International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 1, Number 2, pages 211–229, July 2011

# GPU-based SoftAssign for Maximizing Image Utilization in Photomosaics

Marcos Slomp, Michihiro Mikamo, Bisser Raytchev, Toru Tamaki, Kazufumi Kaneda

Intelligent Systems and Modeling Laboratory
Faculty of Engineering, Hiroshima University
1-4-1 Kagamiyama, Higashi-hiroshima City, Hiroshima Prefecture, 739-8527, Japan

Received: January 31, 2011
Revised: May 2, 2011
Accepted: June 20, 2011
Communicated by Yasuaki Ito

**Abstract**

Photomosaic generation is a popular non-photorealistic rendering technique, where a single image is assembled from several smaller ones. Visual responses change depending on the proximity to the photomosaic, leading to many creative prospects for publicity and art. Synthesizing photomosaics typically requires very large image databases in order to produce pleasing results. Moreover, repetitions are allowed to occur which may locally bias the mosaic. This paper provides alternatives to prevent repetitions while still being robust enough to work with coarse image subsets. Three approaches were considered for the matching stage of photomosaics: a greedy-based procedural algorithm, simulated annealing and SoftAssign. It was found that the latter delivers adequate arrangements in cases where only a restricted number of images is available. This paper introduces a novel GPU-accelerated SoftAssign implementation that outperforms an optimized CPU implementation by a factor of 60 times in the tested hardware.

*Keywords:* Non-photorealistic rendering (NPR), Photomosaic, SoftAssign, Simulated Annealing, General-Purpose GPU programming (GPGPU)

## 1 Introduction

As opposed to photorealistic rendering, non-photorealistic rendering (NPR) algorithms trade-off physical accuracy in exchange for feature highlighting or artistic effects. The current affordability of computers and digital cameras is empowering many inventive outcomes from user-generated content, paving access to NPR effects like photomosaics [6].

Photomosaics comprise special instances of mosaics. A mosaic is a stylization of an image, consisting of a collection of bulky primitives. A photomosaic is then envisaged as a mosaic whose bulky primitives are images themselves, as illustrated in Figure 1. Over the past decades several photomosaic generation algorithms were conceived [8, 5, 2].

Subjects experience different visual responses based on their relative proximity to a photomosaic. Despite of being a fancy effect, photomosaics are widely acclaimed by advertisement producers. In the hands of skilled marketing personnel, photomosaics can capture the essence of a product at

Figure 1: A $20 \times 20$ photomosaic assembled from 1500 images assigned through one of the proposed algorithms (simulated annealing). The input image is miniaturized at the left for reference. The usage of available images was maximized so that no image appears repeatedly in the resulting mosaic.

different perspectives, ultimately entertaining and attracting potential consumers. Many commercial opportunities can arise through clever utilization of photomosaics.

Silvers *et al.* delivered the first efforts on photomosaic generation [8]. Klein *et al.* extended the concept to *video mosaics*, where lower resolution movies are used as bulky primitives to assemble a larger video [5]. Di Blasi *et al.* derived a faster method to generate photomosaics by varying the sizes of the images that appear in the photomosaic, clustered into an *antipole tree data structure* [2]. The interested reader can further refer to Battiato *et al.*'s survey on photomosaic generation techniques [1].

Current photomosaic generation algorithms, however, require huge amounts of images in order to produce attractive results. Repetitions are allowed to occur, which may locally bias the mosaic. Another common strategy, if only a small image set is available, is to reduce the size of the bulky primitives, subdividing the input image into very small chunks, but this ends up breaking the illusion of the photomosaic, unless observed very closely. Tuning photomosaics manually is obviously exhaustive and impractical.

This paper is an extension to our previous work in [7]. The original publication studied three methods to maximize the usage of available images in photomosaics by preventing image repetitions: a greedy-based procedural algorithm, a simulated annealing driven solution [4] and a SoftAssign-based [3] approach. From these strategies, the greedy-based is the fastest, but lacks in quality; simulated annealing, on the other hand, is time-prohibitive. SoftAssign is capable of producing photomosaics that are qualitatively equivalent to simulated annealing in a much faster rate, but

still time demanding nonetheless. The present paper builds upon this investigation and provides an efficient GPU-based implementation of SoftAssign. The performance increase is about 60 times when compared to an optimized CPU implementation, in the tested hardware configuration.

The next three sections of this document presents the studies performed in the above mentioned prior research [7]: Section 2 formalizes the problem of maximizing the number of images used in photomosaics; Section 3 reviews the three image assignment approaches; Section 4 focuses on the initial results originally published.

The remainder of this document is designed based on the analysis of the referred research, paving the road to further investigation of the SoftAssign algorithm in GPU: Section 5 provides a blueprint to implement SoftAssign and discusses issues and parallelization strategies, Section 6 focuses on mapping the SoftAssign algorithm to the graphics pipeline in order to exploit parallelism in GPU, Section 7 extends the initial results with additional mosaics and performance comparisons between CPU and GPU implementations of SoftAssign, and Sections 8 and 9 conclude the paper and points future directions.

## 2 Maximizing Image Utilization on Photomosaics

In general, given a source image $I$ and a set of $n$ images $T = \{t_1, ..., t_n\}$, ordinarily known as *tiles*, synthesizing a photomosaic $P$ digests to:

1. Subdivide the target image into a regular $w \times h$ lattice of rectangular regions; these regions will be referred to as *patches* and denoted as $p_i \in \{p_1, ..., p_m\}$, where $m = w \times h$;
2. For each patch $p_i$, search for an appropriate tile $t_j \in T$ to replace the patch; this step may require manipulation of the tile images (resizing, cropping, etc.)

The goal is to maximize the usage of distinct tiles in the resulting photomosaic. Tile utilization can be maximized by preventing any individual tile to be assigned to more than a single patch. That means that a tile can only be assigned to a patch if it is not currently assigned to any other patch. In the end, every patch must hold a unique tile assigned to it, although it is possible for some tiles to remain unassociated to any particular patch, since typically there are more tiles than patches. When there are more patches than tiles, no solution can be determined.

However, maximizing tile utilization itself does not guarantee that the resulting photomosaic $P$ will resemble the input image $I$. Therefore, the *visual similarity* between patches and tiles can not be neglected, being critical to guide the optimization process. The metric chosen to determine visual similarities is a simple Euclidean distance in RGB color space. More sophisticated metrics or color spaces could be used instead, but such simplistic measure proved to be satisfactory.

In order to determine the visual similarity between a patch $p_i$ and a tile $t_j$, both are further partitioned into smaller $u \times v = s$ rectangular regions: $p_i = \{p_i^1, ..., p_i^s\}$ and $t_j = \{t_j^1, ..., t_j^s\}$. The average intensity of each partition, $\overline{p}_i^k = (\overline{R}_i^k, \overline{G}_i^k, \overline{B}_i^k)$ and $\overline{t}_i^k = (\overline{r}_j^k, \overline{g}_j^k, \overline{b}_j^k)$, is then computed through simple component-wise arithmetic average. Finally, the visual similarity function can be defined as:

$$d(p_i, t_j) = \sqrt{\sum_{k=1}^{s} \left[ (\overline{R}_i^k - \overline{r}_j^k)^2 + (\overline{G}_i^k - \overline{g}_j^k)^2 + (\overline{B}_i^k - \overline{b}_j^k)^2 \right]} \quad (1)$$

Note that the lower the value of the visual similarity function, the more closely tile $t_j$ resembles patch $p_i$; a "perfect" match would evaluate $d(p_i, t_j)$ to *zero*. Therefore it is sound to think of $d(p_i, t_j)$ as a *distance* function. Such formulation will later be used to populate a *distance matrix*, a fundamental component of all of the involved algorithms (Section 3.1, Figure 2).

