Efficient Acceleration of Mutual Information Computation for Nonrigid Registration using CUDA

Kei Ikeda, Fumihiko Ino, and Kenichi Hagihara

Abstract—**In this paper, we propose an efficient acceleration method for the nonrigid registration of multimodal images that uses a graphics processing unit (GPU). The key contribution of our method is efficient utilization of on-chip memory for both normalized mutual information (NMI) computation and hierarchical B-spline deformation, which compose a well-known registration algorithm. We implement this registration algorithm as a compute unified device architecture (CUDA) program with an efficient parallel scheme and several optimization techniques such as hierarchical data organization, data reuse, and multiresolution representation. We experimentally evaluate our method with four clinical datasets consisting of up to $512 \times 512 \times 296$ voxels. We find that exploitation of on-chip memory achieves a 12-fold increase in speed over an off-chip memory version and, therefore, it increases the efficiency of parallel execution from 4% to 46%. We also find that our method running on a GeForce GTX 580 card is approximately 14 times faster than a fully optimized CPU-based implementation running on four cores. Some multimodal registration results are also provided to understand the limitation of our method. We believe that our highly efficient method, which completes an alignment task within a few tens of second, will be useful to realize rapid nonrigid registration.**

Index Terms—**GPU, CUDA, nonrigid registration, mutual information, acceleration.**

I. Introduction

IMAGE registration [1] is a technique for defining a geometric relationship between each point in two images: a reference image and a floating image. This technique plays an important role in computer-assisted surgery. For example, it assists medical doctors by relating preoperative images with intraoperative images [2], [3] and by integrating multiple imaging modalities such as computed tomography (CT), magnetic resonance imaging (MRI), and positron emission tomography (PET) [4]. Such fused images are useful to automate tumor detection and segmentation for image-guided surgery [5], [6], [7].

Typically, registration algorithms consist of three key components: a deformation model for objects, a cost function associated with a similarity measure of reference and floating images, and an optimization scheme. The alignment procedure for nonrigid objects, where rigid or affine transactions are not sufficient [1], [8], is called nonrigid registration. This procedure requires extensive computation because it deals with deformable objects, that compared to rigid objects, require free-form deformation with many degrees of freedom. Thus, minimizing execution time remains a challenging issue to make nonrigid registration feasible for limited-time situations.

The graphics processing unit (GPU) [9], which has recently emerged as a small but powerful energy-efficient accelerator, is promising hardware to meet this challenge. With the release of the compute unified device architecture (CUDA) [10], a C-like programming framework, the GPU now can accelerate general-purpose as well as graphics applications [11], [12], [13]. To exploit the data parallelism inherent in the target application, the GPU adopts a highly-threaded architecture that provides several hundreds of computing cores with off-chip memory bandwidth above 190 GB/s. In addition to these rich resources, the GPU has shared on-chip memory that can be used as a manually managed cache. Though the capacity of the shared memory is in the order of KB, it can greatly improve application performance by data reuse because it provides two orders of magnitude less latency than off-chip memory.

Using CUDA, many researchers have accelerated nonrigid registration algorithms for multimodal images. Plishker et al. [14], [15] accelerated a registration algorithm that uses B-splines [16] as a deformation model, *normalized mutual information* (NMI) as a cost function [17], and gradient descent as an optimization scheme. For $256 \times 256 \times 256$-voxel images, the execution time was reduced to 250 s on a GeForce GTX 285 card. Similar registration algorithms were independently accelerated by Saxena et al. [18] and Modat et al. [19]. Saxena

K. Ikeda, F. Ino, and K. Hagihara are with the Graduate School of Information Science and Technology, Osaka University, 1-5 Yamadaoka, Suita, Osaka 565-0871, Japan. E-mail: {i-kei, ino, hagihara}@ist.osaka-u.ac.jp

Manuscript received December 25, 2012; revised August 17, 2013; revised November 22, 2013.

et al. used a Parzen window approach to estimate NMI values using shared memory. However, this Parzen-based approach usually introduces volatility on account of its heavy sampling sensitivity [20], and the approximation cost can limit the performance [21].

In this paper, we propose a CUDA-based highly efficient method for accelerating nonrigid registration of multimodal images. Our method is based on Rueckert's registration algorithm [8], which consists of hierarchical B-splines, NMI, and gradient descent. Each of these components is commonly used in many registration algorithms [22]. To the best of our knowledge, our method is the first that accelerates nonrigid registration by using fast shared memory without approximating NMI values. The key ideas for enabling this optimization are to exploit the sparsity of *joint histograms* needed for NMI computation and organize joint histograms into a hierarchy. These ideas reduce data size so that a joint histogram can be stored in small capacity shared memory. An efficient merge mechanism, which minimizes the amount of off-chip memory access, is integrated into our method to maximize the benefit of shared resources. Furthermore, our method achieves rapid B-spline deformation by using shared memory with a data reuse technique [2] that reduces the amount of computation. Source code is available at http://www-hagi.ist.osaka-u.ac.jp/research/code/.

The remainder of the paper is organized as follows. Section II introduces related work on acceleration of image registration. Section III presents an overview of the registration algorithm to be accelerated with our parallel method and the key concepts of CUDA needed for achieving high efficiency. Section IV then describes our parallel method. Section V shows several experimental results. Conclusions are presented in Section VI.

II. Related Work

Several groups [15], [18], [19], [23] have accelerated nonrigid registration of multimodal images using CUDA. However, previous methods [15], [23] suffered from numerous accesses to off-chip memory because the capacity of shared memory is not sufficiently large to store a joint histogram. To avoid this, others [18], [19] have reduced the memory usage with approximation. However, this approach raised the issue of sampling sensitivity. In contrast, our method not only avoids approximation but also reduces the amount of expensive off-chip memory access by utilizing shared memory. We also describe in detail how B-spline deformation can be accelerated with optimization techniques.

Mutual information (MI) based similarities can be applied to both rigid and nonrigid registration. Vetter et al. [24] presented a sorting-based algorithm that computes MI values for rigid registration of $128 \times 128 \times 128$-voxel images. Their algorithm reduces the frequency of off-chip memory access by sorting reference voxels in terms of intensity values. After this pre-processing stage, sorted voxels are partitioned into blocks, which are then copied to shared memory to enable rapid counting of the number of intensities in parallel. However, sorting-based methods require additional memory space to store each voxel's position information. Although this amount of memory consumption can be disregarded for small datasets, it is critical for large datasets because graphics cards have a smaller memory capacity than CPUs. For instance, our experimental card has 1.5 GB of off-chip memory, whereas a pair of $512 \times 512 \times 512$-voxel volume datasets requires 2 GB of off-chip memory if each voxel is associated with 4 bytes of an intensity value and 4 bytes of a three-dimensional (3-D) coordinate. Similar sorting-based algorithms were presented by Chen et al. [20], Lou et al. [25], and Shams et al. [21] who aligned $230 \times 230 \times 239$-voxel, $256 \times 256 \times 68$-voxel, and $512 \times 512 \times 29$-voxel datasets, respectively.

Shams et al. [26] also presented alternative methods for MI computation. Their methods allow threads to have an own joint histogram to avoid the performance penalty of *atomic writes* [10], which are needed to serialize simultaneous accesses to the same memory address (i.e., the same bin in a joint histogram). Owing to local joint histograms, parallel threads are allowed to count the number of intensities in their responsible region independently. After this parallel count, local joint histograms have to be merged into a single joint histogram, which is required to compute MI values. This merge operation can be parallelized by applying a tree-based reduction to local joint histograms. Although their reduction-based methods achieved more successful acceleration than a CPU-based method, the performance can be further increased by shared memory. A similar reduction-based method is presented by Cheng et al. [27].

Several groups [28], [29], [30] employed a GPU to accelerate nonrigid registration of single-modal images. Compared to multimodal similarity measures, single-modal measures, such as the correlation coefficient and the sum of squared differences, are less compute-intensive [22] and have a simple structure that can easily be parallelized on a GPU.

Earlier projects accelerated nonrigid registration using high-performance computing systems such as shared-memory multiprocessors [2], [31], clusters of PCs [32], field programmable gate arrays [33], [34], and the Cell Broadband Engine [35]. These parallel algorithms cannot be directly implemented on a GPU, which has a unique memory hierarchy and processor architecture. However, previous optimization techniques, such as data reuse [2] and a multiresolution representation [16], can be adapted to CUDA-based implementations. This will be discussed in more detail in Section IV.

III. Preliminary Considerations

Let R and F be the reference image and the floating image, respectively, with the image domain $\Omega = \{(x, y, z) \mid 0 \leq x < X, 0 \leq y < Y, 0 \leq z < Z\}$, where X, Y, and Z are image sizes in the x, y, and z directions, respectively. Let $T : (x, y, z) \mapsto (x', y', z')$ be a transformation of any voxel (x, y, z) in image F to its corresponding voxel (x', y', z') in image R. Rueckert's registration algorithm [8] then performs an alignment by finding the best nonrigid transformation T_{OPT} that minimizes a cost function $\mathcal{C}$:

$$T_{OPT} = \arg\min_{T} \mathcal{C}(R, T(F)). \tag{1}$$

Here, the cost function $\mathcal{C}$ is associated with a similarity measure $\mathcal{S}$ defined between images R and F:

$$\mathcal{C} = -\mathcal{S}(R, T(F)). \tag{2}$$

A. Normalized Mutual Information

NMI represents the amount of information that one image contains about a second image. According to its definition, our similarity measure $\mathcal{S}$ is given by

$$\mathcal{S}(R, F) = \frac{H(R) + H(F)}{H(R, F)}, \tag{3}$$

where $H(R)$ represents the entropy of image R, and $H(R, F)$ represents the joint entropy of images R and F. The entropy $H(R)$ and the joint entropy $H(R, F)$ are given by

$$H(R) = -\sum_{r \in R} p_R(r) \log p_R(r), \tag{4}$$

$$H(R, F) = -\sum_{r \in R, f \in F} p_{RF}(r, f) \log p_{RF}(r, f), \tag{5}$$

respectively, where $p_R(r)$ is the marginal distribution of R and $p_{RF}(r, f)$ is the joint probability distribution of R and F. The former can be obtained from the histogram of R whereas the latter can be obtained from the joint histogram of R and F.

