image is portioned into are currently defined as squares. Second, the tile size must evenly divide the side length of the "target" image. Otherwise, the program will crash from indexing issues. Third, the user must have a directory containing all the images to be considered in the "database" of photographs for comparison to the "target" tile. Fourth, all images within the directory must be encapsulated by the image-type String specified by the user.

Furthermore, we chose to allow repetition of tile images in the creation of the mosaic. This results in the possibility of many sample images being repeated in the photomosaic, especially if there are large sections that are colored similarly in the "target" image. Our database of tile images is particularly small (~250 .png and ~250 .jpg images) and particularly lends itself to repetition. Although we could have eliminated the repetition in our implementation of the program to remedy the solution, it depends on sequential execution. A parallelized implementation of the search process could not accommodate the deletion of images from the database because the threads simultaneously iterate through all the indices in the red, blue, and green GPU-arrays, causing a scheduling error. A thread may attempt to access an index within these arrays which has been selected and deleted by another thread in parallel, which would result in indexing errors and segmentation faults. As a result, we could not implement this logic in the parallel implementation of the program.

Part 3: Expected Achievements

Desired performance order (essentially your bar for success)

We aim at parallelizing the calculation of average RGB values for each tile in the inputted "target" image and the search of "nearest" sample image from the database because we isolated these steps as the major bottlenecks in the sequential implementation of the program. We hope that our parallelized implementation of the algorithm will offer significant performance improvements for small tile-sizes. We predict that the parallelized and sequential implementations both offer similar performance for large-tile sizes.

In the sequential implementation of the algorithm, the run-time for these steps is $\theta(N*(T^2+D))$, where N is the number of tiles in the "target" image, T is the tile-size, and D is the size of the database. Assuming infinite resources, we believe that the run-time of these steps can be reduced to $\theta(T^2+D)$. If there are finite resources, we believe that

the runtime can be reduced to $\theta((N/F)*(T^2+D))$, where N is the number of tiles in the "target" image, F is the number of finite resources available, T is the tile-size, and D is the size of the database. Either way, parallelizing these steps ought to significantly reduce the run-time of the segmentation of the image into tiles by a factor of N/F.

The overall run-time of the sequential implementation of the algorithm is $\theta(N*(T^2+D+R) + DC^2)$, where N is the number of tiles in the "target" image, R is the run-time of the built-in MATLAB function *imresize*, T is the tile-size, D is the size of the database, and C is the size of sample images within the database. Assuming infinite resources, we believe that the overall run-time can be reduced to $\theta(T^2+D+ NR + DC^2)$. If there are finite resources, we believe that the overall run-time can be reduced to $\theta((N/F)*T^2+D)+ (N/F)R + DC^2))$, where N is the number of tiles in the "target" image, F is the number of finite resources available, R is the run-time of the built-in MATLAB function *imresize*, T is the tile-size, D is the size of the database, and C is the size of sample images within the database.

However, we do not expect the performance improvements to occur exactly according to the above calculations. First, parallelizing the algorithm requires transferring memory from CPU to GPU. It will take longer to initialize the CUDAKernel and all the required parameters to evaluate the kernel. Furthermore, gathering the updated array after the GPU has finished evaluating requires computational complexity because it transfers memory from the GPU to the CPU. Overall, we predict that the transfer of memory from CPU to GPU and visa-versa will be a major hindrance to the performance benefits we predicted above. As a result, the improvement in run-time will likely not be observed for smaller inputs because there is minimal work being parallelized. GPU cores are not designed to handle large tasks, so dividing a few large tasks on the GPU may be slower than running them on the CPU because CPU cores are designed to handle large tasks. However, for large input sizes we expect the GPU code to run significantly faster than sequential code with the gap in performance growing with the size of the input. We hope that the gap in performance between the sequential and GPU code will be a reduction of order $\theta(N/F)$, where N is the number of tiles and F is the number of limited resources which are available. Ideally, the run-time will be reduced by order $\theta(N)$ if there are infinite resources available.

Relation to target application

We believe that the parallelization of our photomosaic algorithm will significantly speed up the creation of the photomosaic in our application. Parallelization offers the greatest benefit for the processing of large images; when the inputted "target" image is larger, there is more work that can be parallelized. Although parallelization is not likely to offer significant improvements over a sequential implementation for large tile sizes, we believe that it facilitates significant performance improvements with smaller tile-sizes because it requires greater granularity.

We believe that the performance benefits from parallelizing significantly enhance user experience and satisfaction with the application; we aim to effectively target the major bottlenecks in the sequential implementation. Although users cannot distinguish between sequential and parallel version of the application, we believe that they are more likely to utilize an application which facilitates the rapid creation of the photomosaics. If we effectively targeted the bottlenecks within the sequential implementation, users will observe significant improvements in application performance. Furthermore, increased

performance facilitates the creation of photomosaics with increased granularity. The sequential implementation of the algorithm does not facilitate a pragmatic selection of small tile-sizes because every tile will have to be serially processed. This creates a large bottleneck because every tile requires traversal of all pixels to calculate average RGB values and traversal of all sample images in the database to find the "nearest" image to the tile in terms of Euclidean distance to their respective average RGB values. As tile-size decreases, the similarity of the outputted photomosaic image to the original "target" image increases. We believe that the performance improvement resulting from our parallelized version of the algorithm facilitates greater photomosaic quality because user's can flexibly specify the granularity. A sequential implementation would require the user to significantly consider the tile-size before executing the program. Photomosaic algorithms have the most application outside of artistic value when the promote increased granularity. Digital copyright functionality may require the production of a photomosaic which strongly resembles a multitude proprietary images; users generally browse proprietary images rapidly to select ones which they are interested in. If the photomosaic production algorithm has too long of a run-time, users will easily become distracted. Furthermore, the application of photomosaic algorithms to compression, database, and diagonalization technique in complexity theory become more relevant with parallelization because users can easily implement a multitude of advanced features which require computation. They are no longer bottlenecked by the sequential execution of the CPU for the entirety of the application.

