Multi-scale Unbiased Diffeomorphic Atlas Construction on Multi-GPUs

Linh Ha*, Jens Krüger*, Sarang Joshi*, Cláudio T. Silva*

August 9, 2010

In this chapter, we present a high performance multi-scale 3D image processing framework to exploit the parallel processing power of multiple graphic processing units (Multi-GPUs) for medical image analysis. We developed GPU algorithms and data structures that can be applied to a wide range of 3D image processing applications and efficiently exploit the computational power and massive bandwidth offered by modern GPUs. Our framework helps scientists solve computationally intensive problems which previously required super computing power. To demonstrate the effectiveness of our framework and to compare to existing techniques, we focus our discussions on atlas construction - the application of understanding the development of the brain and the progression of brain diseases.

1 Introduction, Problem Statement and Context

1.1 Atlas construction problem

The construction of population atlases plays a central role in medical image analysis, particularly in understanding the variability of brain anatomy. The method projects a large set of images to a common coordinate system, creating a statistical average model of the population, and doing regression analysis of anatomical structures. This average serves as a deformable template which maps detailed atlas data such as structural, developmental, genetic, pathological, and functional information on to the individual or entire population of the brain. This transformation encodes the variability of the population understudy. Likewise, the statistical analysis of the transformation between populations reflects the inter-population differences. Apart from providing a common coordinate system, the atlas can be partitioned and labeled, thus providing effective segmentation via registration of anatomical labels.

The brain atlas construction is a powerful technique to study the physiological, evolutionary and developmental of the brain, as well as disease progression. Two desired properties of the atlas construction are that it should be diffeomorphic and non-biased.

In non-rigid registration problems, the desired transformations are often constrained to be diffeomorphic, i.e. continuous, one to one (invertible) and smooth with a smooth inverse so that the topology is maintained. Connected sets remain connected, disjoint sets remain disjoint, neighbor relationships between structures as well as smoothness of features such as curves are preserved, and coordinates are transformed consistently.

Preserving topology is important for synthesizing the atlas since the knowledge base of the atlas is transferred to the target anatomy through topology preserving transformation providing automatic labeling and segmentation. Moreover, important statistics such as the total volume

*Scientific Computing and Imaging Institute, University of Utah, Salt Lake city, UT84112

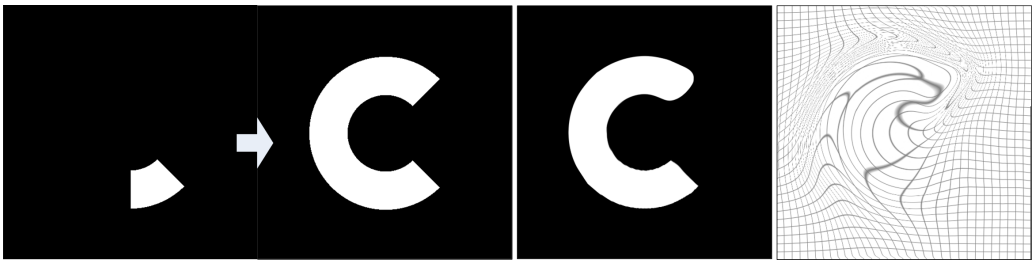

Figure 1: A small part of the letter "C" deforming into a full "C" using 2D Greedy Iterative Diffeomorphism. From left to right: 1. Input and Target Image 2. Deformed template. 3. Grid showing the deformation applied to template

of a nucleus, the ventricles, or the cortical sub region can be generated automatically. The diffeomorphic mapping from the atlas to the target can be used to study the physical properties of the target anatomy such as mean shape and variation. Also, the registration of multiple individuals to a standard atlas coordinate space removes the individual anatomical variation and allows information to be combined with a single conical anatomy. Figure 1 shows that the diffeomorphic setting results in a high quality deformation field which is infinitely smooth on a non-self-crossing grid.

The non-bias property guarantees that the atlas construction is consistent. Our atlas construction framework, first proposed by Joshi *et al.* [9], is based on the notion of Frechet mean to define a geometrical average. On a metric space M, with a distance $d : M \times M \rightarrow R$ the intrinsic average μ of a collection of data x_i is defined as a minimizer of the sum-of-square distances to each individual, that is

$$\mu = \underset{p \in M}{\operatorname{argmin}} \sum_{i=1}^{N} w_i d^2(p, x_i)$$

As the computation of the Frechet mean is independent from the order of the inputs, the atlas is inherently non-biased. The Frechet mean is also rational in terms of minimizing the total energy to deform an average to all images in a population.

The combination of both diffeomorphic and non-bias property results in a minimization energy template problem which is formulated as

$$\{\hat{h}_i, \hat{I}_i\} = \underset{h_i, I}{\operatorname{argmin}} \sum_{i=1}^{N} \int_{\omega} (I_i \circ h_i - I)^2 + \int_0^1 \int_{\omega} ||Lv_i(x,t)||^2 dx dt \tag{1}$$

subject to $h_i(x) = \int_0^1 v_i(h_i(x,t),t)dt$ (*)

This is a dual optimization problem on the image matching (the first term) and deformation energy (the second term). The L-operator is a partial differential operator which controls the smoothness of the deformation field. The constraint (*) comes from the theory of large deformation diffeomorphism that the transformations h_i are generated by integrating velocity field v_i forward in time. The method is the extension of elastic registration to handle large deformations.

While the optimization seems intractable, by noting that for fixed transformation h_i the best estimation of the average $\hat{I}$ is given by $\hat{I}(x) = \frac{1}{N} \sum_{i=1}^{N} I_i(h_i)$, we come up with a simple solution based on an alternating optimization, as shown on Algorithm 1. In each step, we estimate the atlas by averaging the deformed images, then we compute the optimal velocity fields by solving optimization problems on deformation energy, finally the deformed images are then updated. This process is repeated until convergence. Note that, with the assumption of a fixed

template on the second step, the optimal velocity of an image can be computed independently from the others. This velocity is determined by solving the pairwise matching problem. By tightly coupling the atlas construction problem with basic registration problems — the pairwise matching algorithms — our framework allows to implement different techniques and even to combine multiple techniques into a hybrid approach.

Algorithm 1 Atlas construction framework

1: **Input** : N volume inputs
2: **Output**: Template atlas volume
3: **for** $k = 1$ to max_iters **do**
4: Fix images I_i^k, compute the optimal template $\hat{I}^k = \frac{1}{N}\sum_{i=1}^{N} I_i^k$
5: **for** $i = 1$ to N **do** {loop over the images}
6: Fix the template $\hat{I}^k$, solve pairwise-matching problem between I_i^k and $\hat{I}^k$
7: Update the image with optimal velocity field
8: **end for**
9: **end for**

1.2 Challenges

As shown above, unbiased diffeomorphic atlas construction is a powerful method in Computational Anatomy. However, the impact of the method was limited because of two main challenges: the extensive memory requirement and the intensive computation.

The extensive memory requirement is one of the major obstacles of 3D volume processing in general, as the size of a single volume often exceeds the available memory on a processing node. This becomes more challenging on GPUs as they have less memory available. In addition, the atlas construction process requires not just a single volume but a population of hundreds of volumes, which easily exceeds the available memory of pratically any computational system.

The massive size of the problem is compounded with the complexity of the computation per element. These computations are often not just simple, local kernels but global operations, e.g. an ODE integration using a backward mapping technique.

Generating a brain atlas at an acceptable resolution for a reasonably sized population took an impractically long time even with a fully optimized implementation on high-end CPU workstations or small CPU clusters [6]. Acceptable run times could only be obtained by utilizing super computer resources. [4, 3]. Hence, the technique was restricted to the baseline research community. Fortunately, the development of GPUs in recent years and especially the introduction of a general processing model and a unified architecture brings a more accessible solution. Our results show that an implementation on GPU can handle practical problems on a single desktop with a substantial performance gain, on the order of twenty to sixty times faster than a single CPU.

