Eurographics Symposium on Parallel Graphics and Visualization (2009)
J. Comba, K. Debattista, and D. Weiskopf (Editors)

Fast Parallel Unbiased Diffeomorphic Atlas Construction on Multi-Graphics Processing Units

Linh K. Ha, Jens Krüger, P. Thomas Fletcher, Sarang Joshi and Cláudio T. Silva

Scientific Computing and Imaging Institute at the University of Utah

Abstract

Unbiased diffeomorphic atlas construction has proven to be a powerful technique for medical image analysis, particularly in brain imaging. The method operates on a large set of images, mapping them all into a common coordinate system, and creating an unbiased common template for studying intra-population variability and inter-population differences. The technique has also proven effective in tissue and object segmentation via registration of anatomical labels. However, a major barrier to the use of this approach is its high computational cost. Especially with the increasing number of inputs and data size, it becomes impractical even with a fully optimized implementation on CPUs. Fortunately, the highly element-wise independence of the problem makes it well suited for parallel processing. This paper presents an efficient implementation of unbiased diffeomorphic atlas construction on the new parallel processing architecture based on Multi-Graphics Processing Units (Multi-GPUs). Our results show that the GPU implementation gives a substantial performance gain on the order of twenty to sixty times faster than a single CPU and provides an inexpensive alternative to large distributed-memory CPU clusters.

Categories and Subject Descriptors (according to ACM CCS): GPGPU applications, Parallel programming

1. Introduction

Construction of atlases is a key procedure in population-based medical image analysis. In the paradigm of computational anatomy, the atlas serves as a deformable template [Gre94], which is mapped to each individual anatomy. The deformable template provides a common coordinate system for individual or group analysis of detailed imaging data, including structural, biochemical, functional, or vascular information. The transformations mapping each individual anatomy to the atlas encode the anatomical variability of the population under study. Recently, this concept has also been extended to study anatomical change as a function of age in a population by generalizing non-parametric regression [DFBJ07]. A major barrier to the use of such methods is the high cost associated with the atlas construction.

Efficient and scalable solutions for the atlas construction are becoming critical to the analysis of large brain imaging studies due to the ever expanding size of the input data. Advances in magnetic resonance imaging (MRI) are resulting in increasingly higher resolution images. Furthermore, the trend in neuroimaging studies is towards multi-site collection of large numbers of images, including longitudinal data. For instance, the Alzheimer's Disease Neuroimaging Initiative currently includes over 900 subjects, most imaged at multiple time-points. Consequently, fast deformable atlas construction has become a subject of considerable interests [CMVG96,BNG96]. However, current CPU-based solutions depend on expensive parallel systems, either shared memory symmetric multiprocessor machines or distributed memory clusters. Furthermore, there is only a modest amount of parallelism within a single processing core and communication between processing units is expensive. Thus, parallel CPU implementations are still time consuming and do not exhibit well-behaved scalability.

In this paper we present a multiple GPU atlas construction framework based on the unbiased diffeomorphic atlas formulation [JDJG04]. Our method achieves both high quality and extremely fast processing time by exploiting the parallel hardware architecture of multi-GPUs. Our framework includes an optimized 3D image processing library, a hardware supported nonlinear ordinary differential equation (ODE) integration, and a multiscale successive over-relaxation (SOR) solver for Helmholtz-like partial differential equations (PDEs). Our system also exploits the coherency of the vector fields, the massive parallelization of GPU hardware, the scalability of multi-GPU architectures, and the efficiency and robustness of multiscale techniques. The system builds atlases with comparable quality to those constructed by CPU algorithms [JDJG04, LDJ05]. Our system is 20-60 times faster than a well-optimized single core CPU algorithm, and still an order of magnitude faster than optimized multi-core CPU algorithms, while demonstrating a linear scalability curve. In designing an efficient parallel atlas construction, we overcame three challenging issues:

DOI: 10.2312/EGPGV/EGPGV09/041-048

- **Mapping the data structures and algorithms from the CPU domain to the GPU domain.** This is particularly important to exploit the massively parallel GPU hardware and the limited fast internal on-chip *shared* graphics memory as modern GPUs have massively parallel computational power devoted to algorithmic logic units (ALUs) with the instant context switching capability; in contrast the main or *global* memory has a high latency without any caching mechanism. By carefully laying out the 3D volume input data with an aligned flat 2D representation, we take advantage of multi-block threaded and memory coalesced access to hide the memory latency completely and optimize the bandwidth. We exploit hardware supported trilinear interpolation, to resolve the main bottleneck of the ODE integration. We utilize the fast shared memory to achieve maximum bandwidth with our SOR approach. In our algorithm we apply the idea of the arithmetic intensity technique to achieve the highest possible speed gain.
- **Efficiently solving PDEs.** The fastest existing CPU implementation exploits the optimized "FFTW" transformation to solve the PDE explicitly. Although novel GPU architectures make a parallel FFT implementation nearly five times faster than the CPU version, the PDE solver is still the major bottleneck, taking almost 90% of the processing time. To overcome this bottleneck we employ an implicit approach using the SOR framework. While SOR theoretically has a slower convergence rate, it exhibits less pixel-wise dependency and maps extremely well to GPUs. We further exploit the fact that only small changes occur in the vector field at each iteration causing the SOR approach to have a near constant convergence rate which is faster than the FFT approaches and comparable to multigrid schemes.
- **Design of a scalable system that can handle large numbers of inputs and maintain the performance with different input sizes.** We propose a hybrid model: a multiple inputs, multi-GPUs framework that utilizes all the available GPU memory and maximizes the arithmetic intensity, providing the fastest rate and lowest processing cost per volume element. We significantly improve the overall quality and efficiency of the system with the multiscale framework. The techniques presented in this paper can also be used for other highly computationally intense image processing applications.

