ABSTRACT

Title of Document: PARALLELIZATION OF NON-RIGID IMAGE REGISTRATION

Mathew Philip, Master of Science, 2008

Directed By: Professor Raj Shekhar,
Dept of Diagnostic Radiology (University of Maryland, Baltimore) and Dept of Electrical and Computer Engineering

Non-rigid image registration finds use in a wide range of medical applications ranging from diagnostics to minimally invasive image-guided interventions. Automatic non-rigid image registration algorithms are computationally intensive in that they can take hours to register two images. Although hierarchical volume subdivision-based algorithms are inherently faster than other non-rigid registration algorithms, they can still take a long time to register two images. We show a parallel implementation of one such previously reported and well tested algorithm on a cluster of thirty two processors which reduces the registration time from hours to a few minutes.

Mutual information (MI) is one of the most commonly used image similarity measures used in medical image registration and also in the mentioned algorithm. In addition to parallel implementation, we propose a new concept based on bit-slicing to

accelerate computation of MI on the cluster and, more generally, on any parallel computing platform such as the Graphics processor units (GPUs). GPUs are becoming increasingly common for general purpose computing in the area of medical imaging as they can execute algorithms faster by leveraging the parallel processing power they offer. However, the standard implementation of MI does not map well to the GPU architecture, leading earlier investigators to compute only an inexact version of MI on the GPU to achieve speedup. The bit-slicing technique we have proposed enables us to demonstrate an exact implementation of MI on the GPU without adversely affecting the speedup.

PARALLELIZATION OF NON-RIGID IMAGE REGISTRATION

By

Mathew Philip

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park, in partial fulfillment
of the requirements for the degree of
Master of Science
2008

Advisory Committee:
Professor Raj Shekhar, Chair
Professor Shuvra S. Bhattacharyya
Professor Peter Petrov

© Copyright by
Mathew Philip
2008

Acknowledgements

First and foremost, I offer my utmost gratitude to my advisor, Dr. Raj Shekhar without whose supervision, patient direction and technical expertise this thesis would not have been possible., Our many insightful discussions and his constructive feedback at various stages, have been instrumental in shaping this work on to completion. I would also like to thank Dr. Shuvra Bhattacharyya and Dr. Peter Petrov for serving on my thesis committee amidst their busy schedules.

I am also grateful to my colleagues at the Imaging Technologies Laboratory, Baltimore who created a stimulating and cooperative working atmosphere. In particular, I greatly appreciate Dr. William Plishker's invaluable suggestions and assistance on various aspects of my research.

Finally, I am forever indebted to my wife, Pam for her support and encouragement at all times. I also am grateful to my parents, Philip and Mercy who have always guided me and directed my steps. On a different note, many people have been part of my graduate education and I take this opportunity, to thank them all.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction and Motivation

Image Registration is the process of geometrically aligning two images. One of the images known as the reference image is kept unchanged, while the second image, known as the floating image, is allowed to deform to match the reference image.

The floating image is then re-sampled on the grid of the reference image, which allows us to overlay or subtract the two, depending on the specific application needs. This is extremely useful in many areas in the medical field. For example, the field of diagnostic medicine benefits from a new type of image created from complementary information of images acquired using different technologies (modalities). Another application area is the field of image-guided interventions, in which pre-operative images often need to be registered with intra-operative images. Image registration can be classified either as rigid or non-rigid. In rigid registration, the floating image is only allowed to translate and rotate while in non-rigid registration we allow the floating image to deform in a more complex manner.

Many applications need non-rigid registration as rigid registration is insufficient to recover misalignments between two images. However non-rigid registration is computationally very intensive and can take many hours to register two images. One technique for non-rigid registration is the FFD (free form deformation) based approach, which takes approximately 12 hours to align two CT images of the liver having a size of 512 x 512 x 295 on a single 1-GHz Pentium III system [7]. Such a long execution time prevents the use of such an algorithm in many practical

applications. Parallel processing power offered by cluster computing has been used previously to address this issue [7]. In the paper by Ino et al. [7], a cluster of 128 processors is used to parallelize FFD-based registration to reduce the computation time to ten minutes. However, non-rigid registration based on hierarchical volume subdivision is inherently faster than other non-rigid registration algorithms [16]. By parallelizing this type of non-rigid registration algorithm, we are able to perform accurate registration with far less hardware and also have the ability to scale up the hardware to reduce processing time even further.

Contributions of this Thesis

We use cluster computing to parallelize a non-rigid registration algorithm based on hierarchical volume subdivision which has previously been reported by Walime and Shekhar[4]. We are able to reduce processing time from 2 hours for registering CT images of size 256 x 256 x 256 to an acceptable 8 min on a cluster of 32 processors. Mutual Information (MI) is a widely used similarity metric in image registration and is also used in the above mentioned algorithm. Mutual information based registration is an iterative process which tries to maximize the similarity metric (MI). To compute mutual information the probability densities of the intensities of the two images need to be estimated. There are two main techniques for estimating the probability densities for computing mutual information The parzen window approach is computationally more demanding than the mutual histogram approach. The mutual histogram for estimating probabilities for image registration has been shown to be accurate [1]. The mutual histogram approach is also used in the volume subdivision based non-rigid registration algorithm.

We also demonstrate a new technique based on bit-slicing for computing mutual information from the mutual histogram on a parallel computing platform. In contrast to the most common way of constructing the mutual histogram by assigning portions of the reference image to different compute cores on a parallel platform, we divide the reference image based on its histogram. In this approach the portion of the image assigned to a compute core on the parallel platform depends on the area of the histogram assigned to it. We exhibit how this technique further reduces processing time on the cluster and also how this algorithm maps on to a graphics processor unit (GPU). GPUs are becoming increasingly widespread for use in general purpose computing as they are able to accelerate algorithms with the parallel computing power they offer. While registration algorithms have been implemented on GPUs with promising results, exact and efficient MI computation from mutual histogram has not been possible. Previous investigators have not been able to perform a full MI implementation without sacrificing speedup [11] as they were not able to utilize the limited shared memory size of the GPU. By using the bit-slicing technique we show how we are able to work around the shared memory size and also parallelize the computations involved in computing MI without an additional communication overhead.