The problem of maximizing image utilization on photomosaics can then be formalized as an optimization scheme for minimizing the sum of $d(p_i, t_j)$ under the restriction that all patches must have uniquely assigned tiles, as formulated below:

$$min \sum_{i=1}^{m} \sum_{j=1}^{n} d(p_i, t_j)\ y(p_i, t_j) \tag{2}$$

subject to the following constraints:

$$y(p_i, t_j) = \begin{cases} 1 & \text{if } t_j \text{ is assigned to } p_i \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

$$\sum_{j=1}^{n} y(p_i, t_j) = 1 \tag{4}$$

$$\sum_{i=1}^{m} y(p_i, t_j) = 1 \quad \text{or} \quad \sum_{i=1}^{m} y(p_i, t_j) = 0 \tag{5}$$

The membership function $y(p_i, t_j)$ models the patch-tile assignment by assuming the binary values defined in Equation 3. The constraint from Equation 4 ensures that a patch will always have a single tile assigned to it, while the final constraints from Equation 5 guarantee that tiles are assigned *at most* once. All three algorithms detailed in the subsequent section enforce these requirements.

# 3 Photomosaic Optimization Strategies

The photomosaic optimization strategies from [7] can be summarized as follows:

- Greedy-based Search: locates the most similar matches and sequentially revamps repetitions with unmatched ones; likely to fall into local minima solutions.
- Simulated Annealing: attempts to reach a solution close to the global optimum photomosaic through stochastic minimization algorithm that avoids local minima.
- SoftAssign: similar to simulated annealing, but the solution narrows down to the global optimum through a deterministic process that also avoids local minima.

All of these algorithms share a common resource, the *distance matrix*, which will be introduced beforehand.

## 3.1 The Distance Matrix

The purpose of the distance matrix is to track the color similarities (distances) amongst each patch $p_i$ and each tile $t_j$. If we let the rows correspond to the $m = w \times h$ patches and the columns to the $n$ available tiles, then the distance matrix $D^{m \times n}$ can be expressed as:

$$D = d_{ij} = \begin{bmatrix} d_{11} & \cdots & d_{1n} \\ \vdots & \ddots & \vdots \\ d_{m1} & \cdots & d_{mn} \end{bmatrix} \tag{6}$$

where $d_{ij} = d(p_i, t_j)$, according to Equation 1. The distance matrix is also depicted in Figure 2. All of the subsequent algorithms use the distance matrix as an input, and it can be computed in advance. The distance matrix is *immutable*: none of the algorithms ever modify its contents.

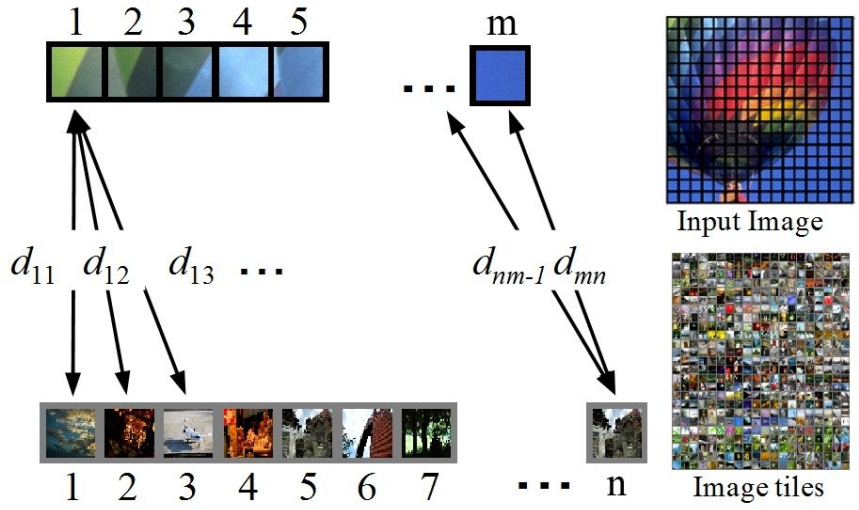


Figure 2: The distance matrix stores the color similarities between patches and tiles.

### 3.2 The Greedy-Based Search

The greedy approach begins by assigning tiles to patches based on the "best-match" criteria. Since repetitions are likely to occur with such criteria, the algorithm iterates once more, sequentially reassigning unused tiles to conflicting patches based once again on the same criteria.

For each patch $p_i$, the corresponding row of the distance matrix $D_i = [\, d_{i1} \ \cdots \ d_{in} \,]$ is examined, searching for the element $d_{iq}$ that holds the lowest distance (most similar). The tile $t_q$ is then assigned to $p_i$ as the best-match.

As repetitions are prone to happen, each tile $t_j$ is further classified as: *assigned once*, *assigned multiple times* and *not assigned*. The rationale is then to keep uniquely assigned tiles unchanged, while replacing multi-assigned tiles by unassigned ones. This can be done by means of the absolute difference between each multi-assigned tile $t_{multi}$ and each unassigned tile $t_{free}$; whichever has the lowest difference wins (Figure 3).

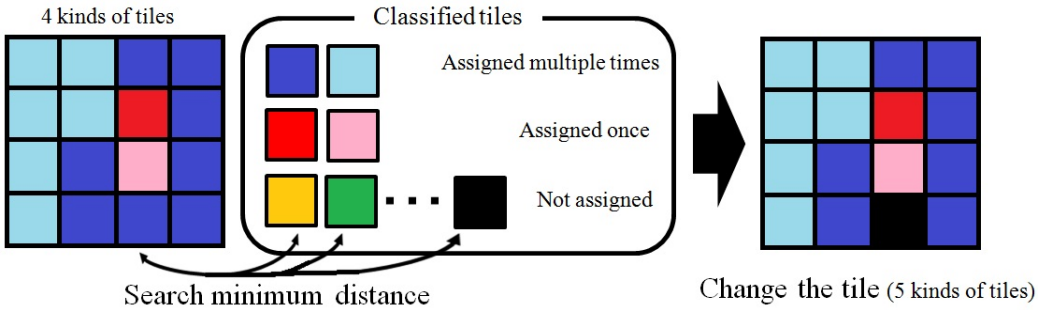


Figure 3: The corresponding patches of a multi-assigned tile are sequentially replaced by unassigned tiles until that multi-assigned tile becomes uniquely assigned. The reassignment process is based on the best-match criteria, i.e., lowest distance (highest visual similarity).

### 3.3 Simulated Annealing

Simulated Annealing (SA) [4] is a generic stochastic technique for optimization problems, identifying potential solutions through random inspection of large search spaces. SA is relatively easy to implement and can account for almost any objective function, with constraints being attached directly into the SA procedure. SA is capable of finding very close approximations to the global optimum if given enough time.

The key component of SA is the dynamic *temperature* parameter. Higher temperatures permit configurations in search space that lead to an increase in the cost function, helping the algorithm to avoid local minima. As the temperature lowers, the solution is progressively enhanced and the search becomes more restricted.

Photomosaic tile matching can be expressed as a SA process through the minimization of the following cost function:

$$E(X, D) = \sum_{i=1}^{m} \sum_{j=1}^{n} (d_{ij} - \alpha)\, x_{ij} \tag{7}$$

where $x_{ij}$ are elements of the *correspondence matrix* $X$ which only takes binary values: if $t_j$ is a possible match to $p_i$ then $x_{ij} = 1$, otherwise $x_{ij} = 0$. The constraints from Equations 3 to 5 must also be respected; therefore every row $X_i = [\ x_{i1}\ \cdots\ x_{in}\ ]$ should hold *exactly* one element $x_{iq} = 1$, while every column of $X$ should have *at most* one element set to 1. The semantics of the correspondence matrix $X$ is thus equivalent to the semantics of the membership function $y(p_i, t_j)$ as expressed through Equations 3 to 5. The parameter $\alpha$ is used to favor a certain range of visual similarities (distances) between patches and tiles.

