A GPGPU Approach for Accelerating 2-D/3-D Rigid Registration of Medical Images

Fumihiko Ino[1], Jun Gomita[2], Yasuhiro Kawasaki[1], and Kenichi Hagihara[1]

[1] Graduate School of Information Science and Technology, Osaka University
1-3 Machikaneyama, Toyonaka, Osaka 560-8531, Japan
ino@ist.osaka-u.ac.jp

[2] Graduate School of Information Science and Technology, The University of Tokyo

Abstract. This paper presents a fast 2-D/3-D rigid registration method using a GPGPU approach, which stands for general-purpose computation on the graphics processing unit (GPU). Our method is based on an intensity-based registration algorithm using biplane images. To accelerate this algorithm, we execute three key procedures of 2-D/3-D registration on the GPU: digitally reconstructed radiograph (DRR) generation, gradient image generation, and normalized cross correlation (NCC) computation. We investigate the usability of our method in terms of registration time and robustness. The experimental results show that our GPU-based method successfully completes a registration task in about 10 seconds, demonstrating shorter registration time than a previous method based on a cluster computing approach.

Keywords. GPGPU, image registration, performance evaluation.

1 Introduction

Image registration technique [1, 2] plays an increasingly important role in computer-aided surgery. For example, as illustrated in Fig. 1, 2-D/3-D registration technique allows us to align a preoperative 3-D CT volume with an intraoperative 2-D fluoroscopy image, giving us point correspondences between the coordinates in the virtual world and those in the real world. These precise correspondences are necessary to exactly perform a preoperative plan in the real world, which is carefully developed using the preoperative volume in advance of surgery. However, naive CPU implementations take several minutes to complete a registration task due to a large amount of computation. Therefore, some acceleration techniques are required to use this technique for surgical assistances, where response time is strictly limited in a short time.

One emerging computational platform is the graphics processing unit (GPU), namely commodity graphics hardware, which is rapidly increasing performance beyond Moore's law [3]. For example, nVIDIA's GeForce 6800 provides approximately 120 GFLOPS at peak performance, which equals to six 5-GHz Pentium 4 processors [4]. Furthermore, recent GPUs provide programmability to users, making themselves a more flexible platform as compared with earlier non-programmable GPUs, which deal only with rendering tasks of 3-D objects. Therefore, many researchers are trying to apply the GPU to a

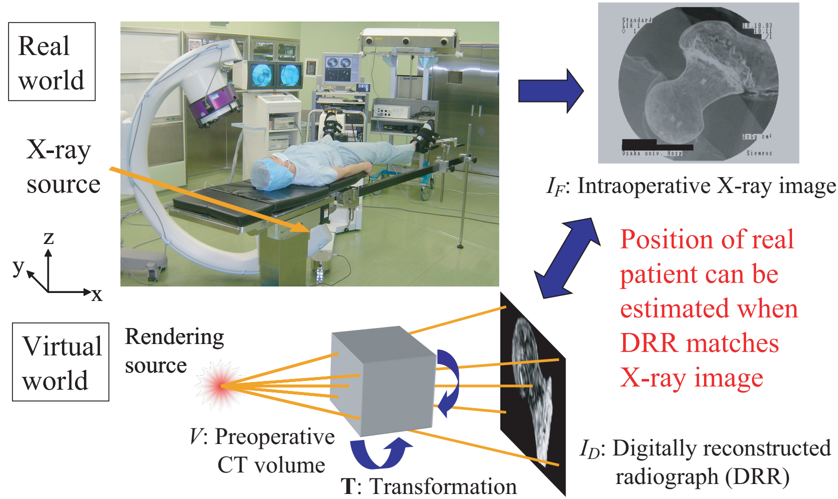

Fig. 1. Overview of 2-D/3-D registration.

variety of problems such as a fluid dynamics simulator [5], numerical application [6], data clustering application [7], and so on.

The objective of our work is to achieve fast 2-D/3-D registration by means of a general-purpose computation on the GPU (GPGPU) approach [8]. We implement the key procedures of a registration algorithm [9] on the GPU: 1) digitally reconstructed radiograph (DRR) generation; 2) gradient image generation; and 3) normalized cross correlation (NCC) computation. The main contribution of our work is the GPU implementation for procedures 2) and 3) based on that for procedure 1) [10]. We compare our GPU-based method with a cluster-based method [9] in terms of performance and robustness. Our method differs from prior methods [10, 11], which employ different strategies to implement a part of the three procedures on the GPU.