Outline of Thesis

This thesis is structured in the following manner. Chapter 2 gives a brief background on image registration and the different techniques commonly used in image

registration. In the same chapter we introduce mutual information which is a widely used similarity metric for image registration. We also go on to present the hierarchical subvolume based non rigid registration algorithm which we have used in the work we present in this thesis.

Chapter 3 covers the cluster implementation of the hierarchical subvolume based non rigid registration algorithm. We also present results for this implementation.

Chapter 4 introduces the bit-slicing algorithm for computing mutual information on a parallel platform. We present details of the implementation of this algorithm on a cluster followed by the associated results.

Chapter 5 describes how the bit-slicing algorithm allows us to map computation of mutual information on to a GPU. Results for this implementation are also presented in this chapter.

Chapter 6 is a discussion of the work we present in this thesis. We also cite areas for future work in this field.

Chapter 2: Background on Image Registration

Image Registration

There are several different techniques for image registration [1]. Landmark-based methods use corresponding landmarks in the two images to align them. Surface-based techniques use delineation of corresponding surfaces in both images. Landmark-based techniques are labor intensive if the landmarks are identified manually and also dependent on how accurately the landmarks are identified. If done automatically, the accuracy of registration will depend on how well corresponding landmarks are identified. Surface-based methods will also depend on how well the surfaces are extracted and this will be highly data dependent. The technique we use comes under voxel-based registration methods, which optimize a function measuring the similarity of geometrically corresponding voxels pairs. This technique is not influenced by segmentation errors. Algorithms which have been proposed for voxel-based registration methods have used metrics like absolute difference between image intensities of regions of interest. However, this is not suitable for multimodality applications as absolute difference assumes that there is a linear dependence between image intensities of the two images which may not be true. Other algorithms in this genre have proposed cross correlation of intensities in corresponding regions. This also assumes some form of linear relationship between intensities of the two images.

Mutual Information-based approach was first suggested by Collignon et al. [15] and Wells, et al. [2]. The advantage with this approach is that no assumptions are made regarding the dependence of intensities of the two images. Mutual Information based

registration has been shown to align images accurately and robustly [18]. Mutual information is a statistical concept used to measure the dependence between two random variables.

$$I(X,Y) = H(X) - H(X/Y) \quad (1)$$

$$= H(Y) - H(Y/X) \quad (2)$$

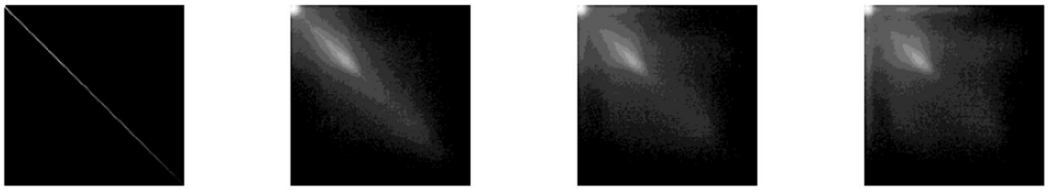
$$= H(X) + H(Y) - H(X,Y) \quad (3)$$

Intuitively, entropy H(X) is regarded as a measure of uncertainty which is also a measure of the amount of information of random variable X. H(X|Y) is a measure of the amount of uncertainty remaining in X after Y is known. The right side of equation 1 can be read as "the amount of uncertainty in X, minus the amount of uncertainty in X which remains after Y is known," which is equivalent to "the amount of uncertainty in X which is removed by knowing Y." This corroborates the intuitive meaning of mutual information as the amount of information (that is, reduction in uncertainty) that knowing either variable provides about each other.

Image registration uses mutual information to measure the statistical dependence of intensities of corresponding voxels in the floating and reference images. We consider intensities from the reference and floating images as random variables. The registration process searches through different candidate transforms to find the best transform. At each candidate transform, the MI value is computed by looking at intensities in corresponding locations to find joint probability density of intensities and also the probability density of the intensity of each image. As discussed the mutual information value indicates the reduction in uncertainty in the distribution of

one random variable (intensity of reference image in this case) by knowledge of the intensity distribution of the floating image. Thus the candidate transform which reduces the uncertainty the most or, in other words, that maximizes mutual information is the best transform as that transform indicates when the two random variables are the most dependent. In order to compute probability densities required for mutual information calculation, Wells et al. [2] used the Parzen window approach. The probability density and joint probability density of intensities of the two images is estimated for each candidate transformation using Parzen windows. However this approach is computationally intensive. Another method that has been used widely is the mutual histogram method of estimating probabilities. This approach has been shown to give accurate registration results by Maes et al. [1].

The mutual histogram is computed by looking at corresponding voxels in the two images to be registered. The intensities of the voxels in corresponding locations are used to index a 2D array and increment the value at that location by 1. This is repeated for all corresponding voxels in the two images. The intensity of one of the images is used to index one axis of the mutual histogram while intensity of the corresponding voxel is used to index the other axis of the mutual histogram. Fig 2.1 shows the mutual histogram for registering a magnetic resonance (MR) image with itself. When the image is aligned with itself then corresponding voxels have the same intensity and the mutual histogram is always incremented along a line. This is illustrated in the first panel of Fig. 2.1. The joint histogram of two images disperses with mis-registration.



0 Degrees 2 Degrees 5 Degrees 10 Degrees

Fig 2.1 Mutual Histogram at different angles of rotation when registering an MR image with itself (Courtesy [3])

When the images are registered then there is clustering in the mutual histogram. Joint entropy is a measure of the uncertainty or dispersion in the mutual histogram. Thus when the images are registered there is more clustering and thus joint entropy is minimal. During the registration process, the joint histogram is created from the overlapping sections of the two images. If we use only the joint entropy as the metric during registration then the mutual histogram could show maximal clustering when only the background of the two images overlap. This will result in a very small value for joint entropy which will give wrong results during registration. However, mutual information incorporates entropy of the two images in the metric and when the images have just the background in them, the individual entropies are low and hence maximization of MI is a more accurate metric to use in registration.

