CUDA Optimization Strategies for Compute- and Memory-Bound Neuroimaging Algorithms

Daren Lee[a], Ivo Dinov[a], Bin Dong[b], Boris Gutman[a], Igor Yanovsky[c], Arthur W. Toga[a,*],

[a]*Laboratory of Neuro Imaging, David Geffen School of Medicine, UCLA, 635 Charles Young Drive South Suite 225, Los Angeles, CA 90095, USA*
[b]*Department of Mathematics, University of California, San Diego, 9500 Gilman Drive, La Jolla, CA 92093, USA*
[c]*Jet Propulsion Laboratory, California Institute of Technology, 4800 Oak Grove Drive, Pasadena, CA 91109, USA*

Abstract

As neuroimaging algorithms and technology continue to grow faster than CPU performance in complexity and image resolution, data-parallel computing methods will be increasingly important. The high performance, data-parallel architecture of modern graphical processing units (GPUs) can reduce computational times by orders of magnitude or more. However, its massively threaded architecture introduces challenges when GPU resources are exceeded. This paper presents optimization strategies for compute- and memory-bound algorithms for the CUDA architecture. For compute-bound algorithms, the registers are reduced through variable reuse via shared memory and the data throughput is increased through heavier thread workloads and maximizing the thread configuration for a single thread block per multiprocessor. For memory-bound algorithms, fitting the data into the fast but limited GPU resources is achieved through reorganizing the data into self-contained structures and employing a multi-pass approach. Memory latencies are reduced by selecting memory resources whose cache performance are optimized for the algorithm's access patterns. We demonstrate the strategies on two computationally expensive algorithms and achieve optimized GPU implementations that perform up to 6× faster than unoptimized ones. Compared to CPU implementations, we achieve peak GPU speedups of 129× for the 3D unbiased nonlinear image registration technique and 93× for the non-local means surface denoising algorithm.

Keywords:
Graphics Processing Unit (GPU), Performance Optimization, Compute-bound, Memory-bound, CUDA, Fermi, Neuroimaging

1. Introduction

Neuroimaging techniques help neuroscientists map the brain to understand the relationships among the structure, function, and connectivity of the human brain using modalities such as computed tomography (CT), magnetic resonance (MR), and positron emission tomography (PET) imaging. These methods help researchers not only learn how the brain functions but also how neurological disorders, such as multiple scelerosis, Parkinson's disease, and Alzheimer's disease, originate and progress. Given the high variability of human brains, computer methods are an invaluable tool to manipulate, analyze, compare, and extract the data in brain images [1]. Advances in neuroimaging continue to produce more complex algorithms as well as higher resolution and larger brain image data sets. As these advances continue to grow faster than CPU performance, leveraging data-parallel hardware and techniques will be increasingly important. Moreover, these neuroimaging algorithms are often applied in large population studies so reducing the computational time from weeks to hours is highly desirable.

*Corresponding Author. Tel.:+1 310 206 2101
Email addresses: daren.lee@loni.ucla.edu (Daren Lee), ivo.dinov@loni.ucla.edu (Ivo Dinov), b1dong@math.ucsd.edu (Bin Dong), boris.gutman@loni.ucla.edu (Boris Gutman), igor.yanovsky@jpl.nasa.gov (Igor Yanovsky), toga@loni.ucla.edu (Arthur W. Toga)

The graphics processing unit (GPU) has evolved from a video processor dedicated to graphics rendering to a programmable, many-core architecture, with cores typically numbering from 128 and above. The high number of cores allows thousands of threads to be executed concurrently, yielding computational power previously only available in large super-computers. Through its attractive cost-to-performance ratio and maturing hardware and software tools, the GPU is emerging as a viable high performance computing platform that can compete with tradition high performance platforms such as supercomputers, clusters, and distributed computing. Researchers have leveraged the massively-threaded, data-parallel architecture of the GPU to reduce the computational time of expensive algorithms typically by an order of magnitude or more [2]. Combined with programming languages that no longer rely on a graphics API, the GPU is being used to accelerate computationally intensive medical and biological applications, such as real-time ultrasound simulation [3], real-time radiation therapy simulation [4], computational chemistry [5], and accelerating gene sequencing alignment [6].

Creating an optimal GPU implementation not only requires redesigning serial algorithms into parallel ones but, more importantly, requires careful balancing of the GPU resources of registers, shared memory, and threads, and understanding the bottlenecks and tradeoffs caused by memory latency and code execution [7]. The challenge is even greater when an implementation exceeds the GPU resources. This paper addresses optimization techniques for algorithms that exceed the GPU resources in either computation or memory resources for the nVidia CUDA architecture. For compute-bound algorithms, the challenge is to increase the data throughput by maximizing the thread count while maintaining the required amount of shared memory and registers. For memory-bound algorithms, the challenge is to reduce the memory latency time by leveraging the fast memory resources and caching features of the GPU.

The techniques to meet these challenges are illustrated on two fundamental pre-processing algorithms – 3D image registration and 3D surface denoising. Registration algorithms automatically spatially align 2D or 3D images. For image-based analysis, such as brain atlas construction [8, 9], statistical brain modeling [10], or brain mapping, registration is a necessary pre-processing step to align the brain images of the study into the same space so meaningful comparisons and calculations can be performed. For shape-based methods, such as morphology analysis [11] or 3D surface reconstruction [12], the shape surfaces are automatically extracted by segmentation algorithms. These computer methods typically produce noisy surfaces which first must be smoothed before any comparisons and computations can be carried out on them.

2. Background

GPU implementations are suitable for neuroimaging algorithms for several reasons. First, neuroimaging algorithms are often intrinsically parallel. For example, the same computations are typically performed on each voxel or vertex. Transitioning to a data-parallel GPU implementation, then, is often natural. Second, neuroimaging algorithms typically analyze a neighborhood of data for each data element. This data locality can be leveraged by storing the data in shared memory for its speed and reuse. Third, neuroimaging algorithms are often intensively iterative which helps yield higher performance gains. For example, since transferring data from CPU to GPU memory and initializing the GPU have major latencies, some GPU implementations do not achieve much total performance gain. However, for iterative algorithms, if all the iterative work can be implemented on the GPU, the effects of these startup latencies become less dominant and the total performance gains are substantially improved.

The application of GPUs to leverage these beneficial characteristics of neuroimaging algorithms have been demonstrated previously. For example, GPU-based 2D and 3D non-rigid registration methods [13, 14, 15, 16] have been presented but most of these are pre-CUDA and are constrained by the graphics rendering pipeline. To work around the graphics pipeline, techniques such as using multiple 2D textures to emulate rendering into a 3D texture, using multi-pass rendering to emulate iterations in a loop, and using floating point textures and blending to emulate histograms were used. The typical performance speedups were 2-5 times faster than CPU implementations. In [17], Fan *et al.* leverage the new hardware features of vertex buffer objects and floating point buffer blending to achieve histogram calculation in a single pass. Their optimized, graphics-pipeline-based 3D registration algorithm performs 44 times faster than the CPU implementation.

More recently, work has been presented that applies CUDA to medical imaging applications. CUDA allows developers to program the massively threaded GPU cores directly without going through a graphics API. This allows more optimal implementations of the algorithms. For example, Shams *et al.* [18, 19] implemented efficient parallelizations for mutual information computation that are 25-50 times faster than a CPU version. Muyan-Ozcelik *et al.* [20] implemented a CUDA version of the Demons algorithm, a non-rigid 3D registration method based on optical flow, entirely on the GPU. Their Demons implementation shows the benefit of using CUDA for image registration as their GPU implementation is 55 times faster than the CPU version. A survey of other 2D and 3D registration methods using high-performance hardware can be found in [21].

Prior work describing GPU optimization techniques have also been presented. Ryoo *et al.* [2] describes general principles and challenges of implementing and optimizing algorithms on the GPU. Their main principles include having efficient code, utilizing many threads to hide latency, and using local memories to minimize global memory transfers. In [22], Ryoo *et al.* also propose an approach to reduce the guessing work in choosing GPU configurations by plotting different configurations where the best ones are those that lie on a Pareto-optimized curve. However, many of the more subtle resource interactions, such as bank conflicts, are not fully captured by their method. In both papers, the authors did not address techniques for compute- or memory-bound applications. Jang *et al.* [23] present similar optimizations based on inspecting disassembled machine code for AMD/ATI graphics hardware. Their strategies include examining ALU, fetch bandwidth, and thread usage for matrix multiplication and back-projection image reconstruction. While they address compute-bound issues, memory-bound applications are not addressed.

