Photomosaic Generation by Rearranging Subimages, with GPU Acceleration

Yi Yang, Yasuaki Ito, and Koji Nakano
Department of Information Engineering, Hiroshima University
Kagamiyama 1-4-1, Higashi-Hiroshima, 739-8527 Japan
Email: {yang, yasuaki, nakano}@cs.hiroshima-u.ac.jp

***Abstract*—The main contribution of this paper is to show a new photomosaic generation method by rearranging subimages of an image. In the photomosaic generation, an input image is divided into small subimages and they are rearranged such that the rearranged image reproduces another image given as a target image. Therefore, this problem can be considered as a combinatorial optimization problem to obtain the rearrangement which reproduces approximate images to the target image. Our new idea is that this rearrangement problem is reduced to a minimum weighted bipartite matching problem. By solving the matching problem, we can obtain the best rearrangement image. Although it can generate the most similar photomosaic image, a lot of computing time is necessary. Hence, we propose an approximation algorithm of the photomosaic generation. This approximation algorithm does not obtain the most similar photomosaic image. However, the computing time can be shortened considerably. Additionally, we accelerate the computation using the GPU (Graphics Processing Unit). The experimental results show that the GPU implementations for the optimization algorithm and the approximation algorithm can accelerate the computation to 40 and 66 times faster than the serial CPU implementation, respectively.**

I. Introduction

Photomosaic is a technique which produces an input image by arranging many smaller images, called *tiles*. Photomosaic is used for works of art and advertisements such as portraits and logo designs [1]. Figure 1 shows an example of photomosaic that produces Lena [2] consisting of small images from the image databases [3], [4], [5]. In general, a method of photomosaic generation with an image database consisting of many small images is widely used. Given a target image, the method produces a photomosaic image, in the following steps:

1) Divide the target image into rectangular subimages,
2) For each subimage, find a small image which is similar to the subimage image from the image database, and
3) Arrange the small images to the corresponding subimages.

The main contribution of this paper is to propose a new method for generating a photomosaic image which reproduces a target image obtained by rearranging tiles of an input image. Figure 2 shows an example of the proposed photomosaic generation method, where an input image is divided into 32×32 tiles. Given an input image to be rearranged and a target image, the proposed method generates a photomosaic image as follows:

1) Divide the input image into tiles
2) Rearrange the tiles such that the rearranged image reproduces a target image

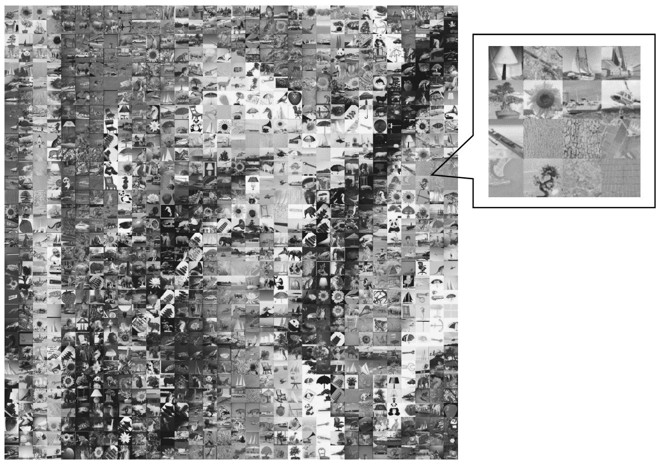


Fig. 1. An example of photomosaic image by arranging small images

This photomosaic generation can be considered as an assignment problem that tiles in an input image are assigned to tiles in a target image. Thus, in this paper, we propose an optimization algorithm and an approximate algorithm for generating photomosaic images.

In the optimization algorithm, we reduce the rearrangement problem of tiles to a minimum weighted bipartite matching problem. By solving the matching problem, we can obtain the rearranged image. Although the optimization algorithm can generate the most similar photomosaic image, a lot of computing time is necessary.

On the other hand, the approximation algorithm is repeating the local search such that for all possible pairs of two tiles, if the total error is reduced by swapping two of tiles compared with the before swapping, the two tiles are swapped. The exchanging for all possible pairs is repeated until no swapping is performed, that is, no more improvement is possible by the local search. The approximation algorithm cannot generate optimal photomosaic images since all possible rearrangements are not considered. However, the resulting photomosaic images by the approximation algorithm are virtually the same as those by the optimization algorithm.

Furthermore, we propose a parallel approximation algo-

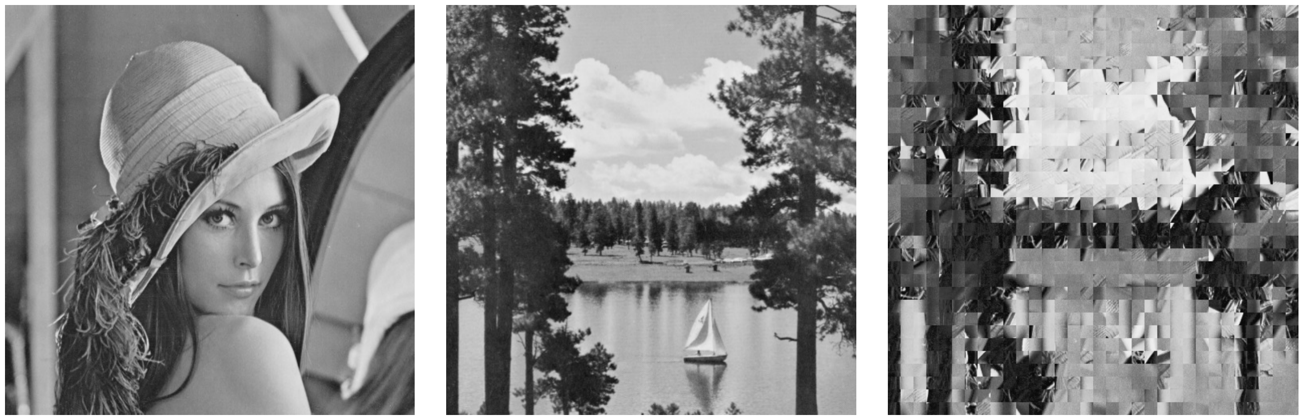

(a) Input image (Lena) (b) Target image (Sailboat) (c) Photomosaic image

Fig. 2. Photomosaic generation in this work

rithm. In the parallel algorithm, the swap operation of two tiles is performed in parallel. To obtain pairs of tiles that can be swapped at the same time, we solve the edge-coloring problem of the complete graph. Using the result of the edge-coloring, we perform the swap operation concurrently.