The rest of the paper is organized as follows. We begin in Section 2 by introducing the 2-D/3-D registration algorithm, and then show an overview of GPU architecture in Section 3. We then present our GPU-based method in Section 4. Section 5 shows some experimental results. Finally, Section 6 concludes the paper.

2 2-D/3-D Registration Algorithm

The problem of 2-D/3-D registration is to compute the rigid transformation parameter **T** that relates the coordinate system of a 3-D volume V and that of a 2-D image I_F (usually, a fluoroscopy image).

We first describe a single-image version of the registration algorithm for easier understanding of the biplane-image version [12]. Our method is based on an intensity-based algorithm [1, 13], which resolves the registration problem into an optimization problem. The algorithm optimizes a cost function C associated with transformation parameter **T**, where **T** represents the translation and rotation of V. The cost function C

here represents the similarity between an image I_F and a DRR I_D produced by projection of V. The optimization is done by the steepest descent optimization technique [14] in a coarse-to-fine manner.

According to an empirical study [13], we currently use gradient correlation (GC) for the cost function C. Given two 2-D images, A and B, GC $G(A, B)$ between them is given by:

$$G(A,B) = \frac{1}{2}\left[N\left(\frac{\partial A}{\partial x}, \frac{\partial B}{\partial x}\right) + N\left(\frac{\partial A}{\partial y}, \frac{\partial B}{\partial y}\right)\right] \tag{1}$$

where N represents NCC between two images, $\partial A/\partial x$ and $\partial A/\partial y$ ($\partial B/\partial x$ and $\partial B/\partial y$) are the horizontal and the vertical gradient images of A (B, respectively). NCC $N(A, B)$ between $n \times n$ pixel images A and B is given by:

$$N(A,B) = \frac{S_{AB} - S_A S_B/n^2}{\sqrt{S_{A^2} - (S_A)^2/n^2}\sqrt{S_{B^2} - (S_B)^2/n^2}}, \tag{2}$$

where S_A and S_{A^2} (S_B and S_{B^2}) represent the sum and the squared sum of A (B), respectively, and S_{AB} represents the multiplied sum of A and B.

The gradient images are produced by the first derivative of a Gaussian. This filter enhances the outline of objects with reducing and smoothing noise in images. Therefore, it contributes to improve the robustness of registration. Given an image A, the horizontal gradient image $\partial A/\partial x$ and the vertical gradient image $\partial A/\partial y$ are given by:

$$\frac{\partial A}{\partial x}(x,y) = \sum_{-R \le i,j \le R} \frac{-i}{2\pi\sigma^4} e^{-\frac{i^2+j^2}{2\sigma^2}} A(x+i, y+j), \tag{3}$$

$$\frac{\partial A}{\partial y}(x,y) = \sum_{-R \le i,j \le R} \frac{-j}{2\pi\sigma^4} e^{-\frac{i^2+j^2}{2\sigma^2}} A(x+i, y+j), \tag{4}$$

where σ and R represent the standard deviation and the kernel size of the filter, respectively. We currently use $\sigma = 3$ and $R = 9$.

In summary, the single-image algorithm optimizes the cost function $G(I_F, I_D)$ with respect to the transformation **T**. On the other hand, our biplane-image version optimizes the sum of two cost functions, each computed for one of the biplane images. The three procedures, namely 1) DRR generation, 2) gradient image generation, and 3) NCC computation, are repeated until finding a local optimum.

3 GPGPU: GPU as a Computational Engine

The original purpose of the GPU is to project 3-D polygonal objects on the 2-D screen. To accelerate this rendering task, the GPU employs a parallel architecture [4] that consists of two different programmable processors: vertex processors (VPs) and fragment processors (FPs), as shown in Fig. 2. Since FPs in modern GPUs provide much higher performance than VPs, most GPGPU implementations use FPs as a computational engine in the GPU [15].

Such implementations employ the stream programming model [16], which exploits the data parallelism inherent in the application by organizing data into streams and

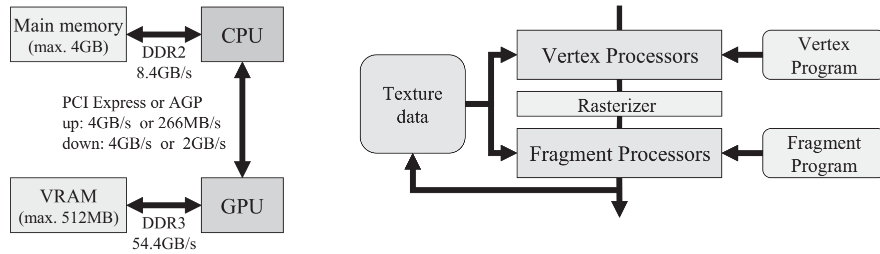

Fig. 2. GPU architecture.

expressing computation as kernels that operate on streams. Streams here are usually stored as texture data on the video memory, which can be fetched to FPs. A kernel is implemented as a FP program. The computed results can be transferred (readback) from the video memory to the main memory by using graphics APIs such as OpenGL [17].

