Fast Deformable Registration on the GPU: A CUDA Implementation of Demons

Pınar Muyan-Özçelik, John D. Owens
University of California, Davis
One Shields Avenue, Davis, CA 95616
{pmuyan, jowens}@ucdavis.edu

Junyi Xia, Sanjiv S. Samant
University of Florida
Gainesville, FL 32611
{junyixia, samant}@ufl.edu

Abstract

In the medical imaging field, we need fast deformable registration methods especially in intra-operative settings characterized by their time-critical applications. Image registration studies which are based on Graphics Processing Units (GPUs) provide fast implementations. However, only a small number of these GPU-based studies concentrate on deformable registration. We implemented Demons, a widely used deformable image registration algorithm, on NVIDIA's Quadro FX 5600 GPU with the Compute Unified Device Architecture (CUDA) programming environment. Using our code, we registered 3D CT lung images of patients. Our results show that we achieved the fastest runtime among the available GPU-based Demons implementations. Additionally, regardless of the given dataset size, we provided a factor of 55 speedup over an optimized CPU-based implementation. Hence, this study addresses the need for on-line deformable registration methods in intra-operative settings by providing the fastest and most scalable Demons implementation available to date. In addition, it provides an implementation of a deformable registration algorithm on a GPU, an understudied type of registration in the general-purpose computation on graphics processors (GPGPU) community.

1 Introduction

Image registration is the process of aligning images so that corresponding features can easily be related [5]. In the medical field, professionals use registration to aid in many critical tasks such as diagnosis, planning radiotherapy/surgery, patient positioning prior to radiotherapy/surgery, dose delivery verification, performing image-guided surgery, monitoring disease progression or response to treatment, and atlas-based registration.

Registration aligns images by applying transformations to one of the images so that it matches the other. Registration techniques can be divided into rigid registrations and deformable registrations. Rigid registration methods only allow rigid transformations. On the other hand, deformable registration methods apply techniques such as elastic transformations to correct deformations that rigid methods are not sufficient to correct.

Medical imaging applications use different deformable registration methods [18]. Among these methods, developers widely use Demons deformable registration technique proposed by Thirion [31] due to its proved effectiveness, simplicity, and computational efficiency. Demons takes two images as inputs and produces the displacement field which indicates the transformations that should be applied to pixels of one of the images so that it can be aligned with the other as shown in Figure 1.
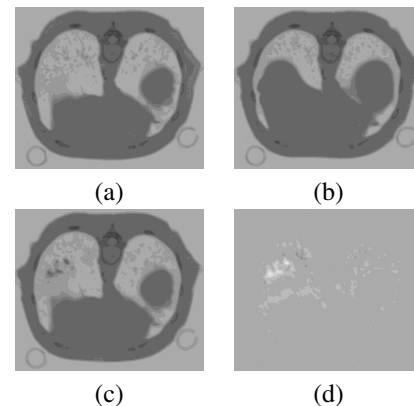

Figure 1. Registering images with Demons: (a) original lung image of a patient, (b) same image after a deformation, (c) corrected image after applying Demons, and (d) difference image of the original and corrected image (helps to verify visually the quality of the match).

While deformable registration methods have broad applicability in intra-operative settings, the long CPU runtimes of current deformable registration techniques do not

allow for time-critical intra-operative applications, which typically require only few seconds as a maximum response time. Hence, developers of time-sensitive medical applications need techniques that considerably reduce the runtime of deformable registration.

Graphics Processing Unit (GPU) computing presents one of the techniques that can address this need. The GPU's substantial arithmetic and memory bandwidth capabilities, coupled with its recent addition of user programmability, has allowed for "general-purpose computation on graphics hardware" (GPGPU) [4]. Many non-graphics-oriented computationally expensive algorithms have been implemented on the GPU. Developers prefer GPUs over other alternative parallel processors due to several advantages including their low cost and wide availability. Owens et al. presented a comprehensive survey of latest research in GPU computing [23]. Since medical imaging applications intrinsically have data-level parallelism with high compute requirements and most of them requires visual interaction, they are very suitable to be implemented on the GPU. As will be explained in Section 3, researchers have conducted several prior GPGPU studies in the field of image registration. However, only a limited number of previous studies concentrate on deformable registration.

In this study, we implemented Demons deformable registration algorithm, on NVIDIA's Quadro FX 5600 GPU with the Compute Unified Device Architecture (CUDA) programming environment. The Demons algorithm, though not necessarily the fastest and most robust deformable registration method in all clinical cases, is widely available and serves as a standard for comparison due to its simplicity. Hence, we have chosen to implement Demons and we hope that other GPGPU researchers in the field will also make the same choice to standardize the performance comparisons. The specific technical contributions of our paper are three-fold:

1) It addresses the need for faster intra-operative deformable registration algorithms by presenting the fastest GPU-based implementation of Demons algorithm: To the best of our knowledge, Sharp et al. [27] reported the fastest implementation using the Brook [1] programming environment. We improved this speed using CUDA [22] and achieved 10% faster runtime.

2) It provides a scalable implementation by achieving speedup which is constant across datasets with different sizes: Independent of the size of the dataset, our CUDA implementation of Demons running on the GPU is 55 times faster than the single-threaded, optimized C implementation running on a CPU. Other implementations either reported speedup for only one dataset or they reported smaller speedups for larger dataset sizes.

3) It concentrates on implementing a deformable registration algorithm exclusively on a GPU, an understudied

type of registration in the GPGPU community.

2 Background

In this section, we provide expanded explanations of some of the concepts we introduced in Section 1. First, we talk about the differences between rigid and deformable registration. Next, we give examples of intra-operative applications that can take advantage of fast deformable registration methods. At the end, we list benefits of using deformable registration in such applications.

Registration techniques are divided into two groups: 1) rigid registration methods, which only allow rigid transformations such as translation, rotation, etc., 2) deformable registration methods which apply techniques such as elastic transformations to register images that rigid methods are not sufficient to register. To minimize the discrepancy between the original image and the image with deformations, rigid registration should be performed before applying any deformable registration technique. Since rigid registration methods are less complex than deformable ones, rigidly aligning two images before performing non-rigid transformation reduces the time needed to non-rigidly align these images.