2.1. CUDA Architecture

CUDA is nVidia's general purpose hardware architecture which provides C language extensions that allow developers to program the GPU directly. A key design difference between the CPU and CUDA platforms is that the CPU has more transistors dedicated to optimizing memory bandwidth whereas the CUDA GPU has more transistors dedicated to computation. Hence, for the GPU, accessing memory is expensive since it lacks an extensive caching hierarchy but computation is cheap as it can perform thousands of operations simultaneously. This compute-oriented philosophy provides great computational power but introduces challenges for algorithms that exceed the limit of the GPU resources.

The basic compute unit of the single-instruction, multiple thread (SIMT) CUDA architecture is the streaming multi-processor (SM). Each SM contains a set of processor cores that share a fixed, limited pool of registers and on-chip memory. Each GPU program, or kernel, is executed on each SM in groups of threads called thread blocks. Each thread block is allocated a set of registers and shared memory from the global pool and can read and write to the global device memory. The registers can only be used by the thread they are allocated to whereas the shared memory can be accessed by any thread in the thread block. The amount of registers and shared memory allocated is dependent on the kernel implementation and is a major factor in determining how many threads can run concurrently.

The registers for each thread block (R_{block}) are allocated in multiples of the register allocation unit size ($R_{allocation}$) and based on the number of warps of the thread block that is the nearest multiple of the warp allocation unit for registers ($W_{allocation}$). The number of registers allocated for a particular thread block configuration is given by:

$$R_{block} = ceil(\ ceil(\ W_{block}, W_{allocation}\ ) * T_{warp} * R_{kernel},\ R_{allocation}\ ) \tag{1}$$

where W_{block} is the number of warps in the thread block, R_{kernel} is the number of registers for the kernel, T_{warp} is the number of threads per warp, and $ceil(x, y)$ is x rounded up to the nearest multiple of y. $R_{allocation}$, $W_{allocation}$, and T_{warp} are fixed values based on the hardware compute capability version, listed in [24]. W_{block} and R_{kernel} are tunable values determined by the thread block configuration and kernel program implementation. The total number of registers per SM must be less than the maximum hardware register size and is constrained by the minimum of the warp, register, or shared memory thread block per SM size [25]:

$$R_{block} * Blocks/SM \leq Max\ Registers/SM \tag{2}$$

$$Blocks/SM = min(\ Blocks_w, Blocks_r, Blocks_m\ ) \tag{3}$$

$$Blocks_w = \frac{Max\ Warps/SM}{W_{block}} \tag{4}$$

$$Blocks_r = \frac{Max\ Registers/SM}{R_{block}} \tag{5}$$

$$Blocks_m \quad = \quad \frac{Max\,SharedMem/SM}{M_{block}} \tag{6}$$

$$\tag{7}$$

Equation (1) coupled with the constraints in Equation (2) dictate the number of registers allocated per thread block for a given hardware compute capability, thread block configuration, and kernel register count. A similar equation for the shared memory allocated per thread block is given in [25].

In addition to shared memory, each thread can access data from global device memory, texture memory, or constant memory. Texture and constant memory have read-only access and are cached whereas global memory has both read and write access. Depending on the hardware generation, the global memory may be cached. Both the global and texture memory allow large allocation sizes whereas the constant memory is limited to 64KB. The constant memory is optimized for broadcasting values to multiple threads. Texture memory provides convenient indexing and filtering but if cached, the global memory has a higher bandwidth.

To date, the CUDA architecture has had three generations. The G80 was the original implementation of the unified graphics and parallel processing architecture. The GT200 generation extended the G80 by increasing the number of cores, registers, and threads per multiprocessor. The latest version, the Fermi generation, introduces major changes, including a configurable cache hierarchy for global memory, tripling the shared memory size, a unified address space, and a dual warp scheduler.

2.2. 3D Unbiased Image Registration

Image registration algorithms automatically or semi-automatically spatially align multiple sets of 2D or 3D images to allow for meaningful comparisons and analysis of deformations. Here we use an information-theoretic approach previously introduced in [26], where the authors proposed to quantify the magnitude of a deformation by means of the symmetric Kullback-Leibler (sKL) distance between the probability density functions associated with the deformation and the identity mapping. This distance, when rewritten using skew-symmetry properties, is viewed as a cost function and is combined with the viscous fluid flow model [27] for registration. The resulting unbiased model generates topology preserving and inverse-consistent maps [28]. In this context, *unbiased* means that the method strives to obtain a zero-mean and symmetric log-Jacobian distribution under the null hypothesis, when a pair of images is matched. This distribution is beneficial when recovering change in regions of homogeneous intensity, and in ensuring symmetrical results when the order of two images being registered is switched. In this paper, we demonstrate optimization techniques for the most computationally intensive part of the algorithm, calculating the force field in Equation (11).

2.2.1. Algorithm Review

Let T be the template image, and R be the reference image. We seek to find the transformation h that maps T into correspondence with R. At every point x on the computational grid, we denote the Jacobian matrix of a deformation $h(x)$ to be $Dh(x)$, with Jacobian denoted by $|Dh(x)|$. The displacement field $u = (u_1, u_2, u_3)$ from the position $x = (x_1, x_2, x_3)$ in the deformed image $T(h(x))$ back to $T(x)$ is defined in terms of the deformation $h(x)$ by the expression $h(x) = x - u(x)$. The Unbiased method solves for the deformation h (or, equivalently, for the displacement u) minimizing the energy functional E, consisting of the regularizing sKL distance and the L_2 norm between the deformed template image and the reference image. The general minimization problem can be written as

$$\min_u \Big\{ E(u) = L_2(u) + \lambda\, sKL(u) \Big\}. \tag{8}$$

Here, $\lambda > 0$ is a weighting parameter. Given images T and R, the L_2 norm between the deformed template $T(h(x))$ and the reference $R(x)$ is given as

$$L_2(u) = \frac{1}{2} \int \Big(T\big(x - u(x)\big) - R(x) \Big)^2 dx. \tag{9}$$

As derived in [26] using information theory, the unbiased regularization term is given as

$$sKL(u) = \int \Big(|D(x - u(x))| - 1 \Big) \log \big|D\big(x - u(x)\big)\big|\, dx. \tag{10}$$

The numerical scheme for simulating the fluid model consists of four steps per iteration: calculating the force field f, evaluating the velocity v, updating the displacement field u, and interpolating the image $T(x-u(x))$.

The force field, which drives T into registration with R, is defined as the negative of the gradient of $E(u)$:

$$f(x,u(x)) \quad = \quad \Big[T(x-u(x))-R(x)\Big]\nabla T(x-u(x))-\lambda\begin{bmatrix} d_1 \\ d_2 \\ d_3 \end{bmatrix}, \tag{11}$$

where

$$\begin{aligned} d_1 = {} & \frac{\partial}{\partial x_1}\Big(\big((1-\partial_2u_2)(1-\partial_3u_3)-\partial_3u_2\,\partial_2u_3\big)L'\Big) \\ & +\frac{\partial}{\partial x_2}\Big(\big(\partial_3u_2\,\partial_1u_3+\partial_1u_2(1-\partial_3u_3)\big)L'\Big) \\ & +\frac{\partial}{\partial x_3}\Big(\big(\partial_1u_2\,\partial_2u_3+(1-\partial_2u_2)\partial_1u_3\big)L'\Big), \end{aligned}$$

$$\begin{aligned} d_2 = {} & \frac{\partial}{\partial x_1}\Big(\big(\partial_2u_3\,\partial_3u_1+\partial_2u_1(1-\partial_3u_3)\big)L'\Big) \\ & +\frac{\partial}{\partial x_2}\Big(\big((1-\partial_1u_1)(1-\partial_3u_3)-\partial_3u_1\,\partial_1u_3\big)L'\Big) \\ & +\frac{\partial}{\partial x_3}\Big(\big(\partial_2u_1\,\partial_1u_3+\partial_2u_3(1-\partial_1u_1)\big)L'\Big), \end{aligned}$$

$$\begin{aligned} d_3 = {} & \frac{\partial}{\partial x_1}\Big(\big(\partial_2u_1\,\partial_3u_2+\partial_3u_1(1-\partial_2u_2)\big)L'\Big) \\ & +\frac{\partial}{\partial x_2}\Big(\big(\partial_1u_2\,\partial_3u_1+\partial_3u_2(1-\partial_1u_1)\big)L'\Big) \\ & +\frac{\partial}{\partial x_3}\Big(\big((1-\partial_1u_1)(1-\partial_2u_2)-\partial_2u_1\,\partial_1u_2\big)L'\Big), \end{aligned}$$

Here, we denote $\partial_j u_i = \dfrac{\partial u_i}{\partial x_j}$ and $L' = 1+\log|D(x-u(x))| - \dfrac{1}{|D(x-u(x)|}$.