Finally, we hope that the parallelization of various parts of the algorithm within our application will promote further development in segmentation and searching strategies. Our application contains a multitude of custom and ready-made modules which developers can easily modify and build upon to improve the processing of tiles within the "target" image. Our MATLAB functions are modular and extensible: they do not contain dependencies with other functions. Similarly, our C++ functions are also extensible and modular. We avoided utilizing hard-coded values so that changes to the various user-environments and structure of the algorithm will not result in errors. Developers can easily improve the segmentation process of the "target image." Object and boundary identification is boosted by the power of GPUs because many of the computations required for such machine-learning capabilities can easily be parallelized and do not depend on sequential logic. We hope that the mono-sized tile approach we implemented can be built on through our parallelized infrastructure. Furthermore, developers can also easily improve the search process for the nearest sample image from the "database." Overall, we believe that the parallelized implementation of the photomosaic algorithm is modular and extensible: future developers can easily improve various components of the algorithm because they are incentivized by the ability to harness to power of parallelization to implement computationally intensive tasks.

Part 4: Bottleneck Identification

High-level execution profile of serial (or previous-version) implementation

Our original implementation of the photomosaic program was a sequential algorithm. Our program took four inputs: directory path to image, directory path to the folder containing the tile photographs, file type of tile-photographs, and the size of tiles. It outputted a single saved file into the user's directory: mosaic.png. It had the same

assumptions and properties listed above for the parallel implementation. There are 6 MATLAB function files in our sequential implementation of the photomosaic algorithm: *Mosaic.m, tileMap.m, AverageColorImage.m, AverageColorTile.m, Distance.m,* and *Nearest.m*.

The main program is *Mosaic.m*; it outputs a saved file into the user's working directory: mosaic.png. It takes four intputs: directory path to image, directory path to the folder containing the tile photographs, file type of tile-photographs, and the size of tiles. First, it initializes a map of image file-names to average RGB values by calling the *tileMap.m* function. Then, it iterates through each of the tiles of the image as specified by the tile-size. For each tile, it first calls the *AverageColorTile.m* function on the tile's dimension to calculate the average RGB value. Then, it calls the *Nearest.m* function to return the image from the database with the closest average RGB value to the tile's average RGB value in terms of Euclidean distance. Then, it resizes the newly returned image from the database to the dimensions of tile-size. Finally, it places the newly resized image, represented as a vector of doubles, into a cell-array. After the program has finished iterating through all the "target" image's tiles, it converts the cell-array containing the newly tiled-images into an ordinary array. Finally, it saves the newly converted matrix into a file into the user's working directory: *mosaic.png*.

The *tileMap.m* function takes in two inputs: the path to the image-database directory and the file-type of images within the image-database. It outputs a map containing the image file-names within the directory mapped to their average RGB values. This function iterates through each of the images within the specified directory. For each image, it calls the *AverageColorImage.m* function to return its average RGB value. Then, it maps the image's name as a String to the average RGB value returned by the *AverageColorImage* function as a 3-tuple.

The *Nearest.m* function takes two inputs: the average RGB value of the image tile as a 3-tuple and the map of image names from the image-database to their average RGB values as 3-tuples. This function iterates through all the keys (image-names) within the map. For each key, it calls the *Distance.m* function to calculate the Euclidean distance between the average RGB value of the image, the mapped value for the key, and the average RGB values of the tile, passed in as an input. The function keeps track of the lowest recorded distance and nearest image name outside of the loop. For each key, if the returned value from the *Distance.m* function is less the lowest previously recorded distance, the program updates the value for lowest recorded distance and nearest image name. The function returns the nearest image name after it has finished looping through all the image-names (the key-list of the map).

The *Distance.m* function takes in two 3-tuples as inputs. It returns the Euclidean distance between the two tuples.

The *AverageColorImage.m* function takes in an image as the input. It returns the average RGB value of the image as a 3-tuple.

The *AverageColorTile.m* function takes five inputs: the image, the starting x-coordinate, the starting y-coordinate, the ending x-coordinate, and the ending y-coordinate. It returns the average RGB value of the tile specified by the input coordinates within the image.

Bottleneck(s)

There are two major bottlenecks in the serial implementation of the program: computing the average RGB value for each tile and searching for the most similar

sample. Neither of these steps require sequential logic: threads can independently calculate the average RGB values for each tile within the "target" image and find the nearest image from the "database" in terms of Euclidean distance. In a sequential implementation, every tile within the "target" image must be processed iteratively, creating a significant bottleneck. Therefore, before the next tile in the "target" image can be processed, the previous tile's average RGB value and the nearest image from the database in terms of Euclidean distance to the RGB values of the tile must be calculated.

The complexity of calculating the average RGB value for the "target" image tile is $\theta(T^2)$, where T is tile-size, because it requires iterating through all the pixels within the tile-area (defined as the square of tile-size).

The complexity of searching for the nearest image from the "database" in terms of Euclidean distance is $\theta(D)$, where D is the size of the database. It requires iterating through all the images within the database and calculating the Euclidean distance between its RGB values and the average RGB values from the relevant tile.