In order to minimize Equation 7, the correspondence space is stochastically sampled via a Markov process. A new solution $X^*$ is produced at each iteration by modifying only a single row of $X$. This row $X_r = [\ x_{r1}\ \cdots\ x_{rn}\ ]$ is selected randomly. The single element currently set to 1 in $X_r$, $x_{rb} = 1$, is flipped in $X_r^*$ so that $x_{rb}^* = 0$, thus leaving the entire row $X_r^*$ filled with zeros. What remains to be done is to pick some aleatory element $x_{rw} \mid w \neq b$ and flip it in $X_r^*$, yielding $x_{rw}^* = 1$. This last step should be performed carefully to prevent the same tile $t_w$ of being assigned twice in the new solution $X^*$. The index $w$ is repeatedly shuffled if necessary until no such assignment conflict occurs.

For the particular case of *square* correspondence matrices, the procedure described above fails since a conflict-free element $x_{rw} \mid w \neq b$ would never be found. In such special case, another aleatory row $X_{\hat{r}} \mid \hat{r} \neq r$ is selected and swapped with $X_r$, thus making $X_r^* = X_{\hat{r}}$ and $X_{\hat{r}}^* = X_r$.

Once a new solution $X^*$ is established, it can be accepted or rejected according to the following transition probabilities:

$$P(X \to X^*) = \begin{cases} 1 & \text{if } \Delta E \leq 0 \\ e^{-\Delta E/\tau} & \text{otherwise} \end{cases} \tag{8}$$

where $\Delta E = E(X^*, D) - E(X, D)$. The rationale is that state changes are allowed as long as the cost function decreases; to prevent local minima, state may also change with increasing costs based on the current temperature $\tau$. The temperature is gradually lowered during the stochastic search process according to the annealing schedule: the temperature $\tau'$ of the next iteration is obtained by $\tau' = f\ \tau$, with a cooling factor of $0.85 \leq f \leq 0.99$ typically.

At the end of the simulation, each row $X_i = [\ x_{i1}\ \cdots\ x_{in}\ ]$ allegedly accommodates a unique element $x_{iq} = 1$, which allots $t_q$ as the best replacement for $p_i$.

### 3.4 SoftAssign

In computer vision, SoftAssign [3] offers a robust solution to match point clouds, ensuring unique matching criteria while still avoiding being trapped into local minima cusps. SoftAssign derives from deterministic annealing and can be seen as simulated annealing (SA) when applied under the condition of mean field approximation, i.e., it does not rely on any stochastic search.

An optimal photomosaic can be reckoned as the best unique match between tiles and patches. This naturally settles SoftAssign as an alluring solution, if readapted to minimize the following cost function $E$:

$$E(X, D) = \sum_{i,j=1}^{m,n} x_{ij} d_{ij} - \alpha \sum_{i,j=1}^{m,n} x_{ij} + \tau \sum_{i,j=1}^{m,n} x_{ij} \ln(x_{ij}) \tag{9}$$

with $x_{ij}$ being elements of the correspondence matrix $X$. In contrast to SA, here $X$ holds "fuzzy" correspondences, with $0 \leq x_{ij} \leq 1$. The rationale is that each tile $t_j$ is a potential match to each patch $p_i$ by some weight $x_{ij}$. The fuzziness is guided by the last entropy term and the current temperature $\tau$.

The elements of $X$ are initialized randomly with very small quantities. Subsequent iterations modify $X$ according to the following expression:

$$x'_{ij} = e^{-x_{ij}(d_{ij} - \alpha)/\tau} \tag{10}$$

The temperature $\tau$ decreases at each iteration akin to SA. Equation 10 is obtained from Equation 9, optimizing the same objective function of Equation 9 [3]. The key component of this equivalence is the fact that once $X'$ is computed, a row-column normalization through *Sinkhorn iterations* [3, 9] is performed.

Such normalization procedure forces all rows and columns of $X'$ to sum up to 1. Because of such behavior, columns that correspond to irrelevant tiles (unassigned) will also retain some quantities in their elements, a fact that can bias the convergence process. SoftAssign originally attached to the correspondence matrix an additional row and column, referred to as *outliers*, in order to isolate discrepant matches, discarding them once the solution is found. For the case of photomosaic optimization, only the outlier row is necessary, thus mapping unused tiles to an imaginary patch. In contrast, an outlier column would allow patches to be assigned to an imaginary tile which, once discarded, would result in patches not being assigned to any tile.

At the end of the simulation, save for the attached outlier row, every row $X_i = [\ x_{i1}\ \cdots\ x_{in}\ ]$ should contain a unique element $x_{iq} = 1$ while all others set to zero. The tile $t_q$ is then settled as the best candidate to replace $p_i$.

## 4 Initial Results

We based the current results on a hot-air balloon reference photograph (Figure 4, upper left). As for the color similarity function evaluation (Equation 1), each patch and tile was partitioned into 4×4 smaller square-shaped regions. A total of three configurations were studied by varying the number of subdivided patches and the number of available tiles, as enumerated below:

1. $10 \times 10 = 100$ patches selected from 100 tiles; Figure 4-uppermost; first row of Table 1.
2. $10 \times 10 = 100$ patches selected from 500 tiles; Figure 4-center; middle row of Table 1.
3. $20 \times 20 = 400$ patches selected from 1500 tiles; Figure 4-bottom; last row of Table 1.

The Simulated Annealing and SoftAssign parameters were determined empirically. Their corresponding performance results in Table 1 were measured through our MATLAB implementation of the corresponding algorithms. The exception is the greedy-based search, which was implemented in C++, the binary being compiled and linked from the Microsoft Visual C++ 2008. The target hardware is an Intel Core2 Duo 2.5GHz with 2GB RAM running Windows XP 32bit SP3.