The joint histogram of two images is a 2-D matrix containing the number of pairs of intensity values at the same position $(x, y, z) \in \Omega$ in the two images. Suppose that reference and floating images consist of d grayscale levels. Their joint histogram then contains d^2 bins, and thus joint histogram computation can be regarded as a mapping problem from XYZ-voxel space onto d^2-bin space. The memory access pattern inherent in this computation is the most important issue to maximize performance: (1) irregular access to bins and (2) serialized access to the same bin.

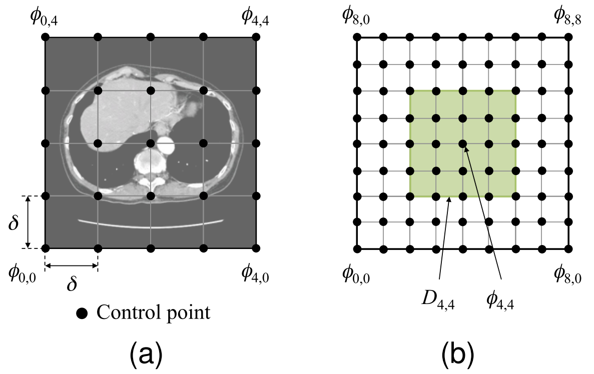


Fig. 1. A 2-D example of the B-spline deformation model: (a) mesh of control points with uniform spacing δ placed over the image domain and (b) $4\delta \times 4\delta$ neighborhood domain $D_{i,j}$ affected by control point $\phi_{i,j}$.

B. B-spline Deformation

As shown in Fig. 1(a), a B-spline free-form deformation represents a nonrigid transformation by manipulating a mesh of control points overlaid on the image domain Ω. Let Φ be a 3-D mesh of control points $\phi_{i,j,k} \in \Phi$, and let δ be the initial distance between the control points. A nonrigid transformation T of any voxel $(x, y, z) \in \Omega$ is then calculated by its surrounding $4 \times 4 \times 4$ neighborhood of control points as follows:

$$T(x, y, z) = \sum_{l=0}^{3} B_l(u) \hat{\phi}_{i+l}, \tag{6}$$

$$\hat{\phi}_{i+l} = \sum_{m=0}^{3} \sum_{n=0}^{3} B_m(v) B_n(w) \phi_{i+l, j+m, k+n}, \tag{7}$$

where B_l $(0 \leq l \leq 3)$ represents the l-th basis function of cubic B-splines [16], $i = \lfloor x/\delta \rfloor - 1$, $j = \lfloor y/\delta \rfloor - 1$, $k = \lfloor z/\delta \rfloor - 1$, $u = x/\delta - \lfloor x/\delta \rfloor$, $v = y/\delta - \lfloor y/\delta \rfloor$, and $w = z/\delta - \lfloor z/\delta \rfloor$. In other words, B-spline deformations are locally controlled because each control point $\phi_{i,j,k}$ affects only its $4\delta \times 4\delta \times 4\delta$ neighborhood subdomain $D_{i,j,k}$, as shown in Fig. 1(b).

Note that the value of $\hat{\phi}_{i+l}$ is identical for all voxels in one row located within the same cell of the mesh Φ [2]. Because such voxels have the same coordinates y and z, they are transformed according to the same indexes j and k (i.e., the same control points) and the same relative positions v and w within the cell (i.e., the same coefficients). Consequently, Rohlfing et al. [2] reduced the amount of computation by reusing Eq. (7) between voxels.

C. Steepest Descent Optimization

Rueckert's algorithm [8] employs steepest descent optimization to find the optimal transformation parameters Φ that minimize the cost function $\mathcal{C}$. To estimate the gradient vector $\nabla\mathcal{C} = \partial\mathcal{C}/\partial\Phi$ with respect to the transformation parameters Φ, the algorithm computes a local gradient $\partial\mathcal{C}/\partial\phi_{i,j,k}$ for each control point $\phi_{i,j,k} \in \Phi$ by using the finite-difference approximation.

Because the deformation of $\phi_{i,j,k}$ affects only its neighborhood domain $D_{i,j,k}$, a precomputation technique [36] is useful to accelerate this gradient computation. That is, a joint histogram of unaffected region $\Omega - D_{i,j,k}$ is computed in advance and is then compounded with a local joint histogram of the affected region $D_{i,j,k}$ for each control point displacement. This precomputation technique reduces the computational requirement to 1/6 because joint histograms are computed for 6 displacements $(\pm x, \pm y, \pm z)$ per control point.

The optimization procedure mentioned above is accelerated with a multiresolution representation that organizes both the images and the control point mesh in a hierarchy. The image resolution γ and the control point spacing δ are then progressively refined at each level of the hierarchy.

D. Compute Unified Device Architecture (CUDA)

In general, CUDA programs [10] consist of host code and device code, which run on a CPU and a GPU, respectively. The host code typically invokes the device code on the GPU to accelerate the time consuming part of the application. The device code can be implemented as a function called *kernel*. The GPU executes a kernel with tens of thousands of CUDA threads to achieve acceleration by exploiting the data parallelism in the application.

These threads compose a series of *thread blocks* to adapt their organization to the hierarchical processor architecture [9] deployed for the GPU. A thread block is then partitioned into a series of *warps*. A warp contains 32 threads, which are executed in a single-instruction, multiple-thread (SIMT) manner [10]. On account of this SIMT execution, a branch within a warp can result in a *thread divergence*, which significantly lowers the efficiency of parallel execution.

As shown in Fig. 2, threads belonging to the same thread block are allowed to share small capacity but fast memory. In the current architecture, the maximum size of shared memory a thread block can allocate is 48 KB. By contrast, any thread can access large capacity but slow off-chip memory. Off-chip memory can be allocated as a texture, which provides hardware accelerated interpolation of texel values. This ability is useful to accelerate deformation of the floating image on a GPU.

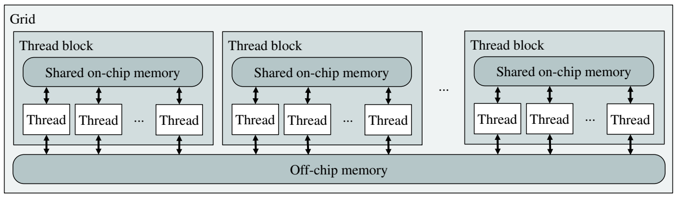


Fig. 2. An overview of CUDA. Shared on-chip memory allows threads in the same thread block to reuse data in order to save off-chip memory bandwidth for memory-bound applications.

CUDA provides a synchronization mechanism for threads of the same thread block, but not for those of different thread blocks. One special exception is the family of atomic operations, which is useful to count the number of intensities for histogram computation. Atomic operations are available to both on-chip and off-chip memories; however, they can cause thread serialization. The only way to achieve global synchronization is to finish and restart the running kernel. However, this increases the amount of off-chip memory access because register files and shared memory are cleared at the end of kernel execution. From this point of view, a series of kernel invocations should be unified into a single invocation if the kernel can be implemented without global synchronization.

In summary, the important GPU concepts that are strongly related to our registration algorithm are fourfold.

- C1: saving off-chip memory accesses by data reuse on shared memory.
- C2: reducing the amount of off-chip memory access by kernel unification.
- C3: maximizing GPU resource utilization by texture-based interpolation.
- C4: maximizing the efficiency of SIMT execution by avoiding thread divergence.

IV. Proposed Parallel Method

Algorithm 1 shows the pseudocode for our parallel method, which produces optimized control points Φ for two images. Similar to Rueckert's algorithm, our parallel method uses multiresolution representation to accelerate the optimization procedure: image resolutions

Algorithm 1 GPU-accelerated nonrigid registration based on Rueckert's registration algorithm [8].

```
Input: A pair $\langle R, F\rangle$ of reference and floating images, image resolutions $\gamma_1, \gamma_2, \ldots, \gamma_L$ and control point spacing $\delta_1, \delta_2, \ldots, \delta_L$
    of $L$ levels, and a threshold $\epsilon$ for minimization.
Output: Optimized control points $\Phi$.
 1: $h \leftarrow 1$                                                    ▷ Initialize the deformation level
 2: while $h \leq L$ do
 3:     Transfer images $R$ and $F$ of resolution $\gamma_h$ from the CPU to the GPU
 4:     Initialize the control points $\Phi$ with $\delta_h$ on the GPU
 5:     repeat                                                          ▷ Optimization step
 6:         Compute the cost function $C(\Phi)$ on the GPU
 7:         Compute the gradient vector $\nabla\mathcal{C} = \partial\mathcal{C}/\partial\Phi$ and the gradient norm $\|\nabla\mathcal{C}\|$ on the GPU
 8:         if $\mathcal{C}(\Phi + \nabla\mathcal{C}/\|\nabla\mathcal{C}\|) < \mathcal{C}(\Phi)$ then
 9:             $\Phi \leftarrow \Phi + \nabla\mathcal{C}/\|\nabla\mathcal{C}\|$
                                                                        ▷ Update control points on the GPU
10:         end if
11:         Transfer $\|\nabla\mathcal{C}\|$ from the GPU to the CPU
12:     until $\|\nabla\mathcal{C}\| \leq \epsilon$
13:     $h \leftarrow h + 1$                                            ▷ Increase the deformation level
14: end while
15: Transfer $\Phi$ from the GPU to the CPU
```

$\gamma_1, \gamma_2, \ldots, \gamma_L$ and control point spacing $\delta_1, \delta_2, \ldots, \delta_L$, where L represents the number of deformation levels.