In a sequential implementation of the photomosaic algorithm, these two steps pose significant computational bottlenecks; the time-complexity of calculating each tile's RGB value and finding the nearest image from the database is $\theta(T^2+D)$, where T is the tile-size and D is the database-size. For each tile within the "target" image, the program must first calculate tile's average RGB values, which requires iterating through all the pixels and aggregating the average red, green, and blue values. Then, before progressing to the next tile within the "target" image, the program must search and find the closest image from the "database" in terms of Euclidean distance. The overall computational complexity of these bottlenecks in the sequential implementation is $\theta(N*(T^2+D))$, where N is the number of tiles into which the image is segmented into, T is the size of tile, and D is the size of the database. In contrast, assuming infinite resources, a parallelized implementation of these two steps would have a complexity of $\theta(T^2+ D)$. As a result, this bottleneck becomes especially significant as the size of the "database" of images becomes larger. Furthermore, this bottleneck becomes significant when the tile-size is large. Although this is less relevant for "target" images with a small size, users may specify large tile-sizes as the size of the input image increases.

Initially, we considered the mapping of image-names to average RGB values to also be a bottleneck. The time complexity of this step is $\theta(D*C^2)$, where D is the size of the database and C is the size of the image within the database. After running the application, we realized the run-time of this step is negligible. The images within the "database" are significantly smaller than the size of the original target image; processing these photographs for average RGB values is not costly. Although this may be because our "database" of images is particularly small, the run-time of this step is static and bound by the number and size of images within the directory. The time complexity of transferring all the images is directly proportional to the size of "database." We believe that the complexity required in transferring all the images to GPU memory is far greater than the time-complexity to iterate through the images sequentially. As a result, a parallel implementation of this step would be more costly because of the required transfer of images from CPU to GPU memory. Loading all the images in the "database" into a GPU-array would also have a run-time proportional to the size of the database. Furthermore, a parallelized implementation would also have to iterate through all the pixels in each of the images within the database. As a result, the run-time would also be proportional to the size of images within the database.

Part 5: Parallel Solution

After implementing a sequential version of our algorithm, we noticed that the run-time was significantly hindered for large input "target" images and large "databases" of images used in the construction of the photomosaic. Our program was bottlenecked in the calculation of each tile's average RGB value and the searching of the image from the "database" with the closest Euclidean distance. As a result, we created CUDA scripts which calculated the average RGB value for each tile within the "target" image and returned an array of indices to images within the "database" with the closest Euclidean distance. Assuming infinite resources, each one thread is assigned per tile and all the tiles can be processed simultaneously in parallel. Every thread simultaneously iterates through all of its' tile's pixels to compute the average RGB values and then iterates through all the images in the "database" to find the closest one in terms of Euclidean distance.

In this implementation, the program first calls the tileMap.m function to return a map of image-names from the "database" to their average RGB values. We implemented this step sequentially, identically to our prior versions' implementation. The tileMap.m function takes two inputs: the path to the image "database" and the type of images within the directory. For each image, it calculates the average RGB value using the AverageColorImage.m function and maps the image's name as a String to its relevant average RGB value as a 3-tuple. It returns a map containing key-value pairs for all images within the specified directory. The AverageColorImage.m function takes one input: an image. It returns the average RGB value of the image as a three-tuple by averaging each color channels value for all the piexels within the image. Then, we initialized a CUDAKernel object using a compiled PTX file of the CUDA script. After executing the code on the GPU, we then constructed a cell array of all the resized "nearest" images and write the matrix version of that array to the user's workspace as a png file: "mosaic.png."

Parallel algorithm description

Our parallelized version of the application still requires an implementation of sequential logic in conjunction with parallelized logic. First, we call the tileMap.m function to sequentially return a map of image-names within the database to average RGB values. This still has a run-time of $\theta(D*C^2)$, where D is the size of the database and C is the size of the image. Then, we initialize all the parameters required to be passed into GPU-memory as GPU-arrays. Then, we call the parallel algorithm to return a row-major indexed array containing the indices of images from the database required to construct the photomosaic. We parallelize the segmentation of the image into tiles: each thread in the GPU simultaneously processes each tile's average RGB value and finds the nearest sample image from the database in terms of Euclidean distance. The run-time of the parallelized sections of the algorithm will be discussed later. Finally, we resize each image within the array, construct the photomosaic array, and write the photomosaic and a .png file sequentially. This still has the run-time of $C^2$where R is the run-time of the built in *imresize* function, L is the length of a side of the "target" image, T is the tile-size.

We chose to allow the repetition of images from the "database" in the photomosaic because otherwise, the searching functionality which we are trying to parallelize will rely upon sequential logic and memory. If images are eliminated from the database, threads

may simultaneously select and delete images from the database, resulting in segmentation faults and scheduling problems (discussed previously and further in depth in the assumptions of the main program).

Under infinite resources, the parallel algorithm assigns one thread to every tile within the "target" image and can process them all simultaneously. Each thread calculates its tile's average RGB value and then iterates through of the images in the "database" to find the closest one in terms of Euclidean distance. If we have a finite number of resources, F, where F is smaller than the number of tiles, we will need to process the tiles in batches of size F. The number of batches required is the number of tiles divided by the batch-size, F.

First, the algorithm calculates the average RGB value of the thread's tile. For each color channel, it iterates all the tile's pixels and aggregates the relevant color channel's mean. Finally, it divides this aggregation by the size of the tile to calculate the relevant color channel's average value. As a result, it initializes three Doubles containing the tile's relevant average red, green, and blue color channel values. The run-time for this step is $\theta(3*T^2) \approx \theta(T^2)$, where T is the size of the tile.

