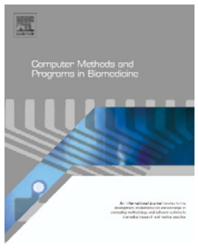
COMPUTER METHODS AND PROGRAMS IN BIOMEDICINE XXX (2009) XXX–XXX

journal homepage: www.intl.elsevierhealth.com/journals/cmpb



Parallel computation of mutual information on the GPU with application to real-time registration of 3D medical images

*Ramtin Shams**, *Parastoo Sadeghi*, *Rodney Kennedy*, *Richard Hartley*

College of Engineering and Computer Science (CECS), The Australian National University, Canberra, ACT 0200, Australia

ARTICLE INFO

Article history:
Received 12 April 2009
Received in revised form
21 August 2009
Accepted 4 November 2009

Keywords:
Image registration
Mutual information (MI)
Histogram
Parallel processing
Graphics processor unit (GPU)

ABSTRACT

Due to processing constraints, automatic image-based registration of medical images has been largely used as a pre-operative tool. We propose a novel method named *sort and count* for efficient parallelization of mutual information (MI) computation designed for massively multi-processing architectures. Combined with a parallel transformation implementation and an improved optimization algorithm, our method achieves real-time (less than 1 s) rigid registration of 3D medical images using a commodity graphics processing unit (GPU). This represents a more than 50-fold improvement over a standard implementation on a CPU. Real-time registration opens new possibilities for development of improved and interactive intraoperative tools that can be used for enhanced visualization and navigation during an intervention.

1. Introduction

Registration is a fundamental problem, frequently encountered in image processing applications. *Virtually all large systems which evaluate images require the registration of images, or a closely related operation, as an intermediate step* [1]. In medical imaging, images of differing modalities often need to be aligned as a preprocessing step for many planning, navigation, data-fusion and visualization applications. Fully automatic image-based registration of images has been extensively researched in the medical imaging domain [2]. Image-based registration of medical images is computationally expensive and as such has been largely confined to pre-operative applications. Development of efficient registration methods can lead to new opportunities for intraoperative applications and will allow for the adaptation of existing tools and new and improved visualization and navigation tools during an intervention.

The emergence of massively multi-processing graphic units with general purpose programing capabilities, in recent years, has brought numerous opportunities and new challenges. On the one hand, super-computing (in the order of 1 TFLOPS) is now cheap and comes in a desktop size with power requirements not much greater than an office computer, on the other hand, existing applications have to be redeveloped for a massively multi-processing architecture and many tools including basic algorithms have to be re-engineered.

Image-based registration typically consists of several iterations of some optimization algorithm with the aim to minimize a suitable cost function subject to an optional smoothness criteria

$$T_{opt} = \underset{T}{\arg\min}\ -\mathcal{S}(F; M(T)) + \mathcal{L}(T), \quad (1)$$

where $\mathcal{S}$ is the similarity function to be maximized,[1] $\mathcal{L}$ is the smoothness term, T is a transformation operator, and F and

* *Corresponding author*. Tel.: +61 2 6125 8612.
E-mail address: ramtin.shams@anu.edu.au (R. Shams).

[1] In line with the optimization literature, our notation shows minimizing the equivalent cost function, $-\mathcal{S}$.

0169-2607/$ – see front matter © 2009 Elsevier Ireland Ltd. All rights reserved.
doi:10.1016/j.cmpb.2009.11.004

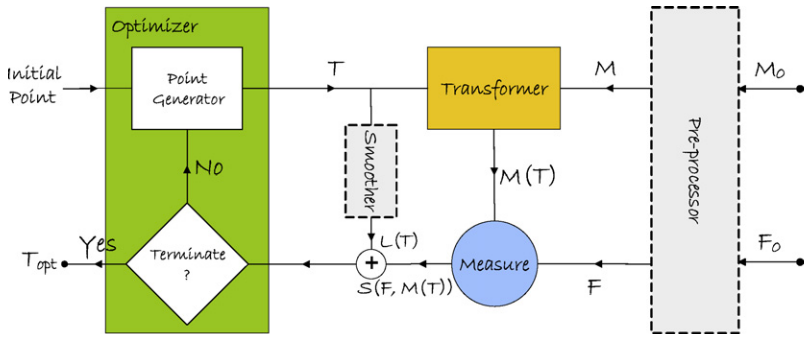


Fig. 1 – A general registration solver and its main components where *F*, *M*, and *M*(*T*) are fixed, moving and transformed moving images, respectively.

$M(T)$ are the *fixed* and transformed *moving* images, respectively. Each iteration of the optimization involves transformation of the moving image and computation of the cost function against the fixed image. The major components of a general registration solver are depicted in Fig. 1.

- *Preprocessor*: an optional preprocessing step, such as denoising or image gradient calculation may be performed on original fixed and moving images (F_0 and M_0) to prepare them for registration. For example, [3] uses image gradients for registration.
- *Optimizer*: the optimizer searches the parameter space for the best match between the images by examining transformations of the moving image and comparing the results with the fixed image. Since an exhaustive search of the parameter space is often not possible, an optimization algorithm such as Powell, simplex, gradient descent, quasi-Newton or Levenberg–Marquardt [4] is used.
- *Transformer*: the transformer maps the points of the moving image to new locations in the transformed image. Depending on the registration problem, the transformation can be rigid, similarity, affine, perspective, projective, or non-rigid.
- *Similarity measure*: a method of measuring the similarity of images is required for automatic registration; ideally the similarity measure attains its maximum, where the images are perfectly aligned and decreases as the images move farther away. Several cost functions and similarity measures have been studied in the literature such as sum of squared differences (SSD), sum of absolute differences (SAD), normalized cross correlation (NCC), correlation ratio (CR) [5], and mutual information (MI) [6,7].

  MI-based registration, in particular, has received much attention in the literature [2] and has become the cost function of choice for registration of multi-modal images.

On a CPU, registration method's execution time is typically dominated by the transformation function. GPUs, however, are specifically designed to perform geometric transformations. Transformations for individual elements are independent and can be efficiently parallelized. Geometric transformations (regardless of their type) require some sort of interpolation that involves adjacent voxels in a cubic region of memory. Standard computer architectures are designed to optimize sequential memory access through their caching mechanism. This does not fully benefit 3D interpolations over a cubic mesh. Modern GPUs, on the other hand, support a 3D texture addressing mode that takes the geometric locality into account for caching purposes. This greatly improves the efficiently of transformations on the GPUs.

Parallel implementation of mono-modality cost functions and similarity measures such as SSD and NCC is straightforward and the entire registration process can be efficiently parallelized. In this paper, we will focus on the more challenging task of multi-modality image registration. For multi-modal registration, efficient computation of a similarity measure such as MI on the GPU is nontrivial. This is due to a need to estimate joint probability density of the fixed and moving images. This typically entails, computing a joint histogram of image intensities, a seemingly simple task which is surprisingly far from trivial on a GPU [8,9]. We emphasize the need for implementation of the similarity measure on the GPU for best performance. This allows the entire registration iteration to be performed on the GPU and avoids costly data transfers between the GPU and the CPU memory.

1.1. *Related work*

Histogram computation on the GPU has been investigated in [10] using a shader program and in [11] using a scattering approach in the vertex shader. A projection-based registration algorithm on the GPU for single-modality rigid registration is presented in [12]. MI-based non-rigid registration on the GPU using OpenGL has been investigated in [13], where histogram computation was also performed on the GPU. These methods use an earlier programing paradigm for the GPUs where the program had to be mapped to graphics processing pipeline and one had to think in terms of vertex and fragment shaders. They also involve clever tricks to get around the limitations of the hardware. Graphics-based programing of the GPUs offers limited capabilities for non-graphic applications, is tied to graphics pipeline, has substantial programing overhead, and has been superseded by modern programing languages for the GPUs such as CUDA and OpenCL. The performance improvements over CPU-based implementations are modest (e.g. authors in [13,12] report only five and three times performance improvement, respectively).