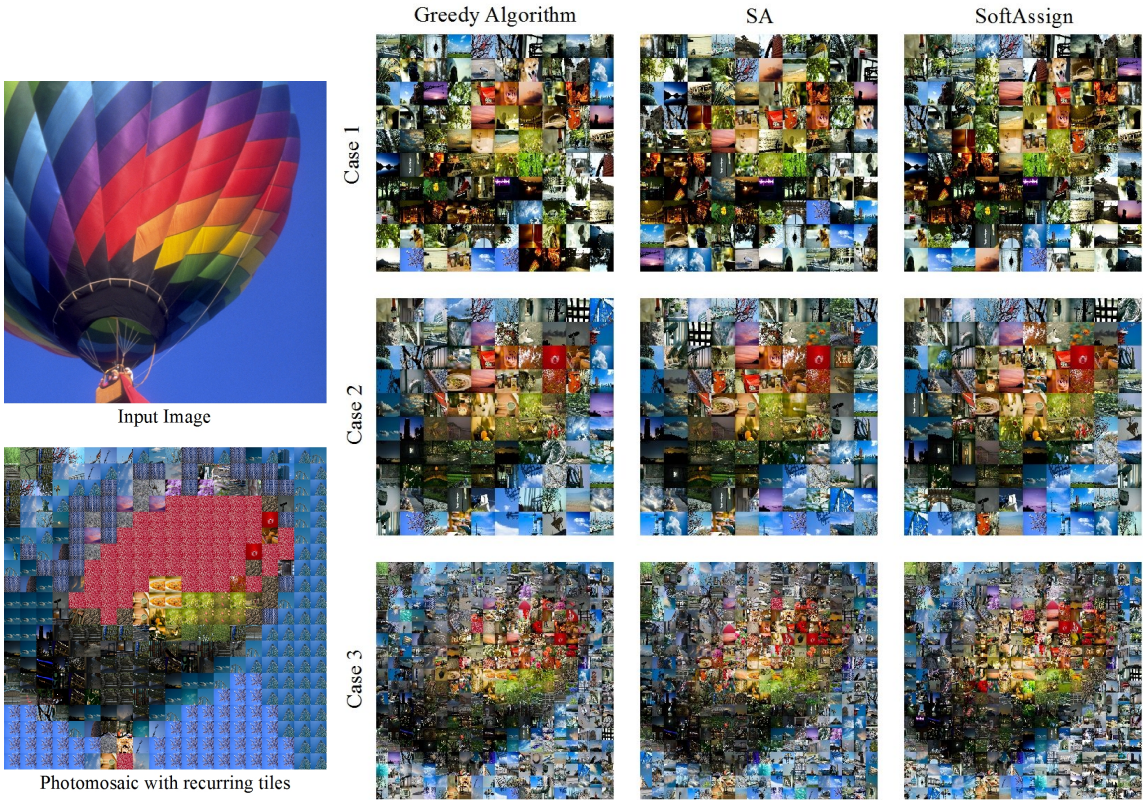


Figure 4: Summary of experimental results: the upper left image is the reference image; the lower left one is a 20x20 photomosaic generated from 1500 tiles (similar to Case 3) through a simple *best-match* (minimal cost) algorithm which allows tile repetition (only 82 tiles were selected; final absolute cost of 79989). Note how such recurring tiles tend to bias the photomosaic, leading to weak aesthetics. The remaining rows corresponds to one of the three studied configurations enumerated in the Experimental Results Section, and each column corresponds to one of the three tile-repetition-free discussed algorithms.

| Patches | Tiles | Measurements | Greedy | Simulated Annealing | SoftAssign |
|---|---|---|---|---|---|
| 100 (10×10) | 100 | abs. cost | 44172 | 43370 | 42713 |
| | | rel. cost | 1.03 | 1.02 | 1.00 |
| | | gen. time | 7s | 13min | 6s |
| 100 (10×10) | 500 | abs. cost | 31997 | 31282 | 31656 |
| | | rel. cost | 1.02 | 1.00 | 1.01 |
| | | gen. time | 6s | 1h32min | 35s |
| 400 (20×20) | 1500 | abs. cost | 128817 | 123350 | 127560 |
| | | rel. cost | 1.04 | 1.00 | 1.03 |
| | | gen. time | 3min | 2 weeks | 8min |

Table 1: Summary of results. The final minimized cost (absolute and relative) of each algorithm in each case is listed, as well as the total time that each algorithm took to find the solution.

In the first two cases, SoftAssign and SA result in more visually appealing photomosaics, rating them as good algorithmic choices when a comparatively small number of tiles is available; although hard to perceptually decide which one feels better, SoftAssign is much faster. As for the third case, SA converges to a qualitatively lower cost than the greedy and SoftAssign approaches, but is excessively time demanding and thus impractical.

The performance and overall quality of the results originated from SoftAssign motivated us to investigate the acceleration of the algorithm with the assistance of the modern programmable graphics hardware (GPU). For that purpose, the following sections will focus on the SoftAssign method, introducing a pseudo-code template, mitigating potential issues, and clarifying decisions made while porting the technique to the GPU.

# 5 SoftAssign Implementation

The blueprint for the SoftAssign technique can be summarized in the following pseudo-code:

Parameters:

```
t0    : scalar, input                          // The initial temperature
tk    : scalar, input                          // The temperature cooling factor
alpha : scalar, input                          // Bias factor for the objective function
D     : matrix, input                          // The distance matrix to be minimized
S     : matrix, output                         // The solution (optimal correspondence)
```

Algorithm:

```
M = matrix(#rows,#cols)
Q = matrix(#rows,#cols)
lowest = +%big
t = t0

for each temperature cooling iteration
  for each temperature stabilization iteration

    Q = M .* (D - alpha)
    M = exp(-Q / t) / sqrt(t)

    outlier_row = array[#cols] of +%tiny       // Refresh outliers
    for each Sinhkhorn iteration               // Sinkhorn method:
      M = M ./ (sum_cols(M) + outlier_row)     //   Normalize columns
      M = M ./ sum_rows(M)                     //   Normalize rows
    end-for

  end-for

  cost = sum_all(Q)                            // Compute the cost (fitting)
  if (cost < lowest)                           // Update solution if necessary
    lowest = cost
    S = M
  end-if

  t *= tk;                                     // Lower the temperature
end-for
```

The terms `#rows` and `#cols` are, respectively, the number of rows and columns of the input distance matrix `D`. The algorithm should initialize `M` with some very small random quantities. The operator `.*` denotes element-wise multiplication. The expression `exp(-Q / t)` does not denote matrix exponentiation, but a simple element-wise exponentiation. The idioms `+%big` and `+%tiny` should evaluate, respectively, to some very big and very small positive constant quantities.

The function `sum_cols(M)` results in a row-vector whose elements correspond to the sum of the respective columns in `M`. Analogously, `sum_rows(M)` yields to a column-vector holding the sum of all elements in the associated rows of `M`. As for `sum_all(Q)`, all elements of `Q` are accumulated, reducing to a single value.

The operator `./` denotes element-wise division, but such division happens in a slightly different fashion. When applied in `M ./ sum_cols(M)`, each row of `M` is divided, element-wise, by the resulting row-vector of `sum_cols(M)`. The semantics is similar for `M ./ sum_rows(M)`, but operating element-wise on columns instead.

The outlier row plays an important role: it accumulates residual weights for tiles that don't seem to fit to any particular patch. Note that the outlier row must be reassigned before triggering the Sinkhorn normalization, each element initialized with the same small value. The outlier row can be attached as an extra row to `M` and ignored during temperature stabilization and row-normalization.

The ideal number of iterations is problem-dependent and usually obtained empirically. The interested reader should refer to the original SoftAssign paper [3] for additional algorithmic details and parameter setup guidelines, as this is out of the scope of this paper.

## 5.1 Addressing Precision Issues

SoftAssign is prone to run into precision issues, pushing values towards infinity as they escape from the representable range of the underlying floating-point scheme. Continuing to operate on such extravagant quantities will eventually cause numerical inconsistencies which will then compromise the entire solution with no turning back.

Having the values of the distance matrix normalized into some small range is preferable. This way the SoftAssign algorithm is unlikely to run into precision issues. However, normalization itself may cause accuracy losses if the fractional part can not be accommodated properly in the underlying representation. When the normalization process is not able to cope with the accuracy required, extra care is necessary in order to prevent precision issues during the algorithm execution. In order to avoid such precision pitfalls in an elegant and efficient way, it is important to understand how and were they can potentially happen.

As the solution converges, individual values of the matrix `M` will approach one. When this happens, the subsequent updates of the associated values in the `Q` matrix will result in progressively larger quantities, based on the magnitude of `D-alpha`. When `M` is then updated based on `Q`, the term `exp(-Q/t)` is likely to evaluate beyond the maximum representable floating-point value, resulting in infinity. The sums from the Sinkhorn Normalization stage are also likely to accumulate to infinity, and the following division would possibly have to deal with $\frac{\infty}{\infty}$, which results in *not-a-number* (`NaN`). At this point, there is no way to remedy the issue and the whole solution is forever spoiled.