Our multi-GPU implementation on a 32-node GPU cluster is roughly a thousand times faster than a 32-core CPU workstation. We are able to handle a large data set of 315 T1 brain images at size $114 \times 192 \times 160$ using only 8 cluster nodes. The ability to solve the problem on small-scale computing systems opens opportunities for researchers to fully understand practical impacts of the method and to enhance their knowledge of anatomical structures.

In addition to providing a practical solution to a specific problem, we also present a general image processing framework and optimization techniques to exploit the massive bandwidth and computational power of multi-GPU systems and to handle compute expensive problems previously available only on supercomputer.

Algorithm 2 Greedy pairwise matching step

1: **Input** : Original image I_0, target I_1, deformed image $I_0(t)$, deformation field h
2: **Output**: New deformed image $I_0(t)$, deformation field h
3: Compute the force $F = -\left[I_0(t) - I_1\right] \bigtriangledown I_0(t)$
4: Solve the PDE $Lv(x) = F(x)$ where $L = \alpha\nabla^2 + \beta\nabla\nabla + \gamma$
5: Update the deformation $h_{new} = h_{cur}\left(x + \epsilon v(x)\right)$
6: Update the transform image $I_0(t) = I_0\left(h_{new}(x)\right)$

While we have integrated a number of different techniques into our system, such as the Greedy Iterative Diffeomorphism, Large Deformation Diffeomorphic Metric Mapping (LDDMM), and Metamorphoses, in this chapter we focus on the implementation of the greedy method on GPUs.

2 Core Methods

The Greedy Iterative Diffeomorphism was proposed by Christensen [5]. The method separated the time dimension from the space dimension of the problem. At each iteration, a new optimal velocity field is computed given that the current deformation is fixed (i.e. the past velocity fields are fixed). The solution is computed by integrating the optimal solution at each step forward in time in a gradient descent approach. The method is locally-in-time optimal, thus in general will not produce the shortest path connecting images through the space of diffeomorphism. However, it is generally preferred in practice since it can produce good results with less computational expense than the other approaches. The Greedy Iterative Diffeomorphism is built on the general framework with the Greedy Pair-wise Matching algorithm at its core (Algorithm 2)

There are several performance keys of a GPU implementation: high throughput data structures and basic functions, high performance advance functions: optimal ODE integration and PDE solvers, and multi-resolution and multi-GPU strategies. We will discuss in detail how to achieve the peak performance in the following section.

3 Algorithms, Implementations and Evaluations

3.1 Data structures

While it is typically recommended to have volume data padding so that the volume dimensions are multipliers of the GPU warp size, our experiment showed that it has negligible effect on improving the performance. The reason is that as GPU data parallel fetching strategies become more sophisticated and efficient, the coalesced condition is greatly relieved from CUDA 1.0 to CUDA 2.0 hardware so it is easy to achieve with regular image functions. Moreover, data padding significantly increases the storage requirement, especially in 3D, and requires extra processing steps. For these reasons, we chose a tight 3D volume representation which can represent a 3D volume as 1D vector, thus most of the basic operations on 1D can be directly applied to 3D. Additionally, we often save two integer shared memory locations and operations per kernel by passing a single volume value instead of three dimensional numbers.

We also define a special structure for 3D vector fields based on a Structure of Array representation. Instead of allocating three separated GPU pointers, we allocate a single memory block and use an offset to address the three components. This presentation allows us to optimize basic operations on the vector field, most of the time as a single image operation. Moreover, it helps us save one shared memory pointer per kernel.

3.2 Basic image operators

The goal of our system design is to be able to run the entire processing pipeline on GPUs. This allows one to maximize the computational benefit from GPUs and minimizes idle time. We keep data-flow running on the GPUs, and only use CPUs for cross GPU and cross CPU operations. With the design goal in mind to be optimal, even on a per function level, we provide n-ary basic-3D functions.

The performance of the basic function is constrained by the global memory bandwidth. To improve the performance we need to minimize the bandwidth usage. Most of the functions provided by regular processing libraries such as Thrust [8] or NPP (`http://developer.nvidia.com/object/npp_home.html`) are unary or binary functions which involve one or two arguments as the inputs. Though any n-argument function can always be decomposed into a set of unary and binary functions, this decomposition requires extra memory to store intermediate results, and increases bandwidth utilization by saving and/or reloading the data. Our n-ary operators, on the other hand, load all the components of an n-argument function to the register files at the same time, hence no extra saving/loading is required. This allows for optimal memory bandwidth usage. For example, if we consider the image loading operator being one memory bandwidth unit, then the linear interpolation $x = a * y + b * z$ requires 7 units with binary decomposition, while optimal bandwidth are 3 units which is achievable with n-ary operators. The bandwidth ratio is also our expected speed up of our n-ary versus binary functions. In terms of storage requirement, the binary decomposition double the memory requirement by introducing an extra template memory per operation, while n-ary functions require no extra memory.

In addition to providing all the basic operation similar to those of the Thrust library [8], we implement n-ary functions combining up to five operations. We also offer n-ary in-place operators which consume fewer registers and less shared memory. Naming of these functions reflects their functionality to preserve the readability and maintainability of the code and to allow further automatic code generation and optimization by the compiler. As shown in Figure 2, our normalization function and linear interpolation achieves speed up factors of up to 2-3 over the implementation using optimized binary operators.

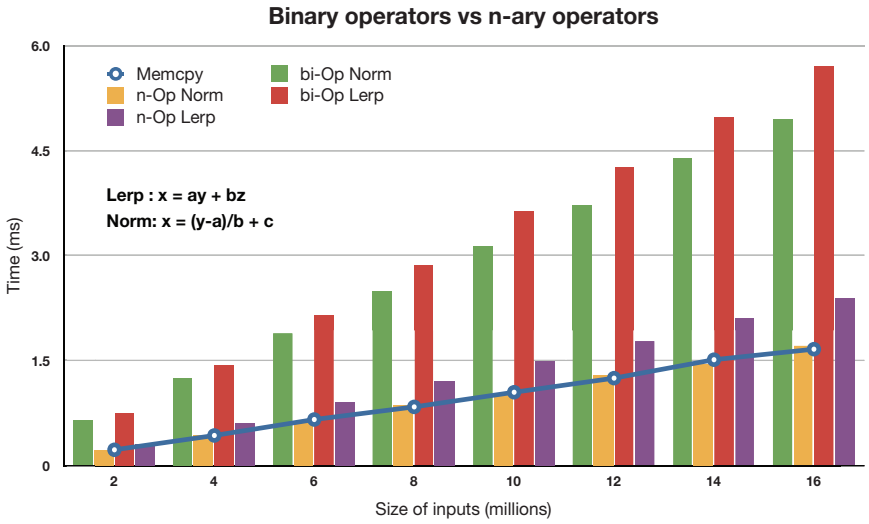

Figure 2: n-ary versus binary operator with linear interpolation and range normalization function in comparision to the device device memory copy. Note the efficiency of our n-ary approach as compared to the classic binary operator approach.

Based on the same strategy of n-ary operators, we propose a parallel efficient average function

with hand-tuned performance for all number of inputs from 1 to 8 as illustrated in Figure 3.