In [8], we presented two fast histogram computation methods on the GPU using CUDA. In [9], we proposed an approximate histogram computation method to speedup MI computation and the registration on the GPU using CUDA. The increased performance comes at the cost of reduced accuracy due to approximating the joint histograms. A 2D implementation of Viola's [14] approximation of MI and its derivatives is given by [15]. Viola's method is based on stochastic sampling of image intensities and Parzen windowing. Smaller sample sizes reduce the accuracy of the estimates [14]. Increasing the sample size improves the shape of cost function and the estimate of MI and its derivatives. However, the cost of method is quadratic in the number of samples. This can become prohibitive if a large number of samples is required to achieve a desired level of accuracy.

A CUDA implementation of *Demons* deformable registration algorithm with a single-modality similarity measure is described in [16]. Compared to multi-modality similarity measures, implementation of single-modality similarity measures is straightforward on parallel architectures. The input data can be processed independently (with the exception of the final *reduction* step) and is suitable for the single instruction multiple data (SIMD) architecture of massively multi-processors.

1.2. *Contributions*

One can argue that GPU-based algorithms are hardware-specific and inevitably bound to be superseded by future generations of hardware. The argument was certainly stronger for previous generations of GPUs with limited general purpose programing capabilities. This is less of an issue for modern GPUs where the programing environment is C/C-like and there is an upgrade path to future generations of hardware. Nevertheless, there is more value in devising algorithms designed for massively parallel architectures with no dependence on the specifics of the hardware. This is the approach taken by this paper.

In this paper, we present a novel algorithm for efficient computation of MI for massively parallel architectures. The method is based on direct computation of joint histograms of image intensives and does not involve approximating the histograms or estimating MI by stochastic sampling. We introduce the concept of *sort and count* for efficient computation of histograms with large number of bins on massively multi-processing architectures. The proposed method allows us to compute histograms in a completely collision-free manner and removes the need for costly atomic operations or data-synchronization that would otherwise be needed to compute a histogram. We present extensive experiments on real 3D medical data using the Vanderbilt database [17] to demonstrate the performance and accuracy of the proposed method and its fitness for registration applications. We demonstrate real-time registration of Vanderbilt images (in less than 1 s), for the first time. While, we have implemented an affine registration method, the misalignment in the Vanderbilt database is known to be rigid and the experiments in Section 4 use a subset of affine transformation parameters for registration.

The main methods and concepts presented in this paper are general and hardware-independent and can be applied to an arbitrary platform. However, we have developed and tested the methods on NVIDIA hardware using the *Compute Unified Device Architecture* (CUDA) v2.0. There are certain implementation details that are specific to this platform. In Section 2.3, we take the time to briefly explain CUDA's terminology, architecture and limitations that are most relevant to our specific application. A reader who is familiar with CUDA may skip Section 2.3 without loss of continuity.

2. Concepts

2.1. *Entropy*

Entropy of a random variable is a measure of the average or expected information content of an event, whose distribution is determined by the marginal probability of the random variable. One such measure was introduced by Shannon in 1948 [18], and is defined as

$$H(X) = \sum_{x \in X} p(x) \log \frac{1}{p(x)}, \qquad (2)$$

where $p(.)$ is the probability mass function (pmf) of the random variable X. Shannon entropy measures the degree of uncertainty of a random variable by scoring less likely outcomes higher than the more likely ones. This is consistent with the notion that knowledge of an outcome that can be easily predicted is considered less valuable.

2.2. *Mutual information*

Mutual information of two random variables is the amount of information that each carries about the other and is defined as

$$\begin{aligned} I(X;Y) &= H(X) - H(X|Y) \\ &= H(X) + H(Y) - H(X,Y), \end{aligned} \qquad (3)$$

$$I(X;Y) = \sum_{x}\sum_{y} p(x,y) \log \frac{p(x,y)}{p(x)p(y)}, \qquad (4)$$

where $H(X|Y)$ is the information content of random variable X if Y is known, $H(X,Y)$ is the joint entropy of the two random variables and is a measure of combined information of the two random variables. $I(X;Y)$ can be thought of as the reduction in uncertainty of random variable X as a result of knowing Y. The uncertainty is maximally reduced, when there is a one-to-one mapping between the two random variables and is not reduced at all if the two random variables are independent and do not provide any information about one another.

A variation is given by the normalized mutual information (NMI) defined as

$$\tilde{I}(X;Y) = \frac{H(X)+H(Y)}{H(X,Y)} \qquad (5)$$

which is also commonly used as an image registration metric and has been shown to be more robust than MI in the presence of non-overlapping regions in the images [19]. We note that MI and NMI are equivalent in terms of computational complexity.

A common method for computing the MI/NMI of two images is to estimate the joint pmf from the histogram of the joint intensities of corresponding positions in the two images. The most straightforward method to compute the joint histogram is to transform the moving image and interpolate image intensities in the transformed image from the original using linear interpolation or similar methods and then compute the joint histogram of intensities from corresponding pairs of points in the fixed and moving images. MI computed in the manner, does not smoothly vary with the registration parameters and may not be suitable for gradient-descent based optimization methods. Other histogram computation methods such as *Parzen windowing* [20] and *partial volume* (PV) histogram [21,22] exist that result in MI functions that smoothly vary with the change in registration parameters. These methods also provide closed-form solutions for MI derivatives which allows the use of gradient-based optimization methods.

2.3. An overview of CUDA

We provide a brief overview of the terminology, main features, and limitations of CUDA. More information can be found in [23]. A reader who is familiar with CUDA may skip this section.

CUDA can be used to offload data-parallel and compute-intensive tasks to the GPU. The computation is distributed in a *grid* of *thread blocks*. All blocks contain the same number of threads that execute a program on the *device* [2], known as the *kernel*. Each block is identified by a two-dimensional block ID and each thread within a block can be identified by an up to three-dimensional ID for easy indexing of the data being processed. The block and grid dimensions, which are collectively known as the *execution configuration*, can be set at run-time and are typically based on the size and dimensions of the data to be processed.

It is useful to think of a grid as a logical representation of the GPU itself, a block as a logical representation of a multi-core processor of the GPU and a thread as a logical representation of a processor core in a multi-processor. Blocks are time-sliced onto multi-processors. Each block is always executed by the same multi-processor. Threads within a block are grouped into *warps*. At any one time a multi-processor executes a single warp. All threads of a warp execute the same instruction but operate on different data.

While the threads within a block can co-operate through a cached but small *shared* memory (16 KB), a major limitation is the lack of a similar mechanism for safe co-operation between the blocks. This makes implementation of certain programs such as a histogram difficult and rather inefficient.

The device's DRAM, the *global memory*, is un-cached. Access to global memory has a high latency (in the order of 400–600 clock cycles), which makes reading from and writing to the global memory particularly expensive. However, the latency can be hidden by carefully designing the kernel and the execution configuration. One typically needs a high density of arithmetic instructions per memory access and an execution configuration that allows for hundreds of blocks and several hundred threads per block. This allows the GPU to perform arithmetic operations while certain threads are waiting for the global memory to be accessed.

The throughput of global memory access is also dependent on the access pattern. When certain requirements are met by threads in a warp, access to global memory by multiple threads can be combined into a single transaction for contiguous memory locations. This is known as memory *coalescing*. Non-coalesced memory access can severely affect the performance of an application and should be avoided were possible. Coalescing global memory access is perhaps the single most important consideration in optimizing CUDA code [24]. It may even be worthwhile to reorganize data prior to execution of a kernel in order to ensure coalesced access. The exact requirements for memory coalescing differs for different generations of GPUs and we refer the reader to [24] for a detailed discussion.

Areas of the global memory can be mapped as read-only *texture memory*. The texture memory is cached and also optimized for 2D and 3D indexing. This is particularly useful for image processing applications that frequently access adjacent data elements in a rectangular or cubic grid. Textures also provide hardware accelerated support for linear interpolation of adjacent data elements in a grid.

The data is transferred between the host and the device via the *direct memory access* (DMA), however, transfers within the device memory are much faster. To give the reader an idea, device to device transfers on GTX 8800 and GTX 280 are around 70 GB/s and 140 GB/s, respectively, whereas, host to device transfers can be around 2–3 GB/s. As a general rule, host to device memory transfers should be minimized whenever possible. One should also batch several smaller data transfers into a single transfer.

