Accelerating Volume Image Registration through Correlation Ratio based Methods on GPUs

Ang Li
National University of Singapore
ang.li@nus.edu.sg

Akash Kumar
National University of Singapore
akash@nus.edu.sg

***Abstract*—Volume image registration is a basic component of medical image processing which traditionally requires long computation time. In this paper, we propose five Correlation Ratio based schemes that explore the design space for Graphics Processing Unit (GPU) acceleration. Through comparisons among these five schemes, we present the trade-off between benefits and overheads of introducing shadow histograms on various storage (shared memory, global memory) by different level execution units (thread, warp, thread block). Compared to Mutual Information based methods, these Correlation Ratio based methods require less resources for shadow histograms, a faster storage therefore could be exploited to achieve better performance which is shown in our experiments. Particularly, the fifth scheme completely avoids updating conflicts of histogram calculation, leading to a substantial performance improvement (over 18x speedup) over the native FLIRT version. It reduces the registration time from over 100s to less than 6s for two typical 256x256x160 3D images.**

I. Introduction

Image registration, the process of generating a transformation that finds the best alignment between two images [1], is one of the fundamental components encountered in many medical image processing applications [2]. Among various medical registration tools, FMRIB's Linear Image Registration Tool (FLIRT) [3], [4] is reported to be effective and robust [5]. FLIRT exploits several similarity functions, among which the default is Correlation Ratio (CR) [6]. Based on information theory, CR exhibits comparative robustness and stability as the Mutual Information (MI) methods [7], [4]. It is also reported that CR is slightly more accurate and easier to compute than MI [4].

Since NVIDIA published Compute Unified Device Architecture (CUDA) in 2007, the graphic processing unit (GPU) has shown its strength in many areas, especially the domain of image processing including image registration due to the inherently massive data parallelism [8]. However, an efficient GPU implementation for multimodality image registration still remains a difficult task since expensive atomic operations are heavily utilized for histogram calculation, which usually turns into a performance bottleneck [9]. Although several approaches are proposed [10], [11], [12], [9], most of them are specifically targeted for MI and still cannot resolve this bottleneck effectively.

In this paper, we discover that, compared to MI, the CR based similarity functions are more suitable for a SIMD platform. We thus explore the design space of CR and propose five CR-based similarity function schemes. The FLIRT registration framework is implemented to embed these similarity functions to construct a complete registration routine. We show the trade-off between benefits and overheads for introducing local sub-histograms (or shadow histograms) on different storage (shared memory, global memory) by different execution units (thread, warp, thread block). It is highlighted that, in the fifth scheme, the updating conflicts of histogram calculation are completely eliminated, leading to significant performance improvement. Our best scheme achieves over 18x speedup compared to the native FLIRT version, which reduces the registration time from over 100s to less than 6s for typical 256x256x160 3D images. Hence the contributions of this paper are:

- Five CR based registration schemes for GPU. This is the first time, to the best of our knowledge, that the CR method is reported to be employed for image registration on GPUs. Experiment results show that CR outperforms MI on both speed and accuracy.
- A design that completely eliminates the updating conflicts, which has not yet been realized by existing works. This highlights the significant advantage of CR over MI on the GPU platforms.
- The trade-off between benefits of exploiting shadow histograms and its concomitant overhead based on comparisons among different schemes.
- An effective implementation of the FLIRT framework specialized for GPUs which can be used for assessment of other similarity functions.

The rest of the paper is organized as follows. Section 2 introduces the background of image registration, FLIRT framework and histogram calculation. Section 3 presents the proposed schemes to implement the CR similarity function. Section 4 compares these schemes through real experiments. Section 5 discusses the related considerations. Section 6 reviews related works and the conclusion is drawn at last.

II. Background

In this section, we first briefly describe the meaning of image registration, the process of FLIRT framework and the definition of Correlation Ratio. We then present histogram calculation and explain why conflicts exist.

A. Image Registration

Image registration is the process of determining a transformation that maps points from one image (source image) to their

homologous points from another image (reference image). It is generally formalized as an optimization problem in which a cost function is defined. The cost function measures similarity degree between the two images. Therefore, the process of optimization is actually to search for a transform that minimize the cost function (or maximize similarity):

$$\begin{aligned} &Minimize\,(f)\\ &where \quad f = cost\_function(A, B),\\ &\qquad A = reference\_image,\\ &\qquad B = Transform(source\_image),\\ &return \quad Transform \end{aligned}$$

In this paper, we only consider *affine registration* meaning that the "Transform" is affine transform. Thus,

$$B = M \times source\_image + b$$

M is a 3×3 matrix. b is a vector. The 3×4 matrix $[M\ b]$ is defined as a *transform matrix*.

During the search process, various searching strategies are utilized to promote the possibility of obtaining the most optimum transform and reduce searching time. These make up the searching framework.

B. FLIRT Framework

FLIRT algorithm [3], [4] is one of the searching frameworks. It is composed of four stages – each stage targeting on a specific resolution, from 8mm, 4mm, 2mm to 1mm progressively. A stage contains a series of local search and four spaces may be covered by them: rotation, translation, scale and skew. Each space is three dimensional (X, Y, Z). Therefore, if one dimension is represented by one degree of freedom (DOF), at maximum a 12 DOF search can be performed.