In addition to the parallelization mentioned above, vectorization is also necessary to maximize the performance on the GPU. FPs support 4-length vector operations because they are designed to deal with pixels, which are four-component RGBA data representing red, green, blue colors and opacity. Since FPs apply vertor operations to a pixel, we must adapt data structure to obtain 400% speedup.

One concern about the GPGPU approach is that the GPU seems not be rigorous with computational errors [18], though it supports the IEEE floating-point representations [19]. Therefore, we should check computational results to verify if the error is acceptable.

4 2-D/3-D Registration on the GPU

Fig. 3 shows an overview of our GPU-based method. The key points of our design are as follows:

- (P1) Performance bottlenecks on the CPU should be implemented on the GPU with an algorithm suitable to the GPU architecture;
- (P2) The amount and frequency of communication between the CPU and the GPU should be minimized to achieve full acceleration on the GPU.

We think that the suitable algorithm mentioned above is an algorithm that (P1-a) resolves the target problem into a rendering problem, which is naturally accelerated by the GPU, or (P1-b) has fully data parallelism so that FPs can simultaneously process different pixels on the image (namely, texture), and if possible, with vector operations.

According to point (P1), we have decided to implement the three procedures on the GPU: DRR generation; gradient image generation; and NCC computation. These procedures take 99% of execution time on a sequential implementation.

1) DRR generation. As LaRose has presented in [10], this procedure can be naturally implemented on the GPU, because the principle of X-ray propagation is similar to that of object projection required for volume rendering. Therefore, we use a texture-based volume rendering method [20] for DRR generation in order to maximize the

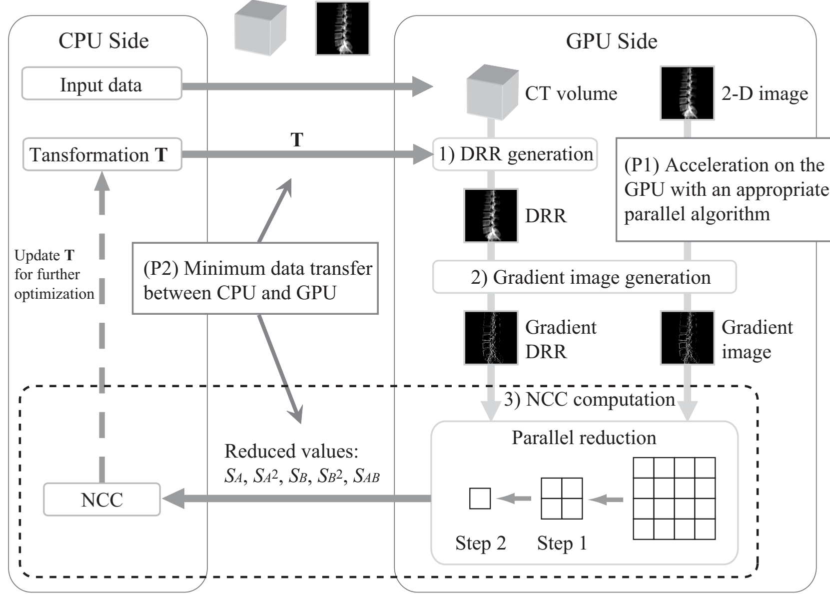

Fig. 3. Overview of 2-D/3-D registration on the GPU.

efficiency on the GPU. This method can be efficiently implemented on the GPU, because the texture-mapping and the alpha-blending procedures are hardware-accelerated on the GPU. Thus, our implementation for DRR generation satisfies point (P1-a).

Our method differs from LaRose's method in using 3-D textures instead of 2-D textures. As compared with 3-D textures, 2-D textures cannot produce higher quality DRRs, because 2-D textures cannot always be perpendicular to the view direction.