2. Related work

2.1. Diffeomorphism

Deformable image registration is an extremely useful but computationally intensive task. Different forms of parallel processing have been applied to this problem since the mid 1990s. In 1996, Christensen *et al.* [CMVG96] developed an elastic and fluid deformation algorithm based on the Finite Difference Method to perform individualized, pair-wise matching, neuroanatomical atlas construction on 128-by-128-by-100 voxel datasets. On a MasPar MP-2, a massive parallel computer containing more than sixteen thousand four-bit processors, it took almost two hours to complete a single matching pair. Also in 1996 Bro-Nielsen *et al.* [BNG96] proposed a convolution filter approach for linear elasticity, based on the assumption of small deformations. Their algorithm ran on a DECmmp 1200 Sx/Model 200 massive computer, and the performance was comparable to Christensen *et al.*'s approach. Recently, Dandekar *et al.* [DS07] developed a special purpose solution based on the field-programmable gate array (FPGA), providing a 40 times speed up, but still took about 6 minutes for a one pair registration with 128-by-128-by-154 inputs (200 iterations).

Multi-resolution techniques significantly reduce the registration time. Zikic *et al.* [ZWK*06] proposed a multilevel framework for deformable registration of 3D ultrasound data that brought down the registration time to the average of half a minute for 256-by-256-by-256 volume inputs. However, their registration method used a variational deformable approach, which is much simpler and less robust than our diffeomorphism based system.

Davis *et al.* [DFBJ07] reported that a fully optimized multi-resolution and multi-core CPU version of diffeomorphism registration on an 8 dual-core processor 3GHz system with 64 GB of RAM took 14 minutes for one pair registration with 256-by-256-by-256 volume inputs. Our GPU implementation on an NVIDIA Quadro FX5600 is about 50 times faster than all of these approaches taking only about 12 seconds for a pair registration of the same data.

2.2. GPU Algorithms

The increasing programmability of graphics processing units (GPUs) coupled with their extremely powerful floating point units, and superior memory bandwidth make them suitable for a variety of computational tasks. Rumpf *et al.* [RS01] proposed to use GPUs as fast vector coprocessors to solve Finite Element Method (FEM) problems. They used multiple textures and buffers as inputs and outputs to communicate between the CPU and the GPU, and they mapped matrix computations to the Graphic APIs. Krüger and Westermann [KW03] built a general framework for efficient linear algebra operators on GPUs. Recently, Hagen *et al.* [HLN06] exploited GPUs to solve the Euler equation in 3D on high-resolution input data. They also proposed a computational scheme to use the GPU as a data stream processing unit to solve systems of Conservation Laws [HHHL07]. Jeong *et al.* [JFTW07] used GPUs to develop a Hamilton-Jacobi solver for iterative visualization of volumetric white matter connectivity in DT-MRI. Strzodka *et al.* [SDR04] exploited DirectX 9 graphics hardware features to develop a 2D image registration framework using streaming gradient flow and a multigrid PDE solver approach. All of these methods have in common that they successfully exploit the computational processing power of modern GPUs. Our work is along the same lines, but hardware and software advances have increased the flexibility of the new GPU architectures, and also enable us to exploit multiple GPUs.

3. Diffeomorphic Atlas Construction

In the template construction framework of the diffeomorphic atlas construction we define the statistical average of the population as the minimizer of the sum-square-distance to each of the data points. In other words, the representative template requires least deformation energy to match input images. To solve the problem, we use the Greedy Iterative framework (Algorithm 1) based on greedy fluid flow algorithm, for more details, see [JDJG04].

Figure 1 illustrates the pair-wise diffeomorphism that defines the transformation deforming a small part the letter 'C'

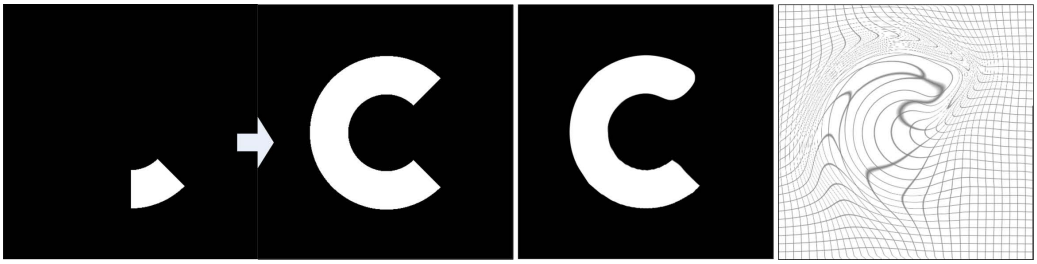

Figure 1: *Small part of the letter 'C' deforming into a full 'C' using 2D Greedy Iterative Diffeomorphism. From left to right: 1. Input and Target Image 2. Deformed template. 3. Grid showing the deformation applied to template*