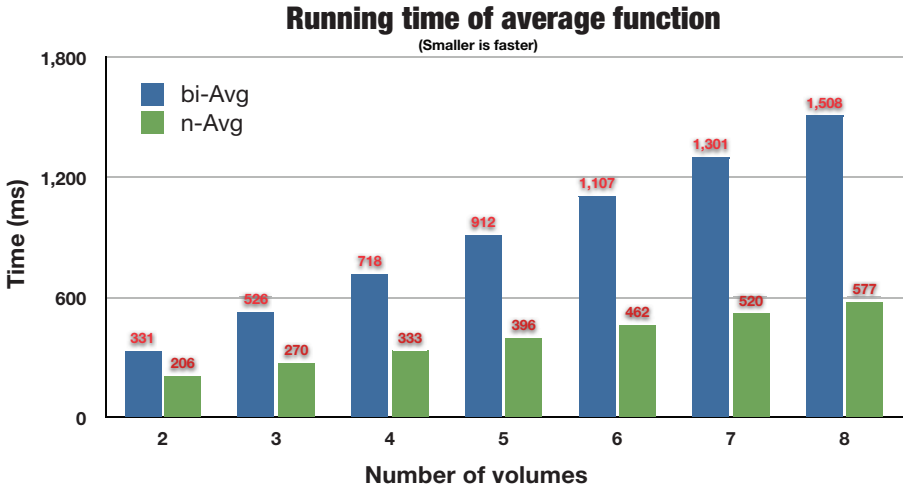

Figure 3: n-ary average function versus binary average operator

Gradient computation is a frequently-used and essential function in image processing. Based on the locality of the computation, several optimization techniques may be applied such as 1D linear texture cache, 3D texture, or implicit cache through shared memory. Among these techniques, we found the 3D stencil method [11] using the shared memory the most effective. The following table shows the runtime comparison in milliseconds of different gradient computations: simple approach, linear 1D texture, 3D texture and our shared memory implementation.

Gradient Method	Simple	1D Linear	3D texture	Shared
$160 \times 224 \times 160$	3.4	3.0	6.8	1.6
$256 \times 256 \times 256$	9.5	8.9	21	5.2

The result shows that gradient computation on shared memory exploiting 3D stencil technique is twice as fast as using the linear texture cache.

3.3 ODE integration

The ODE integration computes the deformation field by integrating velocity along the evolution path. A computationally efficient version of ODE integration is the recursive equation that computes the deformation at time t based on the deformation at time $t-1$, that is $h_t = h_{t-1}(x + v(t-1))$. This computation could be done by the reverse mapping operator (Figure 4), which assigns each destination grid point a 3D interpolation value from the grid neighbor points in the source volume. Fortunately, on GPUs, this 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.

3.4 PDE Solver

As shown in Algorithm 2, optimal velocity is computed from the force function by solving the Navier-Stokes equation

$$\alpha \nabla^2 v(x) + \beta \nabla \nabla v(x) + \gamma v(x) = F(x) \tag{2}$$

Often β is negligible and Equation (2) simplifies to the Helmholtz equation

$$\alpha \nabla^2 v(x) + \gamma v(x) = F(x) \tag{3}$$

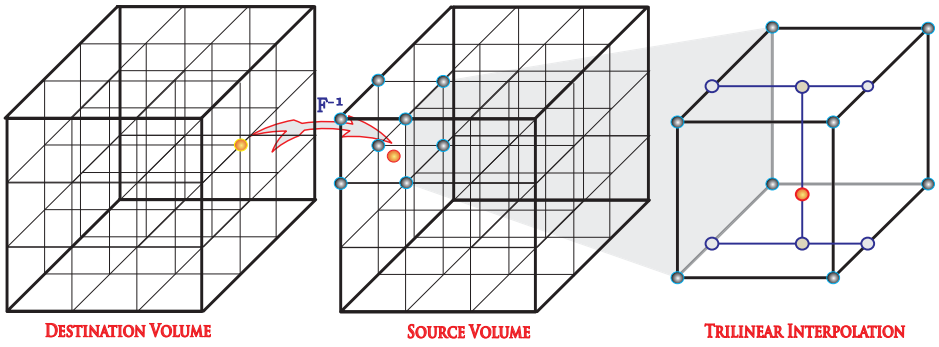

Figure 4: Reverse mapping based on 3D tri-linear interpolation

where $\alpha = 0.01$ and $\gamma = 0.001$ are generally used in practice. Note that, there is no crossing term in Helmholtz equation, that means the solver could independently run on each dimension.

While the ODE computation can be easily optimized simply by utilizing the 3D hardware interpolation, the PDE solver is less amenable to GPU implementation. The PDE is a sparse linear system with size $N^3 \times N^3$, where N^3 is the volume of the input. A direct dense linear package such as CUDA BLAS can not handle the problem. What we need is a sparse solver. There are many different methods to solve a sparse linear system. The two most common and efficient ways are explicit solvers in the Fourier domain and implicit solvers using iterative refinement methods such as Conjugate Gradient(CG), Successive Over Relaxation(SOR) or multigrid.

In our framework we support different methods such as FFT, SOR, and CG. There are multiple reasons to support multiple techniques rather than a single method. Although the FFT solver is the slowest, it produces an exact PDE solution. While the others are significantly faster, they only produce approximate solutions, which often have local smoothing effects. The inability to account for the influence of spatially distant data points in the initial solution slows-down the convergence rate of these methods in the long run. Consequently, they require more iterations to achieve the same result as the FFT approach. Due to smoothing properties of the velocity field, the variance in the solution of the PDE solver between two successive steps is often small. This variance can be captured adequately by the iterative solvers in a few iterations. This is made possible by using the previous solution as an initial guess for the iterative solver in the next step. For the first iteration, without a proper guess, iterative solvers are often slow to converge, so they require a large number of iterations and may quickly become slower than the FFT approach. Therefore, we use an FFT solver in the first iteration and then switch to iterative methods. Our experiments show that the hybrid CG solver that starts with an FFT step produces exactly the same results as an FFT method, but is almost three times faster.

For the details on the FFT solvers we refer reader to [12]. Here we will discuss the implementation of SOR and CG methods.

3.5 Successive Over Relaxation method

Successive over-relaxation (SOR) is an iterative algorithm proposed by Young for solving a linear system [13]. Theoretically, 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 account for 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. The same argument is true for other iterative methods such as CG.

We observe that in the elastic deformable framework with steady fluids, the changes in the velocity field are quite small between greedy steps. The computed velocity field of the previous

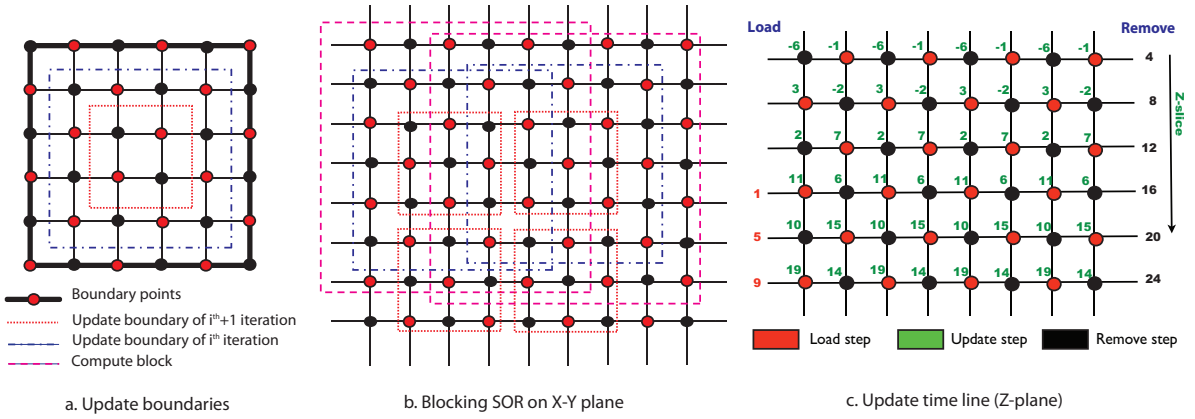

Figure 5: Parallel block SOR, we assign each CUDA thread block a block of data to compute the black points inside the blue boundary, and use that result to compute the red point inside the red boundary. Two neighboring compute blocks share a four grid point-wide region.