Recent GPUs (Graphics Processing Units) have a lot of processing cores that can be used for general purpose parallel computation. CUDA (Compute Unified Device Architecture) [6] is the architecture for general purpose parallel computation on GPUs. We can develop parallel algorithms to be implemented in GPUs using CUDA. Therefore, many studies have been devoted to implement parallel algorithms using CUDA [7], [8], [9]. As the second contribution, we have used the GPU to accelerate the computation for the error values computation that is pre-computation for the above optimization and approximation algorithms and the parallel approximation algorithm. Our experimental results show that the GPU implementation with the optimization algorithm can run up to 40 times faster than the CPU implementation. On the other hand, the GPU implementation with the approximation algorithm can run up to 66 times faster than the CPU implementation.

This paper is organized as follows. Section II explains outline of our photomosaic generation method. In Section III, we propose an optimization algorithm for the photomosaic method by reducing the method to a minimum weighted bipartite matching problem. We then go on to show an approximate algorithm of the photomosaic method to shorten the generating time in Section IV. In Section V, we show a GPU implementation to accelerate the computation. Section VI shows the resulting photomosaic images, and shows the computing time. Section VII concludes our work.

II. Photomosaic Generation Method

The main contribution of this section is to show two algorithms of our proposed photomosaic method that subimages of an input image are rearranged such that the rearranged image reproduces a target image. Let the size of an input image I and a target image T be $N \times N$. Suppose an input image and a target image are divided into S subimages, called *tiles* of size $M \times M$, that is, $S = \frac{N}{M} \times \frac{N}{M}$. Let $\{I_1, I_2, \dots, I_S\}$ be the set of tiles from the input image, and $\{T_1, T_2, \dots, T_S\}$ be the set of tiles from the target image. The rearrangement of tiles $I_1, I_2, \dots, I_S$ can be considered as assignment of tiles of I to T. Since no two tiles can be assigned to the same tile of T, each tile of I is assigned to exactly one tile of T. Let $r : \{I_1, I_2, \dots, I_S\} \to \{I_1, I_2, \dots, I_S\}$ denote a bijective function that is rearrangement of tiles in I. Now we introduce the error between two tiles I_u and T_v $(1 \le u, v \le S)$ as follows;

$$E(I_u, T_v) = \sum_{1 \le i,j \le M} |e_{i,j}|, \tag{1}$$

where $e_{i,j}$ is the error at each pixel location (i, j) of tiles I_u and T_v. Let R be an image of size $N \times N$ obtained by rearranging the tiles of I. The total error between the rearranged image R and the target image T is defined as follows:

$$\mathit{Error}(R, T) = \sum_{1 \le u \le S} E(r(I_u), T_u). \tag{2}$$

We note that we use gray scale images for input and target images in this paper. However, we can easily extend the proposed photomosaic method to deal with color images only by changing the error function in Eq. (1).

A rearranged image R is a good approximation of the target image T if the difference between R and T is small enough. Thus, the best rearranged image R^* that reproduces T is a rearranged image given by:

$$R^* = \arg\min_R \mathit{Error}(R, T). \tag{3}$$

Since S tiles are rearranged, the total number of rearrangement of them is $S!$. Therefore, a straightforward method to find the best rearrangement is to evaluate $\mathit{Error}(R, T)$ for all possible $S!$ rearranged images R.

Using the above error function, our proposed method generates photomosaic images in the following steps:

Step 1: Divide an input image and a target image into S tiles, respectively.

Step 2: Compute the error values between the tiles of two images $E(I_u, T_v)$ $(1 \leq u, v \leq S)$.

Step 3: Rearrange tiles I_u $(1 \leq u \leq S)$ such that the total error in Eq. (2) is small.

In the following sections, we propose an optimization algorithm and an approximation algorithm efficiently to perform Step 3.

The distribution of intensity levels of images is different for each image in general. If the distribution of an input image greatly differs from a target image, it is difficult to rearrange tiles of the input image to reproduce the target image. Therefore, before rearranging the tiles of an input image, we adjust the distribution of an input image to that of a target image using *the histogram equalization* [10]. The histogram equalization is to transform the intensity levels of the image so that the histogram of the resulting image is equalized to become a constant. In this work, the distribution of an input image is changed to that of a target image using the histogram equalization. Figure 3 shows the histogram-equalized image of the input image (Figure 2(a)) such that the distribution of the input image is adjusted to that of the target image (Figure 2(b)). If the distribution of an input image and/or a target image is exceedingly small, it is difficult to adjust the distribution. However, this adjustment is effective when the distribution is concentrated to the certain range. Indeed, to obtain the photomosaic image in Figure 2, the histogram-equalization is applied to the input image. In other words, the photomosaic image consists of the tiles into which Figure 3 is divided. In the following sections, all input images are histogram-equalized to adjust the distribution to the target images in advance.

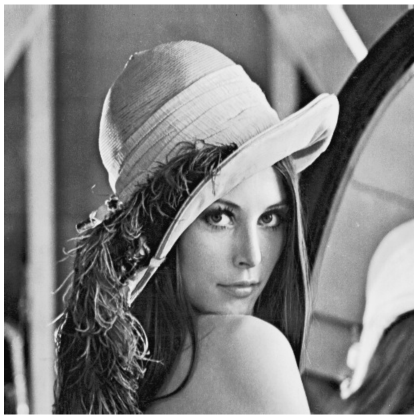


Fig. 3. Adjustment of the distribution of intensity levels in the input image (Figure 2(a)) to that in the target image (Figure 2(b)) using the histogram equalization

III. Optimization Algorithm

The main contribution of this section is to show an optimization algorithm to obtain the rearranged image of which the total error in Eq. (2) is the smallest by reducing the rearrangement problem to a minimum weighted bipartite matching problem.