Algorithm 1 Compute the average representation

1: **Input**: N input volume images I_i , $N > 0$
2: **Ouput**: $\hat{I}$ average template estimation
3: Initialize the time step $k \leftarrow 0$, $maxNumberOfIteration \leftarrow 200$, and $v_i^k \leftarrow 0$ with all volume inputs
4: **while** $k < maxNumberOfIteration$ **do**
5: Compute the global template estimate $\hat{I}^k(x) = \frac{1}{N}\sum_{i=1}^{N} I_i^k(x)$
6: **for all** Transform images I_i^k $i = 0, \cdots N$ **do**
7: Compute the force $F_i^k = -\left[I_i^k(x) - \hat{I}^k(x)\right] \bigtriangledown I_i^k(x)$
8: Solve the PDE $Lv_i^k(x) = F_i^k(x)$ where $L = \alpha\nabla^2 + \beta\nabla\nabla + \gamma$
9: Update the transformation $h_i^{k+1} = h_i^k\left(x + \varepsilon v_i^k(x)\right)$
10: Update the immediate transform images $I_i^k = I_i\left(h_i^k(x)\right)$
11: **end for**
12: $k \leftarrow k+1$
13: **end while**

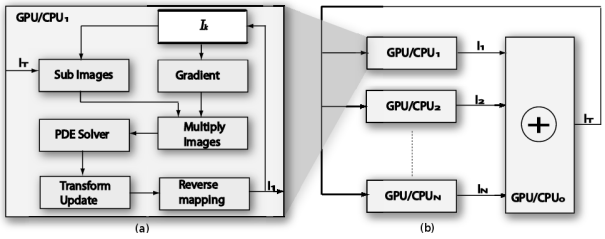

Figure 2: *Greedy Iterative Diffeomorphic framework*

into the target full 'C' image. The smoothly changing curve in Figure 1 is the minimal transformation energy path, and thus, the optimal deforming path.

4. GPU Diffeomorphic Atlas Construction

The Greedy Iterative method directly maps to the parallel computing architecture, as shown in Figure 2. We perform independent operations on each image inside the loop. We assume that the number of processing nodes equals the number of inputs. For each iteration, we send the ith image to the ith node together with the common average template image, and then perform a greedy step on each node independently. The overall processing time is primarily constrained by the processing time of a single node, exactly the time of a single CPU implementation. We address two main bottlenecks in the CPU implementation: the PDE solver (Algorithm 2:8) and the ODE integration (Algorithm 2:10).

4.1. ODE integration

The ODE integration takes 40% of the processing time in the CPU implementation, performing composition using the reverse mapping technique. Reverse mapping is a general strategy in image processing to perform nonlinear integration to handle the hole and missing value issues with forward mapping. The method computes the destination grid value as the interpolation over grid neighbor points from the source. In 3D, trilinear interpolation (trilerp) is commonly used because of its simplicity and satisfactory results, however, it is still an expensive operation. The trilerp requires at least 14 multiplications and 8 additions per scalar component. It is up to 3 times more expensive with a 3D vector field. Moreover, the random access pattern makes it cache unfriendly with the linear CPU cache. Fortunately, this challenging problem has a hardware-assisted solution on GPUs. The interpolation process is fully hardware accelerated with 3D texture volume support from CUDA 2.0 APIs. This optimization greatly reduces the computational cost of the ODE integration. The timing result of the optimized version based on the FFT approach (see Table 2) shows only 9% of overall time spent on ODE integration. The 3D texture can also be used to compute the gradient effectively, also giving a performance boost of 50 to 120 times.

All basic 3D image operations can be efficiently implemented using the 3D flat-array data layout. The 2D multi-threaded block architecture of CUDA matches with this 2D representation, giving performance gains over CPU implementations for these functions from 20 to 50 times. These performance results are shown in Table 1.

The other bottleneck is the PDE solver, which takes the remaining 60% processing time with the optimized CPU version, and 90% in the optimized FFT GPU version.

4.2. PDE solver

PDE solvers have long been studied in the literature. The two most common and efficient ways to solve the problem are explicit solvers in the Fourier domain using FFT transformation and implicit solvers using iterative refinement methods such as Jacobi, Gaussian, SOR or multigrid methods. Demmel [Dem97] summarizes these different approaches and gives a thorough comparison.

It is well known that the FFT solver is among the most efficient methods, and generally preferred on CPU, the others are block cyclic reduction and multigrid. Currently the fastest implementation of the Greedy Iterative Diffeomor-

phism on CPU builds on the FFTW3 library, the most efficient FFT implementation using general processing units. With their high level of parallelization GPUs enable us to perform the FFT faster than their FFTW-based counter part. The current CUDA FFT [NVI07] running on the G80 architecture from NVIDIA is about 5 to 8 times faster than FFTW3, especially with larger volume size. Our result is compared with the GPUs optimized FFT version using CUDA FFT.

Successive over-relaxation (SOR) is an iterative algorithm proposed by Young for solving a linear system [**?**]. As showed by Demmel [Dem97], the 3D FFT solver has a complexity of $O(n\log(n))$ versus $0(n^{5/3})$ for SOR. However, the complexity analysis does not count the fact that SOR is an iterative refinement method whose convergence speed largely depends on the initial guess. With a close approximation of the result as the initial value, it normally requires only a few iterations to converge. We observe that in the elastic deformable framework with steady fluid, the changes in the velocity field are quite small between greedy steps. The computed velocity field of the previous step is inherently a good approximation for the current one. In practice, we typically need 50 to 100 SOR iterations for the first greedy step, but only 4 to 6 iterations for each following step.