2) Gradient image generation. We implement a two-pass 1-D filter to reduce the time complexity of the 2-D filter:

$$P_{x1}(x,y) = \sum_j e^{-j^2/2\sigma^2} p(x, y+j) \tag{5}$$

$$P_{x2}(x,y) = \sum_i \frac{-i}{2\pi\sigma^4} e^{-i^2/2\sigma^2} p(x+i, y) \tag{6}$$

$$P_{y1}(x,y) = \sum_j \frac{-j}{2\pi\sigma^4} e^{-j^2/2\sigma^2} p(x, y+j) \tag{7}$$

$$P_{y2}(x,y) = \sum_i e^{-i^2/2\sigma^2} p(x+i, y) \tag{8}$$

This filter has fully data-parallelism, so that FPs in the GPU are allowed to simultaneously process different pixels in the image. Furthermore, vectorization can be applied to Eqs. (5) and (7) (Eqs. (6) and (8), also), because these computations (1) have no data dependence between them and (2) require pixels on the same location $p(x, y+j)$.

Therefore, the horizontal gradient image and the vertical gradient image can be produced simultaneously by vectorization. To enable this, we use two of four (RGBA) components to process the 1-D filters at the same time. Our vectorization can be represented as follows:

$$\boldsymbol{P}_{xy1}(x, y) = \sum_{j} \left[\begin{pmatrix} e^{-j^2/2\sigma^2} \\ \dfrac{-j}{2\pi\sigma^4} e^{-j^2/2\sigma^2} \end{pmatrix} * \boldsymbol{p}(x, y + j) \right] \tag{9}$$

$$\boldsymbol{P}_{xy2}(x, y) = \sum_{i} \left[\begin{pmatrix} \dfrac{-i}{2\pi\sigma^4} e^{-i^2/2\sigma^2} \\ e^{-i^2/2\sigma^2} \end{pmatrix} * \boldsymbol{p}(x + i, y) \right] \tag{10}$$

where $\boldsymbol{P}_{xy1}$ and $\boldsymbol{P}_{xy2}$ represent two-component data containing pixels of gradient images after applying the first-pass filter and the second-pass filter, respectively, and $\boldsymbol{p}$ represent a vectorized pixel. Thus, our implementation for gradient image generation satisfies point (P1-b).

NCC computation. Finally, Eq. (2) indicates that there is no data-parallelism in NCC computation, because pixel values are merged into a single value (NCC). Therefore, a naive method may process this procedure on the CPU. However, this is not recommended from the viewpoint of (P2). That is, if the CPU takes the responsibility for NCC computation, we have to transfer the DRR from the GPU to the CPU at every optimization step. This communication may result in a lower performance, because the DRR is 2-D data. To tackle this problem, we decompose the computation into two parts: reduction operations on the GPU and the remaining operations on the CPU. This allows us to transfer only five floating point numbers, S_A, S_{A^2}, S_B, S_{B^2}, and S_{AB}, instead of the 2-D DRR. Then, the CPU computes NCC using these numbers according to Eq. (2). Thus, although parallelization cannot be applied to the entire computation, we can parallelize reduction operations with reducing the amount of communication between the CPU and the GPU.

Given an image of $n \times n$ pixels, the parallel reduction can be done in at most $\log n$ steps, as shown in Fig. 3. Although this sequence of steps must be serially processed, each step can be parallelized according to point (P1-b). In our current implementation, we have empirically determined that each of FPs merges nine pixels into a single pixel at a step. Furthermore, we apply vectorization to reduction operations. That is, four of the five sums are computed at the same time.

Note here that Chisu also have presented parallel reduction in [11]. However, this method may suffer in computational (round-off) error, because it uses mipmap textures to compute averages of pixels at each step. Since this error increases with the number of steps, the amount of communication cannot be reduced into optimal five numbers in most cases. Thus, their method has a tradeoff relation between the communication amount and the computational error. In contrast, our method computes sums of pixels, preventing computational error. Therefore, the DRR is fully reduced at the GPU side without any errors.

According to the designs mentioned above, we have implemented the method using the C++ language, the OpenGL library [17], and the Cg (C for graphics) toolkit [21]. Fig. 4 and Fig. 5 show an overview of the CPU and GPU programs implemented

```
void runReduction(Region &region, // Region for drawing
      TextureObject *rtexture, // Initialized texture object
      RenderTexturePBuffer *pbuffer, // Initialized pixel buffer
      BufferSpecifier &rtextureBuffer, // Input buffer, namely gradient images
      BufferSpecifier &drawBuffer) // Output buffer
{
      cgGLEnableProfile(vertexProfile and fragmentProfile);
      cgGLBindProgram(vertexProgram and fragmentProgram); // See Fig. 5
      glDrawBuffer(drawBuffer); // Specify output buffer
      rtexture->bindTexture(); // Bind texture
      pbuffer->bindTexImage(rtextureBuffer); // Bind input buffer
      glClear(GL_COLOR_BUFFER_BIT); // Clear output buffer
      glRecti(region); // Draw specified region
      glFlush(); // Flush issued OpenGL commands
      pbuffer->releaseTexImage(rtextureBuffer);
      cgGLDisableProfile(vertexProfile and fragmentProfile);
}
```

Fig. 4. CPU program for parallel reduction.

for the parallel reduction procedure. Basically, the remaining procedures also can be implemented in the same way.