Recall that our proposed photomosaic method is finding the rearrangement of subimages such that the total error is small. Now, consider a weighted complete bipartite graph (V_1, V_2, E) such that for every two vertices $v_1 \in V_1$ and $v_2 \in V_2$ is an edge in E. As illustrated in Figure 4, each subset of vertices consists of S vertices and they represent subimages of the input image and target image $I_1, I_2, \ldots I_S$ and $T_1, T_2, \ldots T_S$, respectively. Using this bipartite graph, obtaining one of the rearranged images R can be considered as finding a *matching* m in which each vertex has exactly one edge incident on it. In addition, suppose that each edge between I_u and T_v $(1 \leq u, v \leq S)$ has weight $w_{u,v}$, as follows;

$$w_{u,v} = E(I_u, T_v).$$

The sum of the weight values in m is equivalent to the total error between the rearranged image R_m obtained by m and the target image T, that is,

$$\mathit{Error}(R_m, T) = \sum_{(u,v) \in m} w_{u,v}.$$

Thus, obtaining the best rearranged image R^* is finding a matching of minimum weight. In other words, the rearranged problem can be reduced to a minimum weighted bipartite matching problem.

To solve the minimum weighted bipartite matching problem, there are several existing algorithms. The Kuhn-Munkres algorithm, which is also known as the Hungarian method, solves this problem in $O(S^3)$ time [11], [12]. In this algorithm, a weighted complete bipartite graph is considered as a matrix of edge weights, and the problem is solved with respected to these values. On the other hand, Blossom algorithm is an algorithm for consulting maximum matchings on graphs [13]. The algorithm constructs the matching by iteratively finding an augmenting path and shrinking a cycle called a blossom. There are several efficient implementations of Blossom algorithm [14], [15]. Note that Blossom algorithm performs maximum graph matching in a graph that is not bipartite. However, in our experiments, Blossom V [15] that is one of the implementations of Blossom algorithm is the fastest implementation for the case that S is at most several thousands among the above implementations. Therefore, we use Blossom V in the following experiments.

The aforementioned optimized algorithm can find a photomosaic image with the minimum total error. On the other hand, the generating time is also important. Photomosaic is used in interactive photomosaic system [16] and real time video photomosaic [17], [18]. To build such systems, GPUs are utilized to accelerate the generation of photomosaic [19], [20]. Since photomosaic images in the above are generated with the image database, the problem is different from this work. Therefore, in the next section, we present an approximate algorithm using the local search and its parallel algorithm.

IV. Approximation Algorithm

This section presents an approximation algorithm of the proposed photomosaic generation using the local search, and its parallel approximation algorithm.

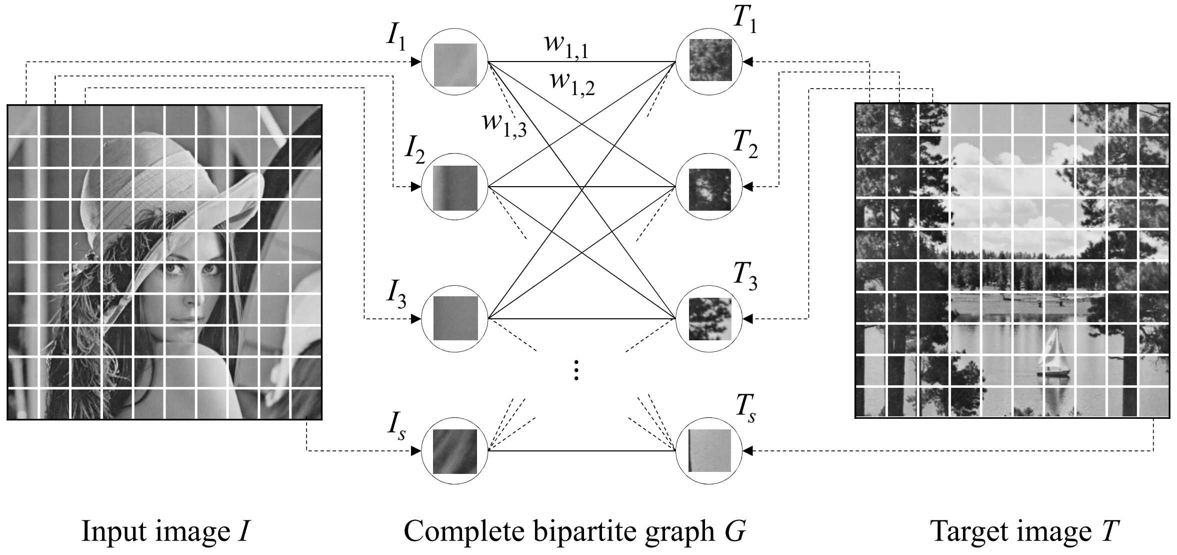


Fig. 4. Complete weighted bipartite graph for the optimization algorithm

A. Serial Approximation Algorithm

The idea of the proposed approximation algorithm is that given an input image divided into tiles, the method repeats the local search such that for all possible pairs of two tiles, if the total error is reduced by swapping two of tiles compared with the before swapping, the two tiles are swapped. The exchanging for all possible pairs is repeated until no swapping is performed, that is, no more improvement is possible by the local search. Given an input image $I = \{I_1, I_2, \ldots, I_S\}$ and a target image $T = \{T_1, T_2, \ldots, T_S\}$, the algorithm of the approximation algorithm is shown in Algorithm 1.

Algorithm 1 Serial approximation algorithm

```
1: repeat
2:     $flag \leftarrow 0$
3:     for all $u, v$ such that $1 \leq u < v \leq S$ do
4:         if $E(I_u, T_u) + E(I_v, T_v) > E(I_v, T_u) + E(I_u, T_v)$ then
5:             swap $I_u$ and $I_v$
6:             $flag \leftarrow 1$
7:         end if
8:     end for
9: until $flag = 0$
```

The local search is performed for $\frac{S(S-1)}{2}$ pairs in each iteration of the “for all” loop. Let k be the number of the ”for all” iteration. The total computing time of this approximate algorithm is $O(kS^2)$. In our experiments as shown in Section VI, the value k takes at most 9, 8, and 16 for $S = 16 \times 16$, 32×32, and 64×64, respectively. Thus, the running time of this approximation algorithm is much shorter than that of the optimization algorithm. On the other hand, since the approximate algorithm cannot generate all possible combinations of tiles, the total error of the photomosaic image obtained by the approximate algorithm must be larger than that by the optimization algorithm. Namely, the quality of the photomosaic image by the approximate algorithm is lower. However, in our experiments, the resulting photomosaic images by the approximation algorithm are virtually the same as those by the optimization algorithm although the total error is larger.