As in [26, 29] the instantaneous velocity v is obtained by convolving the force field f with a Gaussian kernel G_σ of variance σ^2:

$$v = G_\sigma * f(x,u(x)). \tag{12}$$

Given the velocity field v, the following partial differential equation can be solved to obtain the displacement field u:

$$\frac{\partial u}{\partial t} = v - v\cdot\nabla u, \tag{13}$$

where t is an artificial time.

2.3. Level Set Based Non-local Means Surface Denoising

Variational and partial differential equations (PDEs) based image denoising models have had great success in the past twenty years [30, 31, 32, 33, 34]. More recently, Buades, Coll and Morel [35, 36] proposed nonlocal means/filtering for image denoising, which produced excellent results. Later in [37, 38], Gilboa and Osher put the nonlocal means of [35, 36] into a variational framework (an earlier variational formulation was done in [39]), where they introduced some interesting concepts such as nonlocal gradient, divergence and Laplacian. They showed in [37] that excellent denoising results can be obtained by using some properly chosen weight function, which is related to the kernel of nonlocal means of [35, 36].

Some of the models used originally for image denoising have been extended to denoising surfaces [40, 41, 42, 43, 44, 45, 46]. In [47], Dong *et al.* extended the idea of nonlocal means to surface denoising where all surfaces are represented by level set functions. Their idea is to solve a non-local heat equation within a narrow band of the given level set function. The well known advantages of handling implicitly represented surfaces (e.g. surfaces represented by level set functions) over triangulated surfaces are numerical simplicity and flexibility of topological changes. Topological flexibility is important for surface denoising, because the surface noise usually includes both random variation in normal direction and topological errors. This advantage of topological flexibility was illustrated in one of the examples of [47].

Dong *et al.* use a semi-non-local strategy by performing the smoothing with a neighborhood of values near each narrow band element. In this paper, we extend the calculations to the entire narrow band to implement the full non-local means approach. We also demonstrate optimization techniques for the most expensive part of the smoothing algorithm, finding the weights for all the patches in the narrow band in Equation (15).

2.3.1. Algorithm Review

The given surface S is taken as the boundary of some domain Σ. The corresponding signed distance function ϕ satisfies: $\phi(x) < 0$, for $x \in \Sigma$; $\phi > 0$, for $x \notin \Sigma$; and $|\nabla\phi| = 1$ away from its singularities. Thus we have $S = \{x : \phi(x) = 0\}$.

Let ϕ be the signed distance function representing the noisy surface S. Perform the following steps.

1. Extract a narrow band of ϕ
$$\Sigma_\delta := \{x \in \mathbb{R}^3 : |\phi(x)| \leq \delta\} \tag{14}$$
with δ the width of the narrow band.
2. Compute weight $w(x, y)$ and similarity function $D(x, y)$:
$$\begin{aligned} w_{xy} &= e^{-|x-y|^2/c_1} e^{-D(x,y)/c_2} \\ D(x,y) &= ||\phi[x] - \phi[y]||_2^2, \; x, y \in \Sigma_\delta \end{aligned} \tag{15}$$
where $\phi[x]$ is a 3D patch of ϕ centered at x.
3. Iterate the following steps until the maximum allowable number of iteration is exceeded:
$$\phi_j^{k+1} = \phi_j^k + dt \sum_{l \in \Sigma_\delta} w_{jl}(\phi_l^k - \phi_j^k), \tag{16}$$
with w_{jl} calculated as given in step (2), and dt chosen to be
$$dt = 1/max\{\sum_{l \in \Sigma_\delta} w_{jl}\}$$

3. Methods

The computational performance of an algorithm can either be compute-bound, where there are a large number of computations per data element, or memory-bound, where there are many data elements per computation. For both approaches, determining the maximum allowable amount of resources allocated to a thread block is key for optimization. As shown in Equations (1) and (2), the register and shared memory resources are allocated in multiples of fixed sizes. Thus, the amount that is requested for a resource may not be the same as the actual amount allocated. Understanding how these resources are allocated are vital in determining over and underused resources, valid thread block configurations, and optimal thread block configurations.

For compute-bound kernels, the register count typically limits the number of concurrent threads per SM. Using the shared memory can help overcome the register limits. In addition to providing a cache for the slow global device memory, the shared memory can be used to cache intermediate values, preferably ones needed by other threads. Storing the intermediate values can reduce the register as well as instruction count since arithmetic operations can be replaced with a memory lookup. Additionally, since computation, not memory bandwidth, is the main bottleneck, optimizing the GPU resources for a single thread block configuration can lead to increased performance. Increasing the thread block count is generally more useful for hiding memory latencies.

For memory-bound kernels, overcoming memory latencies and fitting the working set of data into the fast but small GPU memory resources are the main bottlenecks. To reduce the memory latency, the GPU supports caching for several memory types. However, the type of memory chosen should be carefully considered based on the access patterns of the algorithm as not all the caches have equal performance.

To fit the data into the small resources such as shared or constant memory, reduce the overall footprint of the data by repartitioning or reorganizing the data into independent data structures. This can be accomplished by partitioning the complete dataset into smaller subparts that can be solved independently or by creating a new self-contained, data structure that holds the working set for a single data element. The self-contained data structure may duplicate values but if the working set for a data element is only a small subset of the total data, the footprint will be substantially reduced. Consequently, higher performance GPU memory resources, such as the shared or constant memory, can be leveraged for their speed or caching abilities. This approach oriented around the data element maps well into the data-parallel architecture of CUDA and achieves a degree of parallelism that is proportional to the total number of independent data structures that can fit into the fast GPU memory resources.

Another strategy for memory-bound algorithms is to use a multi-pass approach. Rather than computing the entire solution in a single kernel invocation, a subset of the data can be computed in each pass of a multi-pass approach. With multiple kernel invocations, the entire data set can be processed. However, the data in the shared memory is not persistent across each pass and bandwidth must be used to write the data to global memory. After each pass, the results in shared memory must be copied to global memory and then reread into the shared memory in the next pass. Also, the algorithm may need to be reformulated to use the multi-pass approach.

3.1. CUDA Implementations

The basic strategy for our CUDA implementations for the image registration and shape denoising are described below. The effect of the optimization techniques are presented in Section 4.

3.1.1. 3D Unbiased Image Registration

A general divide and conquer approach is used to partition the original 3D grid into sub-grids where the flow field for each sub-grid can be computed in parallel. Each sub-grid is assigned to a thread block and each thread in that block is assigned to a grid point. A drawback of the divide and conquer approach is that partitioning the data into independent sub-grids requires padding the target grid points with the necessary data needed for the partial derivatives and the Jacobians. The border grid points need to be double padded to compute the partial derivatives for the neighboring Jacobian for each grid point. Consequently, border grid points are duplicated across sub-grids, increasing the amount of data transferred from global device memory and reducing the amount of available GPU resources.

To ensure there are enough resources to compute the entire flow vector field, a 2D rather than a 3D thread block size is employed. Each thread in the block is assigned to a (x, y) grid point and loops over the z depth dimension. For each xy-plane of threads of size $w \times h$, not only does the data in the x and y dimension need to be double padded but also in the z direction. This results in a working set of five double padded planes per force field dimension for a total size of $15 \times [(w + 4) \times (h + 4)] \times 4$ *bytes*.