However mutual information can actually increase with increasing misregistration. This can occur when the relative areas of object and background even out and the sum of the marginal entropies increases, faster than the joint entropy. We use a

variant of mutual information, called normalized mutual information (NMI), for registration which has been shown to be even more overlap invariant than MI [21]. NMI can be calculated using the following formula.

$$NMI(A,B) = \frac{H(A) + H(B)}{H(A,B)}$$

Rigid Registration

Rigid registration allows only rotations and translations of the floating image during the registration process. Fig 2.2 illustrates the process of rigid registration. The registration process is iterative. Different candidate transforms are applied to reference image. For each candidate transform the metric (mutual information) is evaluated. The iterative process stops when mutual information function reaches maximum. The optimizer block in Fig 2.2 ensures that the search space of the applied transform is stepped through in an optimal fashion. From Fig 2.2 we see that the transform is applied on the reference image. This is to avoid the problem of holes when re-sampling the floating image on the reference image. Once the transform that maximizes MI is computed, the floating image has to be resampled on to the reference grid. Since we know the transformation that takes us from reference image to floating image the intensity at the matching location in floating image space can be determined for each grid location of the reference image. This can be used to resample the floating image on to the reference image grid. If the transformed point in the floating image does not lie on a grid position the algorithm uses trilinear interpolation involving intensities of surrounding voxels. If the transformation had been computed for the floating image, then when we re-sample onto the reference

grid, there could be possible grid locations that we miss resulting in holes in the re-sampled image.

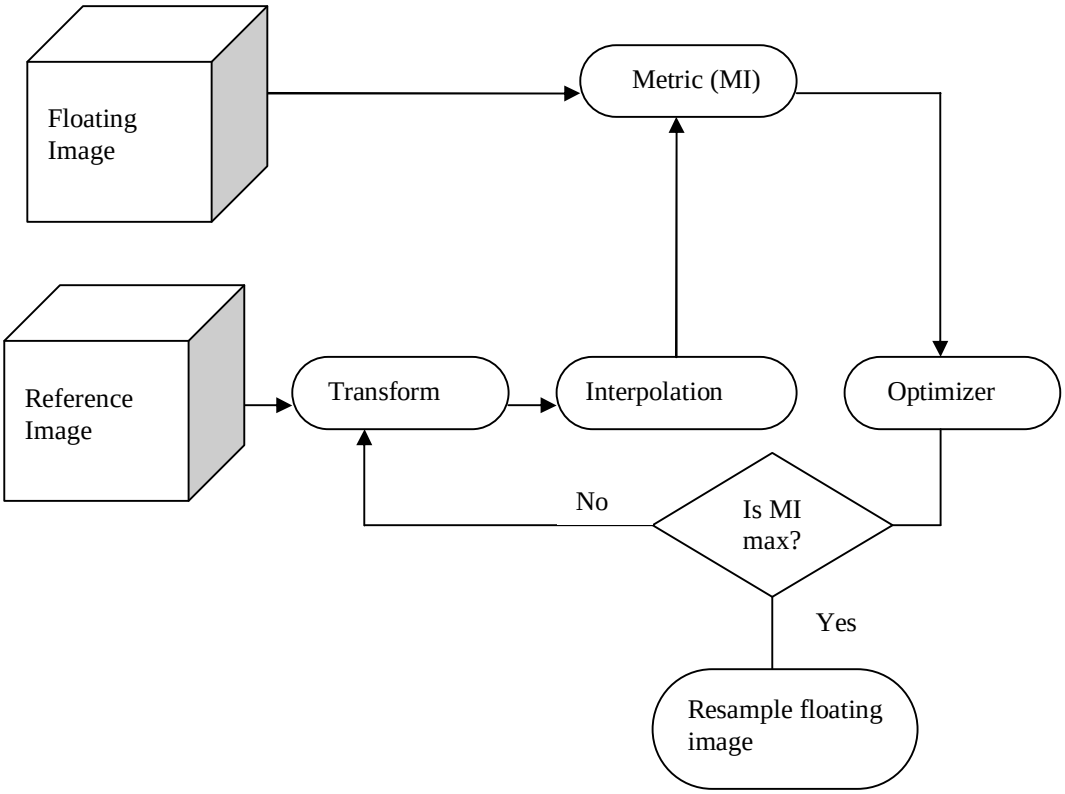


Fig 2.2 A flow diagram showing the process of rigid registration.

Fig 2.2 also indicates an interpolation stage before mutual information is calculated. This is because mutual information is calculated from the mutual histogram which is computed by looking at corresponding voxel locations for each candidate transform as has been described earlier. A transform applied on the reference image may result in a corresponding location in the floating image which is not on a grid position, warranting some form of interpolation. We use a special form of interpolation, called PV interpolation, for computing the mutual histogram as it has been shown [1] to give a smooth registration function which is necessary during optimization to ensure

accurate registration. PV interpolation ensures sub-voxel accuracy. PV interpolation is performed as follows: We update the mutual histogram in a weighted manner by indexing the mutual histogram with joint intensities obtained by the reference image voxel lying in between the grid location of the floating image and the voxels at the nearest eight corners of the floating image.

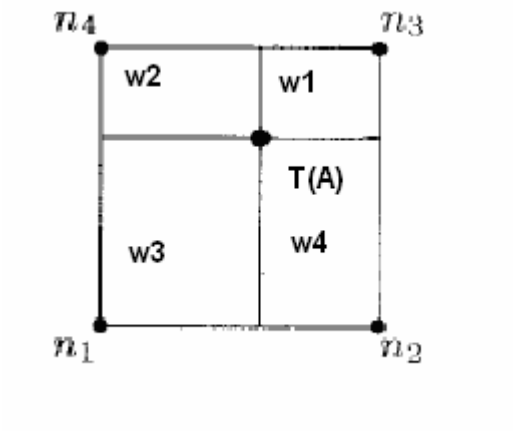


Fig 2.3 PV interpolation on a 2D image (Courtesy [1])

Once the mutual histogram is computed for each candidate transform the MI value is calculated from the joint entropy and the entropy of each image. The mutual histogram is used to compute the entropy from the probability density of the intensities of the two images. The joint probability density is calculated by normalizing each value in the mutual histogram. This is illustrated in Fig 2.4.
Fig 2.4 also illustrates that the probability density of image A (reference image) is calculated by summing the values in the mutual histogram along the rows followed by normalization. Similarly the probability density of image B is computed by summing the values in the mutual histogram by summing along the columns followed

by normalization. Once the probability densities are computed the entropies are calculated from equations 5, 6, 7. Finally MI can be computed from equation 4.