B. Parallel Approximation Algorithm

Next, let us consider the parallel execution of the approximate algorithm to accelerate the computation. As shown in Algorithm 1, the swap operations of two tiles are performed sequentially. Suppose that the exchanges are performed simultaneously. Clearly, two pairs that include the same tile cannot be swapped at the same time. For example, two pairs of tiles (I_1, I_2) and (I_2, I_3) cannot be swapped at the same time since I_2 is included in the both pairs. Therefore, it is necessary to perform parallel swapping of pairs in which the tiles are distinct. To obtain such pairs, we use a graph theoretic result [21] as follows:

Theorem 1. *A complete graph with n-vertices is n-edge-colorable if n is odd, and $(n-1)$-edge-colorable if n is even.*

Figure 5 illustrates an example of a complete graph with 16 vertices painted by 15 colors such that no vertex is connected to edges with the same color. In other words, no two edges with the same color share a vertex. The reader should refer to [21] for the proof of Theorem 1.

Suppose that an input image consisting of S tiles $I = \{I_1, I_2, \ldots, I_S\}$ is given. We draw a complete graph $G = (V, E)$ of S as follows:

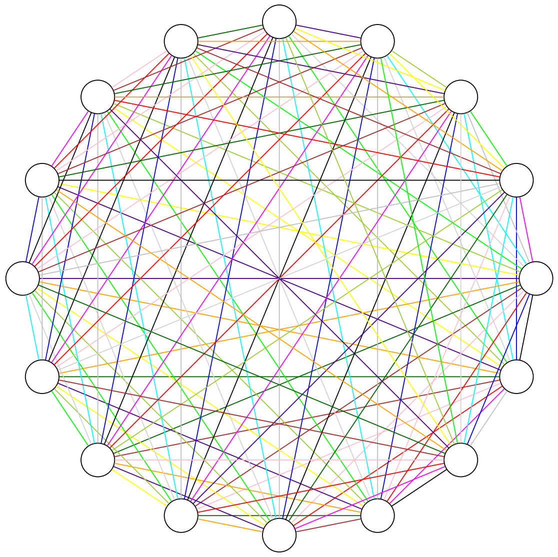


Fig. 5. A 15-edge-coloring of the complete graph with 16 vertices

- $V = \{I_1, I_2, \ldots, I_S\}$ is a set of tiles each of which corresponds to a tile of I.
- For each pair I_u and I_v, E has a corresponding edge connecting I_u and I_v $(\in V)$.

Clearly, an edge (I_u, I_v) $(1 \leq u < v \leq S)$ corresponding to a pair of two tiles to be swapped. Also, $G = (V, E)$ is a complete graph with S vertices. Hence, G is at most S-edge-colorable from Theorem 1. Suppose that all of the edges in E are painted by at most S colors $1, 2, \ldots, S$. Using the result of the edge coloring, we define edge groups $P_1, P_2, \ldots, P_S$. Each group P_i $(1 \leq i \leq S)$ has pairs of index of vertices such that when an edge (I_u, I_v) with color i, tuple (u, v) is included in P_i. We note that when S is even, the graph is $(S-1)$-edge-colorable. If this is the case, we consider that P_S is empty, that is, $P_S = \varnothing$. It should have no difficulty to confirm that all tiles in each group are distinct. For example, a complete graph with 16 vertices is 15-edge-colorable. Using the result of the coloring, edges are divided into 16 edge groups $P_1, P_2, \ldots, P_{16}$, as follows:

$$\begin{aligned}
P_1 &= \{(1,2),(3,15),(4,14),(5,13),(6,12),(7,11),(8,10),(9,16)\}\\
P_2 &= \{(1,4),(2,3),(5,15),(6,14),(7,13),(8,12),(9,11),(10,16)\}\\
P_3 &= \{(1,6),(2,5),(3,4),(7,15),(8,14),(9,13),(10,12),(11,16)\}\\
P_4 &= \{(1,8),(2,7),(3,6),(4,5),(9,15),(10,14),(11,13),(12,16)\}\\
P_5 &= \{(1,10),(2,9),(3,8),(4,7),(5,6),(11,15),(12,14),(13,16)\}\\
P_6 &= \{(1,12),(2,11),(3,10),(4,9),(5,8),(6,7),(13,15),(14,16)\}\\
P_7 &= \{(1,14),(2,13),(3,12),(4,11),(5,10),(6,9),(7,8),(15,16)\}\\
P_8 &= \{(1,16),(2,15),(3,14),(4,13),(5,12),(6,11),(7,10),(8,9)\}\\
P_9 &= \{(1,3),(2,16),(4,15),(5,14),(6,13),(7,12),(8,11),(9,10)\}\\
P_{10} &= \{(1,5),(2,4),(3,16),(6,15),(7,14),(8,13),(9,12),(10,11)\}\\
P_{11} &= \{(1,7),(2,6),(3,5),(4,16),(8,15),(9,14),(10,13),(11,12)\}\\
P_{12} &= \{(1,9),(2,8),(3,7),(4,6),(5,16),(10,15),(11,14),(12,13)\}\\
P_{13} &= \{(1,11),(2,10),(3,9),(4,8),(5,7),(6,16),(12,15),(13,14)\}\\
P_{14} &= \{(1,13),(2,12),(3,11),(4,10),(5,9),(6,8),(7,16),(14,15)\}\\
P_{15} &= \{(1,15),(2,14),(3,13),(4,12),(5,11),(6,10),(7,9),(8,16)\}\\
P_{16} &= \varnothing
\end{aligned}$$

Given an input image $I = \{I_1, I_2, \ldots, I_S\}$ and a target image $T = \{T_1, T_2, \ldots, T_S\}$, Algorithm 2 shows a parallel approximation algorithm of the photomosaic method using the above edge groups. Since all tiles in each P_i $(1 \leq i \leq S)$ are distinct, the swap operation at line 6 can be performed at the same time. We note that edge groups $P_1, P_2, \ldots, P_S$ obtained by edge-coloring depend only on the number of tiles S. In other words, they are not independent from input images and their size. Therefore, we assume that the number of tiles S is fixed and edge groups $P_1, P_2, \ldots, P_S$ are computed in advance. After that using them, photomosaic images are generated for various input images.