In addition to composing the preliminary step for deformable registration, rigid registration methods are used in many other vital clinical applications. For instance, they can be used for positioning a brain tumor patient prior to commencing radiotherapy. Positioning a patient has a crucial role in the effectiveness of therapy, since it arranges the location of the patient's couch and the angle of radiation beam in a way such that healthy tissues would be minimally affected while the tumor is exposed to maximal radiation. In the case of a brain tumor patient, since the skull has a rigid structure, aligning of intersubject brain images can be done by rigid transformations. Thus, positioning a patient, which involves aligning pre-operative and intra-operative brain images of the same subject, can be performed by rigid registration.

In many cases rigid transformations are insufficient for aligning images. As an example, various factors such as respiratory or cardiac motion, tumor growth or shrinkage, and weight gain or loss cause deformations in the images, which can only be matched by non-rigid registration techniques. Similarly, atlas-based registration would require non-rigid transformations. In such cases, a deformable registration method such as Demons is needed to align images.

In addition, some of the critical intra-operative application can be significantly improved by the use of deformable registration methods. These applications are characterized as time-sensitive and can only tolerate few seconds as a maximum response time. However, depending on the technique, image size, and stop criteria, CPU runtimes of de-

formable registration methods range between a couple of minutes to several hours. This is because, unlike rigid registration techniques, deformable registration methods compute the transformations on a pixel by pixel basis, which is computationally very expensive. Hence, unless their runtimes are reduced to a few seconds, deformable registration methods can not be clinically implemented in real-time.

If we could reduce the deformable registration runtime, examples of clinical applications that can take advantage of real-time deformable registration include Adaptive Radiation Therapy (ART) [11] and image-guided surgery [7, 29]. ART improves radiation therapy by modifying a treatment plan based on daily changes of patient anatomy. Changes in patient anatomy can occur during the course of radiotherapy due to the facts such as change in the sizes and shapes of tumor and internal organ motion. If a fast deformable registration algorithm is available, ART can take these changes into account in real-time and modify a treatment plan accordingly, before commencing daily therapy.

In image-guided surgery, intra-operative and pre-operative images are registered before the intervention in order to position the medical equipment and the patient so that the surgery can start as it is planned. During the intervention, however, soft tissues such as brain tend to deform. This fact causes intra-operative and pre-operative images to be mis-aligned and requires adjustments to the surgery plan at important stages of the intervention. Similar to ART, if a fast deformable registration algorithm is available, the plan can be modified in real-time by determining the deformation of the tissue and updating navigation of medical equipment accordingly.

Thus, making deformable registration available in operation rooms by reducing the runtime of these methods to few seconds would improve the accuracy and efficacy of the intra-operative applications such as ART and image-guided surgery. Hence, techniques such as GPGPU that would make the runtime of deformable registration methods compatible with time-critical intra-operative applications are strongly desired in the medical field.

3 Previous Work

In this section, we present the previous research on GPU-based image registration techniques. The current literature has not paid sufficient attention to GPGPU use in deformable methods. Instead, GPUs have been mainly used in Digitally Reconstructed Radiograph (DRR) generation to speed up the rigid registration process. We structure this section by first providing the interested reader with survey papers on image registration techniques, then we present GPU-based image registration techniques by specifically focusing on deformable methods. Finally, we discuss three studies that implemented Demons algorithm on a GPU and

highlight the contributions of our study.

Image registration is an integral part of many vital clinical tasks including diagnosis, radiotherapy/surgery planning, patient positioning, image-guided surgery, and evaluation of radiotherapeutical/surgical procedures. These tasks require performing intrasubject registration which involves aligning multiple images from the same individual. In addition, by performing intersubject registration, images acquired from different patients can be aligned to compare different individuals or to do an atlas-based registration.

Image registration techniques which are used in the medical field were reviewed by Maintz and Viergever [21]. Lester and Arridge [18] also surveyed medical image registration approaches by specifically focusing on deformable methods. In addition to medicine, image registration is also used in different fields such as weather forecasting, environmental monitoring, image mosaicing, map updating, computer vision, etc. Zitova and Flusser [36] provided a comprehensive survey of methods used in image registration in a variety of application areas.

Due to their high parallelism, image registration tasks are computationally very expensive. Hence, GPUs with their high-performance parallel processing power provide great opportunities for speeding up these tasks. As mentioned by Khamene et al. [13], in the domain of medical image registration, GPUs have been mainly utilized to generate DRRs using hardware-accelerated volume rendering techniques. Indeed, in the literature, we see many studies which followed GPU-based DRR generation approaches to speed up the 2D/3D rigid registration process [9,12,15,17,25,26].

Deformable registration algorithms applied to large 3D data sets put an even heavier burden on computational resources. However, as indicated by Vetter et al. [32], there has not been much research into the use of GPU in non-rigid registration methods. Likewise, Sharp et al. [27] do not consider prior work on the GPU-based deformable registration as "well established". In this domain, the research of Soza et al. [28] presents one of the early studies. They used 3D Bézier functions for non-rigid alignment of respective volumes and utilized the trilinear interpolation capabilities of the GPU to accelerate the deformation of the moving volume. In order to better understand the brain shift phenomenon, Hastreiter et al. [7] performed deformable voxel-based registration and, like Soza et al. [28], they also accelerated trilinear interpolation by using the 3D texture mapping capabilities of graphics hardware. In this study, the authors applied piecewise linear transformations proposed in an earlier study [24] to approximate nonlinear deformations instead of using Bézier splines. Levin et al. [19] exploited graphics hardware to implement a high-performance Thin Plate Spline (TPS) volume warping algorithm that could be used in iterative image registration and accelerated the application of the TPS nonlinear transformation by combin-

ing hardware-accelerated 3D textures, vertex shaders, and trilinear interpolation.

Strzodka et al. [30] implemented a non-rigid regularized gradient flow registration introduced by Clarenz et al. [2] on the GPU to match 2D images. Köhn et al. [16] extended regularized gradient flow to 3D and in contrast to earlier studies that used graphics hardware only for the trilinear interpolation (e.g. Soza et al.'s study [28]), they based their entire implementation on a GPU. However, due to memory bottlenecks, they reported that their 3D non-rigid registration is not as fast as one would expect. Vetter et al. [32] implemented non-rigid registration on a GPU using mutual information and the Kullback-Leibler divergence between observed and learned joint intensity distributions. They proposed this method for specifically matching multi-modal data sets and, like Köhn et al. [16], they implemented the entire registration process on the GPU.