Our method mainly consists of similarity computation and gradient computation, which can be found at lines 6 and 7, respectively. Both require histogram computation and B-spline deformation, which limit the registration performance owing to memory intensive operations. We accelerate these processes using the GPU, which minimizes the amount of off-chip memory access. To achieve this, both deformation and histogram computation are implemented within a single kernel execution (concept C2). This implementation strategy avoids storing the transformed image $T(F)$ in off-chip memory because it allows $T(F)$ to be computed in an on-the-fly manner while computing a joint histogram. Similarly, gradient computation and control point update, which can be found at lines 7 and 8–10, respectively, are also implemented within a single kernel execution (concept C2).

We use different parallel schemes for gradient and similarity computations because these computations have slightly different parallelism: the former creates and compounds $|\Phi|$ histograms for local regions, whereas the latter creates a single histogram for the whole domain Ω. As shown in Fig. 3, our method exploits data parallelism in the similarity computation by decomposing the image domain Ω into pieces and assigning each to a thread block. Similar to Shams' method [26], our parallel scheme maintains local histograms in order to reduce the performance penalty of atomic operations. However, our method differs in (1) using shared memory (concept C1), (2) reducing the amount of off-chip memory access (concept C2), and (3) maintaining a histogram per thread block rather than per thread. The details of our contribution will be presented in Section IV-A. After this histogram computation, the joint entropy $H(R, T(F))$ and the entropies $H(R)$ and $H(T(F))$ are computed in parallel to obtain a similarity value. This computation can be easily parallelized by iterating tree-based reduction operations [37] on histograms.

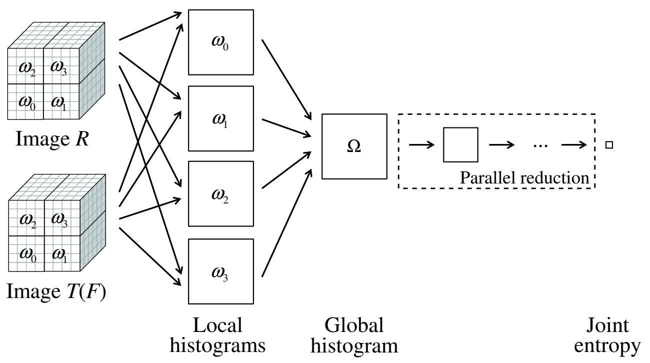


Fig. 3. Parallel scheme for similarity computation. Each thread block is responsible for a piece of the image domain and owns a local joint histogram. Local histograms are merged into a global histogram using atomic operations. The global histogram is then reduced into the joint entropy by a parallel tree-based approach [37].

Data parallelism in the gradient computation is exploited by assigning each control point $\phi_{i,j,k} \in \Phi$ to a thread block, as shown in Fig. 4. In other words, our scheme creates $|\Phi|$ global joint histograms simultaneously, because only a small region $D_{i,j,k}$ is referenced to generate each joint histogram. Thus, more parallelism is exploited by simultaneous computation of multiple joint

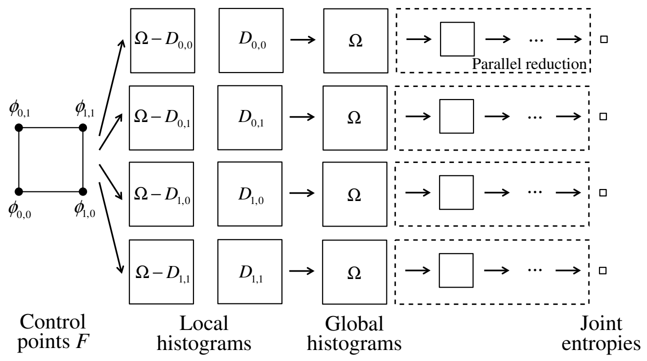


Fig. 4. Parallel scheme for gradient computation. Each thread block is responsible for a control point and owns the global joint histogram for the point. The global joint histogram is computed with a precomputation technique [36].

histograms. However, this suppresses resource usage, and therefore makes it difficult to store per-thread histograms even in large capacity off-chip memory. Accordingly, we decided to maintain a histogram per thread block. Although our scheme requires atomic operations within each thread block, this serialization overhead can be reduced by using shared memory. Note that the global joint histogram obtained in similarity computation can be reused to precompute joint histograms for unaffected regions $\Omega - D_{i,j,k}$ in gradient computation.

A. Acceleration of Joint Histogram Computation

In general, medical images have at least $d = 256$ grayscale levels, and the data size of a joint histogram reaches 256 KB if we use 4-byte bins. To store this data within 48 KB of shared memory (concept C1), our method (1) exploits the sparsity of joint histograms and (2) minimizes the bin size. By using this small data structure, our method stores 8-bit bins in shared memory and 32-bit bins in global memory. We also present an atomic-based parallel mechanism needed to handle simultaneous bin accesses and different bin sizes between shared memory and global memory.

Figure 5 shows a joint histogram generated from clinical images. As shown in this figure, as floating image $T(F)$ converges to an optimal solution, nonempty bins converge around the diagonal of the 2-D matrix. For multimodal images, a modality transformation technique [38], [39] is useful for producing this kind of convergence. Therefore, if all nonempty bins appear around the diagonal, our method switches behavior to use shared memory by eliminating empty bins located far from the diagonal. This implies that, like Shams' method [26], our method initially uses off-chip memory. However, owing to our multiresolution approach, such initial and coarse

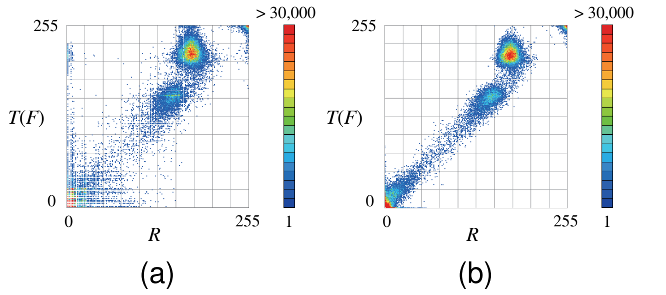


Fig. 5. Joint histograms of dynamic contrast enhanced CT images: (a) before and (b) after registration. Empty bins are white. Nonempty bins are colored from blue to red as values increase.

grained steps take less time than the succeeding fine-grained steps. Because we eliminate only empty bins, the truncated image information never affects registration results.

Let P be a parallelogram with width W and height $d - 1$ having its center on the diagonal, as shown in Fig. 6(a). Suppose that all nonempty bins exist within P. Our method will store bins located within P, as shown in Fig. 6(b). Bins outside P are not stored because they are empty. More formally, bins of the naive joint histogram are transformed into those of our joint histogram as follows:

$$(r', f') = (r, f - r + \lfloor W/2 \rfloor), \tag{8}$$

where (r, f) represents the location of the source bin in the naive data structure and (r', f') represents the location of the destination bin in the proposed data structure. The width W is determined according to d, the bin size b (in bits), and the capacity of shared memory. For instance, the data size of P reaches $256W$ in bytes if 8-bit bins ($b = 8$) are used for 8-bit ($d = 256$) images. Using this configuration, P can be stored in 48 KB of shared memory if $W \leq 192$.

Note that P in Fig. 6(a) includes wasted bins at the top and bottom area. This wasted region simplifies address computation in our kernel (i.e., Eq. (8)). Without this wasted region, we have to add a branch in the kernel to obtain a bin address correctly. In our preliminary experiments, we found that this additional branch increased execution time by 25%.

In addition to the number of bins, we minimize the bin size b according to the thread block size S. Because S threads, which share a joint histogram, increment S bins at a voting step, $b \geq \lceil \log_2 S \rceil$ must be satisfied to avoid an overflow during this parallel step. Therefore, the bin size b of 8 bit is sufficient for our method, which uses $S = 256$ threads. However, overflows cannot be avoided if voting steps are repeated. Consequently, when our method detects a maximum bin in shared

Algorithm 2 Joint histogram computation using shared memory with a merge mechanism.

Input: A pair $\langle R, F\rangle$ of reference and floating images, and their image sizes X, Y, and Z.
Output: A joint histogram h in off-chip memory.

```
1:  Compute the responsible line $(x, y)$ according to the thread index and the thread block index
2:  Initialize 32-bit joint histogram $h$ and 8-bit joint histogram $hs$ in off-chip memory and shared memory, respectively
3:  Synchronization
4:  for $z \leftarrow 0$ to $Z - 1$ do
5:      $r \leftarrow R(x, y, z)$                              ▷ Fetch a voxel value
6:      $f \leftarrow F(x, y, z)$                              ▷ Fetch an interpolated voxel value
7:      $f' \leftarrow f - r + \lfloor W/2 \rfloor$
8:      if $hs(r, f') = $ 0xff then                            ▷ Merge a bin to avoid an overflow
9:          $val \leftarrow$ atomicExch($hs(r, f'), 0$)
10:         atomicAdd($h(r, f), val$)
11:     end if
12:     atomicAdd($hs(r, f'), 1$)
13:     Synchronization
14: end for
15: Merge $hs$ into $h$ using atomicAdd()
```

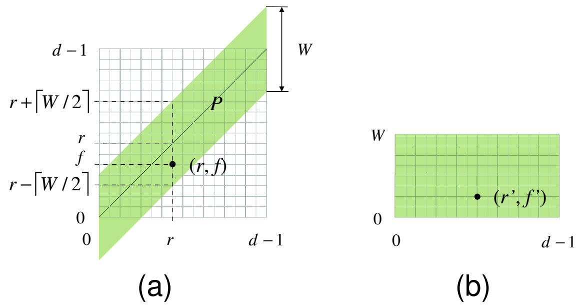


Fig. 6. Joint histogram organization: (a) naive data structure and (b) our data structure.

memory, it merges the bin into the corresponding bin in off-chip memory. Although this merge mechanism causes detection overhead, it reduces the amount of off-chip memory access, which determines the performance of joint histogram computation. Our method uses two merge mechanisms for similarity and gradient computations. For similarity computation, where thread blocks share a single global histogram, the bin is merged with an atomic operation. For gradient computation, where thread blocks possess local histograms, the bin can be merged without an atomic operation.