The SOR and multigrid solvers are generally used in parallel super computing models using SIMD machines. The results of a similar approach running on MasPar 128x128 have been reported by Christensen *et al.* [CMVG96]. The problem of the multi-CPU system is that it is expensive and only suitable for complex problems that require a general computing model, massive computational power, and process very large databases. It is not an economical solution in general. Our framework, based on the CUDA GPU platform, shows an affordable and efficient solution for the problem. The new parallel computational capability of G80 GPUs enables us to perform 32 operations in parallel, like SIMD machines but in a simpler and more efficient way. Our GPU SOR implementation is 2.5 to 3 times faster than the most recent parallel optimized FFT implementation on GPUs [NVI07], hence 14-18 times faster than the well known optimal FFTW3 running on an 8-core 2.33 GHz Intel Core2 Xeon system [FJ05].

4.3. GPU Implementation of SOR

As shown in Algorithm 1, velocity is computed from the force function using the Navier-Stokes equation

$$\alpha\nabla^2 v(x) + \beta\nabla\nabla v(x) + \gamma v(x) = F(x) \quad (1)$$

Often β is negligible and (1) simplifies to the Helmholtz equation

$$\alpha\nabla^2 v(x) + \gamma v(x) = F(x) \quad (2)$$

In practice we chose $\alpha = 1.0$ and $\gamma = 0.02$. We solve the Helmholtz equation on the grid using the SOR (Algorithm 3). The computations are constrained to the interior grid region. On the boundary we use Dirichlet condition with values 0. The convergence rate of SOR is controlled by the overrelaxation factor ω that is optimally defined in 3D as

$$\omega = \frac{2}{1+\sqrt{1-\frac{1}{3}\left[\cos\frac{\pi}{w}+\cos\frac{\pi}{h}+\cos\frac{\pi}{l}\right]^2}} \quad (3)$$

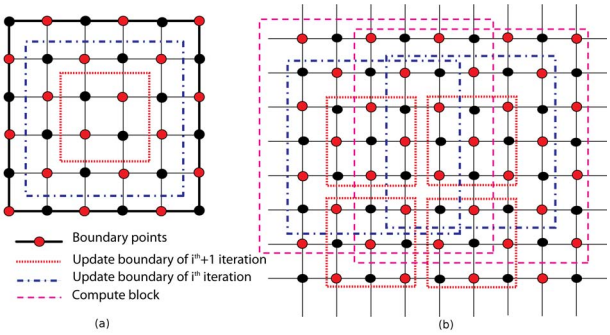

Figure 3: *Parallel block SOR, we assign each CUDA thread block a block of data to compute the black points inside the blue boundary, and use that result to compute the red point inside the red boundary. Two neighboring compute blocks share a four grid point-wide region.*

where w, h, l are the input dimensions. We employ Red-Black ordering, as shown in Figure 3, to update the result. In this configuration we only update points of the same color based on their neighbors, which have different color. Red-Black decoupling is proved to have a well-behaved convergence rate [Dem97], but more importantly, allows us to update points independently and efficiently in parallel.

The color of a grid point (i, j, k) (red or black) is simply defined by the odd/even value of the total indices $i+j+k$. Although similar, the red/black updating function is much more expensive than computing the gradient. Moreover, SOR iterations need to update the results to use as the inputs of the next steps synchronously between red and black iterations, preventing us from using texture memory as effective inputs to exploit the fast cache and neighbor coherence, as is seen in the gradient operator.

In our first version of the CUDA implementation, we used two separated kernels to perform the red/black update. Although it exploited the parallel power, it did not give us the performance we need. Even with only 4 iterations, the performance was only about as fast as the FFT. The speed of the algorithm was limited by the GPU DRAM bandwidth rather than the computation. To improve performance for bandwidth limited applications on GPUs, Buck [Buc05] recommended to increase the efficiency by increasing arithmetic intensity, the ratio of the computation performed in an algorithm relative to the amount of memory read and written. We improve arithmetic intensity of the GPU SOR by merging the red and black update steps. We read the red point value from the global memory to the shared memory, compute the value of the black points, write out the result to global memory *and* also the shared memory, then use the new updated black point values in the shared memory to compute the red point values of the next iteration. This strategy doubles the arithmetic intensity, consequently doubling the speed of the SOR solver. However, this approach is limited by the amount of shared memory available per processing block which is 16Kb on the G80 architecture. To overcome this limitation, we propose the *block-SOR* method. We divide the input volume into blocks, each fitting onto one CUDA execution block. We exploit the available shared

memory to perform the merged black/red updating step locally on the block. Figure 3(a) shows the updating boundary in the Block-SOR approach. For simplicity, we illustrate the idea in 2D, but it is generalized to arbitrary dimensions. We can see that the actual updated volume is two-cells smaller in each dimension than the processing block. The reduction in size explains why we can not merge an arbitrary number of update iterations in one kernel call. To update the whole volume, we allow a data overlap among processing blocks, as shown in Figure 3(b), so the remaining red points inside the blue boundary of the current block will be computed from the neighbor blocks. In this approach we allow data redundancy to increase memory usage. The configuration shown in Figure 3(b), having a 4 point-wide boundary overlap, allows us to do the update of one merging step entirely over the M^2 block using $(M+4)^2$ data block input, while the traditional SOR needs a data block size of $(M+1)^2$. If we want to perform k red/black update iterations in a single kernel call, we need the input data block of size $(M+4*k)^2$. To quantify the benefit of warping multiple SOR steps in one kernel, we define a trade-off factor α such that:

$$\alpha = \frac{\text{Minimum needed data size}}{\text{Actual processing data size}} * \text{Speed up factor} \quad (4)$$