The obvious point to address the precision issues is by preventing `exp(-Q/t)` to ever evaluate out of the representable range. In double precision floating-point (64bit) scenario, `exp(-708)` $\approx 3.3 \times 10^{-308}$ is very close to the minimum allowed positive number, that is, $2.2 \times 10^{-308}$. Similarly, `exp(+709)` $\approx 8.2 \times 10^{+307}$ is close to the maximum allowed positive number, $1.8 \times 10^{+308}$. Single precision floating-point (32bit) is much worse: `exp(-87)` $\approx 1.6 \times 10^{-38}$ and `exp(+88)` $\approx 1.6 \times 10^{+38}$ already sit near the representable limits, respectively, $1.1 \times 10^{-38}$ and $3.4 \times 10^{+38}$.

In order to eliminate numerical inconsistencies, the values of `-Q/t` are preset in a safe range before the evaluation of `exp(-Q/t)`. One could simply bind around the numerical limits highlighted previously, but we found that giving an extra margin to the exponent limits also helps to prevent the posterior division by `sqrt(t)` and subsequent array sums to accumulate to infinity. In this paper we clamped `-Q/t` in the range $[-650, +650]$ when computing in double precision, or $[-70, +70]$ with single precision.

### 5.2 Parallelism in SoftAssign

As can be seen from the pseudo-code, SoftAssign is a heavily sequential algorithm: in order to proceed to the next iteration, all nested iterations should finish, and every operation within each loop is tightly bound to the results of the previous one. Thus, an optimized single-threaded CPU-based implementation of SoftAssign is trivial from the pseudo-code. The challenge is then to harness parallelism from such a conceptually sequential procedure.

Unfortunately *macro-parallelism* in SoftAssign is not feasible due to the sequential nature of the algorithm. What can be done is to exploit *micro-parallelism* from each individual operation within the loops. Most of the operations involved are *one-to-one*: they read-from and write-to individual elements of distinct matrices, and this is simple to parallelize. The exception is for matrix/array sums which are *many-to-one*: multiple elements must be gathered from the input in order to compute a single element of the output.

Even though threads are not intended to optimally deal with micro-parallelism, a multi-threaded CPU-based implementation of SoftAssign can provide significant speedup if designed carefully. The basic idea is to keep each thread responsible for a portion of the matrix address space, synchronizing them before continuing to the next operation. Since threads can keep local internal state indefinitely, implementing sums of arrays is simple if access to the required elements happens without races. It is also possible to further exploit micro-parallelism in CPU if a SIMD instruction subset is available such as Streaming SIMD Extensions (SSE).

On the other hand, a GPU-based implementation of SoftAssign is much more challenging. The following Section discusses in detail the design principles and implementation decisions proposed by this paper to map the SoftAssign algorithm to the GPU.

## 6 SoftAssign on GPU

Even though most of the required operations are one-to-one, thus mapping well to the programmable graphics hardware, the gathering process required by the sums of arrays imposes extra effort. Execution contexts in GPU are much more volatile than threads are in CPU, preventing them to easily hold or share state amongst multiple parallel executions.

There are two philosophies to implement SoftAssign in GPU: a) using traditional GPGPU by wrapping GPGPU concepts around the graphics pipeline, which is more portable, efficient and can inter-operate better with further rendering operations if needed; and b) the more recent GPGPU pipeline exposed through technologies such as CUDA, DirectCompute and OpenCL, which provides better synchronization scheme and read/write memory access patterns.

In this paper the traditional GPGPU approach was adopted. Besides performance and portability, the choice for a traditional GPGPU implementation was made towards future use of the framework in interactive and progressive rendering optimization research problems. The interested reader can refer to Tamaki *et al.*'s CUDA-based implementation of SoftAssign [10], although applied to a different problem domain.

## 6.1 Mapping SoftAssign on the Graphics Pipeline

All matrices and arrays required by the SoftAssign algorithm are stored as floating-point textures; a texture being a fundamental, highly optimized structure that can access memory mostly in a two-dimensional fashion. These textures should reside in video memory whether possible in order to prevent the pipeline to stall while waiting for data to be transfered, and also to eliminate any bottleneck in the video bus while the algorithm executes.

Read operations on matrices assume that the corresponding textures are bound to texture targets, each on a separated texture unit. Write operations on a matrix assume that the related texture is bound to the framebuffer. In order to execute a given operation, a quadrilateral is issued to be rendered around the interest region. The graphics pipeline will then rasterize such rectangular region, producing fragments. Each fragment holds an automatically interpolated texture coordinate, which can be seen as a matrix index, uniquely addressing a particular element position.

A shader, that is, a small GPU program, is invoked for each fragment. The GPU schedules and executes the same shader, for each generated fragment, in multiple processing units, all in parallel. The shader code uses the texture coordinate to access elements from the currently bound texture units/targets. Once the intended computation is performed on such elements, the shader outputs the result as a color component. Such color will be placed, by the graphics pipeline, into the appropriate position in the framebuffer, which is allegedly pointing to the destination matrix (texture).

One limitation of the current graphics pipeline is that it is not permitted to have a texture bound for reading and writing simultaneously, since this could yield to race conditions and shading languages do not expose intra-synchronization directives. When such conflicts happen, an auxiliary texture can be employed to hold partial results and feedback them subsequently. Fortunately, all of the SoftAssign operations, save for the ones in the Sinkhorn Normalization stage, have distinct read and write access patterns.

## 6.2 SoftAssign Implementation on GPU

Overall, the following shaders must be implemented:

Temperature Stabilization:

- one to compute `Q = M .* (D-alpha)`
- one to compute `M = exp(-Q/t) / sqrt(t)`

Sinkhorn Normalization:

- one for the vertical parallel reduction: `V = sum_columns(M)`
- one for matrix-row division: `A = M ./ V`
- one for the horizontal parallel reduction: `H = sum_rows(A)`
- one for matrix-column division: `M = A ./ H`

The shaders for the temperature stabilization stage are trivial to implement as they only require one-to-one operations. The parameters `alpha` and `t` are defined as uniform variables within the respective shaders. The value of `alpha` has to be set only once, while `t` has to be uploaded for every temperature cooling iteration, which happens in a very low-frequency pace.

As for the shaders of the Sinkhorn normalization stage, additional considerations are required. First, the outlier row is assumed to be attached as an extra row of `M`; this is not a requirement, but reduces the amount of shaders to write and intermediate textures to manage. Second, note that an auxiliary matrix (texture), `A`, is being used in order to eliminate the simultaneous read-write

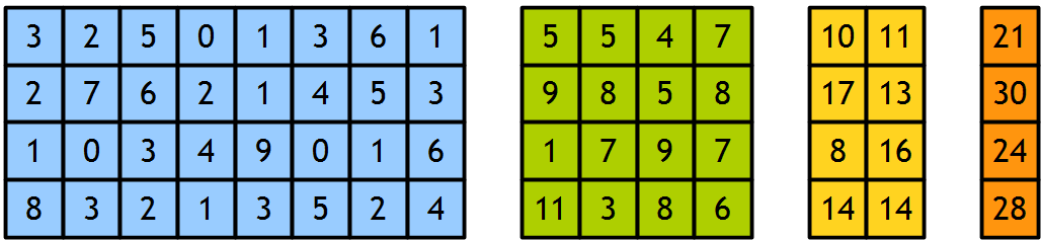


Figure 5: Parallel Reduction instance that performs the total sum of each row of a given table. The result is held in a column-vector whose elements correspond to the sum of all elements of that corresponding row in the original input table. At each step, two consecutive elements are gathered and accumulated together from the partial results of the previous step.

conflict that would otherwise happen in `M`. Third and more important, since both `sum_columns(M)` and `sum_rows(A)` require gathering several elements of `M` and `A`, respectively, the shader must be able to keep track of the partial sums until the total amount is computed.

Although one could simply accumulate all values in a single step, this would significantly compromise the texture cache performance, thus slowing down the entire process. To complicate the matters even more, shading languages do not provide any synchronization directives, and the order of scheduling and execution of the fragments are unpredictable.