For compute-bound algorithms, the challenge is often balancing shared memory versus register usage. To evaluate the performance differences between the two approaches, a CUDA implementation which recomputes intermediate values is compared against a method which stores the intermediate values for reuse. In the recompute approach, the shared memory is only used to store the displacement vector field. None of the partial derivative calculations are stored as all intermediate results are recomputed by each individual thread. However, the thread count for this implementation becomes constrained by the register count. To increase the number of threads in each thread block, the number of registers for the kernel needs to be reduced. In the reuse approach, shared memory is used to relieve the pressure on the registers by storing intermediate results and leveraging the reuse of many of the partial derivatives. With the shared memory, the values for each displacement field, partial derivative in each dimension, and Jacobian at each grid point can be reused by six other grid points. Storing these intermediate values requires changing the overall structure of the algorithm as well as introduces a few other optimizations.

First, if the previous slice's intermediate values are stored in shared memory, only 4 xy planes (instead of the original 5) of the displacement vector field per dimension are required since only the last slice needs to be double

Resource	Recompute (registers)	Recompute (lmem)	Reuse
Shared memory	$15 \times WH = 8640$	$15 \times WH = 4364$	$32 \times WH = 49152$
Registers	63	35	24
Local memory	664	20160	2921
$w \times h$	8×12	8×10	28×12
Thread block	32×16	32×10	32×16
Blocks/MP	1	2	1

Table 1: GTX 480 resource allocation for the flow field kernel for the recompute and reuse approaches. For the recompute approach, a method emphasizing register usage is compared to one which emphasizes local memory. For the reuse approach, the shared memory usage is maximized. For the shared memory, $WH = [(w + 4) \times (h + 4)] \times 4\ bytes$ and memory counts are in bytes.

Variable	# Values
Narrowband ID + $n \times n \times n$ patch	126
m similarity distances	96
m narrow band IDs	96
max similarity value	1
count of number of values	1
Total	320

Table 2: NLM Smoothing memory requirements showing the number of floating point values for each variable. $n = 5$ is the patch size used in the similarity metric and $m = 96$ are the most similar patches.

padded to compute its values. However, the total size of the shared memory must be increased greatly to store the required intermediate values, as listed in Table 1. Second, the access to global memory is minimized by copying and shifting each slice of the displacement vector field in shared memory as the depth slices are marched through. Third, shared memory locations are reused after they are no longer used. For example, only the last slice of the displacement data needs to loaded from global device memory; all other slices already reside in the shared memory. To reuse shared memory, the partial derivatives $\partial u_x/\partial x, \partial u_y/\partial x, \partial u_z/\partial x$ for the current slice $k = 0$ are stored in the displacement field values u_x, u_y, u_z for the previous slice $k = -1$.

In addition to evaluating the performance difference between the recompute and reuse approaches, the effects of the local memory caching and the number of thread blocks per multiprocessor are tested. To study the local memory caching effects, the compiler is forced to use a fixed number of registers smaller than actually required and local memory is used in lieu of the registers. Reducing registers helps to increase the number of thread blocks. However, for compute-bound algorithms, increasing the number of blocks per multiprocessor may not have any effect if the bottleneck is computation and not memory bandwidth.

3.1.2. Level Set Based Non-local Means Surface Denoising

Our general approach to finding the weights for all the patches in Equation (15) is to use a brute force method to compare each surface patch with every other patch since it provides the highest fidelity. The computational intensity arises not from the computational complexity of the similarity metric, which is the l^2-norm of $n \times n \times n$ patches, but from the shear number of $N^2/2$ pair-wise comparisons that must be calculated, where N is the number of narrow band elements. In addition to the intensive computations, another bottleneck is the large memory requirement for storing all the similarity values. With N typically around 100,000 elements, this become impractical. Since patches which are not similar have a weight close to zero and do not contribute to the final result, the memory requirements are reduced by saving only the m patches with the highest similarity, resulting in $N * m$ stored values. Although the number of stored values is reduced, the pair-wise, shape similarities still need to be exhaustively computed. The main challenge of computing these values is fitting the narrow band and patch data for all pair-wise comparisons into the memory resources of the GPU.

In our CUDA implementation, a data-parallel approach is used where the similarity values for multiple narrow band elements are computed concurrently. To work with the GPU memory resources, the following changes are made from the CPU implementation. First, since only the patch data centered around the narrow band elements are of interest, the 3D data array for storing the entire SDF is discarded. Instead, the patch data, similarity distances, and IDs are stored together with each narrow band element, as shown in Table 2. Although storing the patch data with each narrow band element duplicates values, the overall footprint for the memory is greatly reduced. This new narrow band data structure also provides the basic unit for the data parallelism. Second, even with this data structure change, all the data for each pair-wise comparison can not fit into the shared memory. Rather than computing the entire solution in a single kernel invocation, a subset of the comparisons can be computed in each pass of a multi-pass approach. With multiple kernel invocations, the large memory requirements can be overcome and the entire narrow band can be processed with a higher degree of parallelism. A drawback of this multi-pass approach is that the shared memory is not persistent across each pass and bandwidth must be used to copy the results in shared memory to global memory and then reread into the shared memory in the next pass.

In addition to the high level changes for the GPU implementations, more subtle optimizations are also made. First, to reduce the memory requirements for each thread, rather than finding all similarities first and then sorting, each thread computes a single pair-wise similarity value first and then uses an insertion sort to determine if it belongs in the m closest patches. This reduces the memory requirements from $N + m$ to $m + 1$. Second, the bank conflicts in shared memory are minimized. If multiple threads access the same bank in shared memory, the memory accesses will be serialized and therefore will reduce the parallelism. To avoid bank conflicts, the stride between data structures must be odd. In our case, the narrow band data structure of 320 floating point values must be padded by 1 to avoid bank conflicts. However, due to the non-deterministic nature of the sorting, the bank conflicts are only eliminated when computing the similarity distance but not during the sorting step to find the closest m patches.

To evaluate the differences between the GPU memory resources, the subset of patches being compared during each pass are stored in three different memory resources – global, texture, and constant memory. In the Fermi architecture, all three memory types are cached. Since all the threads read the subset of patches in the same order, the CUDA implementation of the non-local surface denoising algorithm should benefit from the Fermi caching hierarchy since cache misses should be minimal. Both the global and texture memory allow large allocation sizes whereas the constant memory is limited to 64KB. The constant memory is optimized for broadcasting values to multiple threads and can store a maximum of 130 patch data structures, allowing 130 patches to be compared during each pass. Texture memory provides convenient indexing and filtering but the global memory has a higher bandwidth cache.

4. Results and Discussion

We evaluate our optimization strategies using the nVidia GeForce GTX 480 (Fermi) and 8800 GTX (G80) graphics cards with CUDA versions 3.0 and 2.2, respectively. The CPU experiments were performed on a dual Xeon 3.8 GHz processors with 3GB of memory running Windows XP SP2. For each of the applications, we present the impact of the optimization techniques over unoptimized implementations.

4.1. 3D Unbiased Image Registration

As shown in Figure 1, the reuse method outperforms the recompute approach in all configurations. For the optimal configuration of a single block per multiprocessor, the reuse method is 3× faster than the recompute one. In the reuse approach, the shared memory usage is increased to store intermediate results and to decrease the register count, thereby increasing the total number of threads that can run concurrently. The effect of applying these optimization techniques on the resources is shown in Table 1.

The results also show that for the force field kernel, increasing the number of thread blocks per multiprocessor increases the computational time. For the recompute method, increasing the thread blocks per multiprocessor from one to two increases the computational time by 30%. For the reuse method, increasing from one to three thread blocks per multiprocessor increases the computational time by 25%. As expected, the bottleneck of the flow field kernel is computation, so maximizing the thread block configuration for a single block per multiprocessor yields the best results. The single block configuration allows a greater number of threads per block and also has the side effect of greater reuse of the intermediate data.