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.

Our framework provides an SOR implementation with Red-Black ordering as shown in Figure 5. This strategy allows for efficient parallelism as we only update points of the same color based on their neighbors, which have different color. Also, Red-Black decoupling is proved to have a well-behaved convergence rate with the optimal over-relaxation factor ω defined in 3D as $\omega = \frac{2}{1+\sqrt{1-\frac{1}{3}\left[\cos\frac{\pi}{w}+cos\frac{\pi}{h}+cos\frac{\pi}{l}\right]^2}}$

We incorporate optimization techniques from the 3D stencil buffer problem to exploit the fast shared memory available in CUDA and improve the register utilization of the algorithm (Algorithm 3). We further improve the performance by increasing the arithmetic intensity of the data. This is done with merging steps of SOR that combines red-update steps and black-update step of traditional SOR in one execution kernel. We also proposed *block-SOR* algorithm that we divide input volume into blocks, each fitting onto one CUDA execution block. We then exploit the shared memory to perform the merging step locally on the block. For simplicity, we illustrate the idea in 2D in Figure 5, but it is generalized to arbitrary dimensions.

As shown in Figure 5 the updated volume is two-cells smaller in each dimension than the input. This reduction in size explains why we can not merge an arbitrary number steps in one kernel. To update the whole volume, we allow data overlaps among processing blocks, as shown in Figure 5(b). Here, we allow data redundancy to increase memory usage. The configuration shown in Figure 5(b), having a 4 point-wide boundary overlap, is able to update of one Red-Black merging step over a M^2 block using $(M+4)^2$ input. Likewise, a k-merging step needs a data block of size $(M+4*k)^2$. To quantify the benefit of SOR merging steps, we compute a trade-off factor α such that:

$$\alpha = \frac{\text{Minimum needed data size}}{\text{Actual processing data size}} * \text{Speed\_up\_factor} \tag{4}$$

In 3D, to update the volume block M^3, we need $(M+4k)^3$ volume inputs, the trade-off factor is $\alpha = (\frac{M+1}{M+4k})^3 * k$. Note that, the size of shared memory constraints the block size M and merging level k. In practice, we see benefits only if we merge one black & red update step per kernel call.

Algorithm 3 shows the pseudocode of our block-SOR implementation on CUDA. We further leverage the trade-off by limiting block-SOR in the 2D plane only, and exploit the coherence between consecutive layers in the third dimension to minimize data redundancy. On the Tesla,

Algorithm 3 Efficient CUDA PDE block-SOR solver

1: **Input** : Old velocity field v and new force function F
2: **Output**: Compute new velocity field v
3: Allocate 4 shared mem arrays s_{prev},s_{cur}, s_{next}, s_{next2} to store 4 slices of data
4: Load F of 3-first slices to the registers of current thread
5: Load v of 3-first slices to registers and shared memory
6: The boundary thread load the padding data of v
7: Update the black point of the second slice s_{cur}
8: **for** $k = 1$ to $Z - 2$ **do** {loop over Z direction}
9: Load the F and v of the next slice to the free shared-mem array s_{next2}
10: Update the black points of the s_{next} slice
11: Update the red points of the s_{cur} slice
12: Write the s_{prev} to the global output, s_{prev} buffer is free to load the next slice
13: Shift the value of v and F in the registers, $cur \rightarrow prev$, $next \rightarrow cur$, $next2 \rightarrow next$
14: Circular shift the shared memory array pointers, s_{prev} becomes s_{next2}
15: **end for**
16: Update the red points on the last slice close to boundary
17: Write out the last slice

our block-SOR implementation using shared-memory is twice faster than equivalent version using 1D texture cache. Figure 5(c) shows the updating time line in Z-dimension, it is clear that each node is computed by its neighbors which are updated in previous steps.

3.6 Conjugate Gradient Method

While the SOR method is specialized for solving PDEs on a regular grid, in practice the Conjugate Gradient approach is often the preferred technique because of several advantages:

- It is capable of solving a PDE on irregular grid as well.
- It is simple to implement as it built on the top of basic linear operations.
- In general, it converges faster than the SOR method.

As shown in Figure 7, CG algorithm is implemented in our framework as a template class with T being the matrix presentation of the system. The only function required from T is a matrix vector multiplication. The template allows for the integration of any sparse matrix vector multiplication package using an explicit presentation such as ELL, ELL/COO [2] and CRS [1], or an implicit presentation which encodes the system matrix with constant values in the kernel.

Figure 6 is the implementation of the implicit matrix vector multiplication with Helmholtz Matrix and zero-boundary condition. The texture cache is used to access neighboring information to achieve maximal memory bandwidth. Our experiments showed that in the case of regular grid, the implicit approach allows for a more efficient matrix vector multiplication as the matrix does not consume memory bandwidth. As shown on the table, it is up to 2.5 times faster than explicit implementations [2]. The performance is measured in GFLOPs with Helmholtz Matrix vector multiplication.

Matrix size	64^3	96^3	128^3	160^3	192^3	224^3	256^3
Implicit	17	37	53	42	54	51	59
Explicit Dia	25	27	27	25	25	25	27
Explicit Ell	16	17	17	16	16	16	16

```
__global__ void helmholtz3D_MV(float* b, float* x,
    float alpha, float gamma, int sizeX, int sizeY, int sizeZ){
    uint xid      = threadIdx.x + blockIdx.x * blockDim.x;
    uint yid      = threadIdx.y + blockIdx.y * blockDim.y;
    uint id       = xid + yid * sizeX, planeSize= sizeX * sizeY;
    if (xid < sizeX && yid < sizeY){
        float zo = 0, zc = fetch(id, x), zn;
        for (uint zid=0; zid<sizeZ; ++zid, id += planeSize){
            zn = (zid + 1 < sizeZ) ? fetch(id + planeSize, x) : 0;
            float r  = zo + zn;
            r += (xid   > 0)      ? fetch(id - 1, x)      : 0;
            r += (xid + 1 < sizeX)? fetch(id + 1, x)      : 0;
            r += (yid   > 0)      ? fetch(id - sizeX, x)  : 0;
            r += (yid + 1 < sizeY)? fetch(id + sizeX, x)  : 0;
            b[id] = zc * (6 * alpha + gamma) - alpha  * r;
            zo = zc; zc  = zn; // shift values on Z-dir
        }
    }
}
```

Figure 6: Matrix vector multiplication CUDA kernel with implicit Helmholtz Matrix

3.7 Multi-scale framework

The concept of our multi-scale framework is derived from the multigrid technique, which computes an approximate solution on a coarse grid and then interpolates the result onto the finer grid. As the solution on the coarse grid generates a good initial guess of solution on the finer grid, it speeds up the convergence on the finer level. In addition to reduce the number of iterations, multi-resolution increases the robustness with respect to noise in the input data while it is capable of handling local optimums inherent to gradient-descent optimization. We design a multi-scale GPU interface (Algorithms 4) based on two main components: the downscale Gaussian filter and the upscale sampler.