Second, after calculating the tile's average red, blue, and green color channel values, the thread searches for the image from the database with the closest Euclidean distance in terms of average RGB values to the tile. The run-time for this step is $\theta(D)$, where D is the size of the "database" of images.

Assuming infinite resources, the overall run-time for the parallel algorithm is $\theta(T^{2*} D)$, where T is the size of the tile and D is the size of the database. If there are a finite number of resources, the overall run-time is $\theta((N/F)*(T^{2*}D))$, where N is the number of tiles within the "target" image, F is the number of available resources, T is the size of the tile, and D is the size of the database.

Code structure overview & concurrency-dependency graph

Our final implementation of the photomosaic algorithm took four inputs: directory path to image, directory path to the folder containing the tile photographs, file type of tile-photographs, and the size of tiles. It outputted a single saved file into the user's directory: mosaic.png. There are 3 MATLAB function files in our parallel implementation of the photomosaic algorithm: *mosaic_cuda.m, AverageColorImage.M, and tileMap.m.* Furthermore, there is one C++ file: *mosaic_cuda.cu*.

The main program is *mosaic_cuda.m*; it outputs a saved file into the user's working directory: mosaic.png. It takes four inputs: directory path to image, directory path to the folder containing the tile photographs, file type of tile-photographs, and the size of tiles. First, it initializes *tiles* a map of image file-names to average RGB values by calling the *tileMap.m* function. Then, it reads in the image. Second, it prepares all the parameters required to evaluate the CUDAKernel on the GPU. It initializes *reds, greens,* and *blues* as three matrix arrays containing the average red, green, and blue values for the keys within the *tiles* map. It does this indexing within the relevant color channel within the values of the *tiles* map. Then, it initializes *nearestTiles* as an array of ones, with a size equivalent to the number of tiles within the original image. Finally, it initializes *redGPU, blueGPU, greenGPU, nearestTilesGPU, and imGPU* as Double GPU-arrays. Third, it initializes and evaluates the CUDAKernel to return an array containing the indices of images within the *tiles* map which are the closest to "target" image's tiles

in terms of Euclidean distance. The number of blocks is calculated by dividing the number of tiles by the number of threads per block. After initializing the kernel's thread-block size and grid size, it evaluates the kernel by calling the built-in *feval* function. In our implementation, each block has 16 x 16 threads. Finally, it gathers the newly returned *nearestTilesGPU*-array from the CUDAKernel, constructs the final photomosaic matrix, and writes it to the user's working directory. After returning from the CUDAKernel, it initializes *nearestImageIndices* as an array by calling the built-in *gather()* function to return the updated *nearestTilesGPU*-array. Then, it iterates through all the indices in *nearestImageIndices*, keeping in mind that it is a row-major indexed linear array. For each index in the *nearestImageIndices* array, the program utilizes the stored value to index within the keys of the *tiles* map and returns the name of the relevant image. Then, it utilizes built-in functions *imread* and *imresize* to read in and resize the image from the database to the size of the tile. After the program has finished iterating through *nearestImageIndices*, all the cells within the cell-matrix *mosaic* have been initialized. The number of cells in *mosaic* is equivalent to the number of tiles in the original "target" image. Then, the program calls the built-in *uint8* and *cell2mat* functions to return *mosaic* as a matrix-array of 8-bit unsigned integers. Finally, the program writes *mosaic* into the user's workspace as *mosaic.png*. This module is custom and ready for developers to build upon because it avoids dependencies with other functions, hard-coded values, and static values. The only hard-coded value in our current implementation is the number of threads per block in the GPU to initialize the CUDAKernel's grid size and thread-block size. Through the localization of repeated logic within our code through refactoring it into smaller helper functions, I believe that the structure of our code promotes future development and improvement.

The *tileMap.m* function takes two inputs: a path to the directory of images and the type of images within the directory. It returns one output: a map of all image names, stored as Strings, within the database to their relevant average RGB values, stored as a 3-tuple. It iterates through all the images to the directory. For each image, it calculates the average RGB values through calling the *AverageColorImage.M* function. It then maps the image's name to the average RGB value. This function is a custom and ready module for developers to utilize whenever they need to construct a map of image-names to average RGB values; it does not contain dependencies with other functions, hard-coded, or static values. Developers can easily improve upon the map-creation algorithm because of the localization of functionality.

The *AverageColorImage.m* function takes in one input: an image. It returns one output: a 3-tuple containing the average RGB values for the image. It iterates through all the pixels in the image and averages their relevant red, green, and blue values into a 3-tuple. This is a ready module for developers to call whenever they need to calculate the average RGB value for an image; this logic may need to be repeated in future improvements.

The *mosaic_cuda.cu* is a C++ file containing the parallelized portion of the algorithm. It takes nine inputs: an array containing the indices of "nearest" images within the *mosaic* map, an array containing the image, an array containing the average red color channel values from the *mosaic* map, an array containing the average blue color channel values from the *mosaic* map, an array containing the average green color channel values from the *mosaic* map, the number of samples in the "database" of

images, the number of tiles in the original "target" images, and the number of threads per block. This file effectively partitions the image into tiles which can be processed in parallel: each tile's average RGB value is calculated and the nearest sample image is identified from the database.

The main method *mosaic_cuda_double* calls the *mosaic* function on all the inputted parameters and returns.