To update the volume block M^3, we need $(M+4k)^3$ volume inputs, the trade-off factor is $\alpha = (\frac{M+1}{M+4k})^3 * k$, α is larger when M increases, however we need to satisfy the memory constraint so that we can fit the entire data block into the limited shared memory. Typically $M = 12$ is the maximum size that we can fit in 16KB shared memory. The trade off factor with $M = 12$ is less than 1 if we try to merge more than $k = 2$ iterations. In practice, we see benefits only if we merge one black & red update step per kernel call. Algorithm 2 shows the pseudocode of our efficient block-SOR implementation on CUDA. We further leverage the trade-off requirement by limiting block-SOR in the 2D plane only, and exploit the coherence between consecutive layers in the third dimension to minimize data redundancy. Our results show that the block-SOR method is optimal because it yields equivalent bandwidth to the gradient computation.

Algorithm 2 Fast 3D parallel SOR step (SOR_Helmhotz3D)

Input : Old velocity field v and new force function F
Output: Compute new velocity field v
Allocate 4 shared mem array to store 4 slices of data
Load v, F of the 1^{ft}, 2^{nd} and 3^{rd} slice to shared-mem
$k \leftarrow 1$
Update the black point of the k^{th} slice
while $k < number\_of\_slice - 2$ **do**
  Load the next slice to free shared mem array slice
  Update the black point of the $(k+1)^{th}$ slice
  Update the red point of the k^{th} slice
  Circulate shifting the shared mem array pointers the shared mem of $(k-1)^{th}$ slice becomes free
  Load the next slice to free shared mem array
  Update the black point of the $(k+2)^{th}$ slice
  Update the red point of the $(k+1)^{th}$ slice
  Circulate shifting the shared mem array pointers, the shared mem of k^{th} slice become free
  $k \leftarrow k+2$
end while
Update the red points on the last slice close to boundary

Algorithm 3 Efficient SOR solver

if first iteration **then**
  Initialize velocity field $v = 0$
  $numIters = 100$
else
  Keep value v from previous iteration
  $numIters = 4$
end if
for iter = 0 to $numIters - 1$ **do**
  call SOR_Helmhotz3D(v,F)
end for

4.4. Multi-scale Greedy Iterative Algorithm

By design the Greedy Iterative Algorithm has the capability to process data in a hierarchical multi-scale manner. The idea is derived from the multi-grid technique in PDE solvers. We perform the greedy algorithm on a coarse grid, compute the approximate solution of the transformation, and then interpolate the transformation onto the finer level grid. This solution for the coarse grid generates a good initial guess of the deformation on the fine grid. This strategy dramatically reduces the number of iterations from 200 iterations to only 25 iterations to achieve adequate results. The results show significant improvements on the speed due to smaller processing sizes. The system is more robust to the high frequency noise of the scanned input data, due to the damping high frequencies during grid down-sampling.

4.5. Computational model for multiple inputs

Handling multiple inputs efficiently is a critical problem. The atlas construction yields meaningful results with a significant number of inputs. Processing a large population requires large amount of memory and computational power, resulting in a significant amount of processing time; days or even months. In this section, we analyze two common models to deal with the problem and present our solution.

4.5.1. One GPU - multiple inputs model

Nowadays, the amount of GPU memory is comparable to the system's main memory, thus we are able to handle multiples inputs locally on the GPU. In our system, a Quadro FX 5600 with 1.5 GB of memory was used. Thus, we should be able to process at least two 256^3 inputs separately. We observed that, when we process multiple inputs in the same GPU context, we can reuse a large amount of memory used during the computation. Thus we can manage up to six 256^3 inputs on the Quadro FX 5600. Further more, as shown in Figure 5(a), all inputs share the same GPU memory space, we can compute the common average template directly on the GPU and shared among inputs. The GPU average function is 20 times faster than the CPU version. The common average template is updated automatically among inputs. Because the updating cost is negligible, the runtime grows linearly over the number of inputs, as shown in Figure 4.

4.5.2. One-to-one multi-GPUs model

While the one-GPU-multiple-inputs-model processes multiple inputs sequentially, a simple one-to-one multi-GPUs system can process multiple inputs fully in parallel (see Figure 2). To each GPU we assign an input image and run

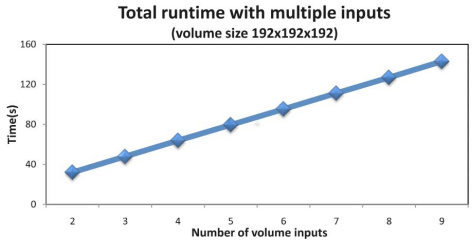

Figure 4: *Overall runtime for multiple inputs with* 192^3 *volumes*

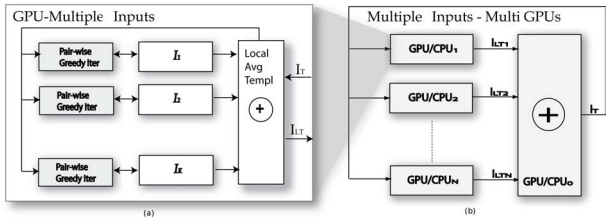

Figure 5: *Multiple Inputs - Multi-GPUs configuration (a) Multiple-Inputs processing node (b) Multi-GPUs configuration*

the greedy step independently. Since each GPU has a separate context, we need a central processing entity to distribute the average template and to collect results from all GPU nodes. The central processing node, as shown in Figure 2, also performs template averaging. Ideally, if the collecting/averaging/updating time on the server is negligible in comparison to clients' processing time, the speed gain of the whole system will increase linearly with the number of processing nodes.