The solution is then to employ *Parallel Reduction*, a well known *multi-pass parallel gather pattern*, keeping the state stored in intermediate sets of data that feedback each other at every subsequent pass, as depicted in Figure 5. Such parallel reduction algorithms are typically stream-based and cache-coherent, and thus GPU-friendly.

In the example of Figure 5, only two elements are gathered and accumulated at each step, but operating on more elements can improve performance and lower memory requirements. The ideal number of gathers per pass is hardware-dependent. Small values may sub-utilize the number of available texture fetch units, while big values may stress them, thus penalizing texture-cache performance. Hence the ideal choice is rather empirical and must be refined manually for each particular hardware.

When allocating space for intermediate tables, if the dimensions of the source table happen not to be multiples of the number of gathering samples, they should be rounded-up. Additionally, any attempt to gather elements outside of the boundaries of a table should yield zero value in order to keep the accumulation semantics sound.

In order to compute `sum_all(Q)`, no additional shader effort is required. Such sum can be computed in two stages, first by a vertical reduction, resulting in a single row-vector, and then by an horizontal reduction on such row-vector, resulting in a single value that corresponds to the entire sum of `Q` (the other way around would yield the same result). Moreover, note that the intermediate memory of the parallel reductions used by the Sinkhorn Normalization stage can be shared amongst this reduction as well, as they happen independently.

Another strategy to compute `sum_all(Q)` would be to perform a *2D Parallel Reduction*, a process that resembles texture mip-map generation. This would only increase performance marginally since such a sum is computed at a very low frequency, once per temperature cooling iteration. Besides the effort of implementing and keeping an additional shader, this would also require dedicated additional memory.

Finally, reading-back from GPU is required when comparing the cost of each iteration. Fortunately, such read-back is very small (only one texel). Furthermore, updating the output matrix `S` from the values of `M` does not require a shader, just a simple texel copy. The same can be said for the reassignment of the outlier row before starting each of the Sinkhorn normalization process: a template of the initial outlier row is kept in video memory in a buffer and simply copied over the additional row of `M` when required.

### 6.3 Implementation Details

Our current implementation is OpenGL 1.4 / GLSL 1.0 compliant, requiring few but widely supported OpenGL extensions: framebuffer object, rectangle textures, floating-point textures and shader objects. This makes the algorithm portable to a wide range of GPUs, the lower bound being the commodity GeForce FX series (now 9 generations old).

All textures are stored in 32bit floating-point format (`GL_LUMINANCE32F` or equivalent). Double precision floating-point texture formats are not yet mainstream, and even when available, the hardware may not achieve full performance because not all arithmetic units in commodity graphics GPUs can operate on double precision quantities. This causes the shaders to stall as they race for these units. Half precision floating-point textures, on the other hand, are widely supported and optimized by the graphics hardware and could potentially double the overall performance, but we found that they do not suffice for stable executions of the SoftAssign algorithm.

Textures are bound for read in rectangular texture targets (`GL_TEXTURE_RECTANGLE` or similar). This is not an enforcement, just a convenience to ease debugging the implementation. Regular 2D texture targets (`GL_TEXTURE_2D`) could be used instead as well.

The OpenGL Shading Language has a built-in function to restrain values within a range called `clamp(val,min,max)`. This is useful to workaround the precision issues when updating `M` as it is hardware accelerated and more efficient than placing a manual conditional logic.

In order to ensure the proper semantics when sampling outside of a texture boundary during the parallel reductions, no special shader control, such as conditionals or extra uniform variables, is required. By simply setting the texture access wrap mode to `GL_CLAMP_TO_BORDER_COLOR`, and specifying `RGBA=(0,0,0,0)` as the border color, is enough to keep the semantics sound.

For parallel reductions, texture filtering can be used to fetch two texels at the same time by sampling at the exact boundary of the corresponding texels, just being careful to multiply the filtered value by two afterwards. Note that some old graphics hardware may not support single precision floating-point texture filtering (GeForce FX Series), even though they may support it for half precision (GeForce 6 Series).

Finally, keep in mind that the hardware has limitations regarding the maximum dimensions for textures. It is possible to split and stitch bigger matrices into smaller textures in order to accommodate all the data, if the limit lies below the required one. Additional control is then required to manage such texture chunks. Also keep in mind that video memory (VRAM) is more scarce than regular RAM. Although most graphics drivers are capable of virtualizing video memory, it is not necessary to do so. Besides, such virtualization is prone to drastically impact the performance.

## 7 Extended Results

The performance comparison between the GPU and CPU implementations is summarized in Table 2 and Figure 6. Regarding the CPU implementation, for a more fair comparison, we decided to move

away from the MATLAB environment and write an optimized multi-threaded implementation in C++ with the Win32 Threads API; the binary was compiled with the Microsoft C Compiler under the Visual C++ 2010 Professional development environment. The GPU implementation is OpenGL 1.4 / GLSL 1.0 compliant.

The hardware configuration used for the performance measurements is an Intel Core2 Quad CPU 2.55GHz with 4GB RAM running Windows 7 Enterprise 32bit, equipped with a GeForce GTX 280 with 1GB VRAM (240 stream processors). All performance results refer to a single temperature cooling iteration, comprised of 10 temperature stabilization iterations, each with 10 Sinkhorn normalization iterations.

| rows | 100 | 100 | 256 | 400 | 512 | 900 | 1024 | 1200 | 1600 | 1800 | 2048 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| columns | 100 | 500 | 1024 | 1500 | 2048 | 3000 | 4096 | 5000 | 6000 | 7000 | 8192 |
| CPUx1 | 0.024 | 0.114 | 0.617 | 1.506 | 2.572 | 6.485 | 10.01 | 14.29 | 22.73 | 29.82 | 39.77 |
| CPUx4 | 0.022 | 0.062 | 0.358 | 0.780 | 1.479 | 3.912 | 5.963 | 8.532 | 13.85 | 17.78 | 23.62 |
| GPU | 0.022 | 0.023 | 0.024 | 0.035 | 0.052 | 0.114 | 0.171 | 0.239 | 0.374 | 0.487 | 0.643 |

Table 2: Performance results for the CPU and GPU implementations of SoftAssign. CPUx1 stands for single-threaded execution, while CPUx4 represents a multi-threaded execution context with 4 threads. All time measurements are expressed in seconds.

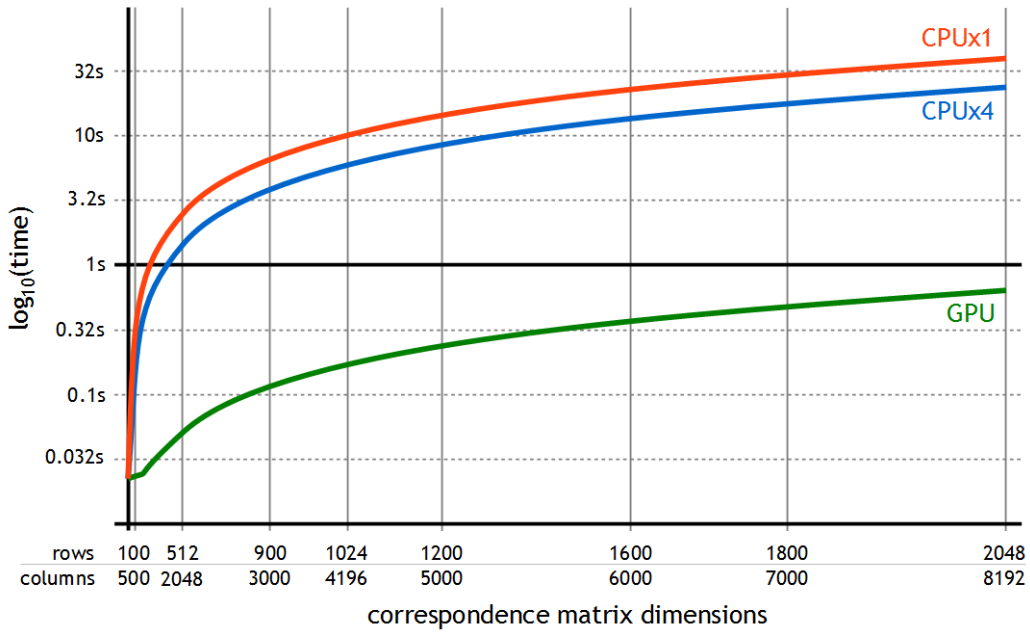


Figure 6: Performance chart comparing the CPU and GPU implementations of SoftAssign. For clarity, the chart was plotted with time being expressed in a base-10 logarithmic scale. The samples in the horizontal axis are spaced linearly according to the total number of elements in the distance matrix, that is, $rows \times columns$. Note that the GPU performance is far better than the CPU: the 1s barrier is never reached by the GPU while in the CPU cases this barrier is crossed at very small distance matrix dimensions.