The downscaled filter is composed of a low-pass filter followed by a down-sampler. The low-pass filter is a 3D-Gaussian filter which is implemented using 1D-Gaussian separable filters along each axis. We discovered that it is more efficient to implement this 3D filter based on the separable-recursive Gaussian filter rather than convolution based or FFT-based approaches. Our recursive version is generalized from the 1D recursive version (see NVIDIA SDK *RecursiveGaussian*) with a circular-dimension shifting 3D transpose. As shown on the table, the 3D recursive version is the fastest, and its runtime, measured in milliseconds, is independent of the kernel size. The other methods in comparison are: a separable filter, a circular-dimension shifting combined with the 1D filter in the fastest dimension, and a FFT-based filter.

Half kernel size	Separable	Dim-shift	Recursive	FFT
2	14	17	10	85
4	26	28	10	85
8	49	47	10	80

While the down-sampler simply fetches values from the grid, the up-sampler is the reverse-mapping operation from the grid point of the finer scale to the point value of the coarser grid based on the trilinear interpolation. Here we used the same optimization as for the ODE integration.

Our multi-resolution framework can be employed in any 3D image processing problem to improve both performance and robustness.

```
template< class T>
void CG_impl(float* d_b, T& d_A, float* d_x, int imax,
             float* d_r, float* d_d, float* d_q){
  int n = d_A.getNumElements();
  computeResidual(d_r, d_b, d_A, d_x);// r = b - Ax
  copyArrayDeviceToDevice(d_d, d_r, n); // d = r
  float delta_new = cplvSum2(d_r, n);  // delta_new = r^Tr
  float delta0 = delta_new, delta_old, eps=1e-4, alpha, beta;
  for (i=0; (i < imax) && (delta_new > eps * delta0); ++i)
    maxtrixMulVector(d_q, d_A, d_d);     // q = Ad
    alpha = delta_new/cplvDot(d_d, d_q, n);// alpha = delta_new/d^Tq
    cplvAdd_MulC_I(d_x, d_d, alpha, n) ;// x = x + alpha * d
    cplvAdd_MulC_I(d_r, d_q, -alpha, n);// r = r - alpha * q
    delta_old = delta_new;
    delta_new = cplvSum2(d_r, n);        // delta_new = r^Tr
    beta = delta_new / delta_old;        // beta = delta_new / delta_old
    cplvMulCAdd_I(d_d, beta, d_r, n);  // d = beta * d + r
  }
}
```

Figure 7: CG Solver template

3.8 Multi-GPU processing model

Computing systems in practice have to deal with a large amounts of data which cannot be processed directly and efficiently by a single processing system. GPUs are no exception to this limitation. Hence, the development of a parallel multi-GPU framework is necessary, especially for exploiting the total power of multi-GPU systems (multi-GPU desktops) or GPU cluster systems.

In the following, we address the two main bottlenecks of multi-GPUs and cluster implementations: the limited CPU-GPU bandwidth, which is about 20 times slower than the local GPU memory bandwidth, and the limited network bandwidth between compute nodes which is an order of magnitude slower than CPU-GPU bandwidth. Our computational model aims at minimizing the amount of data transfer over the slow media, exploiting existing APIs such as MPI, and moving most of the computation from the CPUs to the massively parallel GPUs.

3.8.1 Multi-GPU model

We first proposed a multiple-input multi-GPU model [7]. The key idea was to maximize the total volume of inputs that the system can handle. In other words, by maximizing the number of inputs per node we increase the *arithmetic intensity* of each processing node.

We divide the inputs between GPU nodes and assign a GPU memory buffer at each node to serve both as an accumulation buffer and an average input buffer. As an output buffer, it is used to sum up the local deformed volumes while as an input, it contains the new average and is shared among volumes of the node.

At each iteration, we compute the local accumulation buffers, and send the result to the server to compute the global accumulation buffer. The new aggregate volume is read back to GPU nodes. Next, we perform a volume division on the GPUs to update the average.

Our aggregate model is more efficient than our previous average model [7], since it yields the same memory bandwidth but moves the computation from CPUs to GPUs, hence it is able to

Algorithm 4 Multi-scale atlas construction

Input : N volume inputs, multi-scale information
Output: Template atlas volume
for all $s = 1$ to N_s **do** {loop over the scales}
  Read $factor_s$, $nIters_s$, fluid registration parameters at the scale
  for $i = 1$ to N **do** {loop over the images}
    if $factor_s = 1$ **then** {first level scale - original image}
      $I_{is} \leftarrow I_i$
    else {down sample the image}
      Blur the image $I_i(blur) = GaussFilter(I_i)$
      Down sample $I_{is} = DownSample(I_i(blur))$
    end if
    if $s = 1$ **then** {first iteration}
      $h_{is} \leftarrow Id$, $v_{is} \leftarrow 0$
      Copy the sample image $I^0_{is} = I_{is}$
    else
      Up sample deformation field $h_{is}(x) = UpSample(h_{is}(x))$
      Up sample velocity field $v_{is}(x) = UpSample(v_{is}(x))$ {if needed}
      Update deformed image $I^0_{is} = I_{is}(h_{is}(x)))$
    end if
  end for
  Apply the atlas construction procedure at this scale
end for

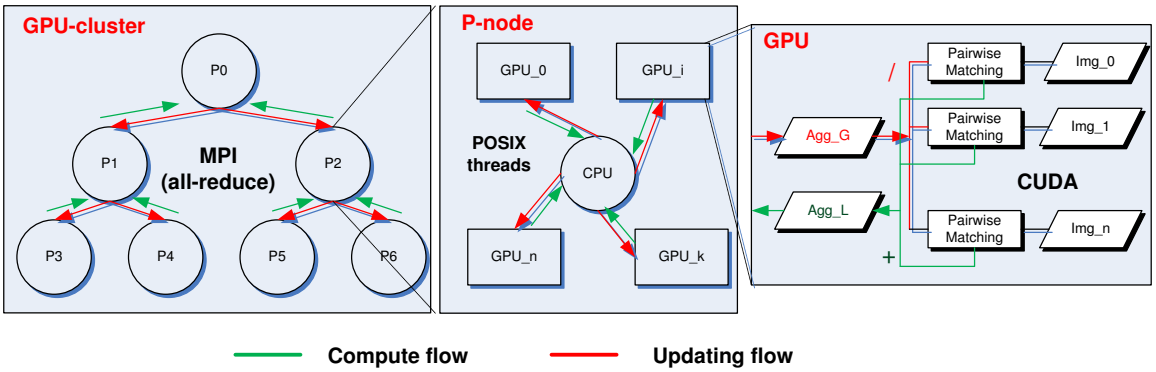

Figure 8: Multi-GPUs framework on the GPU cluster

exploit the computational power of the GPU. This strategy minimizes both the overall cost per volume element as well as the data transfer over the low bandwidth channel.

3.8.2 GPU cluster model

We generalize the multiple-input multi-GPU model to a higher level to build a computationally efficient framework on GPU clusters. As displayed in Figure 8, we maintain two buffers on a CPU-multi-GPU processing node: an output accumulation buffer and an input aggregate buffer which is shared among its GPUs' members. These two CPU memory buffers are used as the interface memory to other processing nodes through the MPIs. As we used the aggregated model instead of the average, we can directly exploit the MPI all-reduce function to efficiently compute and update the accumulated volume to all processing nodes. Next, we address the load distribution problem of GPU cluster implementation.