The recompute method using the local memory cache performs the slowest, taking 6× longer than the optimized reuse method. This method actually performs slower than the optimized reuse method on the older generation of G80 hardware. Given the looping behavior of the flow field kernel, values are read and written to the same variables many

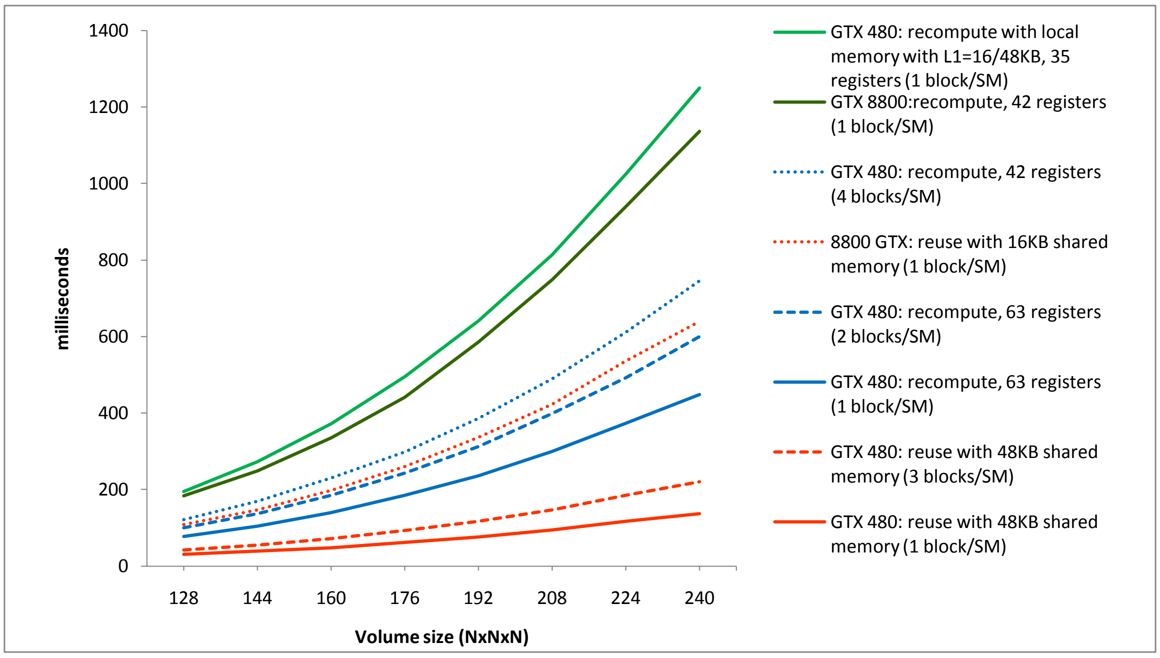


Figure 1: CUDA computational times of computing the force field with varying volume dimensions using the recompute and reuse methods with different thread block sizes.

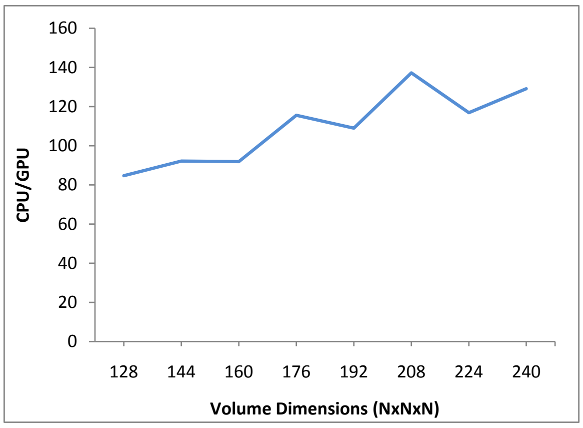
times. As evidenced by the results, the cache for the local memory for register spills is not effective when writes are needed.

The overall performance increase for the complete registration implementation, as shown in Figure 2a, is 85× to 129× faster than a single-threaded CPU implementation. On the lower end, for a $128 \times 128 \times 128$ volume, this reduces the computational time from 35 minutes on the CPU to just 24 seconds on the GPU. At the upper end, for a $240 \times 240 \times 240$ volume, the overall computational time is reduced from over 7 hours to 2.5 minutes. The per iteration gains follow a sawtooth trend where the performance increases to a peak, decreases, and then increases to another peak in a cyclical fashion. This behavior reflects unoptimized load balancing of the GPU SMs. For optimal parallelization, the number of blocks should be a multiple of the number of SMs so there will be no idle processors. In our CUDA implementation, fixed thread block sizes are used so the number of blocks may be less than ideal, depending on the volume dimensions. Ideally, the number of blocks should be allocated dynamically based on the input volume dimensions. However, the interplay of shared memory, register count, and global memory coalescing makes it difficult to choose a thread block size that will be optimal across all volume dimensions.

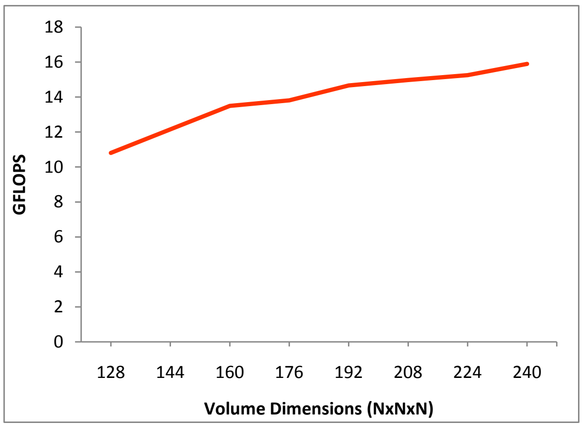
For the flow field kernel, each voxel requires 158 flops and requires 500 bytes of data in the double padded region. The maximum flop rate for the flow field kernel on the GTX 480, then, is given by $(158n\ flops)/(500n\ bytes) \times 177GB/s = 55\ GFLOPS$. As shown in Figure 2b, the optimized reuse kernel achieves 11 to 16 GFLOPS, increasing with the volume dimensions. With larger volume dimensions, more thread blocks are required, and hence more multiprocessors can be utilized to process more voxels concurrently. The results of applying the GPU implementation of the 3D unbiased registration algorithm to MRI patient data are shown in Figure 3. The GPU results align well with the CPU results as the maximum error is 0.5 voxel.

4.2. *Level Set Based Non-local Means Surface Denoising*

The computation times for finding the closest neighbors for each patch in the narrow band using different memory storage are shown in Figure 4. As evidenced by the data, each memory type and their caches do not perform equally. Despite its small size, storing and reading the subset of patches in constant memory produces the fastest computational times while using texture memory is substantially slower for the same shared memory and thread block configuration. The optimized broadcast feature of the constant memory overcomes the overhead of uploading the patch data during



(a)



(b)

Figure 2: Performance metrics of the flow field kernel with varying volume dimensions of (a) the speedup of the optimized GTX 480 reuse method versus the CPU one, and (b) the giga-flops (GFLOPS) for the GTX 480 implementation.