Algorithm 2 Parallel approximation algorithm

```
 1: repeat
 2:   $flag \leftarrow 0$
 3:   for $i = 1$ to $S$ do
 4:     for all $(u, v)$ in $P_i$ do in parallel
 5:       if $E(I_u, T_u) + E(I_v, T_v) > E(I_v, T_u) + E(I_u, T_v)$ then
 6:         swap $I_u$ and $I_v$
 7:         $flag \leftarrow 1$
 8:       end if
 9:     end for
10:   end for
11: until $flag = 0$
```

V. GPU Acceleration for the Error Computation and the Parallel Approximate Algorithm

The main purpose of this section is to show our GPU implementations of the error computation and the approximate algorithm.

We briefly explain CUDA architecture that we will use. NVIDIA provides a parallel computing architecture called *CUDA* on NVIDIA GPUs. CUDA uses two types of memories in the NVIDIA GPUs: *the global memory* and *the shared memory* [22]. The global memory is implemented as an off-chip DRAM of the GPU, and has large capacity, say, 1.5-12 GBytes, but its access latency is very long. The shared memory is an extremely fast on-chip memory with lower capacity, say, 16-96 Kbytes.

CUDA parallel programming model has a hierarchy of thread groups called *grid*, *block* and *thread*. A single grid is organized by multiple blocks, each of which has equal number of threads. The blocks are allocated to streaming multiprocessors such that all threads in a block are executed by the same streaming multiprocessor in parallel. All threads can access to the global memory. However, threads in a block can access to the shared memory of the streaming multiprocessor to which the block is allocated. Since blocks are arranged to multiple streaming multiprocessors, threads in different blocks cannot share data in the shared memories. CUDA C extends C language by allowing the programmer to define C functions, called *kernels*. By invoking a kernel, all blocks in the grid are allocated in streaming processors, and threads in each

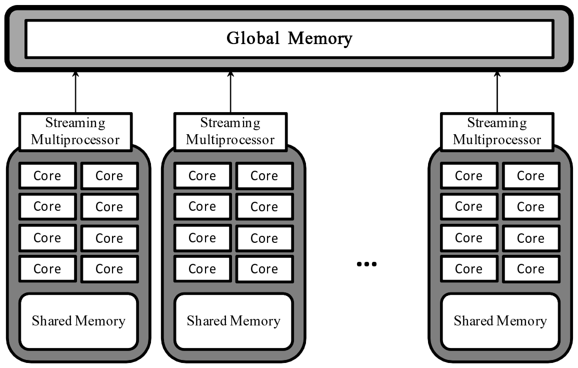


Fig. 6. CUDA hardware architecture

block are executed by processor cores in a single streaming processor.

We are now in a position to explain how we implement the following two computations on the GPU:

(i) the error values computation in Step 2, and
(ii) the parallel approximate algorithm in Step 3.

Regarding the optimization algorithm in Step 3, since it is not easy to parallelize the algorithm, we sequentially perform it on the CPU. We assume that an input image and a target image of size $N \times N$ are stored in the global memory in advance, and the implementation writes the resulting photomosaic image to the global memory. Furthermore, we assume the number of tiles S is fixed and the edge groups $P_1, P_2, \ldots, P_S$ are also stored in the global memory.

In Step 2, given an input image and a target image, the error values of $E(I_u, T_v)$ $(1 \leq u, v \leq S)$ are computed. To implement this step, S CUDA blocks are invoked. Each CUDA block is responsible for computing S error values $E(I_u, T_1), E(I_u, T_2), \ldots, E(I_u, T_S)$ $(1 \leq u \leq S)$. In each CUDA block, multiple threads are used and compute the error values in parallel, as follows. First, threads in each CUDA block read pixel values of tile I_u and store them to the shared memory. After that, the corresponding error values $E(I_u, T_1), E(I_u, T_2), \ldots, E(I_u, T_S)$ are computed using multiple threads one by one.

To implement the parallel approximation algorithm of Step 3, S CUDA kernels are invoked for each "for all" iteration shown in Algorithm 2, where S is the number of edge groups each of which shows pairs enabled to be swapped at the same time. The implementation performs the local search for $P_1, P_2, \ldots, P_S$, in turn. A CUDA kernel that performs the local search for the corresponding edge group is invoked for each group, that is, the execution is synchronized whenever the computation of each iteration is finished. Since the swap operation is concurrently performed based on the edge group, one may argue that the total error by computing Eq. (2) increases compared with that by the sequential one. However, in our experiments, the total error are almost the same and the quality of the resulting photomosaic images cannot be distinguished.

Using the above two GPU implementation, the proposed photomosaic method can be accelerated as follows. First, the GPU implementation of the error values computation of tiles in Step 2 can be used before the optimization algorithm and the approximation algorithm. After that, using the error values, the optimization algorithm is performed on the CPU. On the other hand, the approximation algorithm can be performed by the above GPU implementation of the parallel approximation algorithm.

VI. Experimental Results

In this section, we will show the resulting photomosaic images and the computing time. We have used several images in the image database [3]. To obtain the rearrangement in the optimization algorithm, we have used Bloom V [15] that can solve a minimum weight graph matching efficiently on the CPU. Figure 7 shows the resulting photomosaic images generated using the optimization algorithm and the sequential and parallel approximation algorithms for $S = 16 \times 16, 32 \times 32$, and 64×64. We also shows other three examples in Figure 8.

According to Figure 7, since for $S = 16$, the number of tiles is small and the size of tiles is large, the resulting photomosaic images do not reproduce the target image well. In the resulting images for $S = 32$, the quality becomes better, but the borders of tiles are still distinct. However, for $S = 64$, most of the borders are not clear and the photomosaic image is very similar to the target image.