The primary stage first executes a rotation space searching with a stride of 60 degrees, thus 6x6x6 times to cover the whole space (360 degrees for all X, Y, Z dimensions). For each checkpoint, a 4 DOF local search is done. Then another rotation space search with a finer stride of 18 degrees is performed. This time, 8000 trials are required. However, unlike the coarse searching, for every checkpoint, we only evaluate that specific spot instead of starting a complete local search. Afterwards, three transformation matrices generating the minimum cost are selected to execute a 7 DOF full search. The obtained matrices are marked as candidates for the next stage.

In the second stage with 4mm resolution, a 7 DOF search is applied to the three candidates together with their 30 neighbors (for each candidate, two perturbations on each rotation dimension with 9 degree deviation, four perturbations on scaling with zoom in and zoom out by a factor of 0.1 and 0.2). The best transformation is found out as input for the next stage.

After that, in the 2mm stage, 7 DOF, 9 DOF and 12 DOF local searches are performed alternately, further approaching the optimum transform.

Finally, in the 1mm stage, the expected global optimal is obtained after going through a 12 DOF local search. This transform matrix is believed to generate the minimum value for the cost function.

C. Correlation Ratio Definition

The Correlation Ratio (CR) [6] of two variables X & Y is a measurement of functional dependence between them, which is defined as:

$$\eta(Y|X) = \frac{Var[E(Y|X)]}{Var(Y)} = 1 - \frac{Var[Y - E(Y|X)]}{Var(Y)} \quad (1)$$

and can be measured as:

$$\eta(Y|X) = 1 - \frac{1}{N\sigma^2}\sum_i N_i\sigma_i^2 \quad (2)$$

in which

$$\sigma^2 = \frac{1}{N}\sum_{\omega\in\Omega} Y(\omega)^2 - m^2, \quad m = \frac{1}{N}\sum_{\omega\in\Omega} Y(\omega)$$
$$\sigma_i^2 = \frac{1}{N_i}\sum_{\omega\in\Omega_i} Y(\omega)^2 - m_i^2, \quad m_i = \frac{1}{N_i}\sum_{\omega\in\Omega_i} Y(\omega)$$

Here Ω denotes the images' overlapping region, N is the number of voxels in Ω. Consider the histogram of X, $\Omega_i = \omega \in BIN(i)$, N_i is the number of voxels in Ω_i. The value of CR is between 0 (no functional dependence) and 1 (fully deterministic dependence). Defined as a ratio, CR is invariant to the scaling of Y or $Y(\omega)$. Meanwhile, CR is asymmetrical by definition, meaning that normally $\eta(Y|X) \neq \eta(X|Y)$.

Compared to MI, the computation of CR does not require 2D-histogram calculation, which makes it more suitable for GPU streaming processors with very limited shared memory. Meanwhile, the computation complexity of CR is $O(n_x)$, better than $O(n_x n_y)$ for MI. Furthermore, CR can generate comparatively accurate result while showing better robustness at low resolutions [6] and less sensitive on subsampling [13]. These features are especially meaningful to a multi-resolution framework like FLIRT.

D. Histogram Calculation

The calculation of CR requires the values of N_i, $\sum_{\omega\in\Omega_i} Y(\omega)^2$ and $\sum_{\omega\in\Omega_i} Y(\omega)$ for each $Bin\ i$, which is a histogram calculation process shown in List 1.

```
1 void histogram(int *bins, float *image){
2   for(int idx=tid; idx<imageSize; idx++){
3     bin = calcBin(image[idx]);
4     val = calcVal(idx,image);
5     h[bin]++;
6     y1[bin]+=val;
7     y2[bin]+=val*val;
8   }
9 }
```

Listing 1: Histogram Calculation

For a single thread, histogram calculation is straightforward. As shown in List 1, the thread simply goes through all the voxels of an image, updating the target counters respectively. However, to run on a SIMD machine like GPU, several threads may attempt to update the same counter simultaneously, leading to inconsistent results. Therefore, atomic operations are utilized to sequentialize these access. This

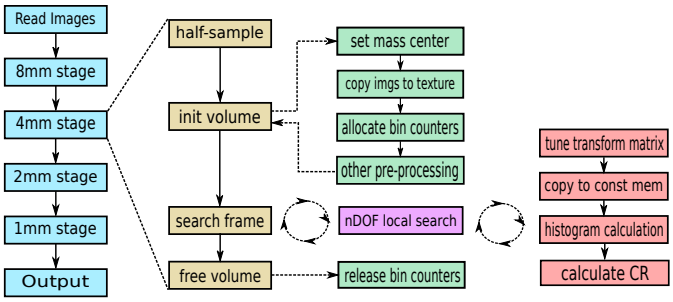


Fig. 1: Algorithm Framework. The vertical arrows indicate module execution sequences. Dashed horizontal arrows implies module hierarchy. The dashed circle means the righthand modules are called by the lefthand module repeatedly.

preserves correctness, but drastically enlarges the operation latency and usually aggravates to be the main bottleneck of the application. The *notion of conflicts* is to describe the scenario that multiple threads updating the same memory location.

Histogram conflicts are generally tackled by replication [14], which is the method of allocating local copies of histogram counters so as to reduce the number of transactions to shared resources. We name these local copies as *shadow histogram* in this paper.