To the best to our knowledge, there are only three studies which mapped Demons registration to the GPU: implementation of Kim et al. [14], Courty and Hellier [3], and Sharp et al. [27]. All these three studies were conducted concurrently and independently with our study and differ mainly on the method they used for smoothing the displacement field. These implementations also used different GPUs and programming environments.

Kim et al. implemented Demons using a simple ramp smoothing. They took the average of the closest six neighbors of each voxel to smooth the displacement field, using an NVIDIA GeForce 6800 GPU and the Cg language. Courty and Hellier presented a GPU implementation of the Demons algorithm using a Gaussian recursive filtering. The advantage of recursive filtering is that number of operations is bounded and independent of the standard deviation of the Gaussian filter. In order to implement recursive filtering, they approximated the Gaussian filter with 4th-order cosines-exponential functions. The authors mentioned that this approximation is fair for Gaussian filters with standard deviation lower than 10. In this implementation, they chose an NVIDIA Quadro FX 1400 GPU, with fragment programs written with a Cg-like syntax. Finally, Sharp et al. implemented Demons algorithm using a separable Gaussian filter. They used the Brook programming environment and an NVIDIA GeForce 8800 GTS GPU.

Using a Gaussian filter for smoothing the displacement field provides the most accurate implementation. However, as will be explained more in detail in Section 5, this is the most expensive part of Demons algorithm. Hence, to achieve faster runtimes some of the researchers simplified or approximated the smoothing process. For instance, Kim et al. used simple ramp smoothing instead of Gaussian smoothing and Courty and Hellier approximated the Gaussian filter as discussed above. Sharp et al. was first to use a separable Gaussian filter. In addition, even if they did not simplify or approximate the smoothing process, they achieved the fastest runtime by using the Brook programming environment and a newer GPU than the other two studies.

In order to provide an accurate implementation, we have also used Gaussian smoothing and performed convolution of the displacement field with separable Gaussian filter like Sharp et al. did. However, we improved the speed of Sharp et al.'s implementation by using NVIDIA's CUDA environment instead of Brook; we report a 10% faster runtime on the same hardware. CUDA specifically targets newer NVIDIA cards (e.g. the GPUs used in Sharp et al.'s implementation and our study) and provides powerful features such as shared memory access, which we extensively used in our implementation, that Brook does not offer. Hence, CUDA is optimized for these cards and provides better support/performance than Brook. In addition, Brook is a legacy academic project and is at best maintenance-mode-only, whereas CUDA is directly supported by NVIDIA, is under active development, and has a broader set of programming tools and libraries available.

In addition to achieving a fast runtime, we obtained a constant speedups across datasets with different sizes. Kim et al. and Courty and Hellier reported speedup for only one dataset. Sharp et al. reported speedups for two datasets. However, they reported a smaller speedup for the larger dataset size. Hence, to the best to our knowledge, our implementation presents the fastest and most scalable GPU-based implementation of Demons algorithm available to date. We will talk more about our runtime and speedup in Section 5. In this section, in contrast to the above studies, which only present single-threaded CPU implementation runtimes, we will also provide multi-threaded CPU implementation runtimes, in order to present a fair comparison between CPU and GPU speeds. Additionally, we will provide a detailed analysis of our implementation by discussing the breakdown of the runtime into kernels and by presenting GFLOP-GiB information, which are not provided in other studies.

4 Implementation

In this section, first, we introduce the general scheme for Demons algorithm. Then, we introduce our GPU implementation environment by first discussing why GPUs are a good fit for medical imaging applications and then presenting NVIDIA's CUDA platform and GeForce 8800 GTX architecture. Next, we talk about the CPU implementation environment. This is followed by description of the test data used in the experiments. Finally, we provide the list of CUDA kernels used in our GPU implementation.

4.1 Demons Algorithm

The Demons algorithm, originally proposed by Thirion [31], is a deformable registration algorithm that is widely used to match medical volumes. Demons is based on the optical flow method [8] which is used to find small deformations in temporal sequences of volumes, often called static and moving volumes.

The optical flow method finds a displacement field that deforms the moving volume, M, so that it is matched with the static volume, S. The basic hypothesis of optical flow is that intensities are constant between M and S, which leads to the following optical flow equation for a given voxel location index, i:

$$\vec{v_i} \cdot \overrightarrow{\bigtriangledown} s_i = M(i) - S(i) \qquad (1)$$

where $\vec{v_i}$ denotes the displacement vector that should be applied to the voxel at location i_d in M, such that $i = i_d + \vec{v_i}$ and $M(i_d) = S(i)$. Here, $\overrightarrow{\bigtriangledown} s_i$ denotes the intensity gradient vector of S at location i. In addition, $M(i)$ and $S(i)$ are the intensity values of voxels at location i in M and S, respectively.

If we find $\vec{v_i}$ for each voxel, we would match M with S. However, since Equation 1 is underconstrained, it cannot be used to define $\vec{v_i}$. To deal with this problem, Thirion follows an iterative approach and proposes the Demons algorithm based on optical flow [31]. This iterative algorithm alternates between computation of additional displacement field and regularization of the total displacement field until convergence. The displacement field is composed of displacement vectors of all voxels. Let $\vec{v}$ denote the total displacement field. In addition, let D denote the deformed image that is produced by applying the total displacement field of the current iteration to the moving image M. The general scheme of the algorithm is shown in Algorithm 1.

1: Compute $\overrightarrow{\bigtriangledown} s_i$ for each voxel, i
2: Initialize $\vec{v}$ to zero
3: **repeat**
4:    **for** each voxel, i **do**
5:       Compute intensity value in deformed image:
          $D(i) = M(i_d)$ where $i_d = i - \vec{v_i}$
6:       Compute additional displacement vector:
          $\overrightarrow{v_{i\_add}} = \frac{(D(i)-S(i))\overrightarrow{\bigtriangledown} s_i}{(\overrightarrow{\bigtriangledown} s_i)^2 + (D(i)-S(i))^2}$
7:       Compute total displacement vector:
          $\vec{v_i} = \vec{v_i} + \overrightarrow{v_{i\_add}}$
8:    **end for**
9:    Regularize $\vec{v}$ by applying Gaussian smoothing
10: **until** D and S converges

Algorithm 1: Pseudo code of Demons algorithm.