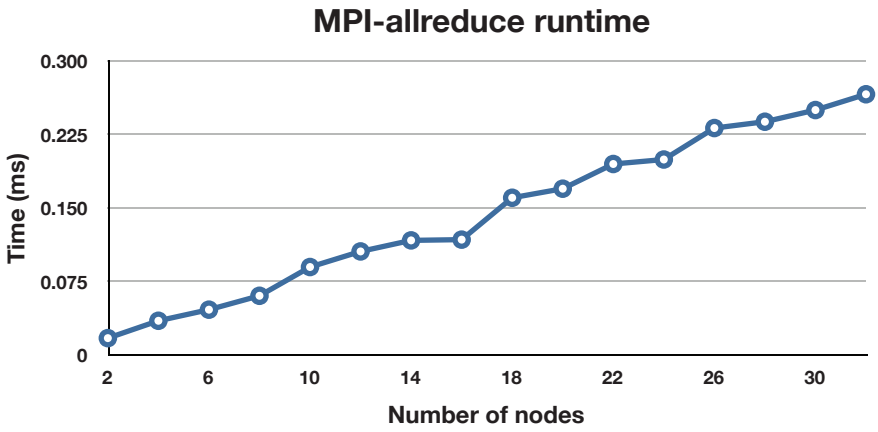

Figure 9: MPI-All reduce runtime on an infiniband network with OpenMPI 1.3 shows a linear dependency on the number of nodes

3.8.3 Load-balancing

We consider load balancing on a system with homogeneous GPUs with N_i, N_g, N_p being the number of inputs, GPUs, and CPUs. Our test system is a Tesla S1070 cluster, each node has dual-GPUs, thereby implying that $N_g = 2 * N_p$.

On the cluster, the total run-time per iteration is computed by $T = T_{GPU} + T_{CPU} + T_{Network}$. As the number of GPUs per node is fixed, T_{CPU}- the amount of time to compute the aggregate among GPUs of the same node - is fixed. Consequently, we must reduce T_{GPU} and/or $T_{Network}$ to improve the run-time.

First, we assume that N_p is fixed, then $T_{Network}$-the amount of time to accumulate and distribute result between CPUs-is defined. T_{GPU} depends on the maximum number of inputs per GPU, which is at least $N_{ig} = \lceil \frac{N_i}{N_g} \rceil$. This number is optimal if inputs are distributed evenly between GPUs, not CPUs as CPUs may have a variable number of GPUs attached. So our first strategy is distributing inputs evenly among GPU nodes. With this strategy, there is at most one unbalanced GPU, and the GPU run-time with synchronization is optimal.

Second, it is highly likely that the MPI all-reduce function performs the binary tree down-sweep to accumulate the volume and binary-tree up-sweep to distribute the sum to all nodes, as shown in Figure 8. This yields a minimal amount of data transferred over the network, that is $2 * N_p$. It is suggested that the amount of data transfer over the network increases linearly with the number of CPU nodes and therefore fewer CPUs implies smaller delay. This hypothesis is confirmed in our cluster in the experiment(see Figure 9)

To reduce the number of CPU nodes, we increase the GPU workload. Note that from the first strategy we want to increase all GPUs with the same number of volumes so that the computation is balanced. Let us increase this number by one, the total run-time then is

$$T = T_{GPU} * \frac{N_{ig} + 1}{N_{ig}} + T_{CPU} + T_{Network} * \frac{N_p - N_{ps}}{N_p}$$

Where N_{ps} is the number of GPUs reduced by increasing the workload. This equation gives us an approximation of running time as the number of volumes per GPU changes. Hence, we can vary the capability on the GPU node to achieve a better configuration. Note that in the dual-GPUs system, if the number of volume per GPU is less than N_g, when we increase the number of volumes per GPU by one, we can decrease the number of CPUs at least by 2.

Our load balancing strategy is as follows: First, the users chose the number of nodes. Based on this the system computes the number of inputs per GPU. The components' run-time is then determined with one-iteration dry run on the zero-initialized volumes which require no data from the host. Next, the optimizer varies the number of volumes per GPUs, recomputes the number of CPU nodes, and computes the total run-time. This heuristic yields an optimal configuration to handle the problem.

3.9 Performance tuning

To further improve the performance, we now present a volume-clipping optimization and the scratch memory model. Those techniques are specially applied for multi-image processing problems.

3.9.1 Volume clipping optimization

Volume clipping is the final step of the pre-processing, which includes

- Rigid alignment and affine registration
- Intensity calibration and normalization
- Volume clipping

The rigid alignment and affine registration guarantee all inputs to be in the same space while the pre-processing distances between them are minimal. This strategy significantly speeds up the convergence of the image registration process. The intensity calibration ensures that the intensity range of the inputs are matched and are normalized for visualization purposes. While these two pre-processing steps are generally applied in a regular image registration framework, the volume clipping is a special optimization scheme applied for the brain image to reduce processing time.

Point-wise computations on zero-data usually return zero, we call this data redundant. This redundancy happens near the boundary of the volume. The volume clipping strategy first computes the non-zero data bounding boxes, and then tightly clips all the volumes to the common bounding box with guarded boundary conditions. In practice, the volume of clipping inputs can be significantly smaller than a typical input volume, for example the 256^3 brain images in our experiment have a common volume of size $160 \times 192 \times 160$, a volume ratio of three. As the runtime of a function is proportional to the volume of the inputs, we experienced three to four times speed up just by applying this volume clipping strategy. Note that this optimization is more effective at PDE SOR solvers than FFT-based solvers as the latter require a power of two volume size to be computationally efficient.

3.9.2 Scratch memory model

It is always a challenge to implement 3D processing frameworks on GPUs as the parallel processing scheme often requires mores memory than it would on CPUs. To deal with this memory problem, we proposed a scratch memory model, a shared-temporary memory space, coupled with different optimization techniques including:

- Zero-copy operation based on pointer swapping to reduce the redundant memory copy from scratch memory (Figure 10a)
- A circular buffer technique to reduce memory copy redundancy and also memory storage requirements for computation in a loop (Figure 10b)

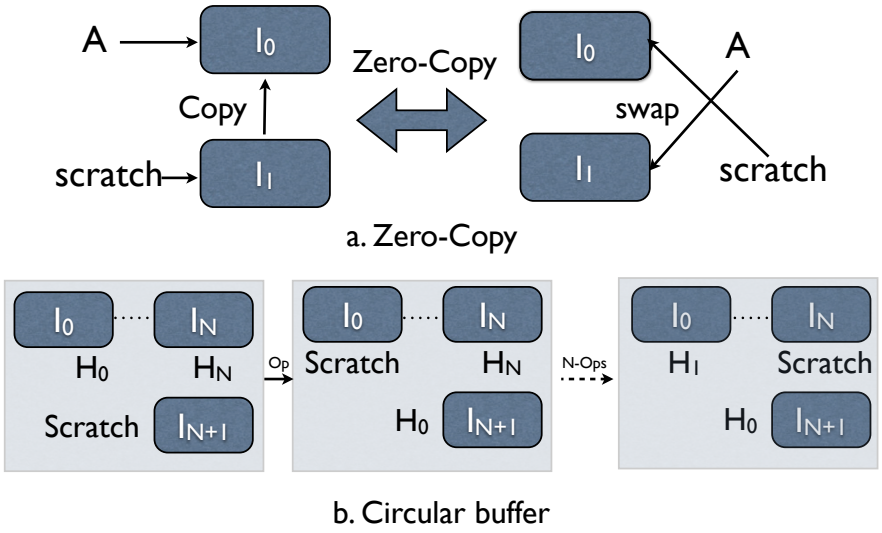

Figure 10: Optimization strategies with scratch memory model

The use of the scratch memory model helps us to significantly reduce the memory requirement. In particular, in the case of the Greedy Iterative Atlas construction, we only need a single image buffer and two 3D-vector buffer for an arbitrary number of inputs on a single GPU device. Consequently, we are able to process 20 brain volumes with 4GB global memory, or 40 brain volumes on a single dual-GPU node.