Even though the measurements are bound to a specific iteration profile, the performance gracefully scales linearly on the number of iterations. Hence, raising the number of temperature stabilization iterations by a factor of $p$ and the Sinkhorn normalization iterations by a factor of $q$ would

yield to a relative increase of $p \times q$ times in the computation time, either in CPU or GPU.

The choice of using four threads is due to the fact that the target CPU, Intel Core2 Quad, is a quad-core microprocessor. Note that the x4 multi-threaded execution is unable to deliver the theoretical 4 times boost in performance when compared to a single-threaded execution. In fact, the x4 multi-threaded performance is very close to a x2 multi-threaded execution. We believe that there are two main reasons for which the implementation is unable to reach peak performance: the first is due the overhead of thread synchronization directives that are essentially managed by the operating system kernel (the thread model was not designed to effective harness micro-parallelism); and the second is the fact that such quad-core processors are arranged in two dies, each holding a dual-core unit that share a common cache.

The GPU implementation, on the other hand, outperforms the 4x multi-threaded CPU implementation by a factor of 35. Compared to a single-thread CPU execution context, the speedup is about 60 times. The only exception is for very small distance matrices: in such cases, the driver overhead congests the actual amount of processing required by the algorithm, and the performance gain is not as significant. However, such small cases are already fast enough to compute anyway and, in practice, neither relevant nor useful at all.

The proposed OpenGL/GLSL-based implementation outperforms Tamaki *et al.*'s CUDA-based SoftAssign [10] by a factor of two in a setup equipped with an identical graphics hardware configuration, a GeForce 8800 GT with 512MB VRAM (112 stream processors). While Tamaki et al. approach takes about 30s to solve a 3000x3000 distance matrix, the proposed approach takes less than 16s.

Finally, a few more examples of photomosaics optimized through the proposed GPU-based implementation of SoftAssign is presented in Figures 7 to 10. They comprise thematic situations where a given input image is transformed into a photomosaic by using image tiles that have similar semantic context to the one of the original input image. All mosaics were subdivided into 900 patches (30x30) assigned from tile sets composed of 8192 images, thus leading to a 900x8192 distance matrix. The total SoftAssign computing time (for 200 cooling iterations) was less than 1 minute in the GPU execution, while the single-threaded CPU run took about 1 hour and the four-threaded took around 35 minutes.

# 8 Conclusion

Novel strategies to maximize the usage of images for photomosaic synthesis based on a greedy procedural algorithm, simulated annealing and SoftAssign were presented. The maximization is ensured by restricting tiles to be assigned only once to any given patch.

The experimental results show that SoftAssign and SA are effective when the number of available tiles is comparatively small. As more tiles become accessible, SA still remains as the most effective choice, but turns to be impractical due to time constraints; the greedy approach still produces plausible mosaics in such cases.

SoftAssign, on the other hand, not only provides an elegant, deterministic solution for the problem, but also requires much less processing time. The algorithm can be implemented in GPU, providing performance improvements higher than 60 times over optimized CPU implementations.

Such performance improvement is welcome not only to optimize the solution, but also to assist the user in identifying the ideal parameters of the SoftAssign algorithm. There are no principal guidelines on how to tune SoftAssign, thus the search for an optimal parameter set can be very tedious if the user has to wait significant amounts of time between each test setup. A GPU-based

implementation of SoftAssign delivers a more immediate feedback to the user.

# 9 Future work

As future work, further investigation is required to determine the impact of more perceptually-aware color similarity metrics to derive the distance matrix and drive the optimization process. The use of different optimization strategies, specially those based on *evolutionary algorithms*, is also a potential and promising target for subsequent analysis.

# Acknowledgements

We would like to thank the open source initiatives of **DevIL**[†] (*Developer's Image Library*, former OpenIL), **GLEW**[‡] (*The OpenGL Extension Wrangler Library*) and **boost**[§] for their generosity and contributions to the open and free software community that accelerated the development of this research project.

# References

[1] S. Battiato, G. Di Blasi, G. M. Farinella, and G. Gallo. Digital mosaic frameworks - an overview. *Eurographics - Computer Graphic Forum*, 26(4):794–812, 2007.

[2] Gianpiero Di Blasi and Maria Petralia. Fast photomosaic. In *In poster proceedings of ACM/WSCG2005, 2005*, 2005.

[3] Steven Gold, Anand Rangarajan, Chien ping Lu, and Eric Mjolsness. New algorithms for 2D and 3D point matching: Pose estimation and correspondence. *Pattern Recognition*, 31:957–964, 1997.

[4] S. Kirkpatrick, C. D. Gelatt., and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[5] Allison W. Klein, Tyler Grant, Adam Finkelstein, and Michael F. Cohen. Video mosaics. In *NPAR 2002: Second International Symposium on Non Photorealistic Rendering*, pages 21–28, June 2002.

[6] Barbara J. Meier. Painterly rendering for animation. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 477–484, New York, NY, USA, 1996. ACM.

[7] Michihiro Mikamo, Marcos Slomp, Shungo Yanase, Bisser Raytchev, Toru Tamaki, and Kazufumi Kaneda. Maximizing image utilization in photomosaics. *International Conference on Natural Computation*, 0:275–278, 2010.

[8] Robert Silvers. *Photomosaics*. Henry Holt and Co., Inc., New York, NY, USA, 1997.

[9] Richard Sinkhorn. A relationship between arbitrary positive matrices and doubly stochastic matrices. *The Annals of Mathematical Statistics*, 35:876–879, 1964.

[10] Toru Tamaki, Miho Abe, Bisser Raytchev, and Kazufumi Kaneda. SoftAssign and EM-ICP on GPU. *International Conference on Natural Computation*, 0:179–183, 2010.