III. Algorithm

The skeleton of our algorithm is illustrated in Figure 1. As can be seen, for every FLIRT stage, a half-sampling procedure is required to generate the images of that resolution. If the source image and reference image are not initially in the same resolution, an additional normalization is performed beforehand. After that, some preparation work are done in the "init volume" phase, mainly the allocation and configuration for GPU. For example, copying images to texture memory and allocating histogram counters. Additional preprocessing follows if necessary. For each local search, depending on search logic, the transform matrix is tuned before transferring to the GPU constant memory. Then histogram calculation is executed and CR is computed.

From the definition of CR, it seems that both $\eta(Y|X)$ and $\eta(X|Y)$ can be selected as the measure. In our schemes, we choose $\eta(source\_image|reference\_image)$ because of two reasons:

- Normally, the size of reference image is smaller than or equal to the size of source image. In such cases, choosing reference image as X can reduce the number of voxels that has to be processed during histogram calculation.
- The reference image is generally fixed, for example a template image. Thus by taking reference image as X, it is possible to reuse the preprocessing outcome from "init volume" module for new source images.

The calculation procedure is illustrated in Figure 2. The reference image is first converted into a "transformed reference image" after applying the transform matrix to all of its voxels. Then the similarity between source image and the "transformed reference image" are measured.

In the remaining part, we present the proposed schemes respectively.

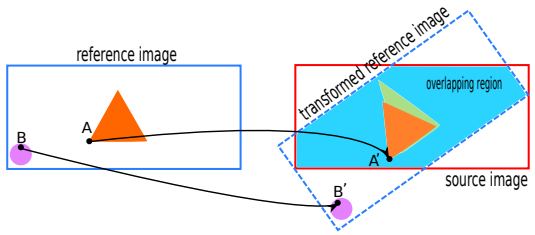


Fig. 2: Voxel Mapping. Voxel A(x, y, z) from the reference image is mapped to A'(x', y', z') in the "transformed reference image" by applying the transform matrix to (x, y, z). A and A' have identical intensity value but different coordinates. Then we measure the similarity between A' and the point from source image with the same coordinate (x', y', z'). Note that we only consider the voxels inside the overlapping region. Thus voxel B is neglected during this computation.

A. First Scheme: Atomic Updating on Global Memory

The code segment is shown in List 2. Three counter arrays $h, y1, y2$ are allocated on global memory, mapping to N, $\sum_{\omega\in\Omega} Y(\omega)^2$ and $\sum_{\omega\in\Omega} Y(\omega)$, respectively. Each counter targets a bin. During execution, from the current position in reference image, a thread executes the transformation indicated by the transform matrix and obtains the homologous coordinate in "transformed reference image". Using this coordinate, the voxel is fetched from source image. The thread then calculates the bin this voxel belongs to and updates the counters accordingly.

```
1  __global__ static void calcCRKernel(float *y1,
2      float *y2, float *h){
3    const int threads=blockDim.x*gridDim.x;
4    const int tid=blockIdx.x*blockDim.x+threadIdx.x;
5
6    for(int idx=tid; idx<refSize; idx+=threads){
7      float x = idx % refsize.x;
8      float y = (idx / refsize.x) % refsize.y;
9      float z = (idx / refsize.x) / refsize.y;
10     float3 srcCorr = transform(x,y,z);
11
12     if(insideRegion(srcCorr)){
13       int bin = floor(refImg[idx])/binWidth;
14       float weight = calWeight();
15       float srcVox = tex3D(srcImg,srcCorr);
16       atomicAdd( &h[bin],weight);
17       if(srcVox>0){
18         atomicAdd(&y1[bin],weight*srcVox);
19         atomicAdd(&y2[bin],weight*srcVox
20                                *srcVox);
21       }
22     }
23   }
24 }
```

Listing 2: Scheme_1 Kernel Code Segment

Note that the statement in line 17 improves the performance noticeably since for this scheme, atomic updates are the bottleneck and most of the intensities from the image background are zero. Meanwhile, the reference image voxels are only retrieved if the mapped coordinates fall in the overlapping region. This trick will not break the coalescing access pattern but may save L2 bandwidth if L1 cache is disabled. This is just our case because every thread maps to a unique voxel and there is little data reuse. It also applies to DRAM when all mapped coordinates are out of overlapping region for threads from the same warp. The weight function in line 14 is employed for smoothing which will be discussed later.

The first scheme is a direct implementation of the histogram calculation taking advantage of the hardware-based atomic primitives of GPU global memory.

B. Second Scheme: Per-thread Shadow Histogram on Global Memory

The bottleneck for the primary scheme is the conflicts of atomic updates. Intuitively, we can allocate extra counters to mitigate its degree. Imagine that if each thread is equiped with a private copy of shadow histogram, there will be no conflicts theoretically. This is our second scheme: every thread updates its local shadow counters which are aggregated afterwards. This scheme will consume huge space, therefore cannot fit into shared memory. Some existing proposals attempted to get around this pitfall by assigning less storage for each bin through bit-shifting operations [15], but such schemes strictly limit the magnitude of bin size and the bits utilized for a bin. For our scenario, however, each bin requires three float counters (12 bytes), hence the overall consumption is far overloaded for the on-device shared memory.

The shadow bins or counter arrays prevent the possible updating collisions. However, due to enormous global memory usage and excessive non-coalescing access, performance of the data cache and TLB are extremely low. Meanwhile, the final merging is mostly meaningless since many bins are zero. It also incurs considerable overhead.

C. Third Scheme: Blockwise Atomic Updating on Shared Memory

Shared memory is exploited in the third scheme in view of its fast speed for both native and atomic access. Involved from the first scheme, we allocate one shadow histogram for each thread block on shared memory and integrate them afterwards. Consequently, the possible conflicts are separated into intra-block conflicts and inter-block conflicts.