4 Final evaluation and validation of results, total benefits, limitations

The system we used in our experiment is a 64-node Tesla S1070 cluster, each containing two GPUs. Communication from the host to GPU is via the external x16 PCIe bus, and inter-node communication is implemented through a 20 GBits 4x DDR infiniband inter connect. The program was compiled with CUDA NVCC 2.3. For multi-resolution, we perform a 2-scale computation with 25 iterations in the coarse level, 50 iterations in the finer level, with parameters set $\alpha = 0.01$, $\gamma = 0.001$, and *maximum step size = 1*. The three solvers used in the comparison are FFT solver, the block-SOR, and Conjugate Gradient(CG). The run-time does not include the data loading time that depends on the hard-disk system.

4.1 Quality improvements

To evaluate the robustness and stability of the atlases, we use the random permutation test proposed by Lorenzen *et al.* [10]. The method is capable of estimating the minimum number of inputs required to construct a stable atlas by analyzing mean entropy and the variance of the average template. We generated 13 atlas cohorts, $C_{l,l=2\cdots14}$, each including 100 atlases constructed from l input images chosen randomly from the original data set. The 2D mid-axial slices of the atlases are shown in Figure 11. The normal average atlases are blurry, ghosting is evident around the lateral ventricles, and near the boundary of the brain. In this case, the Greedy Iterative Average template appears to be much sharper, preserving anatomical structures.

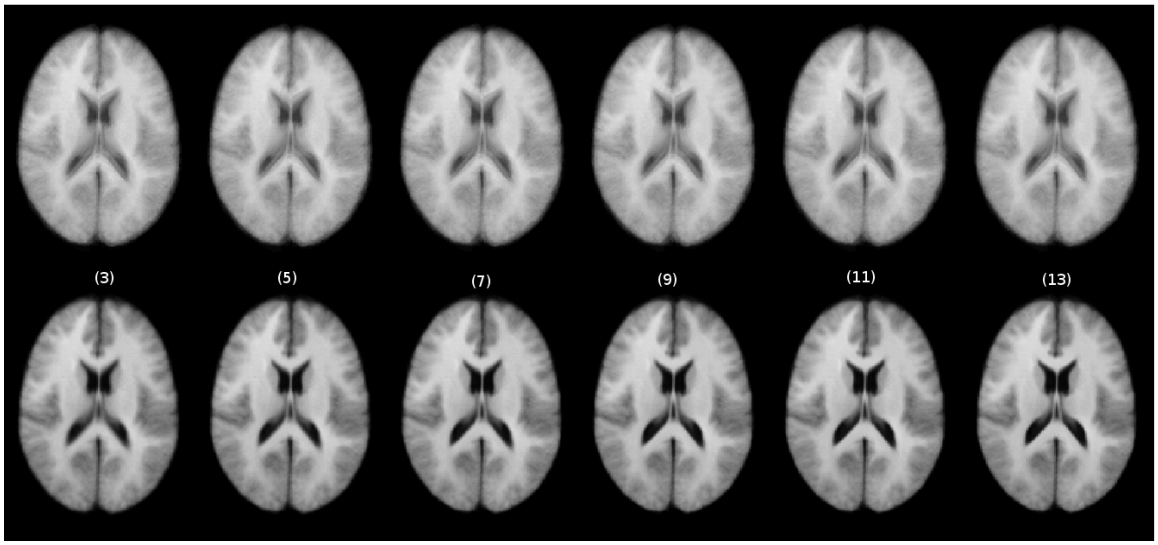

Figure 11: Atlas results with 3, 5, 7, 9, 11 and 13 inputs constructed by (a) arithmetically averaging rigidly aligned images (top row) and (b) Greedy Iterative Average template construction (bottom row)

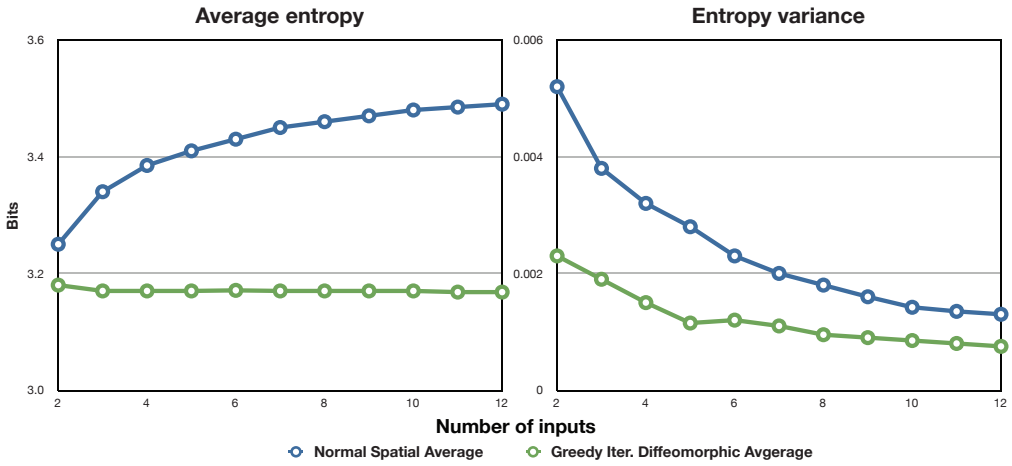

Figure 12: Mean entropy and variance of atlases constructed by arithmetically averaging and the Greedy Iterative Average template

The quality of the atlas construction is visibly better than the least MSE normal average. The entropy results shown in Figure 12 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 decreases due to much higher individual sharpness. This quantitatively asserts the visible quality improvement in Figure 11. The atlases become more stable with respect to the entropy as the standard deviation decreases with an 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.

4.2 Performance improvement

We compare the speedup of the multi-scale framework to the single scale version with a pairwise matching problem to produce comparable results. Experiments show that we generally need 25 iterations in the second level and 50 iterations in the first level to produce similar results

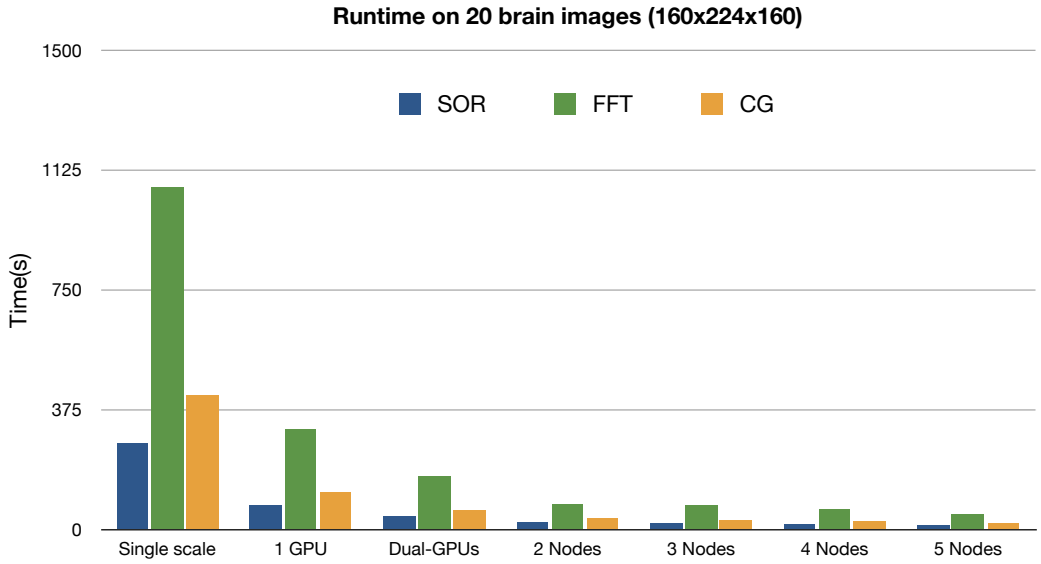

Figure 13: Run-time to compute the average atlas of the 20 T1 brain images ($144 \times 192 \times 160$) with multi-scale and/or multi-GPUs, cluster implementation in reference to one scale version

whereas we need 200 iterations with a single scale implementation. The speed up factor is about 3.5 and comes primarily from lowering the number of iterations in the finest level.