$M(i_d)$ is estimated using trilinear interpolation in M. In addition, since for each voxel we initialize $\vec{v_i}$ to zero, D would be equal to M in the first iteration. On the other hand, D would be very close to S in the last iteration. This would approximate the numerator of $\overrightarrow{v_{i\_add}}$ for each voxel to zero. Hence, the additional displacement field gets closer to zero, and in the last iteration, $\vec{v}$ becomes stable. Although the exact number of iteration for D and S to converge is heavily dependent on the dataset, in most cases, 50 to 100 iterations are sufficient for convergence. In order to measure convergence, we compute the correlation coefficient between D and S at the end of each iteration. As we iterate, the two volumes converge and therefore, the coefficient gets closer to 1.

4.2 GPU implementation environment

We have implemented a GPU version of Demons with NVIDIA's new GPGPU programming environment, CUDA v0.9. The computer used for the GPU implementation has two dual-2.4 GHz AMD Opteron 8216s and 3.6 GB of main memory, and runs Linux distribution 2.6.21-gentoo-r3. This linux box is equipped with an NVIDIA Quadro FX 5600 graphics card (equivalent to the NVIDIA GeForce 8800 GTX GPU with more memory) with 1536 MB of video memory.

The era of single-threaded processor performance increases has come to an end. Programs will only increase in performance if they utilize parallelism. However, there are different kinds of parallelism. For instance, multicore CPUs provide task-level parallelism. On the other hand, GPUs provide data-level parallelism.

Depending on the application area, the type of the preferred parallelism might change. Hence, GPUs are not a good fit for all problems. However, medical imaging applications are very suitable to be implemented on a GPU architecture. It is because these applications intrinsically have data-level parallelism with high compute requirements, and GPUs provide the best cost-per-performance parallel architecture for implementing such algorithms. In addition, most medical imaging applications (e.g. semi-automatic segmentation) require, or benefit from, visual interaction and GPUs naturally provide this functionality.

Recently, GPU vendors have made general-purpose computation on GPU a centerpiece of their future strategy by providing GPGPU programming environments. Hence, the use of the GPU in non-graphics related highly-parallel applications, such as medical imaging applications, became much easier than before. For instance, NVIDIA introduced CUDA to perform data-parallel computations on the GPU without the need of mapping to the graphics API. Since it is cumbersome to use graphics APIs for non-graphics tasks such as medical applications, the graphics-centric nature

of previous environments (e.g. Cg, GLSL, HLSL) makes GPGPU programming more complicated than it needs to be. CUDA makes GPGPU programming easier by hiding GPU-specific details and allowing the programmer to think in terms of memory and math operations as in CPU programs, instead of primitives, fragments, and textures that are specific to graphics programs.

CUDA is available for the NVIDIA GeForce 8800 (G80) Series and beyond. The GeForce 8800 GTX has 16 multiprocessors with 8 processors each, hence, 128 processors in total. Each of these 128 processors can sustain floating point multiply and add per cycle, yielding a measured rate of 330 GFLOPS of computation. The memory bandwidth of the GeForce 8800 GTX is 80+ GB/s. To get the best performance from G80 architecture, we have to keep 128 processors occupied and hide memory latency. In order to achieve this goal, CUDA runs hundreds or thousands of fine, lightweight threads in parallel.

In CUDA, programs are expressed as kernels. Kernels have a Single Program Multiple Data (SPMD) programming model, which is essentially Single Instruction Multiple Data (SIMD) programming model that allows limited divergence in execution. A part of the application that is executed many times, but independently on different elements of a dataset, can be isolated into a kernel that is executed on the GPU in the form of many different threads. Kernels run on a grid, which is an array of blocks; and each block is an array of threads. Blocks are mapped to multiprocessors within the G80 architecture, and each thread is mapped to single processor. Threads within a block can share memory on a multiprocessor. But two threads from two different blocks cannot cooperate. The GPU hardware performs switching of threads on multiprocessors to keep processors busy and hide memory latency. Thus, thousands of threads can be in flight at the same time, and CUDA kernels are executed on all elements of the dataset in parallel. We would like to mention that in our implementation, increasing the dataset size does not have an effect on the shared memory usage. This is because, to deal with larger datasets, we only increase the number of blocks and keep the shared memory allocations in a thread as well as the number of threads in a block the same.

4.3 CPU implementation environment

The CPU version of Demons is implemented in C. The computer used for the CPU implementation is a Dell XPS 710 with dual-core 2.4 GHz Intel Core2 6600s. It runs Windows XP SP2 and has 3 GB of main memory. The CPU implementation was executed on this box with both single-threading mode and multi-threading mode. OpenMP is used to implement the multi-threading part.

4.4 Test data

Our test data consists of 3D Computed Tomography (CT) lung images of three patients. Each patient is represented with two CT volumes, which correspond to static and moving volumes of the patient. These volumes were obtained based on Four-Dimensional CT (4DCT). 4DCT acquires a sequence of three-dimensional CT datasets over ten consecutive phases of a breathing cycle. The end of the inhale phase and the end of the exhale phase are used as static and moving volumes, respectively. The dimensions of static and moving volumes are the same within each patient, and they are different between the three patients. We have chosen these patients such that their dataset sizes are significantly different from each other. As can be seen from Table 1, they provide a good range of possible input sizes and allow us to measure the performance when small (e.g. Dataset 1), medium (e.g. Dataset 2), and large (e.g. Dataset 3) datasets are used.

4.5 CUDA kernels

As explained in Section 4.2, in order to map an algorithm to the CUDA programming environment, developers should identify data-parallel portions of the application and isolate them as CUDA kernels. Hence, in order to implement Demons with CUDA, we identified parts of Algorithm 1 that are executed many times, but independently on different elements of the dataset. Then, we implemented these parts as CUDA kernels and grouped them into five categories:

1) *Gradient*: This group calculates the gradient of the static volume, S, which corresponds to the computation on line 1 of Algorithm 1. It takes S as input and produces a gradient for each voxel in the x, y, and z directions.

2) *Interpolation*: This group performs trilinear interpolation on the moving volume, M, which corresponds to the computation on the line 5 of Algorithm 1. It takes M and a total displacement for each voxel in the x, y, and z direction as inputs and produces a deformed volume, D. Since hardware-based trilinear interpolation is not accessible at this time from the CUDA interface, we could not utilize it and instead, included this kernel in our implementation.