The *mosaic* method calculates the thread's current tile by multiplying the block index with the number of threads per block and adding the thread index. Then, it calculates the top-left pixel of the current tile. Then, it calls the *getTileAverage* method three times to calculate the average red, blue, and green color channel values for the tile. Then, it iterates through all the samples in the database to search for the index of the image with the closest Euclidean distance to the tile in terms of average RGB values. For each sample, if the Euclidean distance is less than the last recorded value for the minimum distance between sample and tile. Finally, it indexes within the *nearestTiles* array and updates the value to the newly retrieved index of the "nearest" sample. This custom and ready module promotes flexibility and extensibility because it does not rely on dependencies with other functions (other than *getTileAverage*) and does not contain hard-coded values. Users can easily determine if the array is being updated if any of the indices contain values other than one (the *nearestTilesGPU* is initialized as an array of ones). Furthermore, this method provides an intuitive platform for developers because repeated logic is refactored into modular helper-functions while the logic for searching for the nearest sample image is localized. This method facilitates user development in the searching of the closest sample image for each tile: maybe the measure of similarity will be improved from the Euclidean distance metric.

The *getTileAverage* method takes in 6 inputs: the row index, the column index, the slice index, tile-size, image-size, and input image. It outputs the average color channel value for the tile as a Double. The splice index indicates the color channel for which the method is calculating the average of. It iterates through all the indices of the tile by calling the *getLinearIndexing* method and aggregates the relevant color channel's values. Finally, it divides the aggregate value by the size of the tile and returns that value. This ready module is flexible and extensible because it does not rely on dependencies with other functions and does not contain hard-coded values. This method provides an easy method for developers to calculate average RGB values for any specified tile within an image.

The *getLinearIndexing* method is a helper function which enables developers to index within the image, which is column major indexed. This function takes in five inputs: the row index, the column index, the splice index, the number of rows, and the number of columns. The splice index indicates the color channel for which the function is returning an index for within the array. It outputs the relevant linear index for the parameters passed in. This module enables developers to easily index within the image even though it is column indexed by calling the helper function instead of having to manually do the calculation everytime. Furthermore, it promotes flexibility if the index ordering changes: the developer only needs to edit the logic for calculating the linear index.

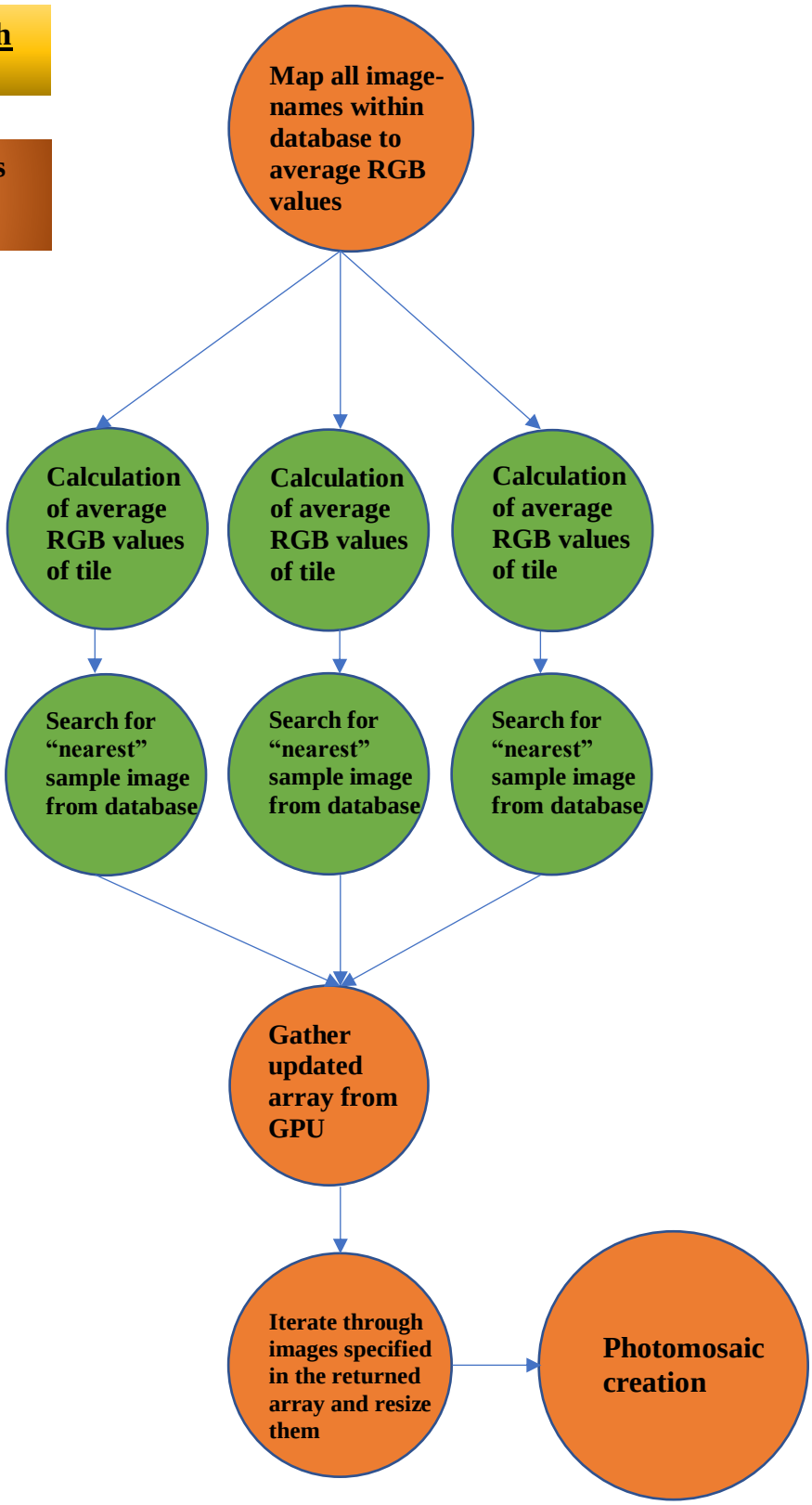
Concurrency Dependency Graph

The parallelized algorithm depends on data from the map

Rather than having to serially segment the image into tiles, our implementation allows the division of this problem into sub-processes which can be executed concurrently. However, these processes also have some dependencies.