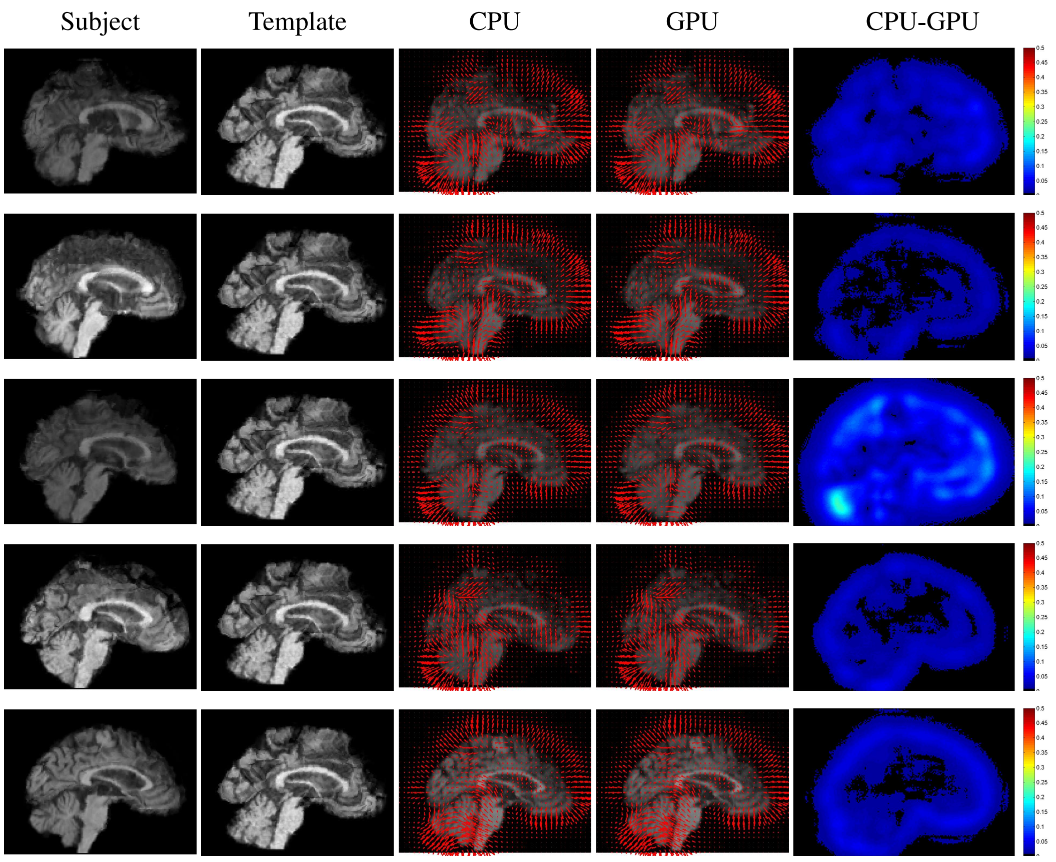


Figure 3: Results of applying the GPU implementation of the 3D unbiased image registration to 192 × 128 × 192 human MRI cases aligned to a template volume. The middle sagittal slice for each of the volumes are shown. The original image volumes are in the first column, the template volume is in the second column, and the aligned output images overlaid with the deformation vector field are shown in the third and fourth columns for the CPU and GPU. The last columns shows the difference between the vector magnitudes in units of voxels. The maximum value for the colorbar is 0.5 voxels.

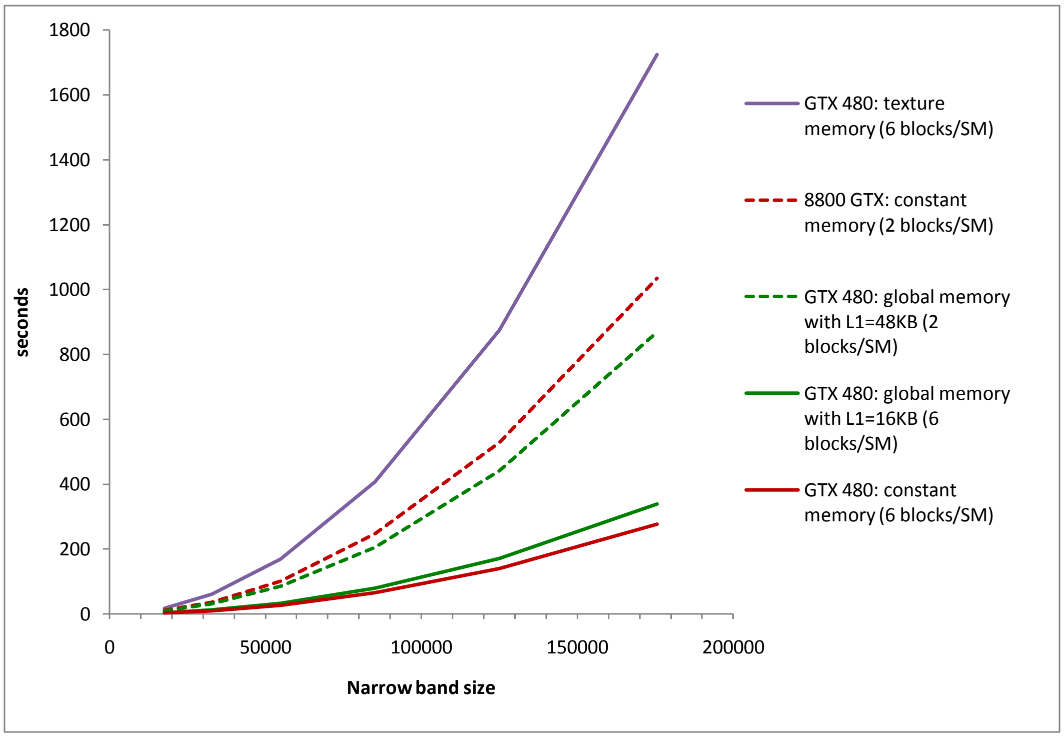


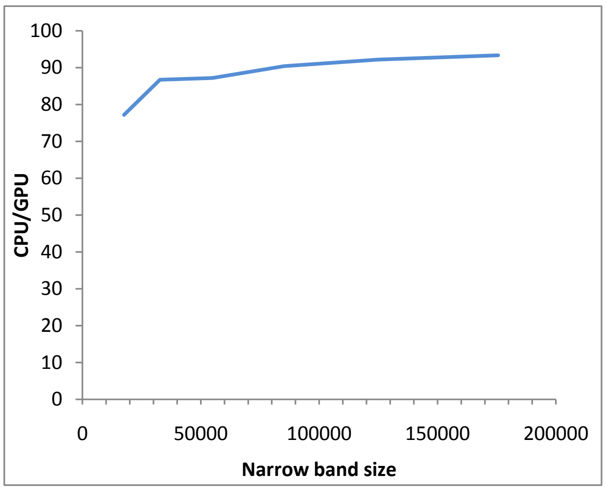
Figure 4: CUDA performance times to compute the patch weights in the non-local surface denoising algorithm with varying narrow band size and with different methods to store the subset of patches compared during each pass using global, texture, and constant memory.

each pass and outperforms both global and texture memory. The method using global memory with a L1 cache size of 16KB yields computational times 20% slower than using constant memory with the same 48KB shared memory size. Unlike the unbiased registration implementation that reads and writes the spilled registers from local memory, the non-local denoising implementation requires read-only access to the patch data stored in global memory. The improved performance of this method shows the beneficial impact of the L1 cache for read-only access with a high degree of locality. This method is limited by the latency incurred for cache misses and any overhead associated with managing the coherence of the cache.

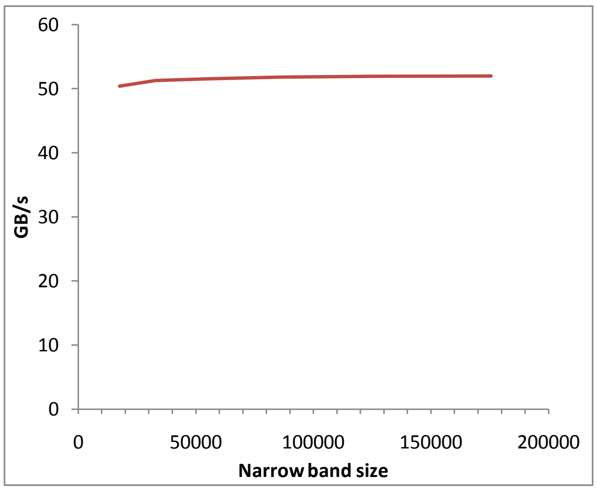
Increasing the L1 cache of the global memory to 48KB increases the computational time by 2.5 times. This method only performs 20% faster than the optimal implementation on the older 8800 GTX. In this case, the method becomes limited by the shared memory as only two blocks, as compared to six blocks, can be run on each SM with the same configuration. In this case, having a larger cache is not beneficial, as the number of patches that can computed concurrently becomes the bottleneck, not the global memory bandwidth.

Using texture memory produces computational times 6× slower than the method using constant memory, 5× times slower than cached global memory, and 66% slower than the optimized version on the G80 hardware. Although cached, the slower bandwidth of the texture memory transfer has a substantial impact on the overall performance. From examining the compiler output, fetching a single element from a 1D texture transfers four floating point values whereas in the other methods, only a single floating point value is transferred. Additionally, there may be added overhead associated with the address computation for the textures.

Unlike the unbiased registration, for the non-local surface denoising implementation, using fewer blocks to maximize the number of patch elements processed by each block decreases the performance. In this case, fewer blocks not only limit the amount of memory latencies that can be hidden by context switching but also increase the shared memory bank conflicts, causing more warps to be serialized. With more threads per block, the probability of shared memory bank conflicts is increased during the sorting of the similar neighbors since the access patterns are undeterministic.



(a)



(b)

Figure 5: Performance metrics of the non-local surface denoising algorithm with varying volume dimensions of (a) the overall speedup of the optimized GTX 480 implementation versus the dual-thread CPU one and (b) the gigabytes per seconds (GB/s) for the GTX 480 implementation.