We tested this hypothesis on a dual-GPU system having two Quadro FX 5600 cards, connected by a PCI 16x bus on the same motherboard. The server in this case is simply the CPU. The processing time with a 160-by-192-by-160 volume input is 20s on the dual-GPU system versus 30s on the single GPU dual-input version. The actual improvement is only 75%, mainly due to the limitation of bandwidth between the CPU and the GPUs. In addition, the averaging function is also 20 times slower on the CPU than on the GPU. These two effects in combination result in the averaging cost being about 30 times more than on one GPU. On the one GPU model, the averaging cost accounts for only 1% of the processing time. The multi-GPUs model will be more efficient with higher CPU-GPU bandwidth, however, a high bandwidth media between CPU-multi-GPUs is expensive.

The cost of memory, computational power, and processing time make up the total cost of the parallel system. We want to design a low cost parallel system while maintaining the capability to handle a large number of inputs. In practice, the number of inputs is larger than the maximum capability of a single processing node, and also bigger than the number of processing nodes in a multi-processor system. Because the inputs may have different sizes, we estimate the cost efficiency of the system on average, per volume element. In terms of efficiency cost, the simple multi-GPU configuration is cost inefficient because the input volume size is normally smaller than the GPU memory; an one-to-one configuration costs the most per volume element. It also wastes processing power as during the synchronization the processors are idle. As the number of processing nodes increases the amount of data transfer between server and clients increases linearly while the processing cost on each GPU remains constant. Consequently, the processors spend more time in the idle state and the collecting/updating process becomes the main bottleneck of the algorithm. To increase the cost efficiency of the system with a large number of inputs, we propose a hybrid model, Multiple Inputs-Multi-GPUs, where each GPU handles multiple inputs.

4.5.3. Multiple Inputs - Multi-GPUs model

The key idea to improve the efficiency is to maximize the total volume of inputs that the system can handle, in other words, we increase the arithmetic intensity of each processing node by maximizing the number of inputs per node. We divide the inputs between nodes. At each processing node, we exploit the one-GPU-multiple-inputs strategy to compute and share a GPU's local template average among inputs of the same context. We send this template average with a weight proportional to the number of inputs of the node to the server to compute the global average, then we send the new average back to GPUs to update it in the local average memory. While retaining the benefits of the former model, this strategy minimizes data transfer to and from the GPUs to the volume of a local common average. This hybrid model, as shown in Figure 5, minimizes both the overall cost per volume element and the data transfer over the low bandwidth channel. Thus, it maximizes the cost efficiency. Our results show that the updating cost on dual-GPU drops from 25% of the overall processing time with the one-to-one input mapping down to only 5% with the 6 inputs per GPU. It gives us the ability to process one hundred inputs with a typical size of 100-by-100-by-80 on the dual Quadro FX 5600 system. This extension allows us to perform several meaningful randomized statistical tests (i.e., Monte-Carlo simulations or permutation tests) which are infeasible with the one-GPU-multiple-inputs or one-to-one Multi-GPUs model.

5. Results

To demonstrate our system's capabilities, we applied the algorithm to a database of 36 MR brain image volumes with 256^3 volume size. The database contains T1-FLASH images from 36 healthy adults at different ages. The images were acquired at a spatial resolution of 1mm$\times$1mm$\times$1mm, and the tissue around to the brain was removed. All images were intensity normalized and aligned using affine registration.

We further pre-process the inputs to improve the performance by removing the redundant zero-data outside the bounding box region. Typical 3D brain volume images have high redundancy ratios, the data volume is about one third to one fourth the unprocessed input volume. As a result, we experienced 3 to 4 times speed up just by tightly clipping the volume to the non-zero data bounding box. However, this optimization could be used with the SOR method only, as the FFT PDE solver requires a power of two input volume size to be computationally efficient, hence it runs slower with non power of two inputs, as shown on Figure 8, the processing

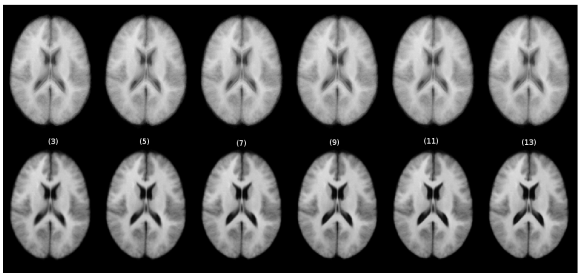

Figure 6: *Atlas results with 3, 5, 7, 9, 11 and 13 inputs constructed by (a) arithmetically averaging rigidly aligned images (top row) and (b) Greedy Iterative Average template construction (bottom row)*

time of 96-by-96-by-96 inputs is equal to that of 128-by-128-by-128 inputs with FFT-based approach.