Shared memory is divided into a number of banks that can be read simultaneously. The efficiency of a kernel can be significantly improved by taking advantage of parallel access to shared memory and by avoiding bank conflicts.

The higher processing power of the GPU compared to the standard *central processor unit* (CPU), comes at the cost of reduced data caching and flow control logic as more transistors have to be devoted to data processing. This imposes certain limitations in terms of how an application may access memory and implement flow control. As a result, implementation of certain algorithms (even trivial ones) on the GPU may be difficult or may not be computationally justified. In particular, CUDA devices with *compute capability* 1.0 (such as GTX 8800) do not support *atomic* operations. GPUs with compute capability 1.1 support some atomic operations on the global memory. Newer GPUs with compute capability 1.3 (such as GTX 280) support some atomic operations in the shared memory. However, existing GPUs still lack other synchronization primitives such as *critical section* and *mutual exclusion* (mutex). The only supported synchronization primitive is the *thread join* which only works among the threads of the same *thread block*.

In designing an algorithm for a massively multi-processing architecture, regardless of the availability of synchronization and atomic features, one should minimize dependence on synchronization among threads, as it essentially causes parallel processes to become serialized and reduces performance.

[2] We use the terms *device* and the GPU, and *host* and the CPU interchangeably.

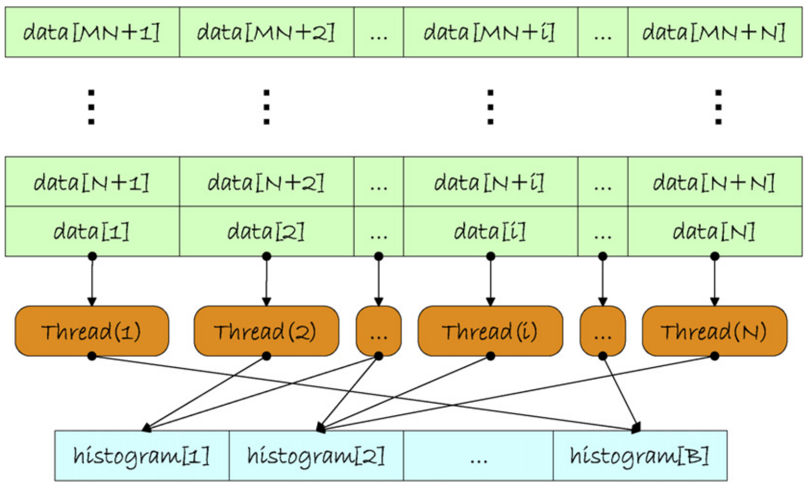


Fig. 2 – Parallel calculation of a histogram with B bins distributed to N threads, where each thread processes a vector of size M + 1. Histogram updates conflict and require synchronization of the threads or atomic updates to the histogram memory.

A typical CUDA implementation consists of the following stages:

(1) Allocate memory on the device.
(2) Transfer data from the host to the device.
(3) Initialize device memory if required.
(4) Determine the execution configuration.
(5) Execute kernel(s). The result is stored in the device memory.
(6) Transfer data from the device to the host.

The efficiency of iterative or multi-phase algorithms can be improved if all the computation can be performed in the GPU, so that step 5 can be run several times without the need to transfer the data between the device and the host.

3. Method

In this section, we first discuss the problem of efficient parallelization of histogram computation. We then introduce a novel algorithm for histogram computation using any sort algorithm that meets certain criteria, known as *sort and count* algorithm. We then use bitonic sort and count for efficient computation of histograms and MI and combine that with efficient computation of transformations and an improved Powell optimization to perform 3D–3D registrations in real-time.

3.1. *Parallel histogram computation*

Histogram calculation is straightforward on a sequential processor as shown in Listing 1.

Listing 1 A simple histogram code snippet for a sequential processor.

```
for (i = 0; i < data_len; i++)
{
    //'data[]' is normalized between 0.0 and 1.0.
    bin = data[i] * (bins - 1);
    //'histogram[]' is already initialized to zero.
    histogram[bin]++;
}
```

Parallelizing a histogram with B bins over N threads is schematically shown in Fig. 2. The input data is distributed among the threads. Updates to histogram memory is data dependent and as such, many threads may attempt to update the same location of the memory resulting in read/write conflicts. Some GPUs lack synchronization primitives to deal with concurrent updates, others that support atomic updates, still suffer reduced performance when the conflicts occur. Histograms have been traditionally difficult to compute efficiently on the GPU [25]. Lack of an efficient histogram method on the GPU, often requires the data to be moved back from the device (GPU) memory to the host (CPU), resulting in costly data transfers and reduced efficiency. Histogram computation can indeed become the bottleneck of an otherwise efficient method.

We propose to use a method which is completely conflict-free and removes the need for any update synchronization and costly data transfers between the GPU and the CPU.

3.2. *Sort and count algorithm*

Let $I = \{1, 2, \ldots, n\}$ be an index set and A be a set with the same cardinality as I ($|A| = n$) and with a *total order* [3] defined in terms of a comparison operator $\leq$. A sequence s is defined as a one-to-one mapping $s : I \to A$. A sort algorithm arranges an arbitrary sequence

$$s : \{a_1, a_2, \ldots, a_n\}, \quad a_i \in A$$

[3] A total order is a binary relation that is transitive ($x \leq y, y \leq z \to x \leq z$), antisymmetric ($x \leq y, y \leq x \to x = y$), and total (either $x \leq y$ or $y \leq x$). A *partial order*, such as the subset operator ($\subset$), does not possess the last property.

to a new sequence

$$s' : \{a_{\phi(1)}, a_{\phi(2)}, \ldots, a_{\phi(n)}\}$$

such that

$$a_{\phi(i)} \leq a_{\phi(j)}, \quad \text{for all} \quad i < j,$$

where ϕ is a permutation of the index set I.

A *binary comparison-only* sort algorithm, is one that reorganizes a sequence using knowledge gained solely by application of a comparison operator on a pair of records. A sort algorithm is *stable* if it maintains the relative order of elements with equal keys (e.g. values) [26]. An arbitrary sort algorithm with a given comparison operator $\leq$ can be *stabilized* by introducing a new comparison operator $\prec$ such that

$$a_i \prec a_j \quad \Leftrightarrow \quad (a_i < a_j) \vee (a_i = a_j \wedge i < j). \tag{6}$$

Note that no two keys are equal under the new comparison operator.

Consider a stabilized binary comparison-only sort algorithm such as S. We introduce a modification to S so that it counts the number of elements that are equal under the original comparison operator while sorting. We assign a counter to each element and initialize the counters to 1 at the beginning

$$s : \{a_1^{(1)}, a_2^{(1)}, \ldots, a_n^{(1)}\},$$

where s is the initial sequence and the superscripts denote the counters. The counters are updated every time a comparison is performed if $a_i^{(n_i)} = a_j^{(n_j)}$ with $i < j$ such that

$$\begin{aligned} n_j &\leftarrow n_j + n_i, \\ n_i &\leftarrow 0. \end{aligned} \tag{7}$$

In other words, every time a comparison of equal elements is performed, the element with a higher position in the sequence accumulates the counter of the lower element. We call this a *sort and count* algorithm.

Here is an example of performing a sort and count on a sequence such as $\{1, 3, 1, 1, 2, 3\}$. The sort and count will result in the following sequence: $\{1^{(0)}, 1^{(0)}, 1^{(3)}, 2^{(1)}, 3^{(0)}, 3^{(2)}\}$. The steps are given below using a simple *bubble sort* algorithm. We can use any sort algorithm that meets the criteria previously defined. In practice, we will use a parallel sort algorithm but it is easier to demonstrate the concept using a simpler sort algorithm such as bubble sort.