Intra-block conflicts occur among threads of the same block, which are resolved by shared memory atomic operations. Inter-block conflicts, on the other hand, take place between thread blocks on global memory. Two alternative methods can be deployed to resolve the inter-block conflicts. One is through global memory atomic operations, marked as scheme_3a. The other is through merging on global memory, marked as scheme_3b. We will compare these two schemes in the experiments.

D. Fourth Scheme: Warpwise Atomic Updating on Shared Memory

The fourth scheme also takes advantage of shared memory. However, in this scheme, we assume that collisions are still significant for threads inside a block. Thus, instead of allocating shadow counters per thread block, we assign one copy for each warp. In this way, the possible conflicts are divided into intra-warp conflicts and inter-warp conflicts. A final merging is still desired. Comparatively, this scheme further alleviates conflicts but requires more shared memory space and merging efforts.

E. Fifth Scheme: Conflict Free

Previous schemes explore design space from various aspects. However, none of them entirely resolve the conflicts problem despite trying different ways to mitigate the degree (actually there is no conflict in scheme_2, however at the expense of enrolling huge global memory access overhead). Motivated by this, we propose our last scheme.

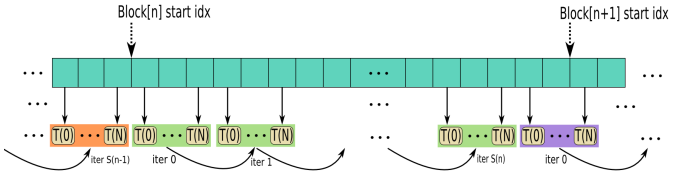


Fig. 3: Execution Procedure. Thread block n traverses from position marked by Block[n] start index. After several iterations, it terminates at the primary element owned by the next block. During this term, each thread handles one element and calculates the corresponding N_i, $Y(\omega_i)^2$ and $Y(\omega_i)$. After all the rotations are accomplished, the threads submit their local accumulated values to shared memory followed by a block-wise reduction.

1) Pre-Processing: In this scheme, an extra pre-sorting on the reference image is required, which takes place in the module of "other pre-processing" shown in Figure 1. A new array V_{index} is constructed with initial values as a sequential number series (0,1,2,...). Then V_{index} is sorted, using reference image intensities I_{ref} as the sorting keys. After that, I_{ref} is in order. For arbitrary element of I_{ref} with index i, we could obtain its original index $V_{index}(i)$, thus its primary coordinate. In this way, we gather the voxels with the same intensity together and distribute them continuously while still conserving the coordinate information.

The sorting process is performed on GPU part to reduce data transfer. Sorting algorithm on GPU are generally radix sort [16] or bitonic sort [17]. In this work we utilize the thrust library [18] for simplicity and high performance. After sorting, I_{ref} is then transferred back to the host part and a routine is employed to traverse the array and record the starting and ending positions for each thread block. It is possible that the block volume is equal to the preestablished bin number, which is the ideal scenario. However, the severe unbalancing of workload for different thread blocks prevents us from doing so. Based on our own experience, for a typical MRI image, due to the large background area, the primary bin can take up as much as 77% of all voxels, meaning that most of processors may be idle for quite a long time. Therefore, we set a threshold to limit the workload for each thread block.

It is important to note that, unlike other conflict-free schemes with pre-sorting [11], the preprocessing and sorting stage in our scheme is executed **only once** for each resolution stage provided that the bin size unchanged. Yet, the sorted data can be reused by the cost function for thousands of times within that stage. Further, if the reference image is invariant, such sorting results can even be reused for registration of other source images. This is a good property for use in the case of multiple image registration.

2) Cost Function: From the pre-processing phase, we know the starting and ending positions for every thread block. Inside the cost function kernel, a thread block will go over the partition of I_{ref} it is in charge of. As all the elements for the whole thread block belong to a unique bin, three registers are sufficient for the counters. They are then stored and merged on shared memory after a block-scope synchronization.

This procedure and code segment are illustrated in Figure 3. Compared to former schemes, the advantages of scheme_5 are:

- During the whole life, a thread only works on a unique bin. Therefore, it has an extremely low demand for storage. In fact, 12 bytes are sufficient, which can be easily fulfilled by the fastest memory – registers.
- All threads from a thread block target on the same bin. So they can be rapidly aggregated through a series of reduction operations on shared memory after accomplishing their own jobs. Besides, one thread only consumes 4 bytes of shared memory. Compared to scheme_3a, scheme_3b and scheme_4, it is more likely that a streaming processor can accommodate extra thread blocks when shared memory size is the limitation.
- All the conflicts are resolved through shadow counters and merging. However, unlike scheme_2, most of them are processed by fast register access and shared memory access.

IV. Experiments

In this section, we compare the proposed schemes under influence of two factors and evaluate the overall performance for the whole application. Several observations are made and discussed through comparisons.

A. Experiment Settings

System configuration for testing is listed in Table I. Compiler optimization level is "-O2". The L1 cache is disabled through compiler option "-ptxas -dlcm=cg". The interpolation method is the nearest point method for stage 1 and stage 2, but trilinear for the remaining stages. Texture addressing model is wrapping for all three spatial axes.