Algorithm 2 shows our pseudocode for the kernel that computes a joint histogram for similarity computation. Note that the atomic functions in the pseudocode operate on 8-bit data. We implemented these functions on the basis of the original atomic functions that operate on 32-bit integer values. Note that atomicExch() is also necessary to deal with the case in which multiple threads detect the maximum value at the same bin simultaneously. The final merge operation at line 15 is performed in parallel by assigning d^2/S responsible bins to each thread. Thus, our hierarchical organization cannot be simply implemented by placing joint histograms in shared memory.

Note that the branch at line 8 can cause thread divergence (concept C4). However, this branch reduces the amount of global memory access, which takes hundreds of cycles. Consequently, the benefit of reduced memory access outperforms the penalty of serial execution. Actually, the performance was decreased by 90% after eliminating this branch from the kernel. This preliminary result indicates that a naive utilization of shared memory is not sufficient to achieve high efficiency on the GPU.

Finally, our method switches behavior according to the degree of convergence. At each optimization step, threads check if all nonempty bins exist within P or not. To achieve this, each thread independently evaluates the following condition for their target bin located at (r, f):

$$r - \lfloor W/2 \rfloor \leq f \leq r + \lfloor W/2 \rfloor. \tag{9}$$

If Eq. (9) is true for all nonempty bins, threads use shared memory in the succeeding steps. Otherwise, all bins are computed using global memory.

B. Acceleration of B-spline Deformation

Our method takes advantage of hardware accelerated trilinear interpolation by storing reference and floating images in textures (concept C3). The data parallelism in the B-spline deformation can be easily exploited by assigning each voxel to a thread. Suppose that voxels are stored in the order x, y, z, and the x-axis has the smallest stride between adjacent voxels. We then decided to run XY threads for XYZ voxels so that warps can minimize the stride of memory access: each thread is responsible for Z voxels in a line, as shown in Fig. 7. The following three optimization techniques are integrated into our parallel scheme.

The first technique is data reuse [2] using shared memory (concept C1). To maximize the data reuse effect mentioned in Section III-B, we rewrite Eqs. (6) and (7) as follows:

$$T(x, y, z) = \sum_{n=0}^{3} B_n(w)\check{\phi}_{k+n}, \quad (10)$$

$$\check{\phi}_{k+n} = \sum_{l=0}^{3}\sum_{m=0}^{3} B_l(u)B_m(v)\phi_{i+l,j+m,k+n}. \quad (11)$$

Our method then precomputes $\check{\phi}_{k+n}$ on the GPU, for all relative positions u and v. These results are stored in register files to save clock cycles for voxels in the same cell of the mesh Φ (see Fig. 7). Consequently, our scheme allows threads to reuse data within their responsible line (i.e., between neighbor voxels along the z direction). Furthermore, threads in the same thread block can reuse $\sum_{l=0}^{3} B_l(u)\phi_{i+l,j+m,k+n}$ between neighbor voxels along the y direction through the use of shared memory. This data reaches $12\delta/\gamma$ bytes because a cell contains δ/γ voxels in the y direction and $\check{\phi}_{i+l}$ is a 3-D vector of single-precision elements.

The second technique is precomputation of B-spline coefficients. Similar to Saxena et al. [18], we first precompute B-spline coefficients B_l $(0 \le l \le 3)$ on the CPU and then store them as a lookup table in shared memory (concept C1). This lookup table, in a single-precision format, consumes $16\delta/\gamma$ bytes because each cell of the mesh Φ has δ/γ voxels in all directions.

Finally, we eliminate divergent branches (concept C4). The threads responsible for border voxels can access outside the image domain Ω because every thread refers its $4 \times 4 \times 4$ neighborhood control points. To avoid such out of boundary accesses, we place dummy control points around the image domain. Such dummy points eliminate divergent branches because threads do not have to check their accessing address in advance.

V. Experimental Results

To evaluate our method in terms of execution time, we conducted experiments using a desktop PC. Our experimental machine had a quad-core Intel Core i5-2500K processor with 16 GB RAM and a 512-core NVIDIA GeForce GTX 580 graphics card with 1.5 GB VRAM. The graphics card was connected with a PCI Express bus (generation 2). We used CUDA 4.2 [10] running on Windows 7.

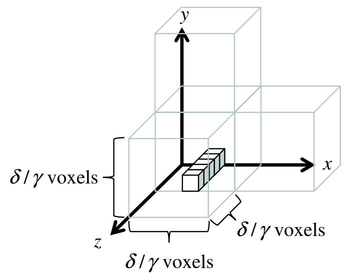


Fig. 7. Parallel B-spline deformation with data reuse. A cube with gray lines represents a mesh cell. Thread (x, y) is responsible for line (x, y) containing Z voxels. Each thread is allowed to reuse data between δ/γ voxels in the same cell of the mesh.

In addition to our GPU-based method, we implemented a CPU-based method. The GPU-based method was implemented with CUDA and the CPU-based method was multithreaded using OpenMP directives [40]. Our CPU implementation ran 3.3 times faster than a single-threaded implementation when using all four cores.

We applied our method to liver datasets consisting of $512 \times 512 \times 296$ voxels with $d = 256$ grayscale levels. These datasets were four pairs of dynamic contrast-enhanced CT images acquired at Osaka University Hospital (Suita, Osaka, Japan). Each pair consisted of volumes at two different time-phases: one was the early arterial phase, where hypervascular tumors were enhanced via the hepatic artery, and the other was the portal-venous phase, where hypovascular tumors were enhanced via the normal liver parenchyma. The volumes acquired at different time-phases are not usually registered due to respiratory movements, because CT data acquisition (i.e., the time phase) is not precisely synchronized with the respiratory cycle. Registration of these volumes is highly desirable to assist hepatic disease diagnosis and surgical planning. For example, aligned images of the liver facilitate lesion identification because they accurately correlate the portal/hepatic veins and tumors enhanced at different phases [41].

Figure 8 shows an example of reference and floating images with checkerboard visualization. For datasets #1–#4, the maximum deformations after registration were 16.1, 11.4, 10.1, and 15.7 mm, respectively. The local large deformations in these datasets were due to the respiratory movements. Table I shows the parameter values used for the experiments. We refined images and control points Φ by three levels. Given $W = 177$, our method used shared memory in all deformation levels except the first level. Note that the hierarchical deformation model is the key to deal with the local large deformations. Actually, we failed to align the images if

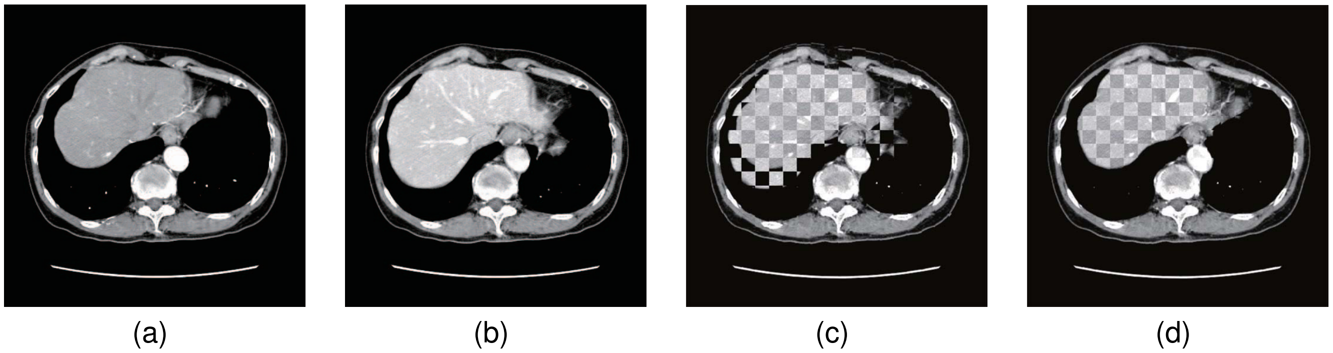


Fig. 8. Examples of registration results: (a) reference image and (b) floating image. Checkerboard visualization showing reference and floating images alternately: (c) before and (d) after registration.

TABLE I
PARAMETER VALUES USED FOR EXPERIMENTS.

Level h	$X \times Y \times Z$ (voxel)	γ (mm)	δ (mm)	ϵ	W (bin)
1	$128 \times 128 \times 74$	2.68	42.88	0.001	177
2	$256 \times 256 \times 148$	1.34	21.44	0.001	177
3	$512 \times 512 \times 296$	0.67	10.72	0.001	177

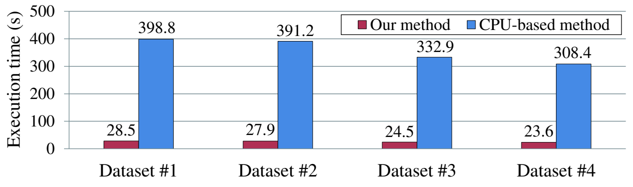


Fig. 9. Execution times for four datasets. The CPU-based method exploits all four cores.

we used only the finest deformation level.