Every thread is assigned a tile and processes the tile concurrently. Each thread must first calculate average RGB value before searching for the "nearest image." As a result, because the threads can execute concurrently and do not depend upon the execution of other threads each thread concurrently handles the segmentation of an image-tile. However, this concurrency has dependencies upon the logic executed by each thread. The concurrency of each thread is dependent on the processing of the tile's average RGB value before finding the "nearest" sample image.

Photomosaic creation depends on the evaluation of the CUDAKernel. The "nearest" images can only be resized to construct the photomosaic after kernel returns the updated array containing the indices of the "nearest "images within the map.



Parallel scheduling and basic/serial sub-problem(s)

As described above, the mapping of image-names in the database to their average RGB values and the resizing/construction of the photomosaic matrix are basic sequential sub-problems within our application. The mapping process must be completed prior to the evaluation of the CUDAKernel because the parallel logic is dependent on the data in the map. After the kernel has finished evaluating, the CPU sequentially constructs the photomosaic because of the built-in MATLAB functions *imresize* and *cell2mat*.

Our parallel algorithm divides the image-segmentation and search process into multiple processes which are executed concurrently by the use multiple cores in GPUs. The bottleneck we are trying to avoid is the processing of tiles: the calculation of each tile's average RGB value and the searching of the nearest sample image in terms of Euclidean distance. There are two basic sequential sub-problems in our parallel algorithm; parallel scheduling must be follow the order of the basic sub-problems because otherwise, threads will attempt to execute logic in parallel which depends on the execution of prior logic. All the threads can execute concurrently because neither the calculation of tile-average RGB value nor the search for the "nearest" sample image requires logic/data from the processes of other threads. The threads can simultaneously process the tiles because each tile contains independent information from the rest of the image and each tile requires an independent search through the database. First, the tile's average RGB values must be calculated. Second, the closest sample image to the tile in terms of Euclidean distance. The second step must be executed after the first step because the process cannot find the "nearest" image without first calculating the average RGB value for the tile. However, these two sequential steps do not need to be executed sequentially for every tile. Assuming infinite resources, each thread can be assigned to one tile and execute the serial sub-problems for each tile concurrently. If there are limited resources, F, these sequential sub-problems would have to be computed on batches of tiles size, N/F, where N is the number of tiles and F are the number of limited resources.

Part 6: Complexity

- Calculating average RGB values for all database images
    - This step of the program involves fully iterating through all of the sample images in the database and computing their average RGB
    - If the database has D images of size C, the overall complexity of this step is $\boldsymbol{\theta(DC^2)}$
    - The runtime of this step is the same for the parallel and sequential versions of mosaic creation
    - Note that for our application, D = 250 and C = 40 on average for both .jpg and .png image types. For any large input size (Eg. a 1000 x 1000 image with tiles of size 40), **the runtime of this step is negligible and not a major bottleneck.**

- Sequentially assembling the mosaic for an image of side length L and tile size T
  - For each tile, it takes $\theta(T^2)$ time to compute its average RGB values
  - For each tile, it takes $\theta(D)$ time to find the image from the database that is the closest match to the tile, assuming the database contains D images
  - The exact complexity of Matlab's built-in *imresize* function is unknown. For our analysis, we will represent the runtime of *imresize* as $\theta(R)$
  - The number of tiles in this case is $\frac{L^2}{T^2}$

Since all of our computation is done sequentially in this version, we take the total amount of work done on each tile, and multiply it by the total number of tiles. We also add the fixed cost of computing the average RGB values for all of the database images.

The overall runtime is $\theta(\frac{L^2}{T^2}(T^2 + D + R) + DC^2)$ then

- Assembling the mosaic in parallel for an image of side length L and tile size T with infinite parallel resources
  - For each tile, it takes $\theta(T^2)$ time to compute its average RGB values
  - For each tile, it takes $\theta(D)$ time to find the image from the database that is the closest match to the tile, assuming the database contains D images
  - Again, we assume that Matlab's *imresize* function rune in $\theta(R)$ time

In our parallel implementation, parallelization changes the runtime of step 1 and step 2 in the list above. For these steps, since the tiles of the source image are disjoint, we can handle each of them in parallel. Assuming we have infinite parallel resources and we dedicated 1 thread to each tile, the runtime of these first two steps is reduced to $\theta(T^2 + D)$

In our parallel code, the final assembly of sample images into a mosaic, as well as the resizing of the sample images, is done sequentially. Therefore the runtime of these steps does not change. The runtime of computing average RGB values for the database images does not change either.