$$MI = H(A) + H(B) - H(A,B) \tag{4}$$

$$H(A,B) = -\sum_{i=0}^{255} \sum_{j=0}^{j=255} p(i,j) \log(p(i,j)) \tag{5}$$

$$H(A) = -\sum_{i=0}^{255} p_A(i) \log p_A(i) \tag{6}$$

$$H(B) = -\sum_{j=0}^{255} p_B(j) \log p_B(j) \tag{7}$$

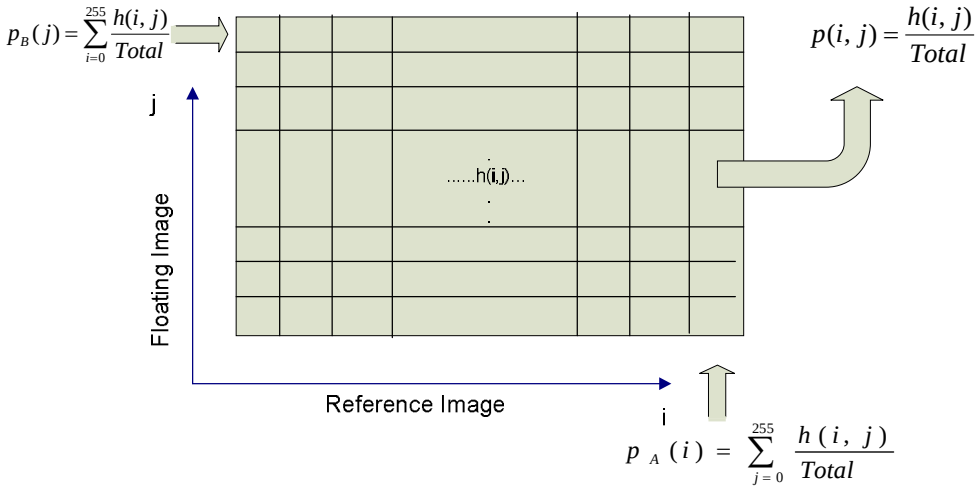


Fig 2.5 An illustration of how to compute probability density form the mutual histogram

Non-rigid Registration

Non-rigid registration allows for the floating image to deform in a complex manner. While rigid registration limited the deformation to rotations and translations, non-rigid registration allows many more degrees of freedom. One popular way of

modeling non-rigid registration is by the using free form deformation (FFD) [5]. In this approach the image is allowed to deform by manipulating a mesh of control points. The degree of non-rigid deformation that can be modeled is dependent on the resolution of this mesh. This form of non-rigid registration is also an iterative process. The mesh of control points are allowed to deform and the resulting deformed image is compared with the reference image to evaluate the degree of similarity between the two. To recover finer misalignments the mesh of control points need to be at a high resolution. However, the computational demand of the algorithm increases as there are more control points to optimize to compare the two images.

Another class of non-rigid registration algorithms models the misalignment between the two images by multiple local rigid body registrations. These algorithms generally subdivide the image into smaller subvolumes and perform independent rigid registration on each subvolume. To recover finer misalignments the registered subvolumes are divided further and registered. A smoothly varying transformation field is generated from the independent subvolume registration results by interpolation to create the deformed image. Image subdivision based algorithms for non-rigid registration are inherently fasters than other non-rigid registration algorithms [5]. In this thesis we worked on the non-rigid registration algorithm of this class proposed by Walimbe and Shekhar[4]. We describe this algorithm in the next section.

Hierarchical Volume Subdivision based Non Rigid Registration

The algorithm as proposed in [4] allows six degrees of freedom (three translations and three rotations) for the rigid registration of each subvolume. Volume subdivision based algorithms previously allowed only translations for the subvolume registrations because interpolating the registration results which allowed for rotations was difficult. This algorithm is able to interpolate rotations by using quaternion interpolation. Thus this algorithm is able to model complex misalignments at larger subvolumes. The previously adopted translation only approach had to register smaller subvolumes to recover the same amount of misalignment and this also is prone to more error from interpolation of many small subvolumes. The ability to recover complex misalignments without having to divide the image into very small subvolumes poses the advantage of less execution time.

This algorithm proceeds by first performing a rigid registration as illustrated by the second cube in Fig 2.6. The rigid registration mechanism proceeds as described in the section on rigid registration. Once the global misalignment has been recovered, local misalignment is recovered by dividing the image into 8 sub-volumes and independently performing local rigid registration on each subvolume. This is illustrated by the 3^{rd} and 4^{th} cubes in Fig 2.6. To further recover misalignments, each subvolume is divided into 8 smaller subvolumes. Thus at the second level of registration there are 64 subvolumes to register. The process of dividing the subvolume carries on till a predefined subvolume size.

However, for small subvolumes there are only a few voxels for creating the mutual histogram. This can lead to artifacts in the registration function which may result in registration errors [22]. The mentioned algorithm addresses this problem by compiling the mutual histogram as the sum of two mutual histograms $MH_{subvoluime}$ and $MH_{rest.}$ $MH_{subvoluime}$ is compiled by the voxels of subvolume being registered while MH_{rest} is compiled from the rest of the voxels. MH_{rest} does not vary as the subvolume transformation is being optimized. It is therefore only a one time overhead for each subvolume. As the subvolume transformation is being optimized $MH_{subvoluim}$ varies and hence the summed mutual histogram is proportionally influenced. This reflects in the NMI and the problem of having a sparse mutual histogram is avoided.

Once all the subvolumes have been registered, a smooth deformation field is created by interpolating the registration results of each subvolume. The deformation field is used to resample the floating image on the reference image grid. The translations can be linearly interpolated to generate the translation component of each voxel in the deformation field. The three rotations are interpolated by quaternion based interpolation.