The overall performance increase for the complete non-local means surface denoising implementation, shown in Figure 5a, is 77× to 93× faster than a multi-threaded CPU implementation per iteration. On the lower end, for a narrow band of 17,500 elements, this reduces the computational time from 4.3 minutes on the CPU to just 4 seconds on the GPU. At the upper end, for a narrow band of 175,000 elements, the overall computational time is reduced from 7.3 hours to 4.6 minutes. In contrast to the registration results, the surface denoising performance gain is fairly constant across narrow band sizes as the latencies due to the repetitive memory transfers dominate the optimizations of well load-balanced SMs. As shown in Figure 5b, for the optimized method using constant memory, the bandwidth is 50 GB/s for all narrow band sizes. This indicates that the multi-pass implementation is limited by the memory bandwidth required to load and unload the shared memory from the global memory as well as load the constant memory each pass.

As shown in Figure 6, the maximum error between the GPU and the CPU is less then 1/4 of a voxel. Examples of applying the GPU implementation of the non-local means surface denoising to white matter surfaces segmented from patient data is shown in Figure 7.

5. Conclusions

This work presents GPU optimization techniques for compute-bound and memory-bound implementations. For compute-bound algorithms, the number of registers must be balanced with the shared memory to maximize the throughput of calculations. Additionally, these resources should be maximized for a single block per multiprocessor when the bottleneck is computation. Increasing the number of thread blocks per multiprocessor in this case can lead to a decrease in performance. Techniques to optimize compute-bound algorithms include storing reusable, intermediate values in shared memory and assigning multiple data points to a thread to overcome the unused resources of idle threads.

For memory-bound algorithms, the challenge is to fit the working set of data into the fast GPU memory resources or leverage the low latency memory caches. This can be achieved by reorganizing the data into self-contained data structures and using a multi-pass approach to process a subset of these self-contained data structures during each pass. Additionally, the type of memory used should be carefully considered in context of the algorithm characteristics as not all the caches perform equally. For example, the cache hierarchy of the global device memory yields performance close to the more optimal methods without investing time to develop complicated implementation schemes. However, the improved performance is only achieved with read-only access of the global memory. Writing to the global memory with caching is substantially slower.

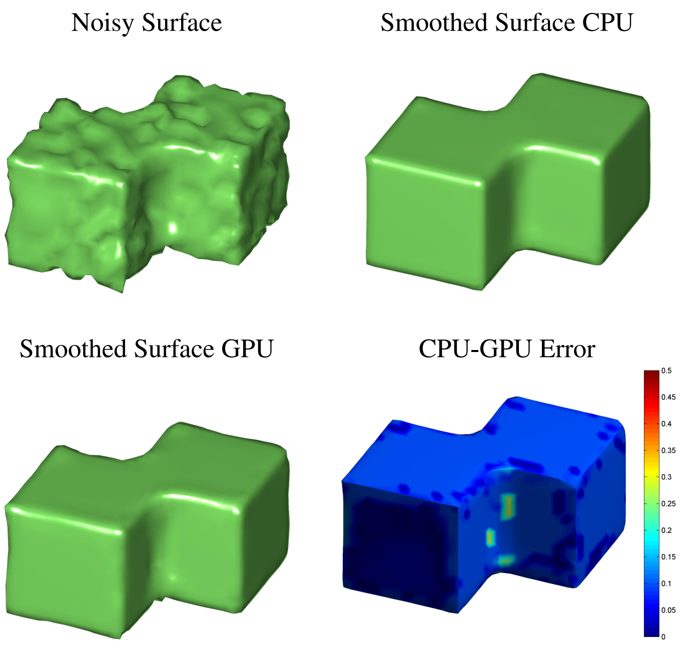


Figure 6: Results of applying the level set based non-local surface denoising technique to an artificially created noisy surface of two square blocks using the CPU and GPU implementations. The maximum error for the colorbar is 0.5 voxels.

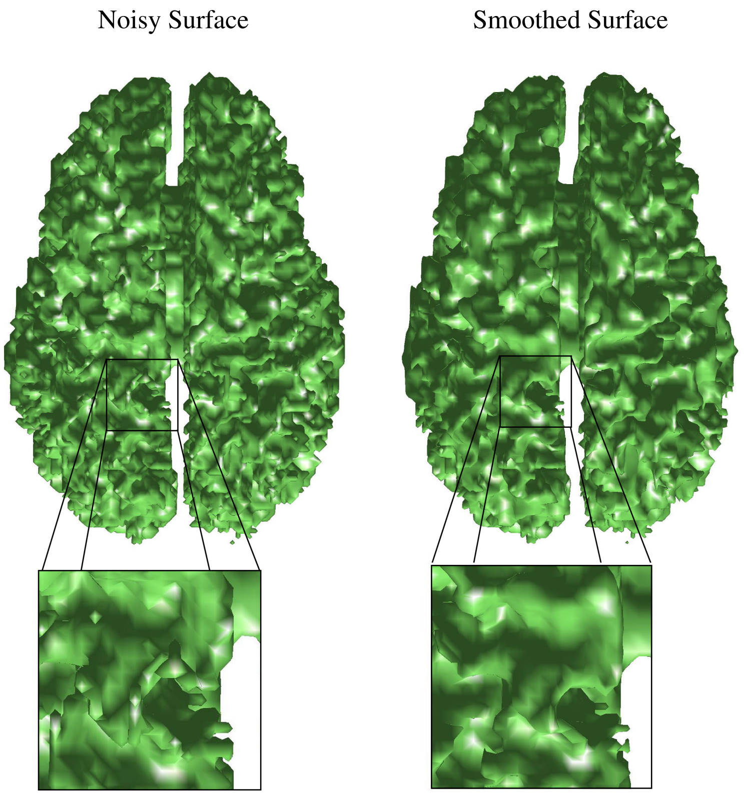


Figure 7: Results of applying the GPU implementation of the level set based non-local surface denoising to white matter surfaces segmented from human subjects. The original surfaces are on the left and the smoothed surfaces are on the right.

For our two applications, we demonstrated that the optimized GPU implementations perform 1.2× to 6× faster than unoptimized ones. Overall, a peak speedup of 129× over CPU implementations for the 3D unbiased image registration and 93× for the level set based non-local means surface shape denoising are achieved. This reduces the registration computational time from 7.2 hours to 2.5 minutes and the denoising time from 7.3 hours to 4.6 minutes. Future work includes extending the optimization techniques to multiple GPU configurations to handle ultra high-resolution images and surfaces.

Acknowledgements

This work is funded by the National Institutes of Health through the NIH Roadmap for Medical Research, Grant U54 RR021813. The research of Igor Yanovsky was carried out in part at the University of California, Los Angeles, and in part at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

[1] A. W. Toga, J. C. Mazziotta, Brain mapping: the methods, Academic Press, 2002.

[2] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, W. W. Hwu, Optimization principles and application performance evaluation of a multithreaded gpu using cuda, in: PPoPP Proceedings: 20-23 February, Salt Lake City, Utah USA, 2008, pp. 73–82.

[3] O. Kutter, R. Shams, N. Navab, Visualization and gpu-accelerated simulation of medical ultrasound from ct images, Computer Methods and Programs in BioMedicine 94 (2009) 250–266.

[4] R. Jacques, R. Taylor, J. Wong, T. McNutt, Towards real-time radiation therapy: Gpu accelerated superposition/convolution, Computer Methods and Programs in BioMedicine 94 (2009) 1–8.

[5] J. E. Stone, J. C. Phillips, P. L. Freddolino, D. J. Hardy, L. G. Trabuco, K. Schulten, Accelerating molecular modeling applications with graphics processors, Journal of Computational Chemistry 28 (2007) 2618–2640.

[6] S. Jung, Parallelized pairwise sequence alignment using cuda on multiple gpus, BMC Bioinformatics 10 (Suppl 7) (2009) A3. doi:10.1186/1471-2105-10-S7-A3.
URL `http://www.biomedcentral.com/1471-2105/10/S7/A3`

[7] D. Kirk, W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, Morgan Kaufmann, 2010.

[8] D. W. Shattuck, M. Mirzaa, V. Adisetiyoa, C. Hojatkashania, G. Salamon, K. L. Narra, R. A. Poldrackc, R. M. Bilderc, A. W. Toga, Construction of a 3d probabilistic atlas of human cortical structures, NeuroImage 39 (2008) 1064–1080.