A. Registration Time

Figure 9 shows the execution times for four datasets. Our GPU-based method and the CPU-based method processed the same number of optimization steps to complete the alignment. Depending on the dataset to be aligned, registration times of our method ranged from 23.6 to 28.5 s. The different results are due to the number of optimization steps at the finest deformation level, which varies according to the target dataset. Despite this difference, our method reached speeds approximately 14 times faster than the CPU-based method, successfully demonstrating the impact of GPU-based acceleration. Data transfer between the CPU and GPU was not a performance bottleneck as the transfer time of 0.11 s was negligible compared to the total time.

We next compared our registration throughput with those reported by previous papers [15], [18], [19], [23], which used different datasets, optimization parameters, and machines. In addition, the original implementation of Modat's method [19] was downloaded and compared using our dataset and machine. Because their method uses a different optimization scheme, we used a default parameter that produces similar alignment results in terms of the maximum deformation. The remaining parameter values in Table I were same as those employed in our method. Furthermore, some previous methods [21], [23], [24], [25] for NMI computation were implemented by ourselves to integrate them into our registration code and measure their throughputs using the same dataset and optimization parameters on the same platform. However, sorting-based methods [21], [24] change the order of voxels, so that they cannot be used with the data reuse and precomputation techniques implemented in our deformation code. The remaining methods [23], [25] used our efficient deformation code.

Table II shows these comparative results, including deployed GPUs and their peak performance in terms of off-chip memory bandwidth and arithmetic instruction throughput, which can be computed from the hardware specification. The peak performance is presented to clarify that the memory bandwidth determines the GPU-based registration performance. The registration throughput is given by XYZ/T_1, where T_1 represents the execution time spent for dataset #4.

Our method achieved a throughput greater than 3200 Kvoxel/s, which is at least five times higher than the throughputs of previous methods. Although a part of this comparison is not completely fair owing to different hardware components, datasets, and optimization parameters, we believe that the exploitation of shared memory is the key to maximizing the GPU registration performance, which is primarily dominated by memory-intensive operations. Note here that the speed increase of the first deformation level was a factor of 10, which is 60% lower than those of the remaining levels. This limited increase was due to the degree of convergence, which was not sufficient to use shared memory at the first level. Consequently, our throughput decreases to

TABLE II
PERFORMANCE COMPARISON OF GPU-BASED NONRIGID REGISTRATION METHODS. THE RESULTS OF PREVIOUS METHODS WITH * WERE QUOTED FROM THE ORIGINAL PAPERS, WHICH USED DIFFERENT DATASETS, OPTIMIZATION PARAMETERS, AND MACHINES. OTHER RESULTS WERE OBTAINED USING DATASET #4 ON OUR EXPERIMENTAL MACHINE.

Method	GPU	Bandwidth (GB/s)	Arithmetic (GFLOPS)	$X \times Y \times Z$ (voxel)	Throughput (Kvoxel/s)	Execution time (s)
This paper	GTX 580	192	2372	$512 \times 512 \times 296$	3288	23.6
Han [23]					581	133.6
Modat [19]					445	174.4
Vetter [24]					393	197.3
Shams [21]					270	287.5
Lou [25]					N/A**	N/A**
Plishker* [15]	GTX 285	159	1063	$256 \times 256 \times 256$	171	98.0
Han* [23]	GTX 280	142	933	$256 \times 256 \times 128$	442	19.0
Saxena* [18]	C1060	102	933	$512 \times 512 \times 98$	382	67.2
Modat* [19]	8800 GTX	86	518	$181 \times 217 \times 181$	169	42.0

**: execution failed due to memory exhaustion

TABLE III
BREAKDOWN OF THE EXECUTION TIMES REQUIRED FOR AN OPTIMIZATION STEP AT DEFORMATION LEVEL h. AVERAGE TIMES ARE OBTAINED USING DATASET#1.

Breakdown	Our method (s)			CPU-based method (s)		
	$h = 1$	$h = 2$	$h = 3$	$h = 1$	$h = 2$	$h = 3$
Gradient comp.	0.08	0.31	2.73	0.83	5.33	43.11
Similarity comp.	0.01	0.01	0.09	0.08	0.54	4.18
Total	0.09	0.32	2.82	0.91	5.87	47.29

approximately 1940 Kvoxels/s for $256 \times 256 \times 148$-voxel datasets.

Table III shows the breakdown of execution times for an optimization step. The speed increases for similarity computation are factors of 8.0, 54.0, and 46.4 at the first, second, and third levels, respectively. The increase between the first and second levels is achieved by the exploitation of shared memory. In contrast, the decrease between the second and third levels is due to the distribution of nonempty bins in joint histograms. As we mentioned in Section IV-A, the nonempty area in joint histograms becomes smaller as the solution converges to the local optimum. Consequently, atomic operations cause more conflicts in the finest level. Although our method minimizes this penalty by using low-latency shared memory and by allowing thread blocks to have local joint histograms, this slowdown cannot be completely avoided owing to the lack of parallelism.

A similar up-down trend can be found among the speed increases of gradient computation. However, at factors of 10.4, 17.2, and 15.8 for the first, second, and third levels, respectively, they are relatively lower than those of similarity computation. These lower speed increases are due to memory bandwidth exhaustion because our method processes $|\Phi|$ joint histograms simultaneously.

B. *Efficiency Analysis*

Table IV shows a comparison of efficiencies, throughputs, and execution times of MI computation with previous methods. The throughputs were measured using dataset #1 at the finest resolution level $h = 3$. For measurement, we used the original code of Shams' method [26]. The remaining methods [21], [24], [25] were implemented by ourselves. We also implemented two additional versions of our method to clarify the impact of our merge mechanism. Owing to shared memory, our method increased throughput from 588 Mvoxel/s to 913 Mvoxel/s. Furthermore, our merge mechanism allows threads to write a single bin instead of all the bins they are responsible for, and thus the throughput reached 7390 Mvoxel/s. This implies a 12-fold speed increase over the off-chip memory version. We also found that previous methods completed MI computation within 40 ms for small datasets. On the other hand, our method realized rapid MI computation for large datasets within 15 ms by using shared memory. We think that our sorting-based implementations [21], [24] achieved reasonable throughputs for our GTX 580 card, because each measured throughput linearly increased with the peak memory bandwidth of the deployed GPU. With respect to Vetter's method, our measured throughput (2452 Mvoxel/s) was 1.3 times higher than the original throughput (1872 Mvoxel/s), and this speedup factor is close to that of the peak memory bandwidth ($1.2 = 192/159$).

Next, we evaluated the efficiency of our MI computation in terms of memory throughput because this computation is a memory-bound operation. Because an increment of a bin loads two 32-bit voxel values and stores a 32-bit integer value, the throughput can be given by $3 \times 4 \times XYZ/T_2$ in B/s, where T_2 represents

TABLE IV
EFFICIENCY COMPARISON OF GPU-BASED MUTUAL INFORMATION COMPUTATION. VERSION 1 USES SHARED MEMORY BUT THREADS MERGE ALL THEIR RESPONSIBLE BINS RATHER THAN A SINGLE BIN EVERY TIME THEY FIND A MAXIMUM BIN. VERSION 2 USES OFF-CHIP MEMORY INSTEAD OF SHARED MEMORY. THE RESULTS OF PREVIOUS METHODS WITH * WERE QUOTED FROM THE ORIGINAL PAPERS, WHICH USED DIFFERENT DATASETS AND MACHINES. THE PERFORMANCE OF SHAMS* [26] IS THAT PRESENTED IN [21].

Method	GPU	Bandwidth (GB/s)	Arithmetic (GFLOPS)	$X \times Y \times Z$ (voxel)	Throughput (Mvoxel/s)	Efficiency (%)	Execution time (ms)
This paper	GTX 580	192	2372	$512 \times 512 \times 296$	7390	46	10.5
Version 1					913	6	85.0
Version 2					588	4	132.0
Vetter [24]					2452	12	41.5
Shams [21]					261	2	297.3
Shams [26]					132	1	589.1
Lou [25]					N/A**	N/A**	N/A**
Vetter* [24]	GTX 285	159	1063	$128 \times 128 \times 128$	1872	14	1.1
Shams* [21]	GTX 280	142	933	$512 \times 512 \times 29$	200	2	38.0
Shams* [26]					100	1	76.0
Chen* [20]	FX 5800	102	933	$230 \times 230 \times 239$	937***	11***	13.5***

: execution failed due to memory exhaustion, *: with deformation

the execution time of MI computation. The throughput of our MI computation reached 88.7 GB/s, which is equivalent to 46% of the peak bandwidth of off-chip memory. Considering the irregular memory access pattern and the conflicts of atomic operations, we believe that, on account of shared memory usage, this efficiency is relatively high. The efficiencies of previous methods ranged from 2% to 14%.

The throughput mentioned above represents the efficiency of similarity computation. Next, we analyzed the efficiency of gradient computation. At the finest deformation level, it took $T_3 = 2.18$ s to compute $|\Phi|$ joint histograms six times for control point displacement. Because each joint histogram covers $(4\delta/\gamma)^3$-voxel space, the throughput can be obtained by $3 \times 4 \times 6 \times |\Phi| \times (4\delta/\gamma)^3/T_3$, where $|\Phi| = \lceil X/(\delta/\gamma)\rceil \times \lceil Y/(\delta/\gamma)\rceil \times \lceil Z/(\delta/\gamma)\rceil$. The throughput of our MI computation in gradient computation reached 168.4 GB/s, which is 88% of the peak bandwidth of off-chip memory. Because this efficiency is higher than that of similarity computation, our parallel scheme is useful for increasing the efficiency of parallel gradient computation.