3) *Displacement*: This group calculates the total displacement field, $\vec{v}$, which corresponds to the computation on the lines 6 and 7 of Algorithm 1. It takes S, the output of the gradient and interpolation kernels, and the total displacement of each voxel from the previous iteration in the x, y, and z directions as inputs. It produces the current total displacement of each voxel in x, y, and z directions.

4) *Smoothing*: This group applies Gaussian smoothing to $\vec{v}$, which corresponds to the computation on line 9 of Algorithm 1. This is accomplished by performing convolution of

$\overrightarrow{v}$ with a three-dimensional Gaussian filter. Since the Gaussian filter is separable, convolution is performed in separate passes for the x, y, and z filters. Kernels in this group are based on NVIDIA's "Separable Convolution" example released in June 2007, which is provided in the CUDA SDK individual sample packages. We have extended this example so that convolution can be performed in three dimensions and the Gaussian filter radius can be different in all dimensions.

5) *Correlation*: This group calculates the correlation coefficient between S and D, which corresponds to the computation on line 10 of Algorithm 1. The correlation coefficient is computed using following formula:

$$\frac{\Sigma_i(D(i)-\overline{D})(S(i)-\overline{S})}{\sqrt{\Sigma_i(D(i)-\overline{D})^2\Sigma_i(S(i)-\overline{S})^2}} \tag{2}$$

where $\overline{D}$ and $\overline{S}$ are the mean values of D and S, respectively. In order to calculate the sums and mean values in Equation 2, we perform a reduction operation.
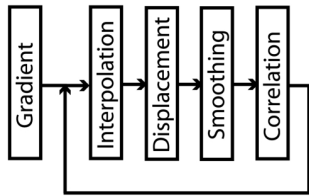

Figure 2. Control flow of the kernels.

After the static and moving images are transferred from CPU memory into GPU memory, our implementation iterates through the above CUDA kernels until convergence as shown in Figure 2.

5   Results

In this section, we first compare the runtimes of our GPU and CPU implementations for datasets with different sizes and present our speedup and error. Then, we show visual results by providing slices from one of the datasets. Next, we provide the breakdown of GPU implementation runtime to the CUDA kernels and present GFLOP (Giga Floating Point Operations = 10^9 FLOP) and GiB (Gibi Bytes = 2^{30} Bytes) instruction and data throughput information. This is followed by the discussion of the factors that limit our performance. Finally, we compare our implementation with Sharp et al.'s implementation and highlight our improvements.

Table 1 shows the performance of our GPU implementation with respect to the optimized CPU implementation. We have achieved around 55x speedup over a single-threaded CPU implementation and 35x speedup over

Dataset	1	2	3
Dimensions (Height × Width × Depth)	256x256x88	256x256x119	512x512x54
Size (MegaPixels)	5.77	7.80	14.16
GPU runtime (s)	5.27	7.07	13.54
Single-threading CPU runtime (s)	297	401.25	745.27
Multi-threading CPU runtime (s)	191	257.37	466.01
GPU speedup over single-threading	56.32	56.78	55.05
GPU speedup over multi-threading	36.22	36.42	34.42
Maximum GPU error	0.000017	0.000015	0.000030

Table 1. Performance of GPU implementation with respect to CPU implementation.

a multi-threaded CPU implementation. We present both single and multi-threading comparisons, since comparing the GPU implementation with only the single-threaded CPU implementation would result in underutilization of the CPU. In all implementations, the algorithm is run for 100 iterations. In each iteration, we computed the error by taking the absolute difference between the correlation coefficients of the GPU and CPU implementations. We reported the error with the largest difference as maximum error for the related dataset. For all datasets, the maximum error was less than $3 \cdot 10^{-5}$, which is a negligible difference. However, introducing error is unavoidable, since the CPU implementation is using double-precision floating-point variables, whereas the GPU implementation is using single-precision floating-point variables. This is because, currently in the G80 architecture, only single-precision floating-point is supported. Using single-precision instead of double-precision floating-point variables in the CPU implementation provides 20% faster runtime.

Figure 1 shows slices from Dataset 2. Figure 1(a), (b), and (c) refer to slices from static, moving, and deformed volumes, respectively. The deformed image is produced by deforming the moving image with the GPU implementation of Demons, so that it can be aligned with the static image. Figure 1(d) and Figure 3(b) show the difference images of the static image and the deformed image generated by the GPU and CPU implementations, respectively. As can be seen from these figures, the GPU and CPU implementations of Demons produced no visible difference, a result which is in line with the negligible error mentioned above. Figure 3(a) presents the difference image between the static and moving images. When Figure 3(a) is compared to Figure 3(b) and Figure 1(d), we see that by applying Demons, a large difference is corrected and reduced to a very small amount.

Table 2 shows the breakdown of GPU implementation runtime into the kernels invoked from the main loop of the
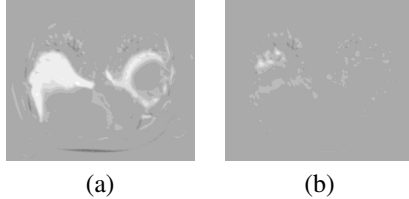
| | (a) | | (b) |

Figure 3. Slices from Dataset 2: (a) difference image of the static and moving image, and (b) difference image of the static and deformed image generated by the CPU.