In this example program, 3×3 neighbor pixels are merged into a single pixel. Before performing rendering operations, the CPU program in Fig. 4 binds a vertex program and a fragment program, which express how a pixel should be computed through the rendering pipeline. For example, the vertex program in Fig. 5(a) computes the coordinates of neighbors in order to pass them to FPs. Then, the fragment program in Fig. 5(b) receives the coordinates from VPs and fetches the corresponding pixels to reduce them into a pixel. These rendering operations are activated by glRecti() in the CPU program and are terminated by glFlush(). After this, glReadPixels() is called to transfer computation results from the video memory to the main memory.

5 Experimental Results

To evaluate the usability of our GPU-based method, we compare it with a cluster-based method [9] in terms of the registration time and the target registration error (TRE) [23]. The GPUs employed for experiments are summarized in Table 1. On the other hand, the cluster consists of 32 PCs each with a Pentium 3 1-GHz CPU. PCs are interconnected by a Myrinet 2000 switch [24], which yields a bandwidth of 2 Gb/s. Note here that the cluster-based method employ a ray casting method for DRR generation. Therefore, the base algorithm is slightly different from our GPU-based method.

Registration is performed using two datasets: the real spine and the femur phantom (see Fig. 6). Because our experiments focus on the comparison of implementation methods, we use DRRs instead of fluoroscopy images. That is, we first generate a DRR from a viewing point, and then use the DRR as an input image I_F to estimate the point from a randomly selected point.

```
// Data structure for passing coordinates data from VPs to FPs
struct ReductionCoords {
    float4 position : POSITION; float2 coord0 : TEXCOORD0; float2 coord1 : TEXCOORD1;
    float2 coord2 : TEXCOORD2; float2 coord3 : TEXCOORD3; float2 coord4 : TEXCOORD4;
    float2 coord5 : TEXCOORD5; float2 coord6 : TEXCOORD6; float2 coord7 : TEXCOORD7;
};
// Coordinates computation for parallel reduction of 3×3 pixels
ReductionCoords reductionVertex9(float4 position : POSITION, // Vertex coordinates in range [0.33, 0.66]
    uniform float4x4 modelViewProjMatrix : state.matrix.mvp) // Transformation matrix
{
    ReductionCoords output;
    output.position = mul(modelViewProjMatrix, position); // Update volume position
    output.coord0 = position.xy * 3 - 1.0f; // Adjust output range [0.33, 0.66] to input range [0, 1]
    output.coord1 = output.coord0 + float2(1.0, 0.0); output.coord2 = output.coord0 + float2(0.0, 1.0);
    output.coord3 = output.coord0 + float2(1.0, 1.0); output.coord4 = output.coord0 + float2(0.0, 2.0);
    output.coord5 = output.coord0 + float2(2.0, 0.0); output.coord6 = output.coord0 + float2(1.0, 2.0);
    output.coord7 = output.coord0 + float2(2.0, 1.0);
    // Address for pixel (2, 2) cannot be precomputed due to limited number of VP registers
    return output;
}
```

(a)

```
// Parallel reduction of 3×3 pixels
float3 reductionSum9RGB(ReductionCoords input,
    uniform samplerRECT sampRect : TEXUNIT0) : COLOR
{
    float2 coord8 = input.coord0 + float2(2.0, 2.0); // Address for pixel (2, 2)
    float3 output = texRECT(sampRect, input.coord0).rgb; // Fetch pixel (0, 0)
    output += texRECT(sampRect, input.coord1).rgb; output += texRECT(sampRect, input.coord2).rgb;
    output += texRECT(sampRect, input.coord3).rgb; output += texRECT(sampRect, input.coord4).rgb;
    output += texRECT(sampRect, input.coord5).rgb; output += texRECT(sampRect, input.coord6).rgb;
    output += texRECT(sampRect, input.coord7).rgb;
    output += texRECT(sampRect, coord8).rgb; // Reduction for (2, 2)
    return output;
}
```

(b)

Fig. 5. GPU programs for parallel reduction. (a) Vertex program and (b) fragment program reduce 3×3 neighbor pixels $(0,0) - (2,2)$ into a single pixel $(0,0)$. Except for $(2,2)$, all of coordinates addresses are precomputed by VPs instead of by FPs in order to reduce computational amount. See [22] for details.