Our GPU-accelerated B-spline deformation was 6.2 times faster than the CPU-based method for all deformation levels. Therefore, the performance of B-spline deformation in our method does not depend on the deformation level or the position of control points. The achieved execution time was 0.08 s at the finest deformation level. This execution time was reduced from the original time of 1.23 s. Rohlfing's data reuse technique [2] reduced the execution time to 0.13 s, and our divergent elimination technique further reduced the time to 0.08 s.

C. Registration Accuracy

As mentioned in Section V-A, our GPU-based method and the CPU-based method processed the same number of optimization steps for our experimental datasets. However, the GPU can yield different numerical results as those produced by the CPU, because they have different instruction sets, and GPU architectures are not fully IEEE-754 compliant [42] in terms of rounding errors [10]. In addition, our method utilizes texture-based interpolation, which employs a lower precision format. These architectural differences can cause different numbers of optimization steps.

To evaluate the registration accuracy, we compared the deformation vector fields of our GPU-based method and the CPU-based method. Compared with CPU results, the maximum errors of deformation vectors were 0.52, 0.41, 0.37, and 0.38 voxels for datasets #1–#4, respectively. These maximum errors were less than the image resolution $\gamma = 0.67$ at the finest level. Thus, the GPU produced nearly same deformation fields as those obtained on the CPU.

Figure 10(c) illustrates an error distribution for dataset #1. Although the maximum deformations were approximately equal between GPU and CPU results, large errors were mainly observed in low-contrast areas. Voxels in such areas have nearly equal values. Consequently, these large errors cannot be clearly seen as differences of voxel values, as shown in the differential image of Fig. 10(d). Because many medical image processing projects such as 3D Slicer [43] and the Insight Segmentation and Registration Toolkit (ITK) [44] adopt the CPU-based method for image-guided procedures, we think that our GPU-based method is also acceptable for such practical situations.

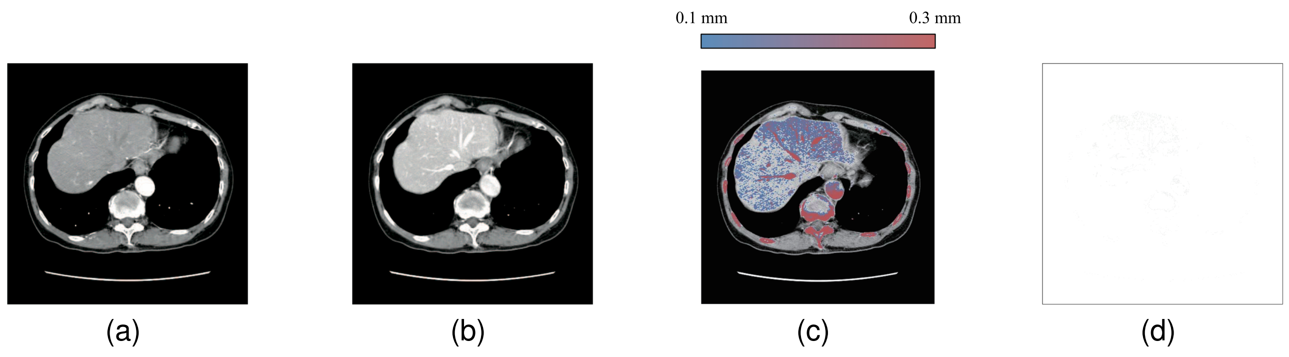


Fig. 10. Error distribution of dataset #1: (a) reference image, (b) aligned floating image, (c) original floating image with error distribution, and (d) differential image between GPU- and CPU-based aligned images. An error is associated with the source of a deformation vector (i.e., a voxel in the original floating image). Smaller errors are blue and larger errors are red.

The source of these errors mainly exists in hardware-accelerated interpolation. Although this capability is attractive in terms of performance, it uses a 9-bit fixed point format [10] to represent the weights of trilinear interpolation. In contrast, the CPU-based method uses a 32-bit floating point representation. For interpolation of similar values, in terms of accuracy, the floating point representation is superior to the fixed point representation.

Figure 11 shows how NMI values increased during optimization. As shown in this figure, in the earlier computational phase, the initial deformation level finished rapidly. This coarsest level, in which global memory is used to compute histograms, occupies only 5.5% of the total computation time. Therefore, our switching strategy efficiently accelerates the optimization procedure in succeeding levels and shows effective integration with a multiresolution approach.

Our method cannot be activated if the deformation of two images is considerably large, because nonempty bins can exist outside a parallelogram area. However, our method was activated at higher resolution levels, which dominate execution time, as shown in Fig. 11. This indicates that the initial large deformations can be reduced at a low resolution level. For datasets #1–#4, the maximum deformations after the first deformation level were 6.7, 4.4, 4.1, and 6.5 mm, respectively. Consequently, our method will reduce execution time provided that the maximum length of deformations reduces at every optimization step.

D. Multimodal Image Registration

We applied our method to multimodal registration of the brain. We used CT, T1-, T2-, and PD-weighted MR images to deal with four registration cases: (1) CT and T1-weighted images (Fig. 12), (2) T1- and T2-weighted images (Fig. 13), (3) CT and T2-weighted images (Fig.

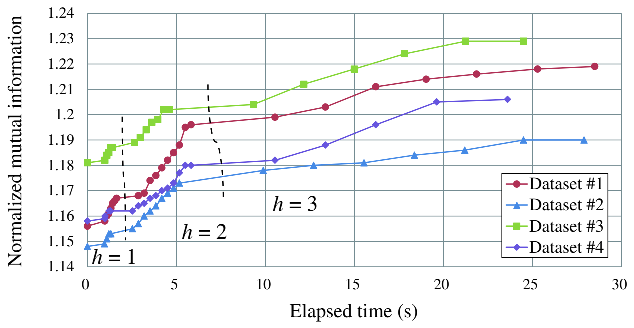


Fig. 11. Optimization progress on the GPU. A plot corresponds to an optimization step. NMI values increase with the deformation level h.

TABLE V
PARAMETER VALUES USED FOR MULTIMODAL REGISTRATION.

Level h	$X \times Y \times Z$ (voxel)	γ (mm)	δ (mm)	ϵ	W (bin)
1	$64 \times 64 \times 25$	5.23	20.92	0.001	177
2	$128 \times 128 \times 49$	2.61	10.46	0.001	177
3	$256 \times 256 \times 98$	1.31	5.23	0.001	177

14), and (4) CT and PD-weighted images (Fig. 15) for reference and floating images, respectively. The data were obtained from the Vanderbilt database [45]. Table V shows the parameter values used for the experiments.

In order to use shared memory for joint histogram computation, we first applied a modality transformation technique [38] to the floating image. This transformation allows the transformed floating image to have the same representation as the reference image. Consequently, non-empty bins appear around the diagonal of the joint histogram of the reference and transformed floating images. Thus, our method used shared memory in all deformation levels.

With respect to the first and second cases (Figs. 12 and 13), we found that our method produced aligned images within 1.31 and 0.38 seconds for the first and second

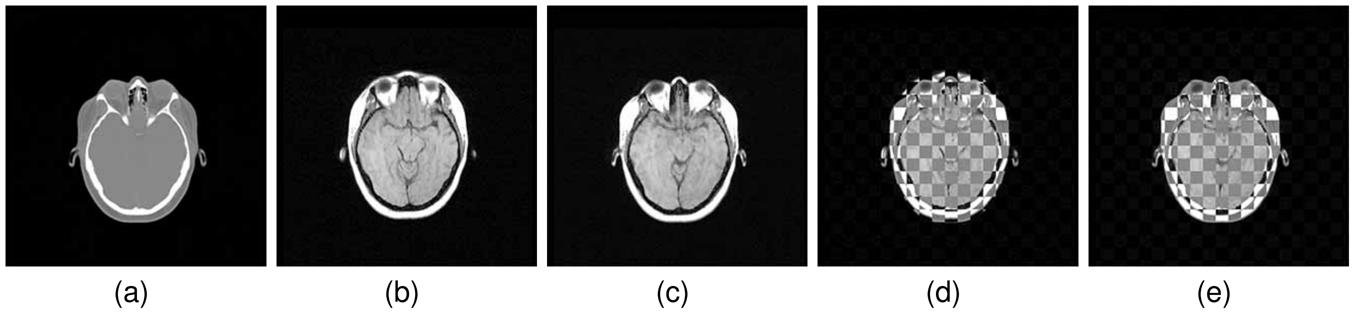


Fig. 12. Successful examples of multimodal registration results: (a) reference CT image and floating T1-weighted MR image (b) before and (c) after registration. Checkerboard visualization (d) before and (e) after registration.

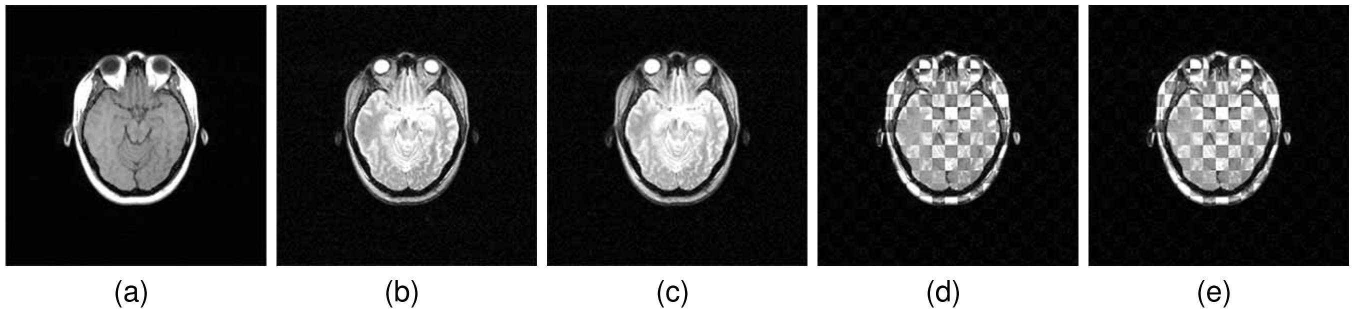


Fig. 13. Successful examples of multimodal registration results: (a) reference T1-weighted MR image and floating T2-weighted MR image (b) before and (c) after registration. Checkerboard visualization (d) before and (e) after registration.