Kernel	Interpolation	Displacement	Smoothing	Correlation	Total
Runtime of 100 iterations (s)	2.75	0.56	2.95	0.71	6.97
GFLOP per iteration	0.29	0.14	0.42	0.10	0.94
GiB per iteration	0.35	0.30	0.54	0.29	1.48
GFLOP/s	10.42	24.54	14.25	13.48	13.50
GiB/s	12.62	53.81	18.43	40.80	21.29
(number of FLOP) / (number of memory operations)	3.08	1.70	2.88	1.23	2.36

Table 2. Breakdown of GPU runtime to the kernels and GFLOP-GiB information.

algorithm. Since the *gradient* kernel is invoked only once (and takes 0.04 seconds to run), it is excluded from the table. In addition, we show GFLOP-GiB data. Dataset 2 is used to generate measurements. Let's denote "GFLOP and GiB per iteration" as "GFLOP-GiB/iteration". If we compare the runtimes of all kernels, we see that *smoothing* is the most expensive part of algorithm and takes the longest time to run. This is not surprising when we consider that this kernel has the most GFLOP-GiB/iteration. *Interpolation* has the second longest runtime and the second largest GFLOP-GiB/iteration. However, even though the GFLOP-GiB/iteration of this kernel is much less than *smoothing*, we do not see a significant decrease in the runtime of *interpolation* when compared to the runtime of *smoothing*. This is because less opportunities exist to perform memory coalescing (accessing a coherent chunk of memory with a block of threads) in *interpolation* than in *smoothing*, due to the nature of the interpolation process. *Displacement* and *correlation* have much less GFLOP per iteration than other kernels. In addition, due to the nature of these kernels, all of the memory accesses can be coalesced. Thus, the runtimes of *displacement* and *correlation* are pretty short and in these kernels, we approach half of the theoretical peak GiB/s measurements of the G80, which is 80+ GB/s. Even though

displacement has slightly more GFLOP-GiB/iteration than *correlation*, its runtime is shorter. This is mainly due to the time-consuming branching inherit in the reduction process implemented in the summing calculations of the correlation coefficient.

In a CUDA implementation, three factors can limit the performance: bus bandwidth, compute bandwidth, and memory bandwidth. In our implementation, since the CPU-GPU data transfer takes a very short time compared to the overall runtime of the algorithm (e.g. the transfer of static and moving images of Dataset 2 from CPU to GPU memory takes 54 milliseconds whereas the whole algorithm takes 7.07 seconds), we can say that bus bandwidth does not limit our performance. Because the ratio of compute throughput to memory throughput is 16:1, as a starting point, we consider a kernel as compute bounded if it its ratio of compute operations to the number of memory operations is larger than 16, and memory bounded otherwise. Table 2 indicates that in our implementation, we have an average of 2.36 floating-point operations per memory operation, when all kernels are considered. This suggests that our implementation is memory bounded rather than compute bounded.

Along the three GPU-based Demons implementations mentioned in Section 3, the study conducted by Sharp et al. [27] has the fastest runtime. In addition, since it uses a separable 3D Gaussian filter to regularize the displacement field, this study has the most accurate displacement computation and also the closest implementation to ours. One difference between Sharp et al.'s and our study is that they do not compute the correlation coefficient in each iteration. Instead, regardless of the dataset used, they run the algorithm for a fixed number of iterations (e.g. 50 or 100) and then stop. The advantage of calculating the correlation coefficient is that for a given dataset, it allows us to see in which iteration the algorithm converges and stop the execution in that iteration. Hence, for each dataset, algorithm would run for a necessary number of iterations instead of fixed number of iterations, which would in turn reduce the overall runtime. For instance, Figure 4 shows that the correlation coefficient values of Dataset 2 stabilizes after around 70 iterations. This suggests that for Dataset 2, the algorithm converges early and we can stop the execution at iteration 70 instead of running it until iteration 100.

We compared our runtime and speedups with Sharp et al.'s results. In order to make a fair comparison, we replaced our NVIDIA Quadro FX 5600 graphics card with an NVIDIA GeForce 8800 GTS, the older graphics card Sharp et al. used in their study. In addition, since Sharp et al's implementation does not compute correlation coefficients, we turn off this computation in our code. Under these conditions, our implementation yielded about 10% faster runtime. For a dataset with dimension $180 \times 424 \times 150$ (height $\times$ width $\times$ depth) when the algorithm is run for 100 iter-
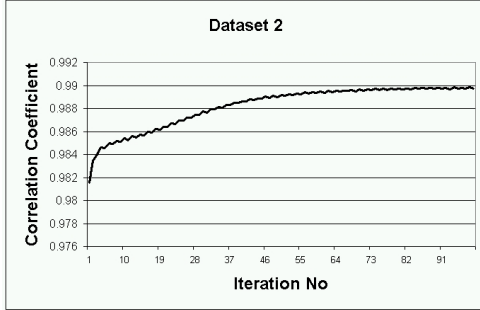
Figure 4. Correlation coefficient versus iteration number for Dataset 2.

ations, Sharp et al. reported their runtime as 13.92 s. For a dataset with the same size, it took 12.46 s for our implementation to run for 100 iterations. In addition, Sharp et al. reported a smaller speedup for a larger dataset size (i.e. $70\times$ speedup for dataset with dimensions $128\times126\times128$ and $45\times$ for dataset with dimensions $180\times424\times150$). As can be seen from Table 1, in our implementation speedups are almost the same and do not significantly fluctuate with dataset size. This suggests that our implementation is more scalable as problem sizes increase.

6 Discussion

In this section, first we share our experiences with optimizing our GPU implementation and list some of the techniques that provided us with better runtimes. Then, we discuss how we can improve the Demons algorithm by utilizing the great speedups we achieved by GPUs. Next, we talk about two main types of data communication on GPUs called gather and scatter, and indicate why we preferred gather over scatter in our implementation. Finally, we conclude this section by making some hardware and software recommendations to vendors, which we believe would improve the runtime of similar applications on GPUs.

Among the optimizations we have performed to reduce the runtime of our GPU implementation, coalescing the global memory accesses [35] provided us with the biggest performance gain. For instance, the runtime of our implementation was reduced from 52.47 seconds to 13.54 seconds for Dataset 3 when our kernels were optimized to perform memory coalescing. Memory pinning was another optimization we used in our implementation and it allowed us to achieve shorter data transfer time between CPU-GPU. For instance, by pinning the memory we reduced the CPU to GPU transfer time of static and moving images of Dataset 3 from 129 ms to 97 ms.

As explained in Section 2, accelerating the runtime of Demons registration by utilizing the GPU allows us to use

this technique in real-time settings such as in operation rooms. The tremendous computation power provided by GPU can also be exploited to improve Demons algorithm. For instance, by implementing Juggler's method [34], a hybrid deformable registration method that combines Demons with free-form deformable registration via the calculus of variations technique [20], we can improve the convergence speed of the registration. Convergence speed can also be improved by adding certain parameters to Demons as proposed in the accelerated Demons study [33]. The methods, such as Juggler, accelerated Demons, or any other technique, which improve the Demons algorithm, might require extra computation power; and the significant speedups provided by GPUs can help us easily address this need.