The experiment dataset is from OASIS database [19]. We use 11 MR1 images indexed from OA_0001 to OA_0012 except OA_0008 (due to unavailability) for registration. The test plan is that we register the last 10 images to the first one (listed in Table II). Note that, while in this experiment the source images and reference image are of the same size, generally they could have distinct dimension sizes or pixel dimension sizes.

The proposed CR-based schemes are also compared with several existing MI-based approaches (see Section 6): Shams' per-warp method [11], sort-and-count method [20], Chens' method [12], Vetters' method [9] and the native CPU implementation (denote as MI_Sham_1, MI_Sham_2, MI_Chen, MI_Vetter and MI_cpu, respectively). (In fact, Sham et al. also propose a per-thread scheme in [11]. However, this method is similar to scheme_2 and the acceleration technique is only feasible for small numbers of bins; it is therefore excluded from the results). We replace the software simulated atomic operations with their hardware based counterparts in these approaches since some of them are based on the older generation GPUs.

B. Cost Function Evaluation

First, we concentrate on the cost functions and evaluate two factors that may influence the performance:

TABLE I: System Configuration

CPU	Intel i7 870(x8) @2.93GHz
Memory	2 x 8G DDR3
Operating System	Linux 2.6.32 - CentOS 6
gcc	gcc 4.4.6
GPU	Nvidia Tesla C2075
Compute Capacity	2.0
CUDA Cores	14(SM) x 32
GPU Clock Rate	1147 MHz
Global Memory Throughput	144 GB/s
CUDA Driver/Runtime Verison	CUDA 5.0

TABLE II: Test Image Information

	Reference Image	**Source Image**
Name	OA_0001	OA_0002 to 12 (except 08)
dim 1-4	256x256x160x1	256x256x160x1
pixdim 1-4	(1,1,1,0) mm	(1,1,1,0) mm
datatype	4 Bytes	4 Bytes
filetype	ANALYZE-7.5	ANALYZE-7.5

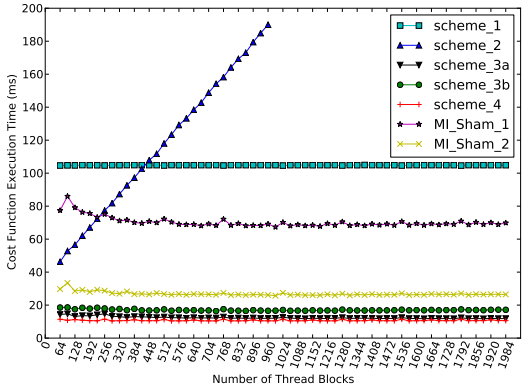


Fig. 4: Variation of execution time with the number of thread blocks. Only scheme_2 is strong correlated.

- Number of thread blocks. If less number of thread blocks are allocated, the remaining ones have to do more jobs. This parameter may make a difference on resource usage but will not actually affect conflicts.
- Number of histogram bins. This factor may greatly impact the volume of resources required and the degree of conflicts.

Figure 4 illustrates the variation of execution time with the number of thread blocks from 64 to 2048 for scheme_1 to scheme_4. The curves of scheme_5, MI_Chen and MI_Vetter are not listed because the number of thread blocks for these approaches depend on the number of bins. The test images are OA_02 and OA_01. The number of bins is 256.

For scheme_2, the curve terminating at 1024 is due to the huge global memory consumption since in that scheme, we allocate shadow counter arrays for every GPU thread. From the figure, we can learn that only scheme_2 has a strong correlation with the number of thread blocks. This is because for other CR-based schemes, the bottleneck is the conflicts instead of global memory access.

Figure 5 shows how execution time varies with the number of bins required. The number of thread blocks is 1024 for scheme_2 and the maximum (since the thread block size is 256

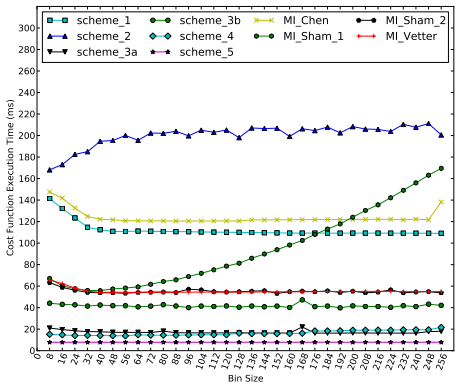


Fig. 5: Variation of execution time with the number of bins. Scheme_2 keeps increasing due to more access and merging operations on global memory. Scheme_1, scheme_3a, scheme_4 reduce at beginning due to alleviation of per-bin conflicts. Scheme_3b is not sensitive to bin number.

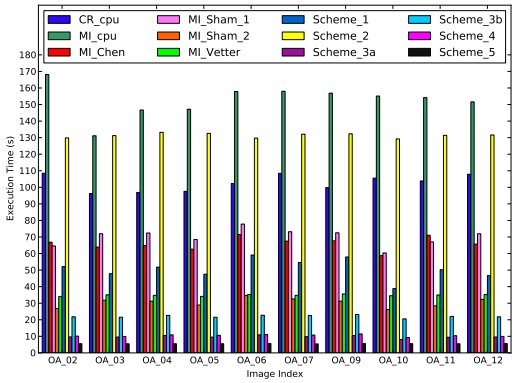


Fig. 6: Application Execution Time

and image size is 256x256x160, at maximum 256x160 thread blocks are allocated) for scheme_3a, scheme_3b, scheme_4, MI_Sham_1 and MI_Sham_2.