Therefore, the overall runtime assuming infinite parallel resources is:

$$\theta\left(T^2 + D + \frac{L^2}{T^2}(R) + DC^2\right)$$

- Parallel mosaic with finite parallel resources
    - In the prior analysis we assumed that we had infinite parallel resources, so if we assigned one thread per tile, the overall parallel runtime was based on the runtime for a single tile
    - If we had a finite number F of parallel resources, where F is smaller than the number of tiles, then the previous runtime analysis is no longer accurate
    - Instead of performing steps 1 and 2 on all the tiles at once, we would have to compute the tiles in "batches" of size F

The number of "batches" is simply the number of tiles divided by F: $\frac{L^2}{T^2 F}$

As before, the time taken to resize the sample images and to compute the RGB averages for all of the sample images would be unchanged, since all of this is done sequentially. The loss of parallel resources only affects steps 1 and 2, in which we compute the RGB average for each tile of the source image, and search for the most similar sample image. Therefore, the overall runtime with a limit of F parallel processes is:

$$\theta\left(\frac{L^2}{T^2F}(T^2+D)+\frac{L^2}{T^2}(R)+DC^2\right)$$

Comments on complexity:

When profiling our code, we noticed that the main bottleneck was computing the average RGB value for each tile and searching for the most similar sample. As seen in the complexity analysis above, as a result of our parallelization we reduce the runtime of these 2 steps by a factor in $O(\frac{L^2}{T^2})$. Therefore, our choice of parallel implementation correctly targets the bottlenecks in our program. Also, note that the scale of theoretical improvement in runtime is directly proportionate to the the number of tiles. Based on this fact we anticipate that the savings in runtime from parallelization will scale with the number of tiles.

Communication:

The only communication that occurs in our program is the transfer of data between CPU and GPU. To minimize transfer latency, we designed the program so that data is only transferred back and forth between CPU and GPU one time.

Granularity:

We determined that the appropriate granularity of the parallel implementation was devoting 1 thread per tile. Every thread has to compute the average RGB value of its tile, and compare that average RGB value to the RGB tuples of all the sample images in the

database. This ensures that even with small tile sizes (eg. 10 x 10), each thread performs a nontrivial amount of work.

We did have the option of giving each thread more work by assigning multiple tiles to each thread. However we chose not to do this because then we would not be fully exploiting the inherent independence of computations in the mosaic problem. Also, because the GPUs we were using had several thousand cores, even for considerable input sizes (1000 x 1000 image, tiles of size 25) we had sufficient parallel resources to maintain granularity without sacrificing speed.

We found that memory access patterns were difficult to manage for our program. Each tile was responsible for a square segment of the image across 3 different color channels. Since the image is transferred to GPU as a 1-dimensional column-major vector, this meant that even a single thread would be accessing many different disjoint sections of the input, dispersed throughout the input. It was therefore hard to manage threads in a way that would exploit locality. This is one of the possible optimizations mentioned in the Discussion section.

Part 7: Experiments

Establishing Correctness

For the sequential version of our program, we verified correctness by running the program on a large (1000 x 1000) image with a small (10 x 10) tile size. We inspected the results visually and noted that the result closely matched the structure and color of the source image. Based on this we were confident that our program correctly matched sample images with source image tiles and assembled and resized the sample images correctly to create an accurate mosaic.

For our parallel program, we were able to verify correctness by comparing the results with the output of our sequential program. A pixel by pixel comparison of the mosaic outputs indicated that the results of the 2 programs were the same.

Testing Performance

To test runtime, we executed our code on a fixed image with varying tile sizes. We executed the programs on GPU-Compute5, using timers in MATLAB to measure the actual computational time.

Our expectation before testing is that there will be no significant performance difference for small input sizes, because both the sequential and parallel programs will run relatively quickly. We predict that as the number of tiles grows large, the runtime of the sequential program will grow exponentially, while the runtime of the parallel program

will grow more slowly. For large input sizes we expect the parallel code to significantly outperform the sequential code.

Results

Image Size (per side)	Tile Size (per side)	Number Tiles	Runtime: Regular (seconds)	Runtime: Parallel (seconds)
2448	1224	4	4	98.5
2448	612	16	3.43	28.3
2448	306	64	3.9	13.09
2448	153	256	6.5	12.5
2448	51	2304	40.56	38.04
2448	17	20736	371.8	330
2448	8	93636	2574	1634

Figure 1: The original image used for testing. 2448 x 2448

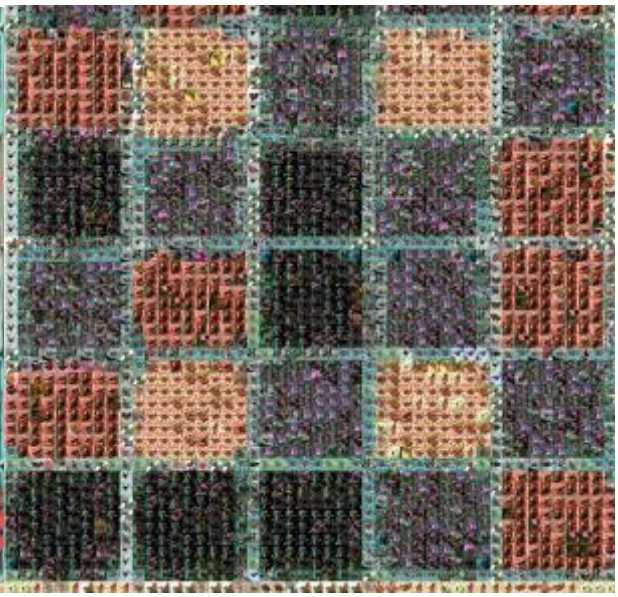


Figure 2: The mosaic output, with a tile size of 51

Comments on results

For small input sizes, the results contradicted our expectations. While we predicted the runtimes to be similar, the GPU code is actually significantly slower for small inputs. This makes sense because GPU cores are not designed to handle large tasks, so dividing 2 or 4 large tasks on the GPU is actually slower than running them on the CPU, which is designed for such tasks.