$\{\underline{1}^{(1)}, \underline{3}^{(1)}, 1^{(1)}, 1^{(1)}, 2^{(1)}, 3^{(1)}\}$: compare $(1, 3)$, $1 < 3$, no order change

$\{1^{(1)}, \underline{3}^{(1)}, \underline{1}^{(1)}, 1^{(1)}, 2^{(1)}, 3^{(1)}\}$: compare $(3, 1)$, $3 > 1$, swap

$\{1^{(1)}, 1^{(1)}, \underline{3}^{(1)}, \underline{1}^{(1)}, 2^{(1)}, 3^{(1)}\}$: compare $(3, 1)$, $3 > 1$, swap

$\{1^{(1)}, 1^{(1)}, 1^{(1)}, \underline{3}^{(1)}, \underline{2}^{(1)}, 3^{(1)}\}$: compare $(3, 2)$, $3 > 2$, swap

$\{1^{(1)}, 1^{(1)}, 1^{(1)}, 2^{(1)}, \underline{3}^{(\underline{1})}, \underline{3}^{(\underline{1})}\}$: compare $(3, 3)$, $3 = 3$, no order change, higher index 3 accumulates the counter

$\{\underline{1}^{(\underline{1})}, \underline{1}^{(\underline{1})}, 1^{(1)}, 2^{(1)}, 3^{(0)}, 3^{(2)}\}$: compare $(1, 1)$, $1 = 1$, no order change, higher index 1 accumulates the counter

$\{1^{(0)}, \underline{1}^{(\underline{2})}, \underline{1}^{(\underline{1})}, 2^{(1)}, 3^{(0)}, 3^{(2)}\}$: compare $(1, 1)$, $1 = 1$, no order change, higher index 1 accumulates the counter

$\{1^{(0)}, 1^{(0)}, \underline{1}^{(3)}, \underline{2}^{(1)}, 3^{(0)}, 3^{(2)}\}$: compare $(1, 2)$, $1 < 2$, no order change

$\{1^{(0)}, 1^{(0)}, 1^{(3)}, \underline{2}^{(1)}, \underline{3}^{(0)}, 3^{(2)}\}$: compare $(2, 3)$, $1 < 2$, no order change

The sequence is sorted at this stage, however bubble sort will take another iteration to realize this fact. As can be seen, at the end of the sort and count algorithm, numbers with the highest position in their groups have their counters set to the number of group elements.

Theorem. *At the completion of a sort and count algorithm, all elements within a subset with equal values have their counters set to zero except for the element with the highest position in the sequence whose counter contains the cardinality of the subset.*

Proof. Let us denote an arbitrary subset of equally valued elements (if one exists) within the sorted sequence by $\{b_{k+1}^{(n_{k+1})}, b_{k+2}^{(n_{k+2})}, \ldots, b_{k+m}^{(n_{k+m})}\}$. For an element such as $b_i^{(n_i)}$ with $i \neq k + m$ to have a non-zero count, it must have never been compared with $b_{i+1}^{(n_{i+1})}, \ldots, b_{k+m}^{(n_{k+m})}$, otherwise the count would have been accumulated by the higher position element. Now let us update b_i in the original sequence with $b_i + \epsilon$ such that $\epsilon > 0$ and $b_i + \epsilon < b_{k+m+1}$. If we run the stabilized sort algorithm again, this change will not affect the outcome, because the updated element is still less than $b_{k+m+1}, \ldots, b_n$, it is greater than $b_{i-1}, \ldots, b_1$ and since it is never compared against $b_{i+1}, \ldots, b_{k+m}$, the algorithm makes the exact same decisions at each step and hence b_i must occupy the exact same position. However, this also means that the algorithm has failed to sort the sequence $(b_i + \epsilon > b_{i+1})$ and the proof is complete. □

3.3. *Sort and count for histogram computation*

Assume that we have two images J_1 and J_2, for which we would like to determine the joint pmf using a joint histogram with $B_1 \times B_2$ bins, where B_1 and B_2 are the number of bins required for calculating marginal pmfs of J_1 and J_2, respectively. We have assumed that $J_1(\cdot)$ and $J_2(\cdot)$ are normalized with intensity values between 0.0 and 1.0. We note that joint histogram computation can be reformulated as a marginal histogram with B bins, where $B = B_1 \times B_2$ by combining the elements of J_1 and J_2 into a single array J such that,

$$J(\mathbf{x}) = \left\lfloor B_1(J_1(\mathbf{x}) + J_2(\mathbf{x})(B_2 - 1)) + 0.5 \right\rfloor, \tag{8}$$

where $J(\mathbf{x})$ is the intensity of the combined images at spatial locations $\mathbf{x}$ with a dynamic range of $[0, B1 \times B2 - 1]$ and with up to B distinct integer values. Calculating a 1D histogram of $J(\cdot)$ with B bins is equivalent to calculating the 2D histogram for $J_1(\cdot)$ and $J_2(\cdot)$ with B_1 and B_2 bins, respectively.

With a sort algorithm that can be efficiently parallelized (such as bitonic sort), sort and count can be used to parallelize histogram computation of J. Since the outcome of sort contains a single non-zero counter for each unique value of J, a number of threads can process the counters in parallel and update histogram locations corresponding to each unique value of J only if the counter is non-zero and as such the algorithm is guaranteed to be free of update conflicts and has no reliance on synchronization primitives. Of course, in practice, we do not sort the entire input data, the data is read in blocks,

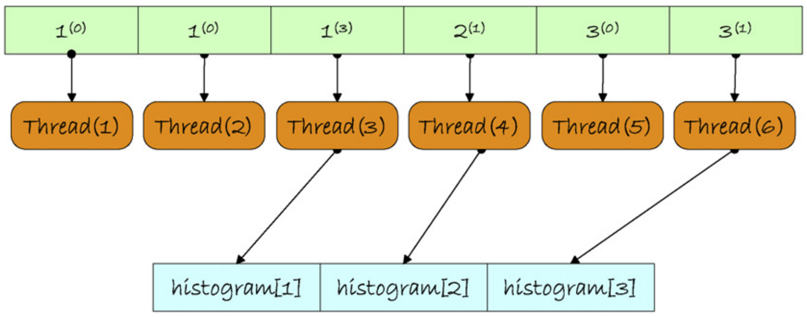


Fig. 3 – Conflict-free update of the histogram with sort and count.

sorted and the histogram is updated with the counters in each iteration. Fig. 3 demonstrates the concept of updating the histogram for the simple example of the previous section, where each thread reads a bin from the sorted sequence and only writes to the histogram memory if the bin contains a non-zero count. As can be seen, some threads may not perform an update and remain idle, however this is a big improvement over having to serialize them.

3.4. *Sort and count for partial volume histogram computation*

The basic sort and count algorithm can be used for computation of a standard joint histogram as discussed in the previous section. The method can also be used for other histogram-based MI computation methods such as B-spline Parzen window [20] and partial volume histogram [21]. These method typically require multiple non-integer histogram updates per joint sample of fixed and moving images. These histogram computation algorithms can be accommodated with minor changes to the basic sort and count algorithm. We briefly discuss PV histogram method in this section.

Under a transformation T, a point such as $\mathbf{y}$ in the fixed image F will correspond to $\mathbf{x} = T\mathbf{y}$ in the moving image M. However, since the coordinates of $\mathbf{x}$ are generally non-integral, the intensity of the transformed point has to be interpolated from the neighboring points with integer coordinates. In a standard joint histogram, once the intensity of $\mathbf{x}$ is computed, the histogram bin corresponding to $(M(\mathbf{x}), F(\mathbf{y}))$ is incremented by 1.

In a partial volume histogram, however, $\mathbf{x}$ contributes to multiple histogram bins associated with its neighboring grid points. The contributions depend on the distance of $\mathbf{x}$ from its neighbors given by

$$P_v(\mathbf{x}, \mathbf{z}) = \prod_{i=1}^{d} (1 - |x_i - z_i|), \qquad (9)$$

where $\mathbf{z}$ is a neighboring point of $\mathbf{x}$, and d is the number of dimensions. The partial volume histogram updates the histogram bin corresponding to $(M(\mathbf{z}), F(\mathbf{y}))$ by $P_v(\mathbf{x}, \mathbf{z})$. For 3D images, partial volume histogram involves 8, generally non-integral, updates to the histogram.