There are two main types of global data communication on GPUs: gather and scatter [6]. If a data elements requires its neighboring elements for its computation, we can choose either of these two methods. If we chose gather, each element performs a pull operation and requests information from other elements. On the other hand, if we use scatter, each element performs a push operation and distributes information to its neighbors who are in need of this element. There are advantages and disadvantages of both methods. Gather performs indirect read(s) from memory and naturally maps to texture fetch. Scatter performs indirect write(s) to memory and is available only on the new GPGPU platforms i.e. AMD's Close To the Metal (CTM) and NVIDIA's CUDA. Scatter has unspecified semantics on collision, which results in undefined behavior when two elements write to the same location at the same time. In addition, scatter is uncached and the performance implications of it are not yet clear. Considering these pros and cons, we preferred gather instead of scatter while implementing our convolution kernel.

We will conclude our discussion by making some hardware and software recommendations to the vendors, which we believe would improve the runtime of similar implementation on GPUs. Currently, we do not have access to the hardware-based trilinear interpolation capabilities of G80 from the CUDA programming environment. Hence, in our implementation we had to implement a kernel that performs trilinear interpolation. If hardware-based trilinear interpolation is exposed to CUDA programmers, interpolation computations in the kernel can be removed by using this feature. Although the cost of memory operations would stay the same, saving computation would simplify this kernel.

Another functionality that we think would be advantageous is having intelligent paging capabilities in the device memory. Processing a large dataset with a size that exceeds the capacity of the device memory creates complications. In order to solve these complications, usually extra complexity is added to the implementation. If the GPU driver is capable of performing intelligent paging, it can hide these

complications in an efficient way and make GPU-based implementations scalable to larger datasets without requiring major changes to the code.

Finally, we think that having more advanced profiling tools would help us get better performance from our implementations. As mentioned above, memory coalescing brings us the biggest performance gain. Hence, it would be very useful to have a profiling tool that provides us with an information about which memory accesses in our kernels are coalesced and which ones are not. In addition, to get a better insight about whether we are memory or compute bounded in our kernels, it would be very advantageous to have a tool that would automatically present us with GFLOP-GiB information. Currently, this information is extracted from kernels manually by counting floating point operations and memory accesses, which is time-consuming and error-prone.

7 Future Work

We would like to improve our implementation by adding a graphical user interface to visualize evolution of the deformed image in each iteration of the registration process. This feature would allow users to better understand the progress of the deformation. In addition, we would like to come up with strategies to handle registration of large medical volumes with sizes that exceed the capacity of the device memory. Automatically dividing the dataset in to smaller pieces in an efficient way, running the Demons algorithm on each subset, and combining the sub-results for the final result would be an effective strategy.

Implementing Demons on GPU has provided us with a proof of concept and standardization for comparison. As a next step, we are planning to implement a more efficient algorithm called Juggler's method (Section 6). Then, we would like to continue working on the implementation of other registration algorithms on GPUs. By investigating several different methods, we are planning to pinpoint common building blocks (i.e. primitives such as algorithms or data structures) of these methods. Then, we would like to implement these algorithms and data structures as high-level parallel primitives for GPU use. Finally, we would like to package these primitives into a library that can be used by other GPU applications in the medical imaging field.

We believe that having a medical imaging GPU library will significantly improve the way GPU-based medical applications are developed. Since currently there are no libraries available, every individual application is written *vertically* from the ground up. This lack of code sharing forces GPU programmers to focus their effort on custom underlying primitives instead of the application where it belongs. On the other hand, CPU developers have a wealth of algorithms and data structures that help them create complex applications. Indeed, analysis of the typical CPU high-performance application would reveal that most code in the application was not written directly by the application writer, but instead comes from various libraries written by others (i.e. BLAS, STL, Boost). Hence, CPU code sharing allows *horizontal* program development where most programmer effort is put into the application rather than the underlying primitives. With construction of a medical imaging GPU library, we are aiming to change the development environment of GPU applications from *vertical* to *horizontal*, which will make the implementation of complex GPU-based medical imaging applications much easier.

We will start construction of our library by adding image registration primitives. However, to make such a library suitable for more general medical imaging tasks, we will also build primitives for related problems such as image segmentation, CT reconstruction, on-line motion correction during MR acquisition, etc. In order to get ideas for potential primitives and applications, we are planning to continue working with the domain experts in the field and also look at The Insight Segmentation and Registration Toolkit (ITK) [10], a CPU-based library.

8 Conclusion

We have presented CUDA implementation of widely used Demons algorithm to address the need for fast deformable registration methods in the medical imaging field. Among the available GPU-based Demons implementations, we have achieved the fastest runtime and provided the most scalable approach. Results show that, independent of the given dataset size, our implementation is 55 times faster than CPU-based implementation. This study also contributes to the literature by implementing a deformable registration algorithm on a GPU, an understudied type of algorithm in the GPGPU community.

References

[1] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream computing on graphics hardware. *ACM Transactions on Graphics*, 23(3):777–786, Aug. 2004.

[2] U. Clarenz, M. Droske, and M. Rumpf. Towards fast non-rigid registration. In *Inverse Problems, Image Analysis and Medical Imaging, AMS Special Session Interaction of Inverse Problems and Image Analysis*, volume 313, pages 67–84. AMS, 2002.

[3] N. Courty and P. Hellier. Accelerating 3D non-rigid registration using graphics hardware. in press, 2007.

[4] GPGPU - General Purpose Computation Using Graphics Hardware. `http://www.gpgpu.org/`.

[5] J. V. Hajnal, D. L. Hill, and D. J. Hawkes, editors. *Medical Image Registration*. CRC Press, 2001.

[6] M. Harris. Mapping computational concepts to GPUs. In M. Pharr, editor, *GPU Gems 2*, chapter 31, pages 493–508. Addison Wesley, Mar. 2005.

[7] P. Hastreiter, C. Rezk-Salama, G. Soza, M. Bauer, G. Greiner, R. Fahlbusch, O. Ganslandt, and C. Nimsky. Strategies for brain shift evaluation. *Medical Image Analysis*, 8:447–464, December 2004.

[8] B. K. P. Horn and B. G. Schunck. Determining optical flow. *Artif. Intell.*, 17(1-3):185–203, 1981.