For large input sizes, our expectations were correct. The GPU code runs significantly faster than the sequential code, and the gap in performance grows with the size of the input.

Part 8: Discussion

Goals

Overall, we accomplished our goals for this project. Both the sequential and parallel versions of our mosaic creation algorithm have been verified to execute correctly. Our performance testing was consistent with our expectations and showed that the parallel code significantly outperforms the sequential code as the size of the input grows large.

Lessons

One of the greatest challenges we encountered while testing our parallel code was issues with data types. Depending on how we cast the data objects being passed to the GPUs, we found at times that our data was hugely distorted when it got transferred to the GPU. As a result we learned that it was important to be careful with the choice of datatypes, and that it was important to be consistent with datatypes throughout the whole program. In a language like MATLAB, which doesn't constantly require the programmer to be aware of types, this posed a significant challenge.

Another important lesson in this project was considering the scale at which parallelization is worthwhile and beneficial. Our initial plan was to work with 400 x 400 images with a tile size of 20 x 20. Our peers in the class suggested that this input size (400 tiles) was too small for us to see significant performance benefits from parallelization. During coding, verification, and analysis, we applied their feedback by looking for input sizes that were more suitable to our choice of architecture. As a result of this process, we ended up significantly scaling up the size of our computation. For verification we used 1000 x 1000 images; for performance analysis we used a 2448 x 2448 image with as many as 93000 tiles in the largest example. By doing this we discovered the scale at which we saw the greatest benefits from parallelization.

Limitations

In its current state our program works correctly for a limited set of cases. The most obvious limitation is that our code right now assumes that the input image is square; for non-square images the program will fail due to indexing issues. Dealing with non-square images would allow the program to be run on a much greater set of input images. Possible adjustments that would make this possible include resizing the input, or calculating an appropriate tile size within the program.

Another limitation at the moment is that if the user inputs a tile size that does not evenly divide the side length of the input image, the program will crash due to indexing issues. One option to correct this is to create an I/O dialogue that warns the user if their tile size is invalid, to adjust the user-given tile size within the program, or to suggest a set of appropriate tile sizes given an input image.

A more challenging limitation is the fact that the number of sample images used by our program is small (~250 .png and ~250 .jpg images). One drawback of this is that many

of the sample images are repeated in the output. If the source image has large sections that are all similarly colored, there are large contiguous sections of the mosaic for which the mosaic image is the same, which is unappealing visually and defeats the purpose of a mosaic. An obvious solution would be to increase the size of the database by finding and saving more images.

However, this approach still leaves open the possibility of repetitions, especially if the source image has large sections that are the same color. In our sequential program we could remedy this by eliminating a sample image from the database after it has been used. This approach is dependent on sequential execution, and it would not work for a parallel implementation. This is why eliminating repetitions from a parallelized mosaic creation program is an outstanding limitation. One potential solution may be to divide the source image into sections and use a disjoint set of sample images for each section to reduce the number of repetitions. This would require a greatly expanded database of sample images, as well as significant work to modify the program.

Potential Optimizations

One optimization would be to compute the RGB values for all of the sample images at one time, and then store this information as a series of key-value pairs on disc. This way, instead of computing the averages every time the program was executed, we could just load the saved information from disc. For a large database of sample images this optimization would be essential.

Another optimization is one that was discussed earlier in the section on complexity. At the moment, the blocks in our program are organized in squares, with each block operating on a square section of the input image. As discussed earlier this method does not take advantage of locality because different threads in each block will be accessing different, and potentially widely dispersed, sections of the input image.

When the source image gets transferred to GPU, it gets rearranged as a column-major indexed 1-D array. Our program requires that each thread access a square section of the original image across all 3 color channels, which means that every thread will be accessing a widely distributed range of locations in the input. Therefore, we were unable to devise an organization of threads and blocks that would take advantage of locality. However by changing the use of threads or rearranging blocks, it may be possible to run the parallel code with a better memory access pattern.

References

The images we used for testing and verification were sourced from Pexels.com under the Creative Commons Zero license.

The butterfly images used as samples for our database were sourced from a publicly available repository hosted by the University of Edinburgh.

1. Tran, N. (1998). Generating Photomosaics: An Empirical Study. Retrieved from http://www-cs.ccny.cuny.edu/~wolberg/capstone/photomosaics/EmpiricalStudySAC99.pdf
2. Kärner, M. (n.d.). Process for creating photo mosaics. Retrieved from http://kodu.ut.ee/~b04866/poster.pdf
3. Junhwan Kim and Fabio Pellacini. Jigsaw image mosaics. ACM Trans. Graph., 21(3):657–664, July 2002.
4. Yoon-Seok Choi, Bon-Ki Koo, and Ji-Hyung Lee. Template based image mosaics. In Proceedings of the 6th International Conference on Entertainment Computing, ICEC'07, pages 475–478, Berlin, Heidelberg, 2007. SpringerVerlag.
5. Linlin Jing, Kohei Inoue, and Kiichi Urahama. An npr technique for pointillistic and mosaic images with impressionist color arrangement. In Proceedings of the First International Conference on Advances in Visual Computing, ISVC'05, pages 1–8, Berlin, Heidelberg, 2005. SpringerVerlag.