The following minor changes allow the basic sort and count algorithm to be used for partial volume histogram computation:

(1) Bin counters and the histogram data types are defined as floating point types.
(2) The program adds multiple bins for each pair of points to the sort and count list.
(3) Bin counters are initialized to corresponding partial volume contributions.

Computation of MI derivatives using the partial volume histogram method also translates into computation of an appropriate histogram for each derivative [22], where the above method can be used again.

3.5. *Bitonic sort algorithm*

A sequence of non-decreasing or non-increasing numbers is called *monotonic*. A *bitonic* sequence is one that consists of an ascending and a descending monotonic sequence. A *bitonic sorter* converts a bitonic sequence into a monotonic sequence. If a bitonic sequence of $2n$ numbers, $\{a_1, a_2, \ldots a_{2n}\}$, is reorganized into

$$\{\min(a_1, a_{n+1}), \min(a_2, a_{n+2}), \ldots, \min(a_n, a_{2n}), \max(a_1, a_{n+1}),$$
$$\max(a_2, a_{n+2}), \ldots, \max(a_n, a_{2n})\},$$

the new sequence is also bitonic and none of the elements in the first half of the sequence can be greater than any elements in the second half [27]. This is known as the *bitonic merge* theorem. So given a bitonic sequence of size $2n$ one can sort the sequence by merging the sequence first and using 2 bitonic sorters of size n. This provides an iterative algorithm for sorting a sequence of $L = 2^m$ elements using a bitonic sorter. Fig. 4 shows a bitonic sort network for an input sequence of eight elements which completes after 6 iterations regardless of the input sequence. The order of the comparison at each iteration is predetermined as shown in Fig. 4. Bitonic sort is not an optimal sort[4] and requires $\mathcal{O}(L(\log L)^2)$ comparisons. How-

[4] An optimal sort algorithm executes in $\mathcal{O}(L \log L)$ time.

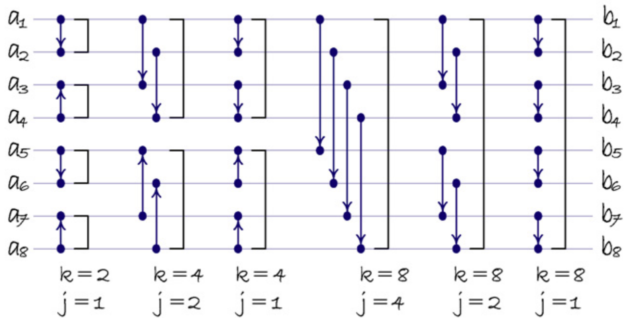


Fig. 4 – Ascending bitonic sort for a sequence of 8 elements. Each connector represents a comparison in the direction denoted by the arrow. Elements involved in each comparison are swapped if they are not in the desired order. *k* and *j* represent block size and comparison offset at each level, respectively.

ever, since the sequence of comparisons is predetermined and data-independent, the algorithm can be efficiently parallelized.

3.6. *Transformations*

Transformations can be efficiently implemented on a massively parallel architecture, as computation for each element is independent of other elements. Additionally, the GPU provides a further speedup through the use of *textures*. A texture is an area of global memory specifically configured for 1D, 2D or 3D access. GPU supports caching for textures and access to adjacent texture coordinates is optimized to minimize a cache-miss. Textures also support linear filtering from neighboring texture coordinates. This is exactly what is needed for linear interpolation of intensities when transforming the moving image and allows us to implement a most efficient 3D transformation method.

Fig. 5 compares the performance of affine transformations for typical 3D images given in million voxels that can be processed per second, for a standard CPU, GTX 8800 and GTX 280 with and without use of textures. Performance of GPU implementation is far superior to the CPU and use of texture improves performance of GPU implementation considerably. We are able to transform 3D images from the Vanderbilt database in less than 1 ms using a GTX 280.

3.7. *Improved Powell optimization*

Powell's multi-dimensional direction set algorithm finds the minimum of a cost function by iteratively minimizing the function along a set of *N* directions, where *N* is the number of independent parameters of the cost function. A line minimization algorithm (typically Brent's) is used to find the minimum in a given direction.

Our implementation of Powell is based on the algorithm described in [4] with one major difference that it also incorporates a resolution parameter per dimension [28]. A minimum distance or resolution for evaluation of the cost function is defined. The cost function keeps track of each point in the *N*-

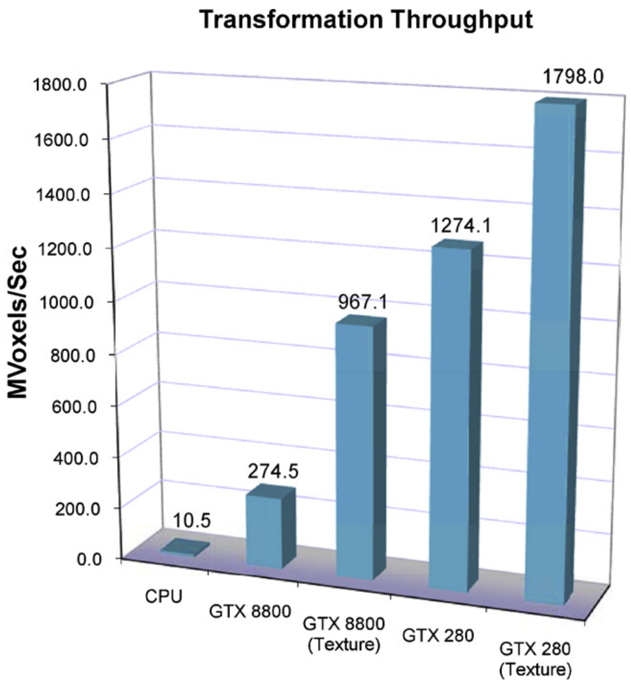


Fig. 5 – Performance of transformations on the CPU and the GPU with and without use of textures. GPU implementation with textures performs best.

dimensional space that it evaluates and will only evaluate a new point if it falls outside all previously evaluated points by the specified minimum distance.

In a standard Powell implementation fractional tolerance and absolute tolerance parameters are used to control convergence of line minimizations. Resolution parameters provide additional flexibility in controlling Powell's overall convergence. Use of an appropriate resolution allows our method to converge much more quickly than standard Powell implementation without affecting the accuracy of registrations as demonstrated in Section 4.2.

3.8. *CUDA implementation*

In our CUDA implementation, the data is distributed among *k* blocks each with N_b threads which process input data in chunks of $2N_b$ size in each iteration. Each block maintains a partial histogram in the global memory. The partial histograms are combined using a parallel *reduction* algorithm at the end to produce the joint histogram used in computation of MI. Input values are stored as 32-bit unsigned integer values, and counters are packed into the higher bits of the same memory location, for most efficient handling.