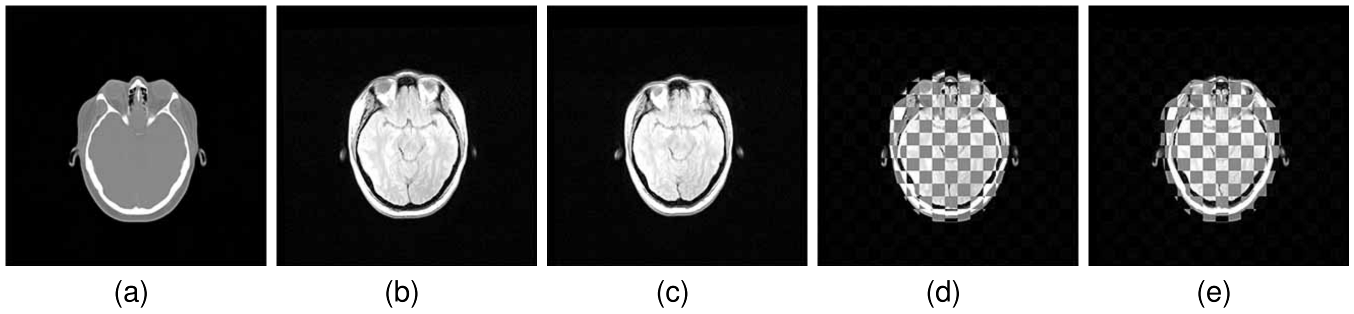


Fig. 14. Failed examples of multimodal registration results: (a) reference CT image and floating T2-weighted MR image (b) before and (c) after registration. Checkerboard visualization (d) before and (e) after registration.

cases, respectively. These timing results are 6.4 and 5.9 times faster than a naive method (version 2 in Table IV) that uses off-chip memory instead of shared memory. The maximum lengths of deformations after registration were 16.7 and 5.3 mm for the first and second cases, respectively.

In contrast to the successful results mentioned above, our method failed to produce aligned images for the third and fourth cases, which deal with multimodal images that have not coarse similarity in terms of intensity values. However, we found that our GPU-based method successfully solved the third case if the modality transformation technique was not applied to the floating image. In this case, our shared memory version was not activated during registration. This result implies that modality transformation can cause a bias issue, which may lead registration process to fail, though the transformation is useful to achieve significant acceleration over the off-chip memory version.

On the other hand, both the CPU- and GPU-based methods failed to solve the fourth case. This failure is due to the basic registration algorithm (i.e., a combination of NMI, B-spline deformation, and steepest descent optimization).

VI. Conclusions

We presented a CUDA implementation of a highly efficient method for accelerating NMI-based nonrigid registration. Our method reduces the data size of joint histograms so they can be stored in shared memory for fast NMI computation. An efficient merge mechanism is integrated into our kernel to minimize the amount of

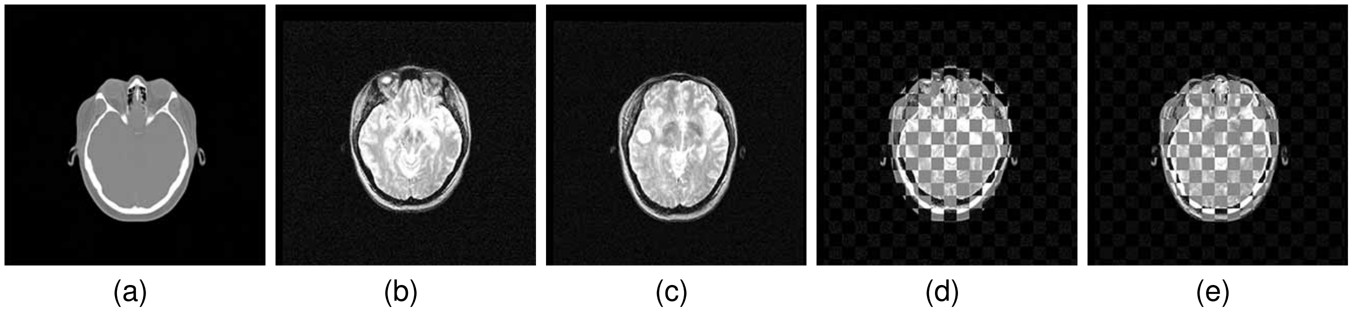


Fig. 15. Failed examples of multimodal registration results: (a) reference CT image and floating PD-weighted MR image (b) before and (c) after registration. Checkerboard visualization (d) before and (e) after registration.

off-chip memory access for fast joint histogram computation. Furthermore, our method achieves rapid B-spline deformation by performing data reuse on shared memory.

We experimentally evaluated our method using four datasets of liver CT images consisting of $512\times512\times296$ voxels. Our alignment procedure completed within a few tens of seconds, which is five times faster than previous GPU-based methods and 14 times faster than a multithreaded CPU-based method. The efficiencies of our method reach 88% and 46% in gradient and similarity computation, respectively. We also demonstrated the impact and limitation of our method for multimodal registration of the brain. Thus, our method demonstrates the effectiveness of using shared memory by realizing rapid nonrigid registration for time-demanding situations.

We think that our method is useful to increase the efficiency of parallel registration on future GPU architectures, because our hierarchical joint histogram organization reduces the amount of resource consumption per thread block. Consumption of fewer resource not only allows registration tasks to be accelerated on a tablet device equipped with a next generation mobile GPU such as the Tegra K1, but also increases the number of active threads [10] on a highly-threaded GPU architecture. Thus, our method will process multiple thread blocks simultaneously on a future GPU that has more than 48 KB of shared memory. Such active thread blocks are useful to overlap a memory fetch instruction with a data-independent arithmetic instruction.

Acknowledgement

This study was partly supported by the JSPS KAKENHI Grant Number 23300007, 23700057, the JST CREST program, "An Evolutionary Approach to Construction of a Software Development Environment for Massively-Parallel Computing Systems," and the MEXT project, "Creating Hybrid Organs of the future" at Osaka University. The authors would like to thank the anonymous reviewers for helpful comments to improve their paper.

References

[1] J. V. Hajnal, D. L. Hill, and D. J. Hawkes, Eds., *Medical Image Registration*. Boca Raton, FL: CRC Press, 2001.

[2] T. Rohlfing and C. R. Maurer, "Nonrigid image registration in shared-memory multiprocessor environments with application to brains, breasts, and bees," *IEEE Trans. Information Technology in Biomedicine*, vol. 7, no. 1, pp. 16–25, Mar. 2003.

[3] S. Oguro, J. Tokuda, H. Elhawary, S. Haker, R. Kikinis, C. M. Tempany, and N. Hata, "MRI signal intensity based B-Spline nonrigid registration for pre- and intraoperative imaging during prostate brachytherapy," *J. Magnetic Resonance Imaging*, vol. 30, no. 5, pp. 1052–1058, Oct. 2009.

[4] F. A. Jolesz, "Image-guided procedures and the operating room of the future," *Radiology*, vol. 204, no. 3, pp. 601–612, Sep. 1997.

[5] O. Clatz, H. Delingette, I.-F. Talos, A. J. Golby, R. Kikinis, F. A. Jolesz, N. Ayache, and S. K. Warfield, "Robust nonrigid registration to capture brain shift from intraoperative MRI," *IEEE Trans. Medical Imaging*, vol. 24, no. 11, pp. 1417–1427, Nov. 2005.

[6] N. Chrisochoides, A. Fedorov, A. Kot, N. Archip, P. Black, O. Clatz, A. Golby, R. Kikinis, and S. K. Warfield, "Toward real-time image guided neurosurgery using distributed and grid computing," in *Proc. Int'l Conf. High Performance Computing, Networking, Storage and Analysis (SC'06)*, Nov. 2006, 13 pages (CD-ROM).

[7] H. Elhawary, S. Oguro, K. Tuncali, P. R. Morrison, S. Tatli, P. B. Shyn, S. G. Silverman, and N. Hata, "Multimodality non-rigid image registration for planning, targeting and monitoring during CT-guided percutaneous liver tumor cryoablation," *Academic Radiology*, vol. 17, no. 11, pp. 1334–1344, Nov. 2010.

[8] D. Rueckert, L. I. Sonoda, C. Hayes, D. L. G. Hill, M. O. Leach, and D. J. Hawkes, "Nonrigid registration using free-form deformations: Application to breast MR images," *IEEE Trans. Medical Imaging*, vol. 18, no. 8, pp. 712–721, Aug. 1999.

[9] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, Mar. 2008.

[10] NVIDIA Corporation, "CUDA Programming Guide Version 4.2," Apr. 2012. [Online]. Available: http://developer.nvidia.com/cuda/

[11] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.

[12] Y. Okitsu, F. Ino, and K. Hagihara, "High-performance cone beam reconstruction using CUDA compatible GPUs," *Parallel Computing*, vol. 36, no. 2/3, pp. 129–141, Feb. 2010.