Table 2 shows the timing results with breakdowns. Our GPU-based method completes a registration task within 10 seconds, which is permissible time for surgical assistances. We can also see that the method successfully reduces the time for DRR and gradient image generation, as compared with the sequential method (C_S). It also demonstrates further acceleration against the cluster-based method (C_P).

The parallel reduction for NCC computation reduces the amount of communication from 56 KB to 20 B. This reduction effect is significant for the laptop PC platform G_1, because such platforms do not have high-speed graphics bus between the CPU and the GPU. Even in desktop platforms, the data transfer time is reduced from 9 ms to 0.2 ms.

By comparing G_2 with G_3, we can see that the growth of GPU speed. That is, the current generation G_3 achieves almost double performance as compared with the previous generation G_2. This result is reasonable because GPU performance has doubled every six months [5].

We also investigate the robustness of our GPU-based method. Table 3 summarizes the alignment results. We repeat registration tasks ten times with different initial points.

Table 1. Experimental environment. Notations C_S and C_P are serial and parallel CPU environment, respectively. The remaining are GPU environments: G_1 is a laptop PC; G_2 and G_3 are desktop PCs with a previous generation GPU and a current generation GPU, respectively.

Environment	C_S	C_P	G_1	G_2	G_3
# of nodes	1	32	1		
CPU	Pentium 4 2.8 GHz	Pentium 3 1.0 GHz	Pentium M 2.0 GHz	Pentium 4 2.8 GHz	Pentium 4 3.4 GHz
GPU	—		Quadro FX Go 1400	Quadro FX 3400	GeForce 7800 GTX
Core clock (MHz)			125	350	430
Memory clock (MHz)			332	900	1200
Memory bandwidth (GB/s)			19.4	28.8	38.4
Fill rate (Gpixels/s)			2.2	4.2	10.3
Network	—	Myrinet 2000	—		

Table 2. Timing results. Total time is the product of time per step and 300 steps.

Breakdown	Real spine V: 512 × 512 × 204 voxels ROI: 300 × 300 × 48 voxels I_F: 300 × 200 pixels					Femur phantom V: 256 × 256 × 367 voxels ROI: 54 × 38 × 55 voxels I_F: 300 × 200 pixels				
	C_S	C_P	G_1	G_2	G_3	C_S	C_P	G_1	G_2	G_3
DRR generation	2940	142	38.6	26.4	17.1	810	31	26.7	14.1	5.8
Gradient image	142	7	8.4	5.2	1.7	197	7	8.4	5.3	1.7
NCC computation	9	46	57.6	5.3	2.7	3	14	55.7	5.3	2.7
Reduction	—	—	7.9	4.9	2.4	—	—	7.9	4.9	2.4
Data transfer	—	—	49.6	0.2	0.2	—	—	47.7	0.2	0.2
Time per step (ms)	3091	195	104.6	36.9	21.5	1010	52	90.8	24.7	10.2
Total steps	300									
Total time	15 m	58 s	31 s	11 s	6 s	5 m	15 s	27 s	7 s	3 s

The registration is regarded as successful if the final TRE is less than 0.5 voxels, namely 0.66 mm for the spine data and 1.56 mm for the femur data.

As we can see in Table 3, our GPU-based method returns successful results if the initial TRE is less than 10 mm. There is no significant difference between the cluster-based method and the GPU-based method in terms of robustness. The results also show that using biplane images instead of a single image is necessary to obtain precise alignments in the depth direction.

Finally, we compare our method with a mipmap-based method [11], which we mentioned in Section 4. Table 4 summarizes the registration results. By comparing this table with Table 3, we can see that our method provides more robust results against the mipmap-based method. Furthermore, the registration performance of the mipmap-based method is almost the same as that of our method. Thus, we think that mipmap textures are not suited to parallel reduction due to computational error.