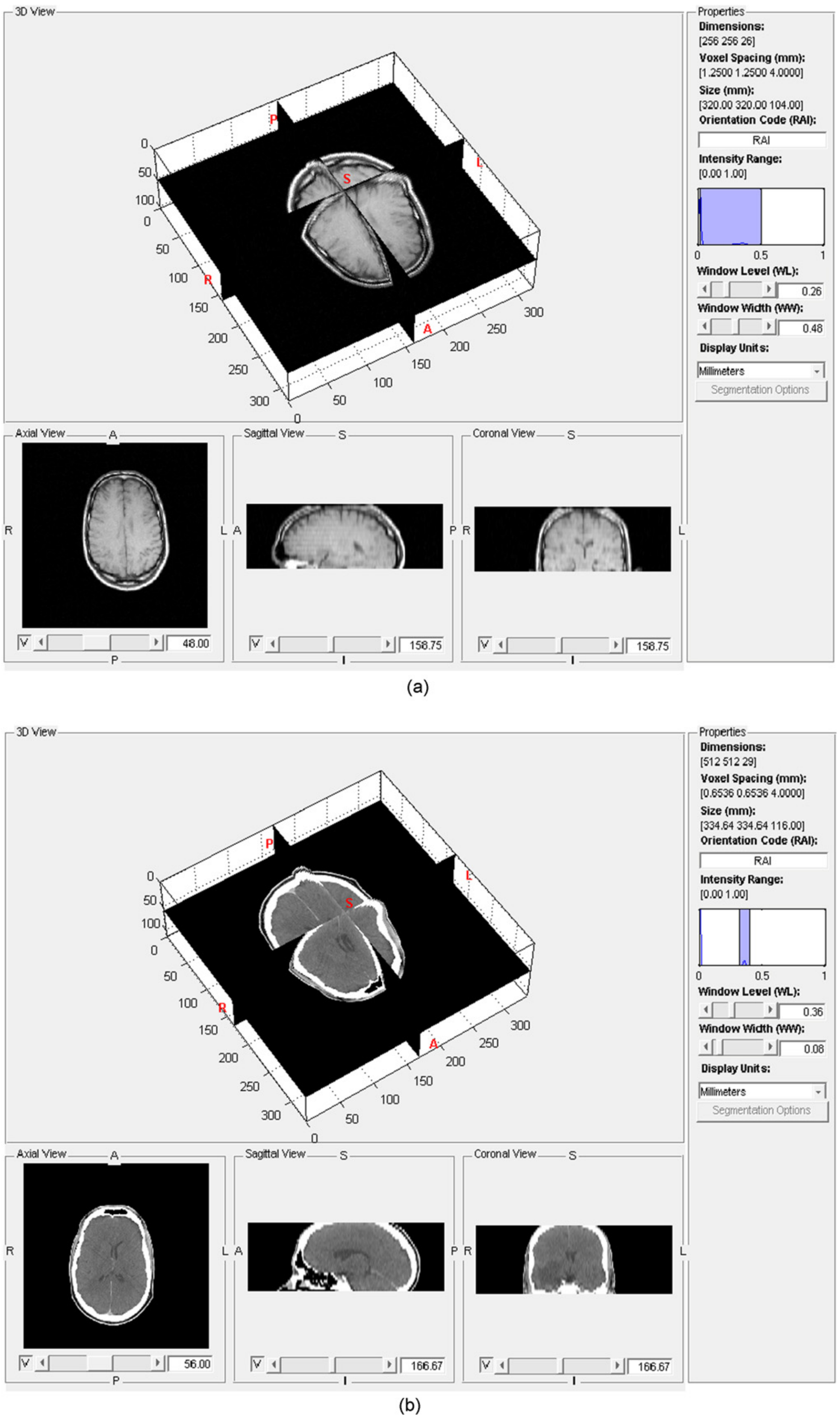
Listing 2 shows the kernel function executed by each thread to perform a bitonic sort and count algorithm. Bins and their associated counters are packed into 32-bit unsigned integer variables, where the lower 16-bit word is used to store the bin number and the higher 16-bin word contains the counter value. The complete source code can be found online at http://cecs.anu.edu.au/ ramtin/cuda/.

Listing 2. Parallel bitonic sort and count.

```
__device__ inline void bitonicSort
(
    // Pointer to data in shared memory
    uint *shared,
    // ID for the current thread
    const uint tid,
    // Number of threads in each block
    const uint threads
)
{
    // Parallel bitonic sort
    // k is the bitonic block length
    // Save log2(k) to speed up computations
    uint log2k = 1;
    for (uint k = 2; k <= threads << 1; k <<= 1, log2k++)
    {
        // Bitonic merge
        // Bitonic block Id
        uint b_id = tid >> (log2k - 1);
        // j is the bitonic offset
        // Save log2(j) to speed up computations
        uint log2j = log2k - 1;
        for (uint j = k >> 1; j > 0; j >>= 1, log2j--)
        {
            uint i1 = ((tid >> log2j) << (log2j + 1))
                + (tid & (j - 1));
            uint i2 = i1 + j;
            uint L1 = shared[i1] & 0x0000ffff;
            uint L2 = shared[i2] & 0x0000ffff;
            // Even blocks are sorted in descending order
            if ((b_id & 1) == 0)
            {
                if (L1 > L2)
                    swap(shared[i1], shared[i2]);
                else if (L1 == L2)
                {
                    // Accumulate counters in i2
                    shared[i2] = (shared[i1] & 0xffff0000)
                        + (shared[i2] & 0xffff0000) + L1;
                    shared[i1] = L1; // Reset i1's counter
                }
            }
            // Odd blocks are sorted in ascending order
            else
            {
                if (L1 < L2)
                    swap(shared[i1], shared[i2]);
                else if (L1 == L2)
                {
                    // Accumulate counters in i1
                    shared[i1] = (shared[i1] & 0xffff0000)
                        + (shared[i2] & 0xffff0000) + L1;
                    shared[i2] = L1; // Reset i2's counter
                }
            }
            // All threads join here
            __syncthreads();
        }
    }
}
```

4. Results

We use Vanderbilt database of brain images (patients 1–9) for our experiments. The database contains MR-T1, MR-T2, MR-PD, CT and PET images of real patients. In total, we performed 47 CT to MR registrations and 41 PET to MR registrations for each experiment. Fig. 6 shows a sample MR-T1 and CT image from the Vanderbilt database. Even though our focus is to demonstrate efficiency of our method, we also provide



(a)

Properties
Dimensions:
[256 256 26]
Voxel Spacing (mm):
[1.2500 1.2500 4.0000]
Size (mm):
[320.00 320.00 104.00]
Orientation Code (RAI):
RAI
Intensity Range:
[0.00 1.00]
Window Level (WL): 0.26
Window Width (WW): 0.48
Display Units: Millimeters
Segmentation Options

Axial View: 48.00
Sagittal View: 158.75
Coronal View: 158.75

(b)

Properties
Dimensions:
[512 512 29]
Voxel Spacing (mm):
[0.6536 0.6536 4.0000]
Size (mm):
[334.64 334.64 116.00]
Orientation Code (RAI):
RAI
Intensity Range:
[0.00 1.00]
Window Level (WL): 0.36
Window Width (WW): 0.08
Display Units: Millimeters
Segmentation Options

Axial View: 56.00
Sagittal View: 166.67
Coronal View: 166.67

Fig. 6 – Sample images from the Vanderbilt database. (a) An MR-T1 image. (b) A CT image.

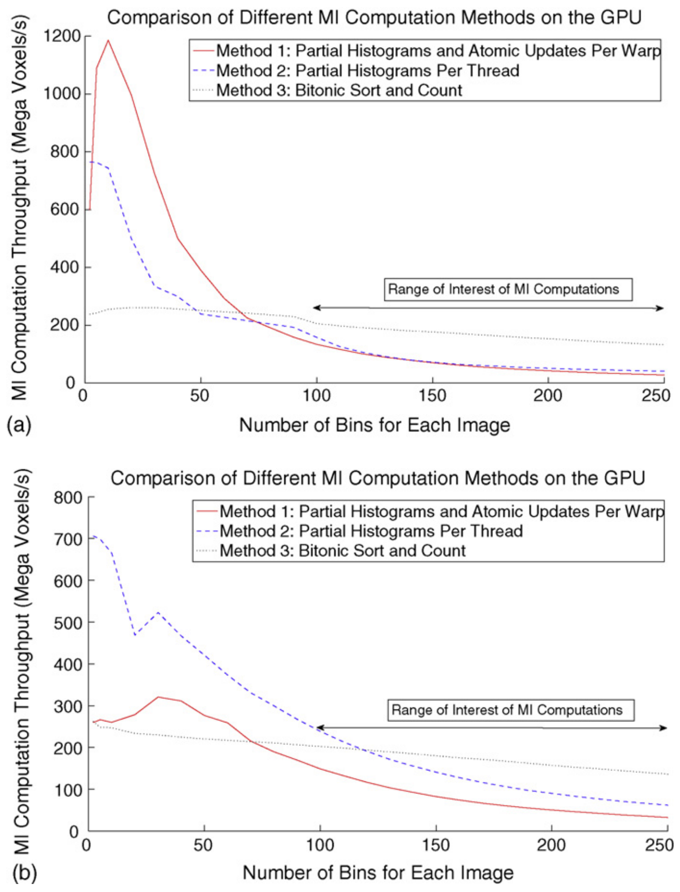


Fig. 7 – Performance of different MI computation methods on the GPU. For the range of bins that is of most interest for MI applications, bitonic sort and count performs best. Also note method 3's independence on data's underlying distribution. (a) MI for two random variables with a uniform distribution. (b) MI for two 3D medical images from the Vanderbilt database.

target registration errors (TRE) for completeness. Accuracy is measured at multiple volumes of interests (VOIs) in the brain that are of neurological significance. The TREs are computed against VOIs registered using the gold standard transformation which were obtained in the original study by using fiducial markers [17].