[9] F. Ino, J. Gomita, Y. Kawasaki, and K. Hagihara. A GPGPU approach for accelerating 2-D/3-D rigid registration of medical images. In *Parallel and Distributed Processing and Applications (ISPA)*, volume 4330 of *LNCS*, pages 939–950, 2006.

[10] The Insight Toolkit. `http://www.itk.org/`, 2003.

[11] S. C. Joshi, M. Foskey, and B. Davis. Automatic organ localization for adaptive radiation therapy for prostate cancer. Technical Report ADA428913, North Carolina University at Chapel Hill, May 2004.

[12] A. Khamene, P. Bloch, W. Wein, M. Svatos, and F. Sauer. Automatic registration of portal images and volumetric CT for patient positioning in radiation therapy. *Medical Image Analysis*, pages 96–112, Feb. 2006.

[13] A. Khamene, R. Chisu, W. Wein, N. Navab, and F. Sauer. A novel projection based approach for medical image registration. In J. P. W. Pluim, B. Likar, and F. A. Gerritsen, editors, *WBIR*, volume 4057 of *LNCS*, pages 247–256. Springer, 2006.

[14] J. Kim, S. Li, Y. Zhao, and B. Movsas. Real-time intensity-based deformable fusion on PC graphics hardware. In *XVth International Conference on the Use of Computers in Radiation Therapy*, 2007.

[15] K. Kim, S. Park, H. Hong, and Y. G. Shin. Fast 2D-3D registration using GPU-based preprocessing. In *7th International Workshop on Enterprise networking and Computing in Healthcare Industry*, pages 139–143. HealthCom, 23-25 June 2005.

[16] A. Köhn, J. Drexl, F. Ritter, M. König, and H. O. Peitgen. GPU accelerated image registration in two and three dimensions. In H. Handels, J. Ehrhardt, A. Horsch, H.-P. Meinzer, and T. Tolxdorff, editors, *Bildverarbeitung für die Medizin 2006*, Hamburg, 3 2006. Springer-Verlag, Berlin, Heidelberg, New York.

[17] D. LaRose. *Iterative X-ray/CT Registration Using Accelerated Volume Rendering*. PhD thesis, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, May 2001.

[18] H. Lester and S. R. Arridge. A survey of hierarchical non-linear medical image registration. *Pattern Recognition*, 32(1):129–149, 1999.

[19] D. Levin, D. Dey, and P. Slomka. Acceleration of 3D, non-linear warping using standard video graphics hardware: implementation and initial validation. *Comput Med Imaging Graph.*, 28:471–483, 2004.

[20] W. Lu, M. L. Chen, G. H. Olivera, K. J. Ruchala, and T. R. Mackie. Fast free-form deformable registration via calculus of variations. *Physics in Medicine and Biology*, 49:3067–3087, 2004.

[21] J. Maintz and M. Viergever. A survey of medical image registration. *Medical Image Analysis*, 2(1):1–36, 1998.

[22] NVIDIA Corporation. NVIDIA CUDA compute unified device architecture programming guide. `http://developer.nvidia.com/cuda`, Jan. 2007.

[23] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[24] C. Rezk-Salama, M. Scheuering, G. Soza, and G. Greiner. Fast volumetric deformation on general purpose hardware. In *Proc. SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 17–24, 2001.

[25] M. Roth, M. Dötter, R. Burgkart, and A. Schweikard. Fast intensity-based fluoroscopy-to-CT registration using pattern search optimization. In *Computer Assisted Radiology and Surgery (CARS)*, pages 165–170, 2004.

[26] O. Sadowsky, J. D. Cohen, and R. H. Taylor. Rendering tetrahedral meshes with higher-order attenuation functions for digital radiograph reconstruction. In *IEEE Visualization*, pages 303–310. IEEE Computer Society, 2005.

[27] G. C. Sharp, N. Kandasamy, H. Singh, and M. Folkert. GPU-based streaming architectures for fast cone-beam CT image reconstruction and Demons deformable registration. *Physics in Medicine and Biology*, 52(19):5771–5783, 2007.

[28] G. Soza, M. Bauer, P. Hastreiter, C. Nimsky, and G. Greiner. Non-rigid registration with use of hardware-based 3D Bézier functions. In *MICCAI '02: Proceedings of the 5th International Conference on Medical Image Computing and Computer-Assisted Intervention-Part II*, pages 549–556, London, UK, 2002. Springer-Verlag.

[29] G. Soza, P. Hastreiter, M. Bauer, C. Rezk-Salama, C. Nimsky, and G. Greiner. Intraoperative registration on standard PC graphics hardware. In *Bildverarbeitung fr die Medizin*, pages 334–337, 2002.

[30] R. Strzodka, M. Droske, and M. Rumpf. Image registration by a regularized gradient flow: A streaming implementation in DX9 graphics hardware. *Computing*, 73(4):373–389, Nov. 2004.

[31] J.-P. Thirion. Image matching as a diffusion process: an analogy with Maxwell's demons. *Medical Image Analysis*, 2(3):243–260, 1998.

[32] C. Vetter, C. Guetter, C. Xu, and R. Westermann. Non-rigid multi-modal registration on the GPU. In J. P. W. Pluim and J. M. Reinhardt, editors, *Medical Imaging 2007: Image Processing.*, volume 6512, page 651228. Society of Photo-Optical Instrumentation Engineers (SPIE), Mar. 2007.

[33] H. Wang, L. Dong, J. O'Daniel, R. Mohan, A. S. Garden, K. K. Ang, D. A. Kuban, M. Bonnen, J. Y. Chang, and R. Cheung. Validation of an accelerated 'demons' algorithm for deformable image registration in radiation therapy. *Physics in Medicine and Biology*, 50:2887–2905, 2005.

[34] J. Xia, Y. Chen, and S. Samant. The Juggler algorithm: A hybrid deformable image registration algorithm for adaptive radiotherapy. volume 6510, page 65105J. Society of Photo-Optical Instrumentation Engineers (SPIE), 2007.

[35] C. Zeller. CUDA performance. *ACM SIGGRAPH GPGPU Course Notes*, 2007. `http://www.gpgpu.org/s2007/slides/09-CUDA-performance.pdf`.

[36] B. Zitova and J. Flusser. Image registration methods: a survey. *Image and Vision Computing*, 21(11):977–1000, October 2003.