For scheme_1, the curve drops substantially at the beginning phase from 8 to 32. After that, additional bins do not help to alleviate the conflicts further. The curve for scheme_2 keeps increasing because scheme_2 avoids the possible conflicts at the expense of consuming excessive memory resources according to the number of bins. Scheme_3a, scheme_3b and scheme_4 seems insensitive to bin size, showing that the atomic operations on shared memory are well balanced by multi-threading. Meanwhile, the comparison between scheme_3a and scheme_3b indicates if the updating conflicts are not severe on global memory, the overhead of merging is often larger than that of atomic operations. For scheme_3a, after intra-block conflicts are resolved on shared memory, inter-block conflicts are much lighter and thus can be easily hidden by multithreading since thread blocks are independent and progres in their own contexts.

C. Application Evaluation

We then proceed to the experiments for the whole registration process. Figure 6 illustrates the measured application execution time. The number of thread blocks is set to be the maximum for those schemes in which this factor matters. For scheme_2, it is 14 for the sake of performance. The size

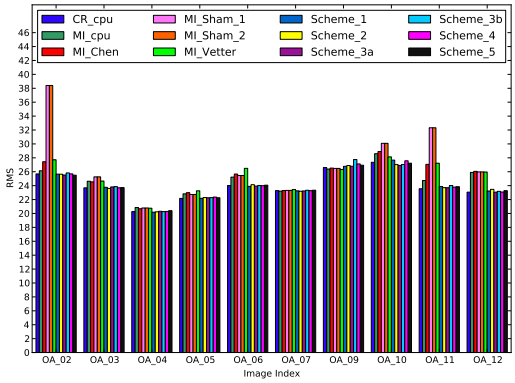


Fig. 7: RMS Error

of thread block is 256 which is one of the optimal values suggested by the CUDA programming guide [21]. The number of bins is varied depending on the stage, say 64, 128, 256, 256 for stage 1, 2, 3, 4, respectively.

As can be seen, in general the CR based methods run faster than the MI based ones except scheme_2, on both CPU and GPU. Particularly, scheme_5 achieves the best performance among all the approaches. The average speedup of the proposed five schemes over CR_cpu is listed in Table III.

Figure 7 shows the RMS error between output images and reference image. The measurement of error is not straightforward due to the lack of a standard baseline. So we simply calculate the intensity standard deviation (RMS) between the output images and the reference image. As can be seen, the CR based methods still behave better than the MI based methods, which highlights the great advantage of CR.

Meanwhile, except scheme_2, all the other proposed schemes reach a speedup of 2 compared with CR_cpu. In particular, the conflicts-free scheme achieves an average speedup of 18.55 over the baseline with less RMS. The low performance of scheme_2 suggests that the overhead of excessive global memory access is much higher than some atomic operations (referring to scheme_1). Meanwhile, comparing between scheme_1 and scheme_3, it is obvious that atomic operations on shared memory are more efficient. Finally, the throughput of scheme_3 is slightly higher than scheme_4 indicating that for this dataset, conflicts inside thread block are not so severe or maybe inter-warp conflicts are the main bottleneck [14].

V. Discussion

In this section, we analyze three topics about the conflict free scheme and their impact on performance.

A. Per-Thread-Block Workload Unbalancing

CUDA distributes thread blocks among streaming processors following a round robin fashion based on the assumption that the workload for each thread block is roughly identical. However, this is not the case for scheme_5 if each thread block accounts for an entire histogram bin. Table IV shows an example of one histogram calculation between OA_0002 and OA_0001 with 8 bins.

As can be seen, the proportion of voxels belonging to Bin 1 is about 77% where Bin 8 is less than 0.02%. This is

TABLE III: Average Speedup Over Native Version

Index	Scheme_1	Scheme_2	Scheme_3a	Scheme_3b	Scheme_4	Scheme_5
Average Speedup	2.027	0.782	10.530	4.661	9.832	18.548

TABLE IV: Histogram Result for 8 Bins

Bin	1	2	3	4	5	6	7	8
Voxels	8065900	1073218	742400	369655	113187	90263	29056	2081
Percentage	76.92%	10.24%	7.08%	3.53%	1.08%	0.86%	0.28%	0.02%

extremely unbalanced. So if 8 thread blocks are allocated, the 8th streaming processor will be idle for more than 99.97% of the total execution time. Meanwhile, if a heavy-loaded thread block is dispatched at the last period, the processor utilization will be even lower since all the remaining streaming processors are idle. So there must be a threshold to limit the maximum workload for a thread block. Though some overhead may be introduced, it is well worth as the total execution time drops by a factor of over 3 in our observation. The code segment for this post-sorting procedure is presented in List 3.

```
for(int i=0; i<refSize; i++){
  if(i-prel >= MAX_BLK_WORKLOAD
      ||(val=iref[i]/binWidth)!=preVal){
    blkpos[blkId]=i;
    blk2bin[blkId]=val;
    preVal=val;
    prel=i;
    blkId++;
  }
}
```

Listing 3: Scheme_5 Post-sorting Process

The question arises what the optimal threshold is. Since more thread blocks introduce extra overhead (initialization, merging, dispatch etc.), this value is in fact a compromise between workload balance and performance. Figure 8(a) illustrates the variation of execution time with the workload for each thread block. From this figure, 768 appears to be the optimal which means at maximum one thread processes three voxels.