[†] `http://openil.sourceforge.net`
[‡] `http://glew.sourceforge.net`
[§] `http://www.boost.org`

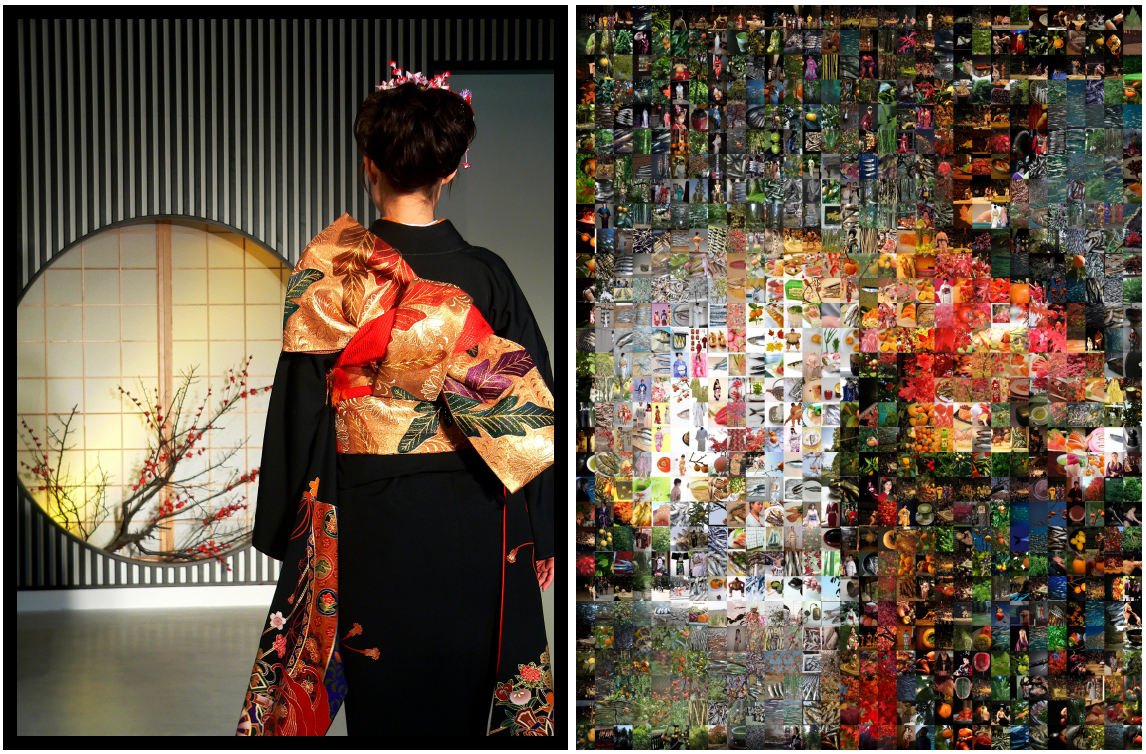

Figure 7: A kimono photomosaic made of 30x30 patches selected from a thematic Japanese tile set of 8192 images. The original input image is shown in the left.
Image and tile set courtesy of `http://www.image-net.org`

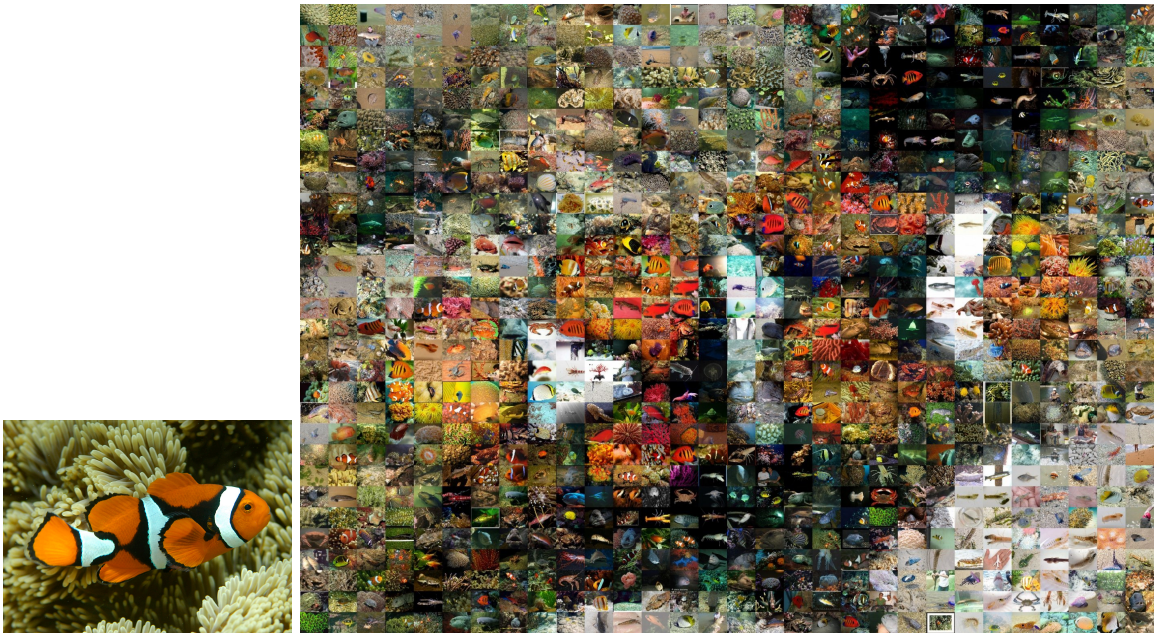

Figure 8: A fish image photomosaic made of 30x30 patches selected from a thematic fish tile set of 8192 images. The original input image is shown miniaturized in the left.
Image and tile set courtesy of `http://www.image-net.org`

Figure 9: A chair image photomosaic made of 30x30 patches selected from a thematic chair tile set of 8192 images. The original input image is shown miniaturized in the left.
Image and tile set courtesy of `http://www.image-net.org`

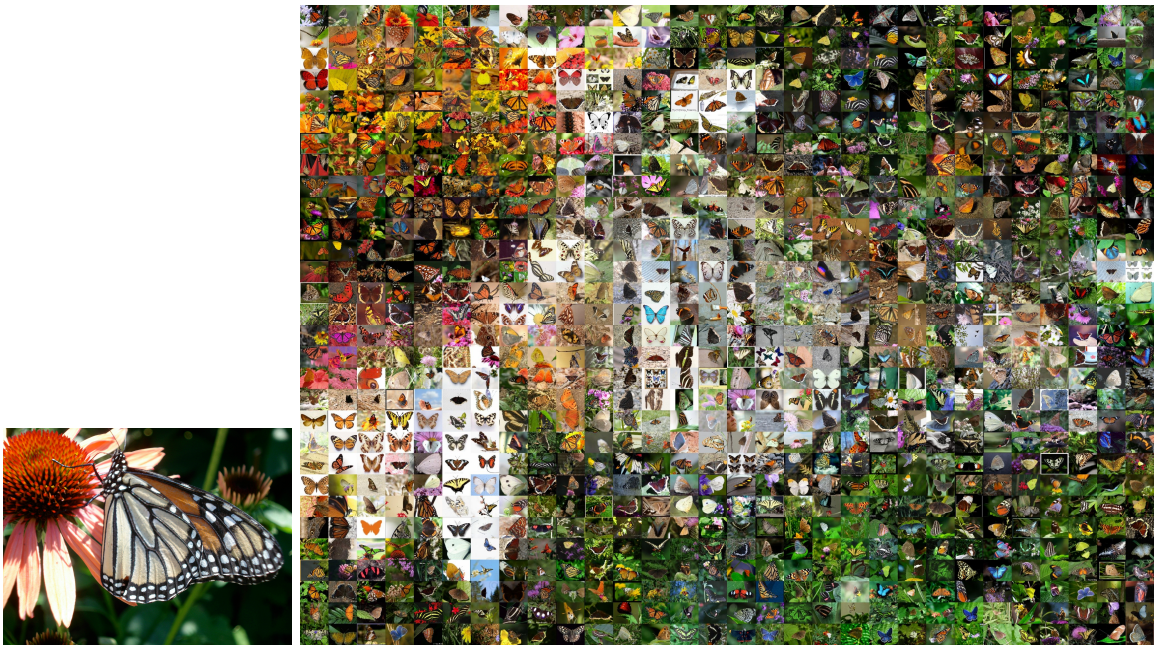

Figure 10: A butterfly image photomosaic made of 30x30 patches selected from a thematic butterfly tile set of 4096 images. The original input image is shown miniaturized in the left.
Image and tile set courtesy of `http://www.image-net.org`