[13] F. Ino, Y. Munekawa, and K. Hagihara, "Sequence homology search using fine grained cycle sharing of idle GPUs," *IEEE Trans. Parallel and Distributed Systems*, vol. 23, no. 4, pp. 751–759, Apr. 2012.

[14] W. Plishker, O. Dandekar, S. S. Bhattacharyya, and R. Shekhar, "Towards systematic exploration of tradeoffs for medical image registration on heterogeneous platforms," in *Proc. IEEE Biomedial Circuits and Systems Conf. (BioCAS'08)*, Nov. 2008, pp. 53–56.

[15] ——, "Utilizing hierarchical multiprocessing for medical image registration," *IEEE Signal Processing Magazine*, vol. 27, no. 2, pp. 61–68, Mar. 2010.

[16] S. Lee, G. Wolberg, and S. Y. Shin, "Scattered data interpolation with multilevel B-splines," *IEEE Trans. Visualization and Computer Graphics*, vol. 3, no. 3, pp. 228–244, Jul. 1997.

[17] C. Studholme, D. L. G. Hill, and D. J. Hawkes, "An overlap invariant entropy measure of 3D medical image alignment," *Pattern Recognition*, vol. 32, no. 1, pp. 71–86, Jan. 1999.

[18] V. Saxena, J. Rohrer, and L. Gong, "A parallel GPU algorithm for mutual information based 3D nonrigid image registration," in *Proc. 16th European Conf. Parallel Computing (Euro-Par'10), Part II*, Sep. 2010, pp. 223–234.

[19] M. Modat, G. R. Ridgway, Z. A. Taylor, M. Lehmann, J. Barnes, D. J. Hawkes, N. C. Fox, and S. Ourselin, "Fast free-form deformation using graphics processing units," *Computer Methods and Programs in Biomedicine*, vol. 98, no. 3, pp. 278–284, Jun. 2010. [Online]. Available: http://sourceforge.net/projects/niftyreg/

[20] S. Chen, J. Qin, Y. Xie, W.-M. Pang, and P.-A. Heng, "CUDA-based acceleration and algorithm refinement for volume image registration," in *Proc. 8th IEEE Int'l Conf. Future BioMedical Information Engineering (FBIE'09)*, Dec. 2009, pp. 544–547.

[21] R. Shams, P. Sadeghi, R. Kennedy, and R. Hartley, "Parallel computation of mutual information on the GPU with application to real-time registration of 3D medical images," *Computer Methods and Programs in Biomedicine*, vol. 99, no. 2, pp. 133–146, Aug. 2010.

[22] R. Shams, P. Sadeghi, R. A. Kennedy, and R. I. Hartley, "A survey of medical image registration on multicore and the GPU," *IEEE Signal Processing Magazine*, vol. 27, no. 2, pp. 50–60, Mar. 2010.

[23] X. Han, L. S. Hibbard, and V. Willcut, "GPU-accelerated, gradient-free MI deformable registration for atlas-based MR brain image segmentation," in *Proc. IEEE Computer Society Conf. Computer Vision and Pattern Recognition Workshops (CVPRW'09)*, Jun. 2009, 6 pages (CD-ROM).

[24] C. Vetter and R. Westermann, "Optimized GPU histograms for multi-modal registration," in *Proc. 8th IEEE Int'l Symp. Biomedical Imaging (ISBI'11)*, Apr. 2011, pp. 1227–1230.

[25] Y. Lou, X. Jia, X. Gu, and A. Tannenbaum, "A GPU-based implementation of multimodal deformable image registration based on mutual information or Bhattacharyya distance," May 2011. [Online]. Available: http://hdl.handle.net/10380/3268/

[26] R. Shams and R. A. Kennedy, "Efficient histogram algorithms for NVIDIA CUDA compatible devices," in *Proc. Int'l Conf. Signal Processing and Communications Systems (ICSPCS'07)*, Dec. 2007, pp. 418–422. [Online]. Available: http://users.cecs.anu.edu.au/~ramtin/cuda.htm

[27] W.-H. Cheng and C.-C. Lu, "Acceleration of medical image registration using graphics process units in computing normalized mutual information," in *Proc. 5th Int'l Conf. Image and Graphics (ICIG'09)*, Sep. 2009, pp. 814–818.

[28] S. S. Samant, J. Xia, P. Muyan-Özçelik, and J. D. Owens, "High performance computing for deformable image registration: Towards a new paradigm in adaptive radiotherapy," *Medical Physics*, vol. 35, no. 8, pp. 3546–3553, Aug. 2008.

[29] A. Ruiz, M. Ujaldon, L. Cooper, and K. Huang, "Non-rigid registration for large sets of microscopic images on graphics processors," *J. Signal Processing Systems*, vol. 55, no. 1-3, pp. 229–250, Apr. 2009.

[30] Y. Liu, A. Fedorov, R. Kikinis, and N. Chirsochoides, "Non-rigid registration for brain MRI: faster and cheaper," *Int'l J. Functional Informatics and Personalised Medicine*, vol. 3, no. 1, pp. 48–57, May 2010.

[31] M. P. Wachowiak and T. M. Peters, "High-performance medical image registration using new optimization techniques," *IEEE Trans. Information Technology in Biomedicine*, vol. 10, no. 2, pp. 344–353, Apr. 2006.

[32] F. Ino, K. Ooyama, and K. Hagihara, "A data distributed parallel algorithm for nonrigid image registration," *Parallel Computing*, vol. 31, no. 1, pp. 19–43, Jan. 2005.

[33] C. R. Castro-Pareja, J. M. Jagadeesh, and R. Shekhar, "FAIR: A hardware architecture for real-time 3-D image registration," *IEEE Trans. Information Technology in Biomedicine*, vol. 7, no. 4, pp. 426–434, Dec. 2003.

[34] O. Dandekar and R. Shekhar, "FPGA-accelerated deformable image registration for improved target-delineation during CT-guided interventions," *IEEE Trans. Biomedical Circuits and Systems*, vol. 1, no. 2, pp. 116–127, Jun. 2007.

[35] J. Rohrer and L. Gong, "Accelerating 3D nonrigid registration using the Cell Broadband Engine processor," *IBM J. Research and Development*, vol. 53, no. 5, pp. 768–777, Sep. 2009.

[36] C. Studholme, R. T. Constable, and J. S. Duncan, "Accurate alignment of functional EPI data to anatomical MRI using a physics-based distortion model," *IEEE Trans. Medical Imaging*, vol. 19, no. 11, pp. 1115–1127, Nov. 2000.

[37] M. Harris, "Optimizing parallel reduction in CUDA," Nov. 2007, http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf.

[38] D.-J. Kroon and C. H. Slump, "MRI modality transformation in demon registration," in *Proc. 6th IEEE Int'l Symp. Biomedical Imaging (ISBI'09)*, Jun. 2009, pp. 963–966.

[39] M. Khader and A. B. Hamza, "An information-theoretic method for multimodality medical image registration," *Expert Systems with Applications*, vol. 39, no. 5, pp. 5548–5556, Apr. 2012.

[40] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon, *Parallel Programming in OpenMP*. San Mateo, CA: Morgan Kaufmann, Oct. 2000.

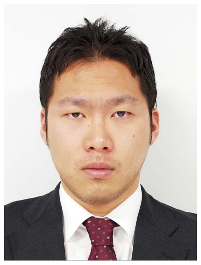
[41] J. Masumoto, Y. Sato, M. Hori, T. Murakami, T. Johkoh, H. Nakamura, and S. Tamura, "A similarity measure for non-rigid volume registration using known joint distribution of targeted tissue: Application to dynamic CT data of the liver," *Medical Image Analysis*, vol. 7, no. 4, pp. 553–564, Dec. 2003.

[42] D. Stevenson, "A proposed standard for binary floating-point arithmetic," *IEEE Computer*, vol. 14, no. 3, pp. 51–62, Mar. 1981.

[43] S. Pieper, M. Halle, and R. Kikinis, "3D slicer," in *Proc. 1st IEEE Int'l Symp. Biomedical Imaging (ISBI'04)*, Apr. 2004, pp. 632–635. [Online]. Available: http://www.slicer.org/

[44] T. S. Yoo and D. N. Metaxas, "Open science — combining open data and open source software: Medical image analysis with the insight toolkit," *Medical Image Analysis*, vol. 9, no. 6, pp. 503–506, Dec. 2005.

[45] J. M. Fitzpatrick, "The retrospective image registration evaluation project," 2008. [Online]. Available: http://www.insight-journal.org/rire/



Kei Ikeda received the M.E. degree in information and computer sciences from Osaka University, Osaka, Japan, in 2012. He is currently working toward the Ph.D. degree at Osaka University. His current research interests include high performance computing, medical image processing, and computer assisted surgery.

Fumihiko Ino (S'01–A'03–M'04) received the B.E., M.E., and Ph.D. degrees in information and computer sciences from Osaka University, Osaka, Japan, in 1998, 2000, and 2004, respectively. He is currently an Associate Professor in the Graduate School of Information Science and Technology at Osaka University. His research interests include parallel and distributed systems, software development tools, and performance evaluation.

Kenichi Hagihara received the B.E., M.E., and Ph.D. degrees in information and computer sciences from Osaka University, Osaka, Japan, in 1974, 1976, and 1979, respectively. From 1994 to 2002, he was a Professor in the Department of Informatics and Mathematical Science, Graduate School of Engineering Science, Osaka University. Since 2002, he has been a Professor in the Department of Computer Science, Graduate School of Information Science and Technology, Osaka University. From 1992 to 1993, he was a Visiting Researcher at the University of Maryland. His research interests include the fundamentals and practical application of parallel processing.