B. Overhead of Presorting

The sorting process is already presented. Here we evaluate its overhead. Since the stages have distinct resolutions hence different image sizes, we list the time expense and throughput of sorting for each stage in Table V. It can be seen that the sorting kernel tends to achieve higher throughput for larger dataset.

TABLE V: Sorting Overhead for Stages

Stage	Vox Size	Sorting Time (μs)	Throughput (Vox/μs)
1	20480	708	28.9
2	163840	1650	99.3
3	1310720	4115	318.5
4	10485760	26907	389.7

The phase of sorting is only beneficial if its outcome can be reused sufficiently. From this perspective, the more searches inside a stage, the more benefit we can expect from this process. For other schemes, the scale of reference image dictates the capacity of workload, thus the degree of conflicts. Therefore, a larger volume reference image will benefit more from the sorting, further highlighting the advantage of scheme_5.

In fact, scheme_5 does not need a complete pre-sorting but a process to aggregate voxels with the same intensity values together. This is also reflected in List 3. Further, if a series of registration jobs share the same reference image, the sorting results can be recycled among them. Note we do not apply this optimization in the experiments of this paper.

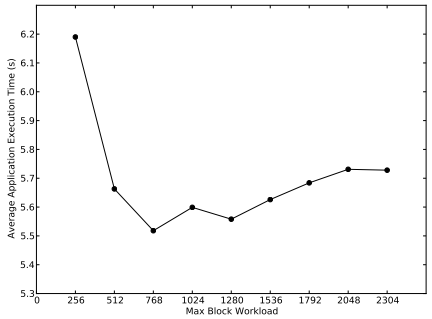
C. Smoothing of the Border Region

The FLIRT algorithm deploys a weighting method to smoothen the cost function at the marginal region [3]. This improvement is also implemented in the proposed schemes (The function named calWeight in List 2). In the previous experiments, we disabled this function by simply returning 1. Here we evaluate its performance impact. The results are presented in Figure 8(b) and Figure 8(c).

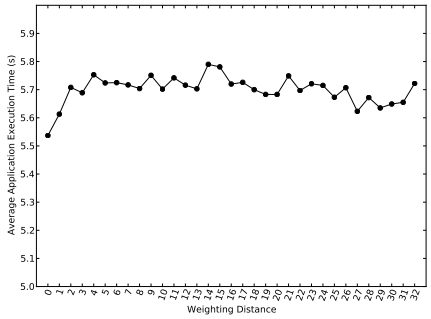
As weighting distance increases, more points will fall into the boundary region hence more extra process and computation are needed. However, from the figures, we can observe that the execution time is almost unchanged, showing that these additional computations can be fully hidden through multi-threading and the long latency of texture fetching. Meanwhile, the RMS curve exhibits a decreasing trend in general but suffers some large deviation for big distances. This suggests that the weighting function may incur some bias for the cost function, especially when the overlapping region is small. Maybe developing a new weighting function that takes intensity values into consideration can address this problem but is out of the scope for this paper. Here, the value of 26 appears to be the optimal weighting distance.

VI. Related Work

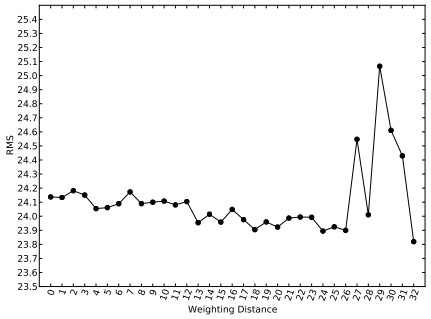
Ever since CUDA has been published, several works are proposed about realizing the image registration algorithms on GPUs [10], [11], [12], [9], [22]. However, the interleaved and concurrent writing access to a limited memory region by massive threads make this migration a difficult task because atomic operations are employed to preserve updating correctness meanwhile introducing serious overheads, especially when numerous threads are competing for the same bin location. Shams et al.[10], [11] maintain a number of shadow histograms in global memory (when shared memory is out of range) and aggregate afterwards to alleviate the degree of conflicts, or keep partial number of shadow histograms in shared memory but traverse the image several times to cover the entire bin range. Chen et al. [12] sort the reference volume beforehand to restrict the shadow histograms' range when counting the joint entropy. Vetter et al. [9] also perform a pre-sorting to narrow the space required to fit into shared memory and guarantee coalescing access. They further mitigate the collisions by allocating more counters for the "fat



(a) Variation of average execution time over the maximum per-block workload. The curve drops drastically at the beginning and increases thereafter indicating that after the flex point, the impact of load unbalancing appears to be prominent.



(b) Variation of average execution time over weighting distance. The execution time seems irrelevant to the size of weighting distance indicating that these extra computation overhead are completely hidden by multithreading and the long latency texture fetching.



(c) Variation of average RMS error over weighting distance. The curve decreases globally but may suffer significant bias for large weighting distance.

Fig. 8: Additional Experiment Results on Scheme_5.

bin" that appears to incur more conflicts based on an intensity distribution figure generated from a pre-profiling. All of these methods, however, are mutual information based and conflicts unavoidable. Though Shams et al. proposed a delegate sort-and-count algorithm to achieve atomic operations free in [20], this sort-and-count phase has to be performed by each thread block every time the cost function is invoked, thus leads to additional performance overhead.