The Vanderbilt database is used for evaluation of rigid 3D registrations. While we limit our experiments to rigid registrations, we note that our GPU-based implementation is capable of performing affine registrations. In addition, the MI computation algorithm can be readily used for non-rigid registrations.

4.1. *Comparison with other histogram computation methods on the GPU*

We have previously proposed other methods for histogram computation in [8,9], which can be used for MI computation. In this section, we compare performance of these methods with the bitonic sort and count and explain why bitonic sort and count is most suited for MI computation.

4.1.1. *Method 1 – partial histograms and atomic updates per warp*

This method, first presented in [8], maintains a partial histogram per warp. The 32-threads in each warp share the same partial histogram and access to histogram memory is synchronized using a software-based mutex.[5] The method allows for computation of histograms with an arbitrary number of bins, but the efficiency of the method decreases as bins are increased. The reason is due to GPU's small shared memory size (4K 32-bit words). To allow calculation of an arbitrary number of bins, we sub-divide the bin ranges into a number of sub-ranges that fit in the shared memory. For a given execution configuration, we run the algorithm as many times as required to cover the entire bin range. At each iteration the kernel will only process those data elements which fall in the specified bin range. For example, with 4 warps and a limit of 1024 bins per execution, a 10,000 bin histogram requires 10 iterations of the algorithm.

The disadvantage of this method is that the throughput is dependent on the distribution of the data. An input with uniform distribution is close to the best case scenario, as for large inputs the histogram is almost uniform and the histogram update collisions are close to minimal. A degenerate distribution results in maximum histogram update collisions, as all the threads try to update the same histogram bin and as such

[5] This method was introduced prior to availability of compute capability 1.3. For 1.3 devices one can use hardware-based atomic updates in the shared memory.

represents the worst case scenario. The performance for a real application is somewhere in between these lower and upper bounds.

4.1.2. *Method 2 – partial histograms per thread*

This method, first presented in [8], allocates a histogram array per thread in the global memory. A partial histogram is calculated per thread and finally the partial histograms are reduced into a single histogram. The benefit is that, given the size of the global memory, for any practical number of bins the algorithm only requires a single iteration to complete. In addition, there will be no concurrent updates of the same memory location by multiple threads and as such no update synchronization is required, which in turn means that the performance of the method is less data-dependent.[6]

However, there are two drawbacks to this method; firstly, a much larger memory for partial histograms needs to be allocated and initialized to zero at the beginning; secondly, histogram updates need to be done on the global memory, this entails non-coalesced read/writes per input data and is inefficient.

To avoid excessive non-coalesced updates of the global memory, we pack multiple bins in a 32-word in the shared memory and only update the corresponding bin in the global memory when the packed bin overflows. This reduces the updates to the global memory by a factor of 2^b, where b is the number of bits available for storage of a bin in the shared memory. b depends on the number of threads per block and the number of bins and is calculated as

$$b = \frac{s_{\max} \times 32}{B \times N_b}, \qquad (10)$$

where $s_{\max}$ is the maximum number of 32-bit words that can be allocated in the shared memory, B is the number of bins and N_b is the number of threads per block.

There is a trade-off between the number of threads and the number of bits per bin. Increasing the number of threads, decreases b, resulting in more global memory updates, while reducing the number of threads can under-utilize GPU resources and affect the performance. We optimize these parameters to achieve the best performance.

4.1.3. *Method 3 – sort and count*

Previous methods, both, suffer from a sharp reduction in performance as the number of bins increases. This is a problem for MI computation where 100 × 100 to 256 × 256 bins are common for computation of joint histograms. Fig. 7 compares the performance of bitonic sort and count with other methods for two random variables with a uniform distribution (Fig. 7(a)) and for two real images (Fig. 7(b)). As can be seen, method 3 scales much better with increasing number of bins and performs well for computation of 100 × 100 or more bins. The method is also virtually data-independent since it does not use a synchronization method and unlike previous methods can be readily used in any massively multi-processing architecture. For these reasons, we chose bitonic sort and count for MI computations in our registration method.

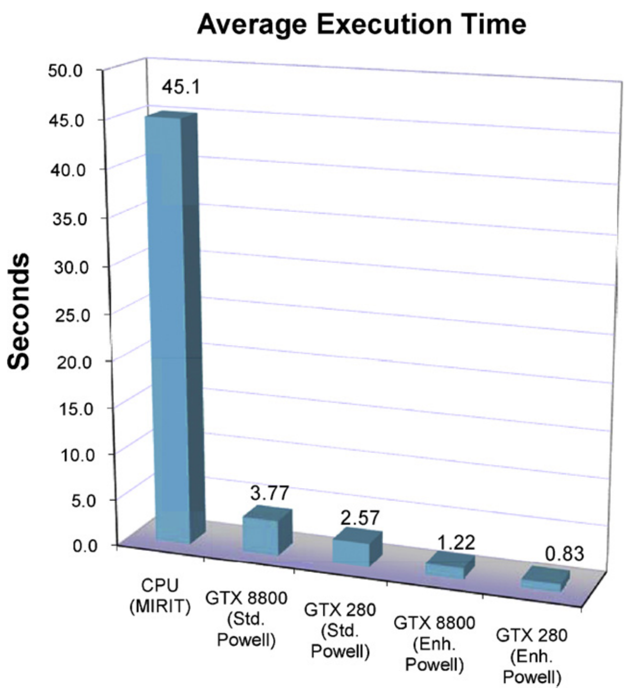


Fig. 8 – Average execution speed for pairs of images from the Vanderbilt data-set. Our method on GTX 280 is more than 50 times faster than MIRIT.

4.2. *Comparison with a CPU implementation*

We compared performance and accuracy of our method against MIRIT[7], a CPU implementation of MI-based registration by Maes et al. [21]. We ran MIRIT with recommended options except for the partial volume option [21]. We found the accuracy of the registrations were actually better without this option. Both methods use Powell optimization algorithm. Our registrations were performed using the modified Powell method with a resolution of 0.02 *mm* for translation parameters and 0.05° for rotation parameters. The results with standard Powell (no resolution parameters) are also included to demonstrate that the accuracy of registrations is not noticeably affected as a result of introducing the resolution parameters, while the performance is considerably improved.

Fig. 8 shows the performance of our method for 3D–3D registrations. On average, we were able to register each image pair in less than one second which represents more than 50 times speedup compared to MIRIT running on a CPU. We also note that by using the enhanced Powell method the performance of GPU-based method can be improved by around 300%.

Table 1 shows the median TRE for MIRIT, our method with enhanced Powell, and our method with standard Powell optimization. Our method with or without resolution parameters, achieves sub-voxel accuracy and even outperforms MIRIT.

[6] The method is not completely data-independent. Bin-packing (explained later) introduces some dependence on the data distribution.

[7] Multi-modality image registration using information theory.

Table 1 – Accuracy comparison (median errors in mm).

Modality	MIRIT	Our method	Our method (std. Powell)
CT–T1	4.73	1.48	1.35
CT–T2	5.30	1.44	1.50
CT–PD	3.50	1.48	1.42
PET–T1	6.33	3.99	4.08
PET–T2	7.33	3.38	3.20
PET–PD	3.19	3.78	3.77

MIRIT failed to converge for a few registrations. MIRIT would be able to reach convergence with a different set of parameters for these cases, however, this would result in failed registration of other successful cases. We set to compare the methods in a fully automatic setting and it was only fair to run both with a single set of parameters. We note that, where convergence was achieved by both methods the TREs were more or less comparable.