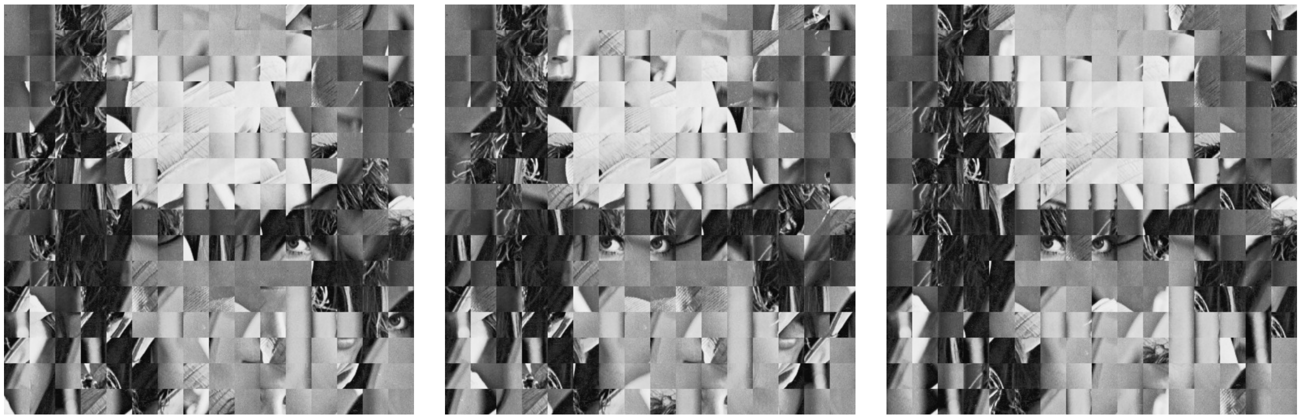
Table I shows the total error in Eq. (2) of the resulting images in the figure. The total errors of the optimization algorithm are smaller than those of the approximation algorithm. However, according to Figure 7, the difference of the quality between the optimization algorithm and the approximate algorithm is very small. Furthermore, since the order of executing the local search between the sequential and parallel approximation algorithm is not the same, their total errors differ, but the difference is small.

TABLE I
The total error of the photomosaic images in Figure 7

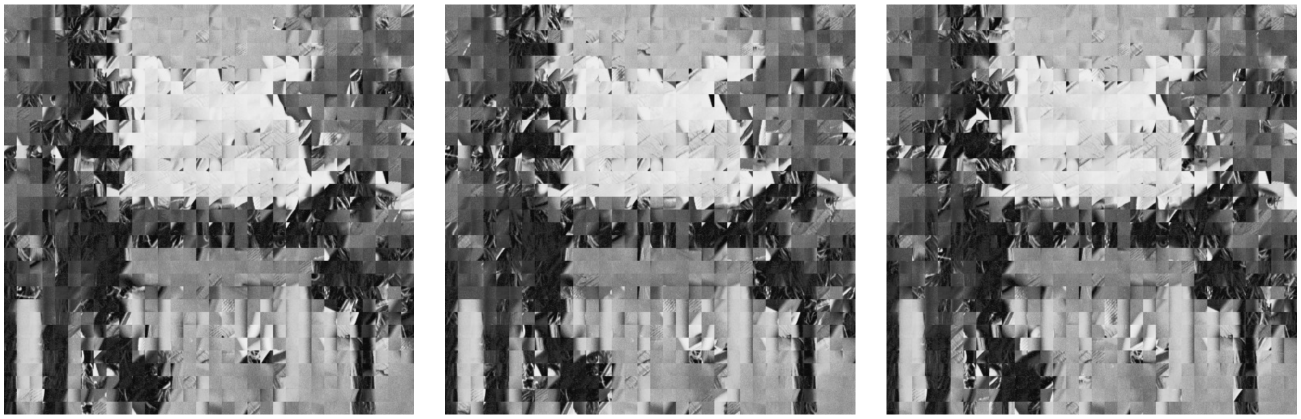
S	Optimization	Approximation	
	CPU	CPU	GPU
16×16	7529146	7701450	7676311
32×32	5410140	5520554	5506782
64×64	3877820	3945836	4047410

Next, let us evaluate the computing time of generation photomosaic images. We have used a PC with Intel Core i7-3770 CPU (3.9GHz) and NVIDIA Tesla K40 GPU (875MHz) [23]. Using this, a sequential CPU implementation and a parallel GPU implementation have been evaluated. The following computing time is the average time of generating four kinds of photomosaic images shown in Figures 7 and 8.

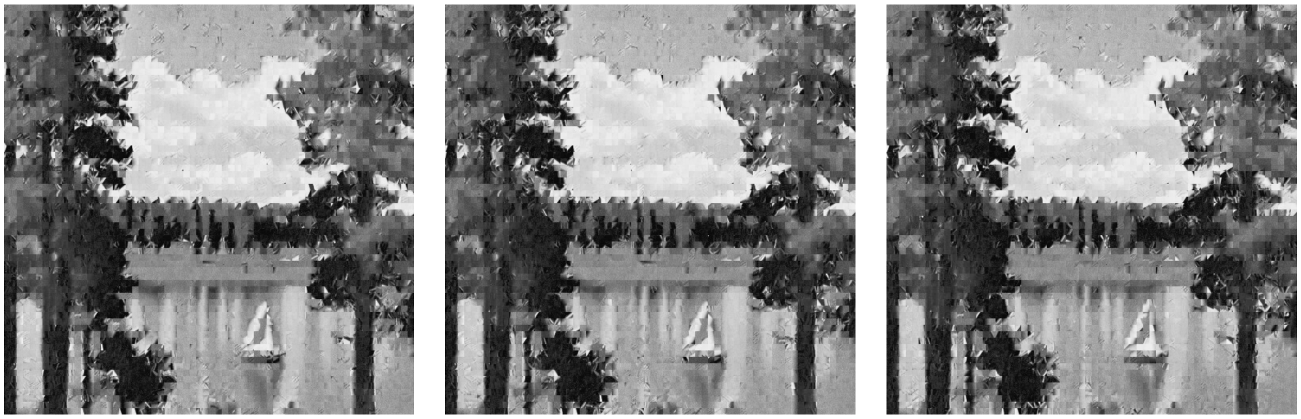
Sequential algorithms are executed as they are using a single thread running Intel Core i7-3770. We may accelerate these sequential algorithms using multiple threads and/or SIMD instructions. However, these acceleration techniques for