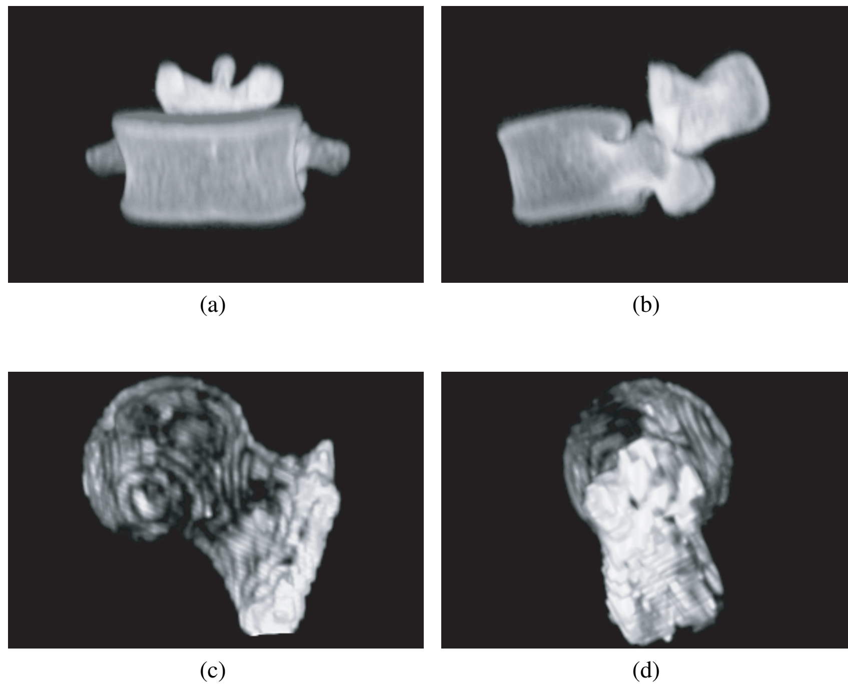

Fig. 6. Biplane images of real spine and femur phantom datasets.

6 Conclusion

We have presented a fast 2-D/3-D registration method for biplane images using a GPGPU approach. Our method reduces registration time by eliminating performance bottlenecks on the CPU: DRR generation; gradient image generation; and NCC computation. Our method performs reduction operations on the GPU in order to minimize the amount of communication between the CPU and the GPU.

The experimental results show that our GPU-based method successfully completes a registration task in ten seconds. This timing result on the GPU is faster than that on the 32-node cluster. With respect to registration errors, the method demonstrates similar results as compared with CPU implementations. Thus, we think that our GPU-based method is useful for computer-aided surgery in terms of performance and robustness. As compared with the cluster-based method, we also think that the GPU-based method provides more attractive solution to medical doctors, because it needs less maintenance cost and less power consumption with higher fault tolerance.

Acknowledgments. This work was partly supported by JSPS Grant-in-Aid for Scientific Research on Priority Areas (17032007) and Scientific Research (B)(2)(18300009).

Table 3. Robustness results in terms of TRE (mm). Registration is done ten times for each. "Pass" represents the number of successful registration in ten trials.

Initial	Real spine						Femur phantom					
TRE	GPU single		GPU biplane		CPU biplane		GPU single		GPU biplane		CPU biplane	
(mm)	TRE	Pass	TRE	Pass	TRE	Pass	TRE	Pass	TRE	Pass	TRE	Pass
2– 4	2.37	4	0.10	10	0.27	10	7.05	0	0.53	10	1.13	8
4– 6	3.38	2	0.13	10	0.25	10	5.49	1	0.45	10	1.63	6
6– 8	3.98	3	0.09	10	0.24	10	6.04	1	0.51	10	3.27	2
8–10	6.84	0	0.09	10	1.70	9	7.18	0	0.94	9	12.71	1
10–12	5.10	3	1.46	8	3.12	8	8.48	0	0.68	9	9.34	0
12–14	7.48	0	1.11	7	8.92	4	6.63	0	0.73	9	11.19	1
14–16	12.41	1	5.87	4	7.25	5	6.91	0	1.86	9	15.18	1
16–18	13.73	2	7.09	4	13.73	2	7.09	0	5.37	5	16.46	0
18–20	19.88	0	11.13	2	13.17	3	7.19	0	4.83	5	22.63	0
20–22	10.84	1	10.71	2	18.48	1	8.72	0	6.58	5	22.23	0

Table 4. Robustness results obtained by the mipmap-based method [11]. In this experiment, we perform biplane registration with GPU-based DRR generation for the mipmap-based method, which originally performs single-image registration using the CPU-based DRR generation.

Intial	Real spine		Femur phantom	
TRE	TRE	Pass	TRE	Pass
2– 4	2.09	2	1.22	10
4– 6	1.85	1	1.17	10
6– 8	3.37	1	6.12	5
8–10	6.20	0	9.86	1
10–12	10.74	0	11.98	0
12–14	8.24	0	11.27	0
14–16	19.47	0	15.96	0
16–18	19.39	0	14.50	0
18–20	16.98	0	19.31	0
20–22	20.18	0	21.04	0