[9] A. MacKenzie-Graham, E.-F. Lee, I. D. Dinov, M. Bota, D. W. Shattuck, S. Ruffins, H. Yuan, F. Konstantinidis, A. Pitiot, Y. Ding, G. Hu, R. E. Jacobs, , A. W. Toga, A multimodal, multidimensional atlas of the c57bl/6j mouse brain, Journal of Anatomy 2 (2004) 93–102.

[10] D. Rueckert, A. Frangi, J. Schnabel, Automatic construction of 3d statistical deformation models using non-rigid registration, in: Medical Image Computing and Computer-Assisted Intervention, 2001, pp. 77–84.

[11] E. R. Sowell, P. M. Thompson, S. N. Mattson, K. D. Tessner, T. L. Jernigan, E. P. A. Riley, A. W. Toga, Voxel-based morphometric analyses of the brain in children and adolescents prenatally exposed to alcohol, NeuroReport 12 (2001) 515–523.

[12] P. M. Thompson, C. Schwartz, A. W. Toga, High-resolution random mesh algorithms for creating a probabilistic 3d surface atlas of the human brain, NeuroImage 3 (1996) 19–34.

[13] A. Köhn, J. Drexl, F. Tietter, M. König, H. O. Peitgen, Gpu accelerated image registration in two and three dimensions, in: Bildverarbeitung fur die Medizin 2006, Springer Berlin Heidelberg, 2006, pp. 261–265.

[14] A. Khamene, R. Chisu, W. Wein, N. Navab, F. Sauer, A novel projection based approach for medical image registration, in: BioMedical Image Registration, Springer Berlin / Heidelberg, 2006, pp. 247–256.

[15] C. Vetter, C. Guetter, C. Xu, R. Westermann, Non-rigid multi-modal registration on the gpu, in: J. P. Pluim, J. M. Reinhardt (Eds.), SPIE Medical Imaging 2007: Image Processing, 2007, pp. 555–560.

[16] D. Ruijters, B. M. ter Haar-Romeny, P. Suetens, Efficient gpu-accelerated elastic image registration, in: Proc. Sixth IASTED International Conference on Biomedical Engineering (BioMed), Innsbruck, Austria, 2008, pp. 419–424.

[17] Z. Fan, C. Vetter, C. Guetter, D. Yu, R. Westermann, A. Kaufman, C. Xu, Optimized gpu implementation of learning-based non-rigid multi-modal registration, in: Proceedings of SPIE, Vol. 6914, 2008, p. 69142Y.

[18] R. Shams, N. Barnes, Speeding up mutual information computation using nvidia cuda hardware, in: Proceedings of the 9th Biennial Conference of the Australian Pattern Recognition Society on Digital Image Computing Techniques and Applications, IEEE Computer Society Press, 2007, pp. 555–560.

[19] R. Shams, P. Sadeghi, R. Kennedy, R. Hartley, Parallel computation of mutual information on the gpu with application to real-time registration of 3d medical images, Computer Methods and Programs in Biomedicine (2009) doi:10.1016/j.cmpb.2009.11.004.

[20] P. Muyan-Ozcelik, J. D. Owens, J. Xia, S. S. Samant, Fast deformable registration on the gpu: A cuda implementation of demons, in: Proceedings of the 2008 International Conference on Computational Science and Its Applications (ICCSA), IEEE Computer Society Press, 2008, pp. 223–233.

[21] R. Shams, P. Sadeghi, R. A. Kennedy, R. I. Hartley, A survey of medical image registration on multicore and the gpu, IEEE Signal Processing Mag 27 (2) (2010) 50–60.

[22] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, J. A. Stratton, W. mei W. Hwu, Program optimization space pruning for a multithreaded gpu, Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization (2008) 195–204.

[23] B. Jang, S. Do, H. Pien, D. Kaeli, Architecture-aware optimization targeting multithreaded stream computing, Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units 383 (2009) 62–70.

[24] nVidia, Santa Clara, CA, CUDA Programming Guide (August 2009).
[25] Cuda occupancy calculator [online].
[26] I. Yanovsky, P. M. Thompson, S. Osher, A. D. Leow, Topology preserving log-unbiased nonlinear image registration: Theory and implementation, in: IEEE Computer Vision and Pattern Recognition, 2007, pp. 1–8.
[27] G. Christensen, R. Rabbitt, M. Miller, Deformable templates using large deformation kinematics, IEEE Transactions on Image Processing 5 (10) (1996) 14351447.
[28] G. Christensen, H. Johnson, Consistent image registration, IEEE Transactions on Medical Imaging 20 (7) (2001) 568582.
[29] E. DAgostino, F. Maes, D. Vandermeulen, P. Suetens, A viscous fluid model for multimodal non-rigid image registration using mutual information, Medical Image Analysis 7.
[30] L. Rudin, S. Osher, E. Fatemi, Nonlinear total variation based noise removal algorithms, Physics D: Nonlinear Phenomena 60 (1992) 259–268.
[31] S. Osher, M. Burger, D. Goldfarb, J. Xu, W. Yin, An iterative regularization method for total variation based image restoration, Multiscale Modeling and Simulation 4 (2005) 460–489.
[32] M. Burger, G. Gilboa, S. Osher, J. Xu, Nonlinear inverse scale space methods, Commun. Math. Sci. 4 (1) (2006) 179–212.
[33] J. Xu, S. Osher, Iterative regularizaiton and nonlinear inverse scale space applied to wavelet based denoising, in: IEEE Image Proceedings, Vol. 16, 2007, pp. 534–544.
[34] C. Tomasi, R. Manduchi, Bilateral filtering for gray and color images, in: Sixth International Conference on Computer Vision, 1998, pp. 839–46.
[35] A. Buades, B. Coll, J.-M. Morel, On image denoising methods, Multiscale Modeling and Simulation 4 (2) (2005) 490–530.
[36] A. Buades, B. Coll, J.-M. Morel, Neighborhood filters and pde's, Numer. Math. 105 (1) (2006) 1–34.
[37] G. Gilboa, S. Osher, Nonlocal linear image regularization and supervised segmentation, Multiscale Modeling and Simulation 6 (2) (2007) 595–630.
[38] G. Gilboa, S. Osher, Nonlocal operators with applications to image processing, in: CAM-Report, 2007, pp. 7–23.
[39] S. Kindermann, S. Osher, P. W. Jones, Deblurring and denoising of images by nonlocal functionals, Multiscale Modeling and Simulation 4 (4) (2005) 1091–1115.
[40] U. Clarenz, U. Diewald, , M. Rumpf, Anisotropic diffusion in surface processing, in: T. Ertl, B. Hamann, A. Varshney (Eds.), Proceedings of IEEE Visualization, 2000, pp. 397–405.
[41] M. Desbrun, M. Meyer, P. Schröder, A. Barr, Implicit fairing of irregular meshes using diffusion and curvature flow, in: ACM SIGGRAPH, 1999.
[42] M. Desbrun, P. S. M. Meyer, A. Barr, Anisotropic featurepreserving denoising of height fields and bivariate data, Graphics Interface.
[43] H. Hoppe, T. DeRose, T. Duchamp, M. Halstead, H. Jin, J. McDonald, J. Schweitzer, W. Stuetzle, Piecewise smooth surface reconstruction, in: ACM SIGGRAPH, 1994, pp. 295–302.
[44] T. Tasdizen, R. Whitaker, P. Burchard, S. Osher, Geometric surface processing via normal maps, ACM Transactions on Graphics 22 (2003) 1012–1033.
[45] T. Tasdizen, R. T. Whitaker, P. Burchard, S. Osher, Geometric surface smoothing via anisotropic diffusion of normals, in: Proceedings of IEEE Visualization, 2002, pp. 125–132.
[46] S. Yoshizawa, A. Belyaev, H. P. Seidel, Smoothing by example: Mesh denoising by averaging with similarity-based weights, in: IEEE International Conference on Shape Modeling and Applications, 2006, pp. 38–44.
[47] B. Dong, J. Ye, S. Osher, I. Dinov, Level set based nonlocal surface restoration, SIAM Multiscale Modeling & Simulation 7 (2008) 589–598.