Optimization Approximation (CPU) Approximation (GPU)

(a) $S = 16 \times 16$


Optimization Approximation (CPU) Approximation (GPU)

(b) $S = 32 \times 32$

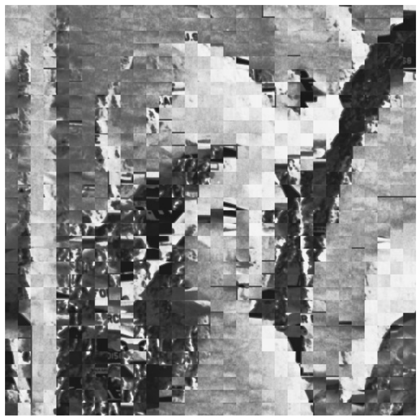

Optimization Approximation (CPU) Approximation (GPU)

(c) $S = 64 \times 64$

Fig. 7. The Comparison of the optimization algorithm and the approximation algorithm on the CPU and the GPU

Input image (Airplane)

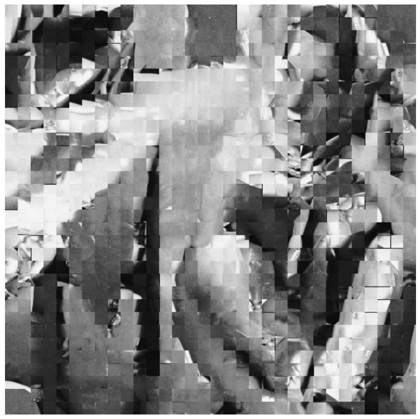
Target image (Lena)


Photomosaic image

(a) Airplane $\rightarrow$ Lena

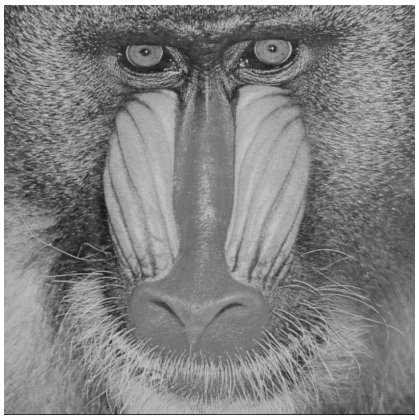
Input image (Peppers)

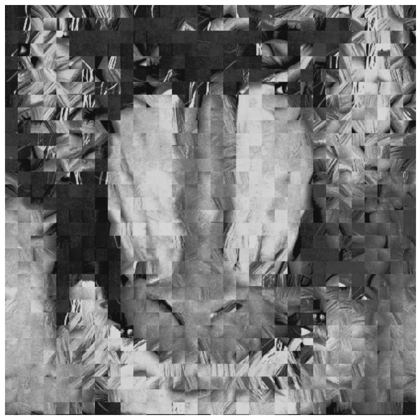
Target image (Barbara)


Photomosaic image

(b) Peppers $\rightarrow$ Barbara

Input image (Tiffany)


Target image (Baboon)


Photomosaic image

(c) Tiffany $\rightarrow$ Baboon

Fig. 8. Examples of our photomosaic method by the optimization algorithm with 32×32 tiles for images of size 512×512

Core i7 CPU are out of scope of this work, because our goal is not to compare the capability of NVIDIA GPU and Intel Core i7 CPU. The speedup factors of GPU implementations over CPU implementations shown by our experiments are just for reference purpose.

Table II shows the computing time of the error values between tiles in Step 2. When the size of images is larger, the computing time is longer. Also, when the number of tiles is larger, the computing time is longer. According to the results, the GPU implementation of the error values computation can run 66 to 92 times faster than the CPU implementation.

TABLE II
THE COMPUTING THE ERROR VALUES BETWEEN TILES IN STEP 2

Size of images	number of tiles S	CPU[s]	GPU[s]	Speed-up
512×512	16×16	0.397	0.005	78.30
	32×32	1.599	0.017	92.12
	64×64	6.253	0.107	58.22
1024×1024	16×16	1.574	0.020	77.28
	32×32	6.178	0.077	80.00
	64×64	24.890	0.269	92.70
2048×2048	16×16	6.238	0.079	78.56
	32×32	20.980	0.316	66.39
	64×64	98.485	1.230	80.08

Table III shows the computing time of rearrangement of tiles in Step 3. According to the table, the computing time of rearrangement does not depend on the size of image but on the number of tiles. When $S = 16 \times 16$, the GPU implementation of the approximation algorithm is slower than the CPU implementation because the number of tiles is small and most of processor cores in the GPU are not used. However, the GPU implementation of the approximation algorithm is at least 2.6 and 18 times faster than the CPU implementation for $S = 32 \times 32$ and 64×64, respectively.

TABLE III
THE COMPUTING TIME OF REARRANGEMENT OF TILES IN STEP 3

Size of images	number of tiles S	Optimization CPU[s]	Approximation CPU[s]	Approximation GPU[s]	Approximation Speed-up
512×512	16×16	0.062	0.006	0.012	0.50
	32×32	15.686	0.179	0.063	2.84
	64×64	1209.082	6.660	0.343	19.42
1024×1024	16×16	0.070	0.006	0.011	0.55
	32×32	15.518	0.180	0.069	2.61
	64×64	1280.027	6.906	0.372	18.56
2048×2048	16×16	0.070	0.008	0.014	0.57
	32×32	15.877	0.169	0.065	2.60
	64×64	1304.024	7.467	0.352	21.21

Recall that the photomosaic generation with the optimization algorithm can use the GPU implementation of the error values computation. On the other hand, that with the approximation algorithm can use the GPU implementation of the error values computation and the rearrangement of tiles. Table IV shows the total computing time of the photomosaic generation to evaluate the GPU acceleration. The optimization algorithm can be accelerated up to 40 times faster than the CPU implementation for $S = 16 \times 16$. However, when the number of tiles is larger, since the computing time of the rearrangement of tiles on the CPU dominates the total execution time, the speed-up factors take almost one. On the other hand, in the approximation algorithm, the rearrangement of tiles can be executed on the GPU. For the approximation algorithm, our GPU implementation attains a speed-up factor of up to 66 over the serial CPU implementation.