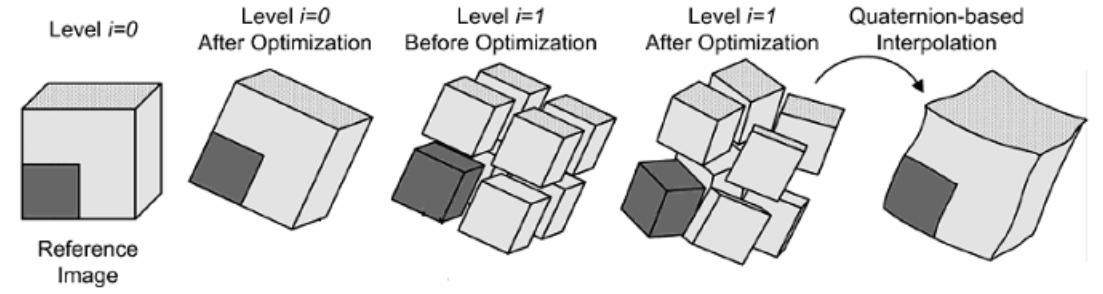


Fig 2.5 Hierarchical subvolume division based non rigid registration (Courtesy [6])

Chapter 3: Parallelization of Hierarchical Volume Subdivision based Registration

Introduction

We parallelized the hierarchical volume subdivision based registration algorithm [4] to run on a cluster of eight nodes each with four cores, effectively giving a total of 32 processors for processing. Though this algorithm is inherently faster than other non-rigid registration techniques, a single processor implementation can take several hours for processing. The parallel implementation we present reduces the processing time significantly. The technique used for parallelizing this registration algorithm is undertaken in two parts. First the rigid registration part is parallelized followed by parallelization of the non-rigid volume sub-division part. We present results for various configurations of the cluster and for four image sets.

Hardware Setup

We use an eight node cluster each with two dual-core Intel Xeon processors running at 2.33 GHz. Each of the nodes has 4GB RAM that is shared by the four cores. The nodes are connected together using a 1-Gbps Ethernet switch. APIs provided by MPI are used for communication between the individual cores [12]. MPI is a language independent communications protocol that is widely used to program parallel computers. We used the MPICH2 implementation of MPI which is developed by Argonne National Laboratory (Argonne, IL). The nodes run Linux. During run time the MPI environment assigns a single process to each core. With the above mentioned cluster setup, MPI can abstract the number of cores and nodes to effectively give

thirty two processors at our disposal. Figure 3.1 shows the cluster setup used in this research.

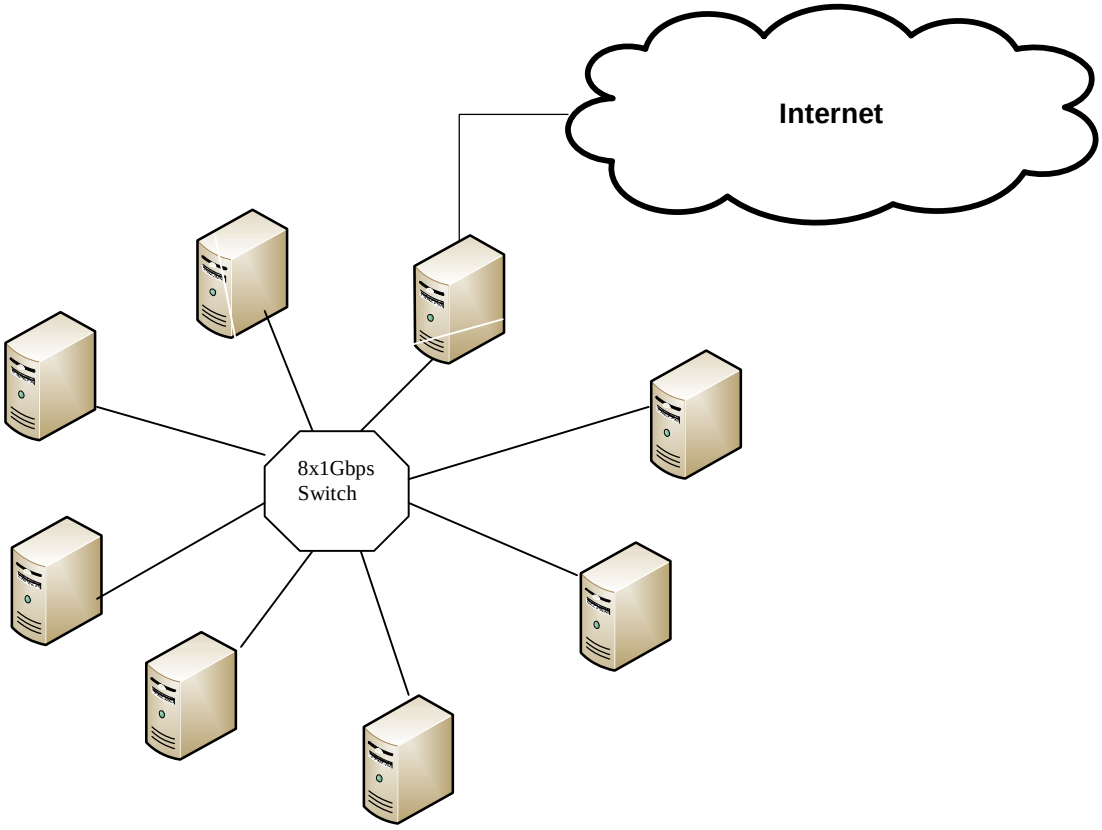


Fig 3.1 Eight node cluster connected together by a 1Gbps Ethernet switch. Each node has two duo core Intel Xeon processors running at 2.33GHz with 4GB RAM.

Implementation

As presented earlier, the volume subdivision based registration algorithm [4] proceeds by first performing a global rigid registration, followed by levels of rigid registration of hierarchically divided subvolumes to recover local misalignments. The volume sub-division part of the algorithm cannot proceed without first finishing the global rigid registration. As a result we parallelize the rigid registration part followed by the non-rigid registration part.

The most intuitive way of parallelizing rigid registration is to have each processor in the cluster process a part of the image. For load balancing the reference image is divided equally among the processors so that each processor processes the same number of voxels. The individual processors compute the mutual histogram with the voxels assigned to it. Although a processor only needs the portion of reference image during the rigid phase of the sub-volume division based non rigid registration algorithm, the complete reference image and floating image were stored on each node for following reasons. The complete reference image is store on each node because we require access to the entire reference image for parallelizing the non rigid part of the algorithm where as the floating image was stored on each node for this same purpose as well as the fact it is required during the rigid part of the algorithm because although we are processing only a part of the reference image on each processor, the candidate transform being evaluated may cause the corresponding voxel to lie anywhere in the floating image. Memory was not a limiting size for storing the images on the cluster. However we store only one copy on each node because the hardware schedules access to the data by four cores on each node. The mutual histogram is a 2D array that indicates how many times a pair of intensities at corresponding voxel locations (in the reference and floating images) occur. Since we are processing a part of the image on each node, each node has to store a 2D array the size of the original mutual histogram, which we call a partial mutual histogram. The final mutual histogram is computed by summing all the partial mutual histograms. This is illustrated in Fig 3.2.