5. Conclusion

With the processing power of modern GPUs, and with support for general purpose programing, high performance computing is now possible at an incredibly low cost and low power. Many problems, previously reserved for super-computing facilities, can be re-tackled on commodity hardware. To unlock the full potential of GPU processing, one must be prepared to rethink existing methods and algorithms and adapt them for a massively parallel processing environment. We have demonstrated application of such an approach for registration of medical images. Our main goal was to show the superior performance of the proposed histogram computation method used in MI computation and we did so in the context of 3D rigid registration of images form the Vanderbilt database. The MI computation method is not limited to rigid/affine registration and can be used for deformable registration as well. For deformable registration, gradient-based optimization is typically used which requires computation of derivatives of the cost function. Derivatives of MI can be computed by finite differences or in closed form using B-spline Parzen histogram [20] or partial volume histogram method [22], as discussed in Section 3.4.

Acknowledgements

The authors would like to thank Prof. Fredrik Maes for kindly providing a license for MIRIT. The data and the standard transformations used in our experiments were provided as part of the project, “Retrospective Image Registration Evaluation”, National Institutes of Health, Project Number 8R01EB002124-03, Principal Investigator, J. Michael Fitzpatrick, Vanderbilt University, Nashville, TN. We would like to thank Kitware, Inc. for hosting and providing free access to the data-set. We would also like to thank Mr. Oliver Kutter for kindly providing access to a GTX 280 card for our experiments.

Table A.1 – Host specification

Processor	AM2 Athlon 64× 2 6000+ 3.0 GHz
Memory	4 GB, 800 MHz DDR2
Motherboard	ASUS M2N-SLI Deluxe
Operating system	Windows XP 32-bit

Table A.2 – Device specification (GPU)

Model	GTX 8800	GTX 280
Number of multi-processors	16	30
Number of cores per multi-processor	8	8
Memory	768 MB GDDR3	1 GB GDDR3
Memory interface	384 bits	512 bits
Shared memory per block	16 KB	16 KB
Max number of threads per block	512	512
Warp size	32	32

Appendix A. Hardware configuration

The hardware specification of our experimental setup is given in Tables A.1 and A.2.

REFERENCES

[1] L.G. Brown, A survey of image registration techniques, ACM Comput. Surv. 24 (4) (1992) 325–376.
[2] J.P.W. Pluim, J.B.A. Maintz, M.A. Viergever, Mutual-information-based registration of medical images: a survey, IEEE Trans. Med. Imaging 22 (8) (2003) 986–1004.
[3] R. Shams, P. Sadeghi, R.A. Kennedy, Gradient intensity: a new mutual information based registration method, in: Proceedings of the IEEE Computer Vision and Pattern Recognition (CVPR) Workshop on Image Registration and Fusion, Minneapolis, MN, June, 2007.
[4] W.H. Press, B.P. Flannery, S.A. Teukolsky, W.T. Vetterling, Numerical Recipes in C., second edn., Cambridge University Press, Cambridge, 1992.
[5] A. Roche, G. Malandain, X. Pennec, N. Ayache, The correlation ratio as a new similarity measure for multimodal image registration, in: Medical Image Computing and Computer Assisted Intervention (MICCAI), October, 1998, pp. 1115–1124.
[6] A. Collignon, F. Maes, D. Delaere, D. Vandermeulen, P. Suetens, G. Marchal, Automated multimodality medical image registration using information theory, in: Proceedings of the International Conference Information Processing in Medical Imaging: Computational Imaging and Vision, vol. 3, April, 1995, pp. 263–274.
[7] P. Viola, W.M. Wells III, Alignment by maximization of mutual information, in: Proceedings of the International Conference on Computer Vision (ICCV), June, 1995, pp. 16–23.
[8] R. Shams, R.A. Kennedy, Efficient histogram algorithms for NVIDIA CUDA compatible devices, in: Proceedings of the International Conference on Signal Processing and Communications Systems (ICSPCS), Gold Coast, Australia, December, 2007, pp. 418–422.
[9] R. Shams, N. Barnes, Speeding up mutual information computation using NVIDIA CUDA hardware, in: Proceedings of the Digital Image Computing: Techniques and

Applications (DICTA), Adelaide, Australia, December, 2007, pp. 555–560.
[10] O. Fluck, S. Aharon, D. Cremers, M. Rousson, Gpu histogram computation, in: SIGGRAPH Research Posters, July, 2006.
[11] T. Scheuermann, J. Hensley, Efficient histogram generation using scattering on GPUs, in: Proceedings of the Symposium on Interactive 3D Graphics and Games, April, 2007.
[12] A. Khamene, R. Chisu, W. Wein, N. Navab, F. Sauer, A novel projection based approach for medical image registration, in: Third International Workshop on Biomedical Image Registration (WBIR), Utrecht, The Netherlands, June, 2006, pp. 247–256.
[13] C. Vetter, C. Guetter, C. Xu, R. Westermann, Non-rigid multi-modal registration on the GPU, in: Proceedings of the SPIE Medical Imaging: Image Processing, February, 2007.
[14] P. Viola, W.M. Wells III, Alignment by maximization of mutual information, Int. J. Comput. Vision 24 (2) (1997) 137–154.
[15] Y. Lin, G. Medioni, Mutual information computation and maximization using GPU, in: Proceedings of the IEEE Computer Vision and Pattern Recognition (CVPR) Workshops, June, 2008, pp. 1–6.
[16] 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 International Conference on Computational Science and its Applications (ICCSA), June, 2008.
[17] J. West, J.M. Fitzpatrick, M.Y. Wang, B.M. Dawant, C.R. Maurer, R.M. Kessler, R.J. Maciunas, C. Barillot, D. Lemoine, A. Collignon, F. Maes, P. Suetens, D. Vandermeulen, P.A. van den Elsen, S. Napel, T.S. Sumanaweera, B. Harkness, P.F. Hemler, D.L.G. Hill, D.J. Hawkes, C. Studholme, J.B.A. Maintz, M.A. Viergever, G. Malandain, X. Pennec, M.E. Noz, G.Q. Maguire, M. Pollack, C.A. Pelizzari, R.A. Robb, D. Hanson, R.P. Woods Jr., Comparison and evaluation of retrospective intermodality brain image registration techniques, J. Comput. Assist. Tomogr. 21 (4) (1997) 554–566.
[18] C.E. Shannon, A mathematical theory of communication, Bell Syst. Tech. J. 27 (1948) 379–423, 623–656.
[19] C. Studholme, D.L.G. Hill, D.J. Hawkes, An overlap invariant entropy measure of 3D medical image alignment, Pattern Recognit. 32 (1999) 71–86.
[20] D. Mattes, D.R. Haynor, H. Vesselle, T.K. Lewellen, W. Eubank, PET-CT image registration in the chest using free-from deformations, IEEE Trans. Med. Imaging 22 (1) (2003) 120–128.
[21] F. Maes, A. Collignon, D. Vandermeulen, G. Marchal, P. Suetens, Multimodality image registration by maximization of mutual information, IEEE Trans. Med. Imaging 16 (2) (1997) 187–198, April.
[22] F. Maes, D. Vandermeulen, P. Suetens, Comparative evaluation of multiresolution optimization strategies for multimodality image registration by maximization of mutual information, Med. Image Anal. 3 (4) (1999) 373–386.
[23] Compute Unified Device Architecture (CUDA) Programming Guide, NVIDIA, http://developer.nvidia.com/object/cuda.html, 2007.
[24] NVIDIA CUDA C Programming Best Practices Guide, NVIDIA, http://developer.nvidia.com/object/cuda.html, 2009.
[25] Podlozhnyuk, V. 64-bin histogram, Technical Report, NVIDIA, 2007.
[26] D.E. Knuth, The Art of Computer Programming. Volume 3 – Sorting and Searching, Addison-Wesley, 1973.
[27] K.E. Batcher, Sorting networks and their applications, in: Proceedings of the AFIPS Spring Joint Computer Conference, vol. 32, 1968, pp. 307–314.
[28] R. Shams, R.A. Kennedy, P. Sadeghi, R. Hartley, Gradient intensity-based registration of multi-modal images of the brain, in: Proceedings of the IEEE International Conference on Computer Vision (ICCV), Rio de Janeiro, Brazil, October, 2007.