References

1. Lemieux, L., Jagoe, R., Fish, D.R., Kitchen, N.D., Thomas, D.G.T.: A patient-to-computed-tomography image registration method based on digitally reconstructed radiographs. Medical Physics **21**(11) (1994) 1749–1760
2. Hajnal, J.V., Hill, D.L., Hawkes, D.J., eds.: Medical Image Registration. CRC Press, Boca Raton, FL (2001)
3. Moore, G.E.: Cramming more components onto integrated circuits. Electronics **38**(8) (1965) 114–117
4. Montrym, J., Moreton, H.: The GeForce 6800. IEEE Micro **25**(2) (2005) 41–51
5. Fan, Z., Qiu, F., Kaufman, A., Yoakum-Stover, S.: GPU cluster for high performance computing. In: Proc. Int'l Conf. High Performance Computing, Networking and Storage (SC'04). (2004)

6. Galoppo, N., Govindaraju, N.K., Henson, M., Manocha, D.: LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware. In: Proc. Int'l Conf. High Performance Computing, Networking, Storage and Analysis (SC'05). (2005)
7. Takizawa, H., Kobayashi, H.: Multi-grain parallel processing of data-clustering on programmable graphics hardware. In: Proc. 2nd Int'l Symp. Parallel and Distributed Processing and Applications (ISPA'04). (2004) 16–27
8. GPGPU: General-Purpose Computation Using Graphics Hardware (2005) `http://www.gpgpu.org/`.
9. Ino, F., Kawasaki, Y., Tashiro, T., Nakajima, Y., Sato, Y., Tamura, S., Hagihara, K.: A parallel implementation of 2-d/3-d image registration for computer-assisted surgery. In: Proc. 11th Int'l Conf. Parallel and Distributed Systems (ICPADS'05), Volume II Workshops. (2005) 316–320
10. LaRose, D.A.: Iterative X-Ray/CT Registration Using Accelerated Volume Rendering. PhD thesis, Carnegie Mellon University, Pittsburgh, PA (2001)
11. Chisu, R.: Techniques for Accelerating Intensity-based Rigid Image Registration. PhD thesis, Technische Universität München, München, Germany (2005)
12. Li, S., Pelizzari, C.A., Chen, G.T.Y.: Unfolding patient motion with biplane radiographs. Medical Physics **21**(9) (1994) 1427–1433
13. Penney, G.P., Weese, J., Little, J.A., Desmedt, P., Hill, D.L.G., Hawkes, D.J.: A comparison of similarity measures for use in 2-D–3-D medical image registration. IEEE Trans. Medical Imaging **17**(4) (1998) 586–595
14. Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P.: NUMERICAL RECIPES in C: The Art of Scientific Computing. Cambridge University Press, Cambridge, UK (1988)
15. Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A.E., Purcell, T.J.: A survey of general-purpose computation on graphics hardware. In: EUROGRAPHICS 2005, State of the Art Report. (2005) 21–51
16. Khailany, B., Dally, W.J., Kapasi, U.J., Mattson, P., Namkoong, J., Owens, J.D., Towles, B., Chang, A., Rixner, S.: Imagine: Media processing with streams. IEEE Micro **21**(2) (2001) 35–46
17. Shreiner, D., Woo, M., Neider, J., Davis, T.: OpenGL Programming Guide. fourth edn. Addison-Wesley, Reading, MA (2003)
18. Hillesland, K.E., Lastra, A.: GPU floating point paranoia. In: Proc. 1st ACM Workshop General-Purpose Computing on Graphics Processors (GP^2'04). (2004) C–8
19. Stevenson, D.: A proposed standard for binary floating-point arithmetic. IEEE Computer **14**(3) (1981) 51–62
20. Cullip, T.J., Neumann, U.: Accelerating volume reconstruction with 3D texture hardware. Technical Report TR93-027, University of North Carolina at Chapel Hill (1993)
21. Mark, W.R., Glanville, R.S., Akeley, K., Kilgard, M.J.: Cg: A system for programming graphics hardware in a C-like language. ACM Trans. Graphics **22**(3) (2003) 896–897
22. Ikeda, T., Ino, F., Hagihara, K.: A code motion technique for accelerating general-purpose computation on the GPU. In: Proc. 20th IEEE Int'l Parallel and Distributed Processing Symp. (IPDPS'06). (2006) 10 pages (CD-ROM).
23. Fitzpatrick, J.M., West, J.B., Maurer, C.R.: Predicting error in rigid-body point-based registration. IEEE Trans. Medical Imaging **17**(5) (1998) 694–702
24. Boden, N.J., Cohen, D., Felderman, R.E., Kulawik, A.E., Seitz, C.L., Seizovic, J.N., Su, W.K.: Myrinet: A gigabit-per-second local area network. IEEE Micro **15**(1) (1995) 29–36