The algorithm for parallelizing the rigid part of the volume subdivision based non-rigid registration can be summarized as follows:

1) At each iteration of the rigid registration process, a processor in the cluster processes only a portion of the reference image for computing the MI value
2) The mutual histogram is computed on each processor. Each processor has a partial mutual histogram at the end of the processing stage.
3) The partial mutual histogram on each processor is summed together to get the mutual histogram. The total voxels processed by each node is summed to get the total voxels in the mutual histogram. This stage involves network communication between the nodes.
4) The MI value is computed and passed on to the optimization stage to check whether we can stop the iteration

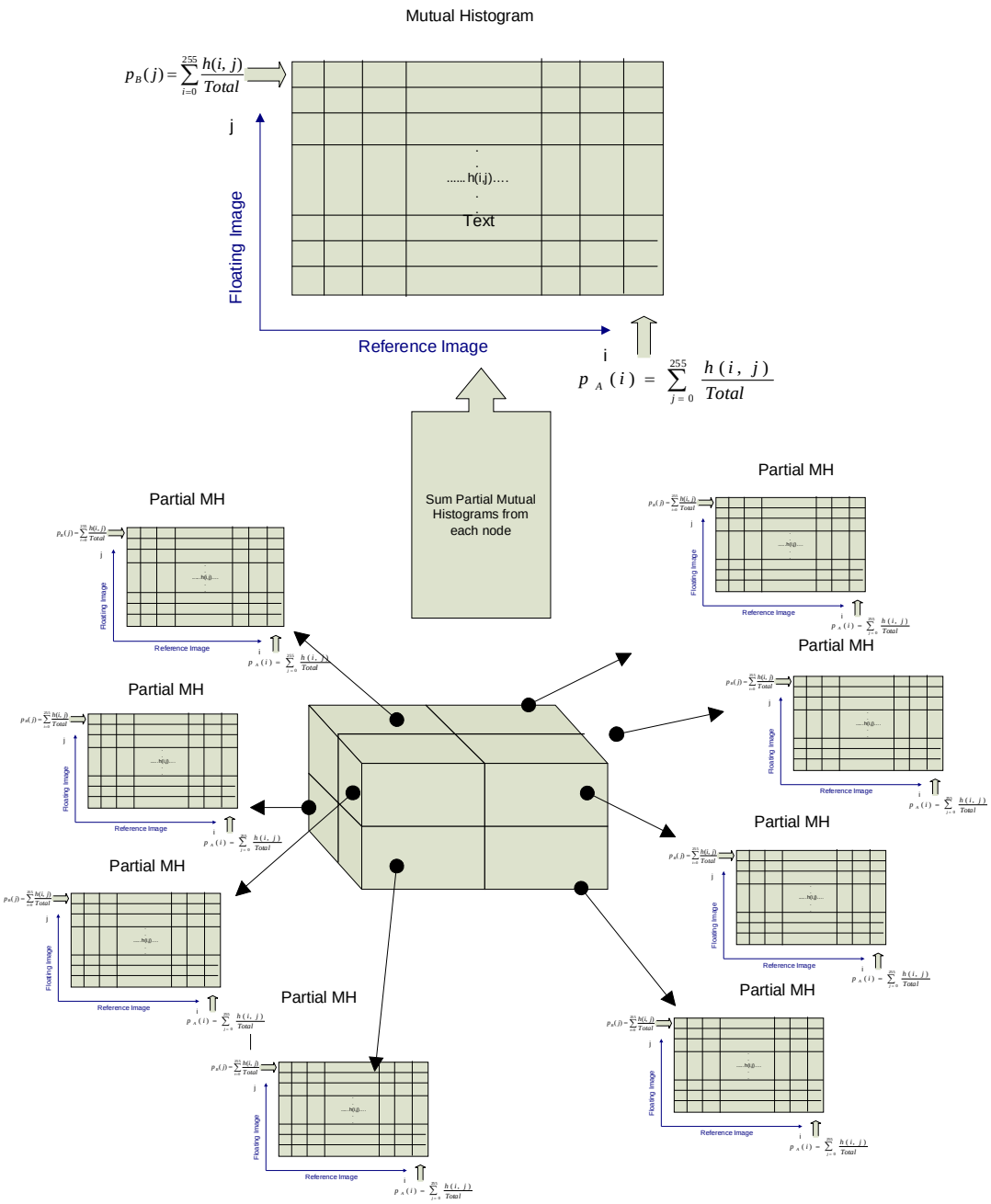


Fig. 3.2 Parallelization of the rigid part of hierarchical subvolume based registration algorithm. Each node computes a partial mutual histogram which are summed together to get the mutual histogram.

The non-rigid part of the subvolume division based non rigid registration takes place over different levels. At each level of sub-division we perform rigid registration on the subvolumes. The sub-volume registrations are independent and thus they can be assigned to different processors. Once all the subvolumes at a level have been

processed by the processors in the cluster, we communicate across the registration parameters to all the other processors so that registration parameters for all the subvolumes are available on each processor. We then go on to the next sub-division level. At the first sub-division level, there are eight sub-volumes to be registered. If the number of processors in the cluster is more than eight we use only eight of the processors while the other processors remain idle. At the next sub-division level we have sixty four sub-volumes while we only have 32 two processors in the cluster hence none of the processors are idle. To assign a sub-volume to a processor for processing we first number each sub-volume sequentially. Then we assign to each processor the subvolume whose number modulus the total number of processors in the cluster is equal to the processor number which is assigned by MPI during run time.