5.1. Atlas Stability Test

To evaluate the robustness and stability of the atlases, Lorenzen *et al.* [LDJ05] used a random permutation test on 14 brain images to estimate the minimum number of inputs required to construct a stable atlas, proving the stability of greedy iterative diffeomorphism in 2D by analyzing mean entropy and variance of the average template. However, this work is limited to 2D atlas building on mid-axial slices, due to the high computational complexity of the problem in 3D. In our work, we are able to perform similar tests in full 3D in reasonable time even with a larger input data set and a larger number of cohorts. We generated 13 atlas cohorts, $C_{l,l=2\cdots14}$, each including 100 atlases constructed from l input images chosen randomly from the original 36 brain volumes. The 2D mid-axial slices of the atlases are shown in Figure 6. The normal average atlases are blurry, ghosting is evident around the lateral ventricles and near the boundary of the brain, while the Greedy Iterative Average template appears to be much sharper and anatomical structures are preserved.

The quality of the atlas construction is visibly better than the least MSE normal average. The entropy results on Figure 7 also confirm the stability of our implementation. As the number of inputs increases, the average atlas entropy of the simple averaging intensity increase while the Greedy Iterative Average template decrease due to much higher individual's sharpness. This quantitatively asserts the visible quality improvement in Figure 6. The atlases become more stable with respect to the entropy as the standard deviation decreases with increasing number of inputs. After cohort C_8 the atlas entropy mean appears to converge. So we need at least 8 images to create a stable atlas representing neuroanatomy, this number is already over the capability of one GPU. Once again this shows the benefit of our proposed framework.

5.2. Performance result

In the GPU processing framework, we build a CUDA image processing library, which is optimized to give the best performance with each image processing function and on specific range of input sizes. To achieve this, we apply the C++ template design and the parallel unrolling technique to reduce the loop overhead. We employ the Structure of Arrays

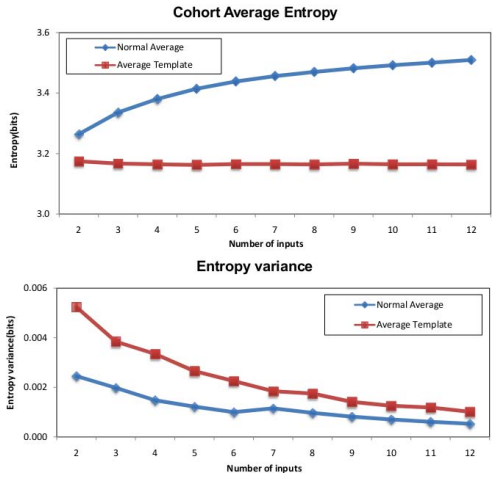

Figure 7: *Mean entropy and variance of atlases constructed by arithmetically averaging and the Greedy Iterative Average template*

Table 1: *Speed up factor of the GPU and optimized-GPU functions over the reference CPU version. Note that the non-optimized GPU reduction is slower than that of the CPU*

N	Basic Opers		Gradient		Reduction		Rev. Map	
0.5M	23	24	18	104	.78	17	5	61
1 M	23	24	18	109	.86	23	9	56
2 M	23	24	18	117	.91	26	5	62
4 M	24	25	20	120	.93	30	10	61
8 M	24	25	21	126	.94	31	4	58
16 M	24	26	21	129	.95	32	10	54

instead of Array Of Structures, and utilize the CUDA Visual Profiler to analyze the performance and guarantee that the global memory access is coalesced. Table 1 shows the speed gain comparison of a trivial non-optimized GPU implementation and the optimized GPU version over a optimized CPU implementation. The results show that we only gain performance when the size of inputs is big enough, so that we efficiently hide the memory latency in the GPU implementation. We experienced a typical 20-50 time speedup over CPU version with optimized GPU function in image processing applications that yields an overall speed up factor of 20 with image diffeomorphism problem. Together with tightly trimming volume redundancy, an overall 30-60 performance gain over optimized CPU FFT version can be seen.

Table 2 shows the runtime contribution of each step in the algorithm using FFT and the SOR PDE solver with 192^3 inputs over 100 iterations. In our optimized SOR version, PDE solver accounts only 50 to 64% overall processing time while it took 90% with FFT-based PDE solver. Figure 8 shows the overall runtime per 100 iterations of pair-

Table 2: *Runtime contribution of FFT and SOR method*

	Diff	Grad	Force	PDE	ODE	Total
FFT	.32%	1.8%	.72%	88%	8.75%	59.6s
SOR	1%	5.6%	2.3%	64%	26.8%	20.5s

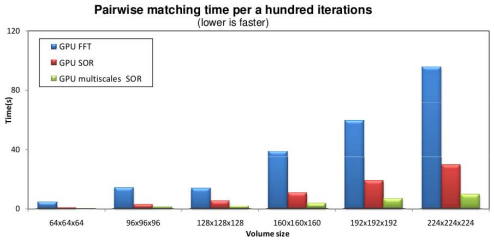

Figure 8: *Overall runtime of the SOR and FFT implementations over 100 iteration with different volume size*

wise matching using FFT, SOR, and the multiscale SOR approach. It is clear that the SOR implementation is a magnitude faster than the FFT, and the multiscale SOR is a magnitude faster still. The linear growing curves also shows the scalability of the SOR approaches.

6. Conclusion and Future work

We have presented a multi-GPUs framework for unbiased diffeomorphic atlas construction. This framework gives a substantial performance increase over previous implementations. We were able to perform expensive tasks like random permutation tests over a large number of input volumes.