VII. Conclusion

In this paper, we have proposed a photomosaic generation method by rearranging divided images. In the photomosaic generation, an input image is divided into small subimages and they are rearranged such that the rearranged image reproduces another image given as a target image. We have shown an optimization algorithm and an approximation algorithm of rearranging tiles. Our new idea is that this rearrangement problem is reduced to a minimum weighted bipartite matching problem. In addition, we have also proposed an approximation algorithm of the photomosaic generation. This approximation algorithm does not obtain the most similar photomosaic image, but the computing time can be shortened considerably.

Furthermore, we accelerate the computation using the GPU. The experimental result shows that the GPU implementation of the optimization and approximation algorithms attains a speed-up factor of up to 40 and 66 times over the sequential CPU implementation, respectively.

References

[1] R. Silvers, *Photomosaics*, M. Hawley, Ed. Henry Holt and Company, Inc., 1997.

[2] L.-M. Po, "Lenna 97: A complete story of Lenna," http://www.ee.cityu.edu.hk/~lmpo/lenna/Lenna97.html [18 January 2016], 2001.

[3] "The USC-SIPI image database," http://sipi.usc.edu/database/.

[4] L. Fei-Fei, R. Fergus, and P. Perona, "Learning generative visual models from few training examples: an incremental bayesian approach tested on 101 object categories," in *IEEE. CVPR 2004, Workshop on Generative-Model Based Vision*, 2004.

[5] G. McCarter and A. Storkey, "Image sequence segmentation data," http://homepages.inf.ed.ac.uk/amos/afreightdata.html.

[6] NVIDIA Corporation, "CUDA ZONE," https://developer.nvidia.com/cuda-zone.

[7] T. Fujita, K. Nakano, and Y. Ito, "Bulk execution of Euclidean algorithms on the CUDA-enabled GPU," *International Journal of Networking and Computing*, vol. 6, no. 1, pp. 42–63, 2016.

[8] D. Takafuji, K. Nakano, and Y. Ito, "A CUDA C program generator for bulk execution of a sequential algorithm," in *Proc. of International Conference on Algorithms and Architectures for Parallel Processing*, Aug. 2014, pp. 178–191.

[9] K. Tani, D. Takafuji, K. Nakano, and Y. Ito, "Bulk execution of oblivious algorithms on the unified memory machine, with GPU implementation," in *Proc. of International Parallel and Distributed Processing Symposium Workshops*, May 2014, pp. 586–595.

[10] M. S. Nixon and A. S. Aguado, *Feature Extraction and Image Processing*, 2nd ed. Academic Press, 2008.

[11] H. W. Kuhn, "The Hungarian method for the assignment problem," *Naval research logistics quarterly*, vol. 2, no. 1-2, pp. 83–97, 1955.

[12] J. Munkres, "Algorithms for the assignment and transportation problems," *Journal of the Society for Industrial and Applied Mathematics*, vol. 5, no. 1, pp. 32–38, 1957.

[13] J. Edmonds, "Maximum matching and a polyhedron with 0,1-vertices," *Journal of Research of the National Bureau of Standards*, vol. 69B, no. 1, pp. 125–130, 1965.

[14] W. Cook and A. Rohe, "Computing minumum-weight perfect matchings," *INFORMS Journal on Computing*, vol. 11, no. 2, pp. 138–148, 1999.

[15] V. Kolmogorov, "Blossom V: A new implementation of a minimum cost perfect matching algorithm," *Mathematical Programming Computation*, vol. 1, no. 1, pp. 43–67, 2009.

TABLE IV
THE TOTAL COMPUTING TIME OF THE PHOTOMOSAIC GENERATION

Size of images	Number of tiles S	Optimization			Approximation		
		CPU [s]	CPU+GPU [s]	Speed-up	CPU [s]	GPU [s]	Speed-up
512×512	16×16	0.459	0.068	6.76	0.403	0.017	23.24
	32×32	16.597	15.705	1.10	1.777	0.081	21.98
	64×64	1215.335	1209.202	1.01	12.914	0.450	28.67
1024×1024	16×16	1.644	0.092	17.89	1.580	0.033	47.79
	32×32	21.705	15.598	1.39	6.367	0.148	43.04
	64×64	1304.918	1280.309	1.02	31.797	0.643	49.45
2048×2048	16×16	6.308	0.155	40.74	6.245	0.098	63.57
	32×32	36.857	16.199	2.28	21.149	0.386	54.75
	64×64	1402.510	1305.270	1.07	105.953	1.587	66.76

[16] M. Fujisawa, T. Amano, T. Taketomi, G. Yamamoto, Y. Uranishi, and J. Miyazaki, "Interactive photomosaic system using GPU," in *Proc. of the 20th ACM international conference on Multimedia*, 2012, pp. 1297–1298.

[17] Y.-S. Choi, S. Jung, J. W. Kim, and B.-K. Koo, "Real-time video photomosaics with optimized image set and GPU," *Journal of Real-Time Image Processing*, vol. 9, no. 3, pp. 569–578, 2014.

[18] G. Wijesinghe, S. B. M. Sah, and V. Ciesielski, "Grid vs. arbitrary placement of files for generating animated photomosaics," in *IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*, 2008, pp. 2734–2740.

[19] K. Dongwann, S. Sang-Hyun, and R. Seung-Taek, "A parallel framework for fast photomosaics," *IEICE TRANSACTIONS on Information and Systems*, vol. E94-D, no. 10, pp. 2036–2042, 2011.

[20] M. Slomp, M. Mikamo, B. Raytchev, T. Tamaki, and K. Kaneda, "GPU-based SoftAssign for maximizing image utilization in photomosaics," *International Journal of Networking and Computing*, vol. 1, no. 2, pp. 211–299, 2011.

[21] R. J. Wilson, *Introduction to Graph Theory*, 4th ed. Addison Wesley, 1996.

[22] *CUDA C Programming Guide Version 7.0*, NVIDIA Corporation, 2015.

[23] NVIDIA Corporation, "Tesla K40 GPU active accelerator," 2013.