We quantify the compute capability and scalability of our system in two cases. First by applying the scratch memory technique, we are able to handle 20 T1 image of size $160 \times 224 \times 160$ on a single GPU device. We measure the performance with one GPU device (multi-scale), one single node with dual-GPUs (multi-scale multi-GPUs), two, four and five GPU nodes (multi-scale cluster) in reference to a single scale version on the 20-brain input set. As shown in Figure 13 the multi-scale version is about 3.5 times faster than single scale version, while our multi-GPU version is twice as fast as a single device. The cluster version shows a linear performance improvement to the number of nodes.

Second, we experiment with the full data set of 315 volumes of T1 input size $144 \times 192 \times 160$. For the first time we handle the whole data set on 8 nodes of the GPU cluster. We measure the performance with 8, 12, 16, 20, 24, 28 and 32 nodes. On the 32-core Intel Xeon server X7350, 2.93 Ghz with 196 GB shared memory, which is able to load the whole data set in-core, the CPU-optimized greedy implementation took 2 minutes for a single iteration. As shown on Figure 14, it only takes the SOR solver about 70 seconds to compute the average on 8 nodes of the GPU cluster and only 20 seconds on 32 nodes which is two order of magnitude faster than the 32-core CPU server.

5 Future directions

In this chapter we have presented our implementation of the Unbiased Greedy Iterative Atlas construction on multi-GPUs; however, this is only a showcase to illustrate the computing power and efficiency of our processing framework. As we mention in the introduction, the atlas construction problem is a basic foundation for a class of diffeomorphic registration problems to study the intra-population variability and inter-population differences. The ability to produce the result in real-time give us the ability to understand the research influence of this powerful technique. Also the framework allows us to implement more sophisticated registration problem such as LDDMM, Metamorphosis, or Image Current to name just a few. While each technique has different trade-off between quality of results and the computation involved, our framework

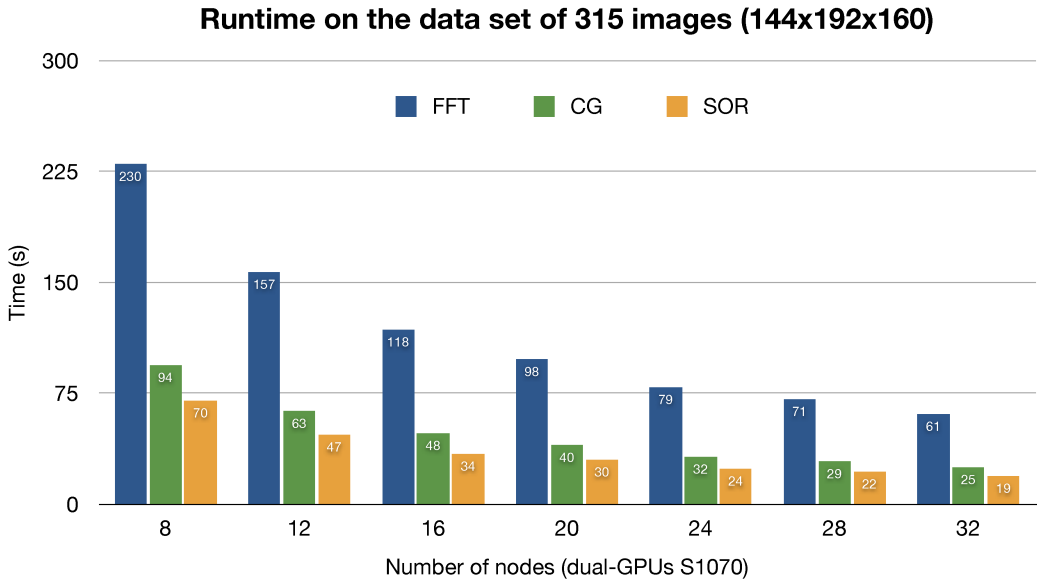

Figure 14: Multi-scale runtime to compute the average atlas of the 315 T1 brain images ($144 \times 192 \times 160$) with different PDE solver

is capable of quantifying those trade-off to suggest a good solution for the practical problem suitable with inputs and the accessible computational power.

Though the system has the capability to handle large amounts of data, it requires a single matching pair to be completely solvable on single GPU node, however, such compute power is not always available. So we are considering extending the processing power of single GPU system using an out-of-core technique. This requires a major redesign of our system; however, it is a required feature of our processing system to handle the ever growing amounts of data in the futures.

Note that our code is a part of the CompOnc project which is free for research purposes and is available to download at `http://www.sci.utah.edu/devbuilds/CompOnc/`

6 Acknowledgements

This research has been funded by the National Science Foundation grant CNS-0751152 and National Institute of Health Grants R01EB007688 and P41-RR023953. L. Ha was partially supported by the Vietnam Education Foundation fellowship. Specially thank to Sam Preston, Marcel Prastawa and Thomas Fogal for their time and feedback on the work.

References

[1] Muthu Manikandan Baskaran and Rajesh Bordawekar. Optimizing sparse matrix-vector multiplication on gpus. Technical report, IBM Technical Report, 2008.

[2] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, New York, NY, USA, 2009. ACM.

[3] Morten Bro-Nielsen and Claus Gramkow. Fast fluid registration of medical images. In *VBC '96: Proceedings of the 4th International Conference on Visualization in Biomedical Computing*, pages 267–276, London, UK, 1996. Springer-Verlag.

[4] Gary E. Christensen, Michael I. Miller, Michael W. Vannier, and Ulf Grenander. Individualizing neuroanatomical atlases using a massively parallel computer. In *Computer*, volume 29, pages 32–38. IEEE Computer Society, 1996.

[5] G.E. Christensen, R.D. Rabbitt, and M.I. Miller. Deformable templates using large deformation kinematics. *Image Processing, IEEE Transactions on*, 5(10):1435 –1447, oct 1996.

[6] B.C. Davis, P.T. Fletcher, E. Bullitt, and S. Joshi. Population shape regression from random design data. *Computer Vision, 2007. ICCV 2007. IEEE 11th International Conference on*, pages 1–7, Oct. 2007.

[7] Linh K Ha, Jens Krüger, P. Thomas Fletcher, Sarang Joshi, and Cláudio T. Silva. Fast parallel unbiased diffeomorphic atlas construction on multi-graphics processing units. In *EUROGRAPHICS Symposium on Parallel Graphics and Visualization 2009*, 2009.

[8] Jared Hoberock and Nathan Bell. Thrust: A parallel template library, 2009. Version 1.3.

[9] S. Joshi, B. Davis, M Jomier, and G. Gerig. Unbiased diffeomorphic atlas construction for computational anatomy. *Neuroimage*, 23 Suppl. 1:S151–S160, 2004.

[10] P. Lorenzen, B. Davis, and S. Joshi. Unbiased atlas formation via large deformations metric mapping. In J.S. Duncan and G. Gerig, editors, *Med Image Comput Comput Assist Interv Int Conf Med Image Comput Comput Assist Interv (MICCAI)*, volume 8 (Pt. 2), pages 411–418, 2005.

[11] Paulius Micikevicius. 3d finite difference computation on gpus using cuda. In *GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 79–84, New York, NY, USA, 2009. ACM.

[12] NVIDIA. Cuda technical training. Technical report, NVIDIA Corporation, 2009.

[13] David Young. *Iterative Solution of Large Linear Systems*. Academic Press, 1997.