Atlas building on the GPU provides an inexpensive and fast alternative for many applications, such as 3D volume segmentation, anatomical labeling, and robust image registration. We can see many potential applications for this framework in many other image processing problems that require huge computational power and massive input data set, especially in brain image processing. We optimized all operations involved in the computational process, it is necessary when we have to process thousands of inputs where small optimizations would yield hours less in the running time.

In the future, we want to investigate the scenarios with even a larger scale multi-GPUs supercomputing systems, providing petaflop processing power at an affordable price. With these multi-GPUs systems, we can perform very large population tests. We believe that our GPU framework can be generalized to a dataflow parallel processing model, providing a replacement for the existing sequential general computing model.

7. Acknowledgements

This research has been funded by the National Science Foundation (grants CNS-0751152, CCF-0528201, OCE-0424602, CNS-0514485, IIS-0513692, CCF-0401498, OISE-0405402, CNS-0551724), the Department of Energy SciDAC (VACET and SDM centers), and IBM Faculty Awards (2005, 2006, and 2007); It also was made possible in part by the NIH/NCRR Center for Integrative Biomedical Computing, P41-RR12553-10 and by Award Number R01EB007688 from the National Institute Of Biomedical Imaging And Bioengineering. L. Ha was partially supported by the Vietnam Education Foundation fellowship.

References

[BNG96] BRO-NIELSEN M., GRAMKOW C.: Fast fluid registration of medical images. In *VBC '96: Proceedings of the 4th International Conference on Visualization in Biomedical Computing* (London, UK, 1996), Springer-Verlag, pp. 267–276.

[Buc05] BUCK I.: *Stream computing on graphics hardware*. PhD thesis, Stanford University, Stanford, CA, USA, 2005.

[CMVG96] CHRISTENSEN G. E., MILLER M. I., VANNIER M. W., GRENANDER U.: Individualizing neuroanatomical atlases using a massively parallel computer. In *Computer* (1996), vol. 29, IEEE Computer Society, pp. 32–38.

[Dem97] DEMMEL J.: *Applied Numerical Linear Algebra*. SIAM, 1997.

[DFBJ07] DAVIS B., FLETCHER P., BULLITT E., JOSHI S.: Population shape regression from random design data. *Computer Vision, 2007. ICCV 2007. IEEE 11th International Conference on* (Oct. 2007), 1–7.

[DS07] DANDEKAR O., SHEKHAR R.: Fpga-accelerated deformable image registration for improved target-delineation during ct-guided interventions. *Biomedical Circuits and Systems, IEEE Transactions 1*, 2 (2007), 116–127.

[FJ05] FRIGO M., JOHSON S. G.: Design and implementation of fftw3. In *Special Issue on Program Generation, Optimization, and Platform Adaptation* (Feb 2005), vol. 93, pp. 216–231.

[Gre94] GRENANDE U.: *General Pattern Theory: A Mathematical Study of Regular Structures*. Oxford Unversity Press, 1994.

[HHHL07] HAGEN T. R., HENRIKSEN M. O., HJELMERVIK J. M., LIE K.-A.: *How to Solve Systems of Conservation Laws Numerically Using the Graphics Processor as a High-Performance Computational Engine*. Springer Berlin Heidelberg, 2007, pp. 211–264.

[HLN06] HAGEN T. R., LIE K.-A., NATVIG J. R.: *Solving the Euler Equations on Graphics Processing Units*. Springer, 2006, pp. 220–227.

[JDJG04] JOSHI S., DAVIS B., JOMIER M., GERIG G.: Unbiased diffeomorphic atlas construction for computational anatomy. *Neuroimage 23 Suppl. 1* (2004), S151–S160.

[JFTW07] JEONG W.-K., FLETCHER P. T., TAO R., WHITAKER R.: Interactive visualization of volumetric white matter connectivity in dt-mri using a parallel-hardware hamilton-jacobi solver. *IEEE Transactions on Visualization and Computer Graphics 13*, 6 (2007), 1480–1487.

[KW03] KRÜGER J., WESTERMANN R.: Linear algebra operators for gpu implementation of numerical algorithms. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers* (New York, NY, USA, 2003), ACM, pp. 908–916.

[LDJ05] LORENZEN P., DAVIS B., JOSHI S.: Unbiased atlas formation via large deformations metric mapping. In *Med Image Comput Comput Assist Interv Int Conf Med Image Comput Comput Assist Interv (MICCAI)* (2005), Duncan J., Gerig G., (Eds.), vol. 8 (Pt. 2), pp. 411–418.

[NVI07] NVIDIA: *CUDA FFT Library*. Tech. rep., NVIDIA Corporation, 2007.

[RS01] RUMPF M., STRZODKA R.: Using graphics cards for quantized fem computations. In *In Proceedings VIIP'01* (2001), pp. 193–202.

[SDR04] STRZODKA R., DROSKE M., RUMPF M.: Image registration by a regularized gradient flow - a streaming implementation in DX9 graphics hardware. *Computing 73*, 4 (Nov. 2004), 373–389.

[ZWK*06] ZIKIC D., WEIN W., KHAMENE A., CLEVERT D.-A., NAVAB N.: Fast deformable registration of 3d-ultrasound data using a variational approach. In *MICCAI (1)* (2006), pp. 915–923.