Results

All the Results shown are form 256x256x256 CT images. The voxel sizes in mm are:
Reference Image: 1.56, 1.56, 1.5
Image #1: 1.484375, 1.484375, 1.605468
Image #2: 1.484375, 1.484375, 1.716798
Image #3: 1.484375, 1.484375, 1.68750
Image #4: 1.484375, 1.484375, 1.68750

	#1	#2	#3	#4
Single CPU accuracy (mm)	0.71	0.82	0.85	0.73
32 CPU accuracy (mm)	0.71	0.82	0.85	0.73
Single CPU Time (seconds)	7380	7234	7403	7374
32 CPU Time (seconds)	452	452	454	451
32 CPU Speedup	16.3	16	16.3	16.4

Table 3.1: Execution time, speed up and accuracy for registering four CT 256x256x256 images. of the abdomen using four subdivision levels of subvolume division based non rigid registration

	#1	#2	#3	#4
Accuracy(mm)				
Single CPU	0.71	0.82	0.85	0.73
(Run on 8 Nodes)				
8 Processor	0.71	0.82	0.85	0.73
16 Processor	0.71	0.82	0.85	0.73
32 Processor	0.71	0.82	0.85	0.73
Execution Time(s)				
Single CPU	7380	7234	7403	7374
(Run on 8 Nodes)				
8 Processor	1158	1102	1153	1128
16 Processor	686	714	697	682
32 Processor	452	452	451	455
Speedup				
8 Processor	6.37	6.56	6.42	6.54
16 Processor	10.75	10.13	10.62	10.81
32 Processor	16.3	16	16.4	16.2

Table 3.2: Accuracy, execution time and speedup for different configurations of the cluster for registering two 256x256x256 CT images of the abdomen four subdivision levels of subvolume division based non rigid registration

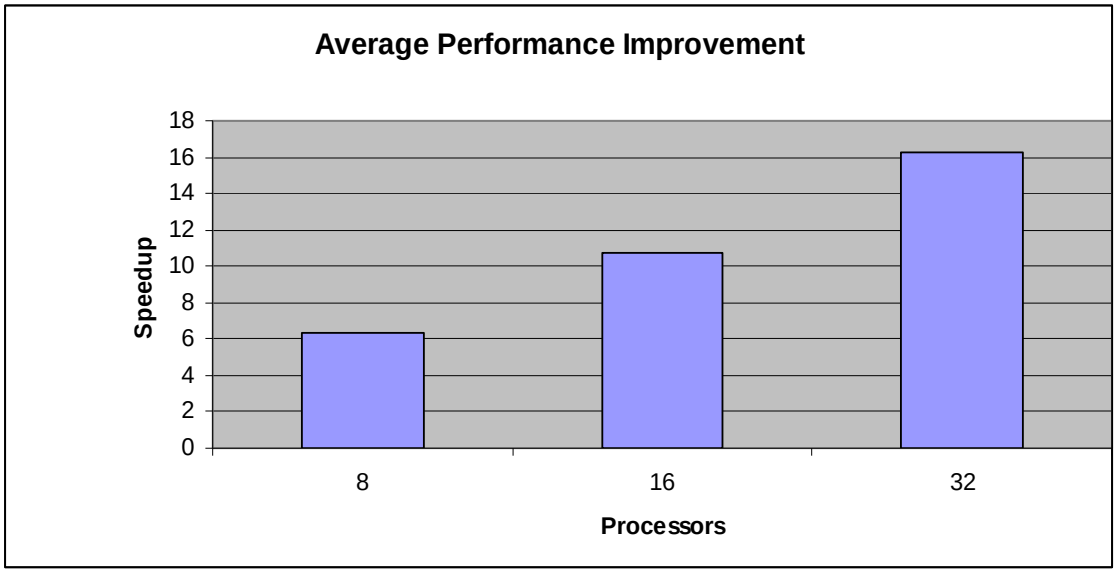


Fig 3.3: Average speedup for different configurations of the cluster

	#1	#2
1 CPU Execution Time (s)		
1 CPU Rigid	104	136
1st Level	231	229
2nd Level	374	362
3rd Level	1226	1175
4th Level	5445	5332
Total	**7380**	**7234**
8 CPU Execution Time (s)		
Global Rigid Level	21	26
1st Level	79	72
2nd Level	95	91
3rd Level	206	200
4th Level	757	713
Total:	**1158**	**1102**
16 CPU Execution Time (s)		
Global Rigid Level	15	19
1st Level	80	71
2nd Level	73	69
3rd Level	121	122
4th Level	397	433
Total:	**686**	**714**
32 CPU Execution Time (s)		
Global Rigid Level	11	15
1st Level	80	71
2nd Level	52	56
3rd Level	88	87
4th Level	221	223
Total:	**452**	**452**

Table 3.3: Break up of execution time for different configurations of the cluster for registering two 256x256x256 CT images of the abdomen using four subdivision levels of subvolume division based non rigid registration.

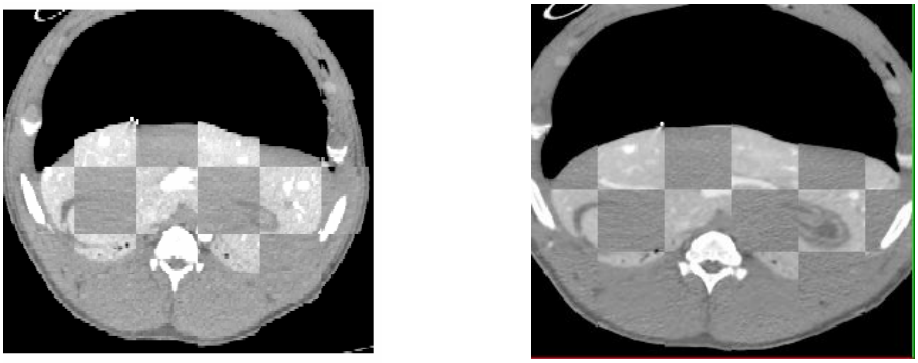


Fig 3.4: Overlay of 256X256 CT images of the liver before and after registration using four subdivision levels of subvolume division based non rigid registration

.

	Processing Time(s) max	Processing Time(s) Min	Communication Time(s)
Non Rigid Registration 8 CPU (8 Nodes)			
Level 1:	71.399802	51.463611	0.002232
Level 2:	94.331788	67.189734	0.003423
Level 3:	204.667939	132.011615	0.014778
Level 4:	757.449903	653.332129	0.255158
Non Rigid Registration 16 CPU (On 8 Nodes)			
Level 1:	71.711030	34.015598	0.053994
Level 2:	72.194589	47.387165	0.061570
Level 3:	121.247008	81.714842	0.018530
Level 4:	395.980341	296.191406	0.259062
Non Rigid Registration 32 CPU (On 8 Nodes)			
Level 1:	71.407932	33.996349	0.076562
Level 2:	57.329186	41.848857	0.252386
Level 3:	87.428625	60.840469	0.023571
Level 4:	219.763279	166.062945	0.270316

Table 3.4 Computation time for processors that take the maximum time and processors that take the minimum time given for different levels and different configurations of the cluster. Communication time is also shown

Global Rigid Registration	Histogram Computation Time(s)	Communication Time(s)	Percentage Comm. Time to Tot. Time(s)
1 Processor	1.97	N.A	N.A
(Different Cluster configurations on 8 Nodes)	0.305947	0.029223	8.6
8 Processor	0.211320	0.030716	11
16 Processor	0.119552	0.035948	23
32 Processor			

Table 3.5 Breakup of computation time in rigid registration on various configurations of the cluster (All run on 8 nodes)

Discussion

The accuracy figures shown in Table 3.1 and 3.2 are for CT images which have been deformed with a known deformation field. The registration process recovers tries to recover the deformation field.. [The accuracy figure is obtained by taking a mean square difference with the known deformation field]. We observe that the cluster version of the code generates the same result as the single CPU version.