VII. Conclusion

In this paper, we present five Correlation Ratio based image registration schemes dedicated for GPUs. Through comparisons between different schemes, we show that 1) Atomic operations on shared memory are much faster than on global memory as expected 2) The overhead of massive access is larger than that of atomic operations on global memory 3) If collisions are relatively small, atomic operations are a better choice than merging. Finally, we propose a scheme that totally avoids conflicts and achieves more than 18 times' speedup compared to native implementation with lower error. This design is based on the algorithmic characteristics of CR which exhibits its great advantage on GPUs when compared with the MI based approaches. Finally we evaluate the impact of balancing, sorting and smoothing on performance and accuracy, as well as provide some optimization suggestions.

References

[1] B. Zitova and J. Flusser, "Image registration methods: a survey," *Image and vision computing*, vol. 21, no. 11, pp. 977–1000, 2003.

[2] J. Maintz and M. A. Viergever, "A survey of medical image registration," *Medical image analysis*, vol. 2, no. 1, pp. 1–36, 1998.

[3] M. Jenkinson, P. Bannister, M. Brady, and S. Smith, "Improved optimization for the robust and accurate linear registration and motion correction of brain images," *Neuroimage*, vol. 17, no. 2, pp. 825–841, 2002.

[4] M. Jenkinson and S. Smith, "A global optimisation method for robust affine registration of brain images," *Medical image analysis*, vol. 5, no. 2, pp. 143–156, 2001.

[5] A. Klein, J. Andersson, B. A. Ardekani, J. Ashburner, B. Avants, M.-C. Chiang, G. E. Christensen, D. L. Collins, J. Gee, P. Hellier *et al.*, "Evaluation of 14 nonlinear deformation algorithms applied to human brain MRI registration," *Neuroimage*, vol. 46, no. 3, pp. 786–802, 2009.

[6] A. Roche, G. Malandain, X. Pennec, and N. Ayache, "The correlation ratio as a new similarity measure for multimodal image registration," in *Medical Image Computing and Computer-Assisted InterventationMICCAI*. Springer, 1998, pp. 1115–1124.

[7] J. P. Pluim, J. A. Maintz, and M. A. Viergever, "Mutual-information-based registration of medical images: a survey," *Medical Imaging, IEEE Transactions on*, vol. 22, no. 8, pp. 986–1004, 2003.

[8] R. Shams, P. Sadeghi, R. Kennedy, and R. Hartley, "A survey of medical image registration on multicore and the GPU," *Signal Processing Magazine, IEEE*, vol. 27, no. 2, pp. 50–60, 2010.

[9] C. Vetter and R. Westermann, "Optimized GPU histograms for multimodal registration," in *Biomedical Imaging: From Nano to Macro, IEEE International Symposium on*. IEEE, 2011, pp. 1227–1230.

[10] R. Shams and N. Barnes, "Speeding up mutual information computation using NVIDIA CUDA hardware," in *Digital Image Computing Techniques and Applications, 9th Biennial Conference of the Australian Pattern Recognition Society on*. IEEE, 2007, pp. 555–560.

[11] R. Shams and R. Kennedy, "Efficient histogram algorithms for NVIDIA CUDA compatible devices," in *Proc. Int. Conf. on Signal Processing and Communications Systems (ICSPCS)*, 2007, pp. 418–422.

[12] S. Chen, J. Qin, Y. Xie, W.-M. Pang, and P.-A. Heng, "CUDA-based acceleration and algorithm refinement for volume image registration," in *BioMedical Information Engineering. FBIE. International Conference on Future*. IEEE, 2009, pp. 544–547.

[13] A. Roche, G. Malandain, N. Ayache, X. Pennec *et al.*, "Multimodal image registration by maximization of the correlation ratio," 1998.

[14] J. Gomez-Luna, J. M. Gonzalez-Linares, J. I. B. Benitez, and N. G. Mata, "Performance Modeling of Atomic Additions on GPU Scratchpad Memory," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 11, pp. 2273–2282, 2013.

[15] V. Podlozhnyuk, "Histogram calculation in CUDA," *NVIDIA Corporation, White Paper*, 2007.

[16] N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore GPUs," in *Parallel & Distributed Processing. IPDPS. IEEE International Symposium on*. IEEE, 2009, pp. 1–10.

[17] H. Peters, O. Schulz-Hildebrandt, and N. Luttenberger, "Fast in-place sorting with cuda based on bitonic sort," in *Parallel Processing and Applied Mathematics*. Springer, 2010, pp. 403–410.

[18] N. Bell and J. Hoberock, "Thrust: A 2 6," *GPU Computing Gems Jade Edition*, p. 359, 2011.

[19] D. S. Marcus, T. H. Wang, J. Parker, J. G. Csernansky, J. C. Morris, and R. L. Buckner, "Open Access Series of Imaging Studies (OASIS): cross-sectional MRI data in young, middle aged, nondemented, and demented older adults," *Journal of Cognitive Neuroscience*, vol. 19, no. 9, pp. 1498–1507, 2007.

[20] R. Shams, P. Sadeghi, R. Kennedy, and 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*, vol. 99, no. 2, pp. 133–146, 2010.

[21] C. Nvidia, "Programming guide," 2008.

[22] P. Muyan-Ozcelik, J. D. Owens, J. Xia, and S. S. Samant, "Fast deformable registration on the GPU: A CUDA implementation of demons," in *Computational Sciences and Its Applications, ICCSA'08. International Conference on*. IEEE, 2008, pp. 223–233.