From Table 3.1 we observe a speedup of the order of 16 when using 32 nodes. We are able to reduce processing time from over 2 hours to 7.5 minutes. This is practical for many applications. However, we note that more speedup can be achieved.

From Table 3.2 and Fig 3.2, we note that the speedup decreases with increasing number of processors. There are two reasons for this. From Table 3.4 we see that some processors take less time than others to register the subvolumes assigned to it. Thus there is a load imbalance resulting from volumes being assigned statically. In the future this can be resolved by adopting a dynamic strategy for assigning subvolumes to processors that are idle. Another reason is that even when we assign more processors to the registration algorithm, the first subdivision levels only uses eight of the available processors. We see from Table 3.3 that for the different configurations of the cluster, the execution time of the first subdivision level remains the same because of this reason. Thus our total execution time has the first subdivision level execution time as a fixed cost even if we increase the number of processors. Table 3.3 shows that the speedup of the global rigid registration level

does not improve much with increasing processors. This is because there is a large amount of data being transferred over the network for computing the mutual histogram. This is evident from Table 3.4 which shows that the communication time becomes significant as we keep on increasing the number of processors used in rigid registration. We address a way to handle this in the next chapter.

Chapter 4: Mutual Information Calculation from Mutual Histogram based on Bit Slicing

Introduction

As part of parallelization of the global rigid portion of the subvolume division based registration algorithm we presented results for a parallel implementation of mutual information in the previous chapter. In that implementation mutual information was computed using the most commonly available technique which is to sub-divide the data (image) between processors where each processor is assigned a portion of the reference image. This technique has the limitation that each processor has to hold a partial mutual histogram which is the size of the original histogram. To calculate the mutual information value the full mutual histogram has to be calculated by summing the individual partial mutual histograms followed by arithmetic operations for entropy calculation. Summing individual partial histograms entail transfer of big chunks of data between processors in the cluster. This is followed by a large number of arithmetic operations (66048 multiplications and logarithms) for entropy calculation which has to be carried out on the master processor or it can be parallelized but it will involve an additional communication overhead. We propose a new technique which allows us to maintain only a portion of the final mutual histogram on each processor during mutual histogram computation instead of a full-size partial mutual histogram which has to be reduced to get the final mutual histogram. This allows us to transfer much less data between processors and also to do arithmetic operations for entropy calculation before communicating data between processors for computing the mutual

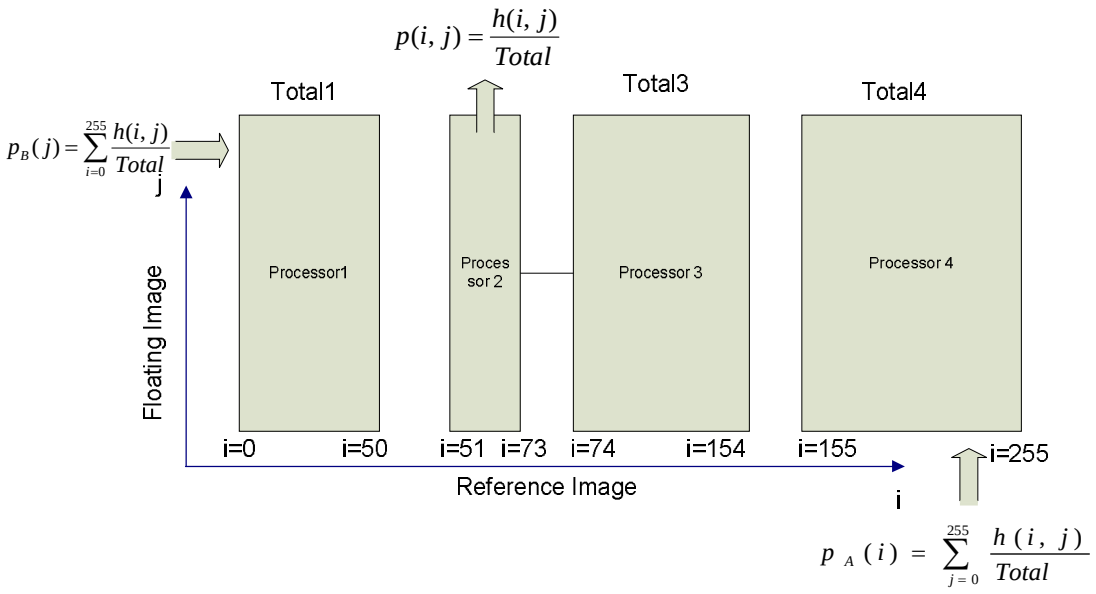
information value. In essence, we have been able to parallelize arithmetic computations involved in the mutual information computation process as well as transfer less data between processors.

Implementation

As described in the introduction, this concept allows us to maintain a part of the mutual histogram on each processor instead of a partial mutual histogram which needs to be summed with other partial mutual histograms to yield the mutual histogram. This is accomplished by pre-processing the reference image. To each processor we assign an intensity range of the reference image that it will process. The pre-processing involves finding all the data (voxels) that each processor will process based on the intensity range it will be processing. Figure 4.1 shows the mutual histogram after processing has completed calculation on each processor. We reason that the reference image is available before the floating image in image registration applications and so we can pre-process it and that the pre-processing time does not need to be considered part of the time required for registration. Another reason that the pre-processing does not need to be part of the registration process is that the reference image can be stored in the desired format during the acquisition process itself.

The data associated with the intensity range (bins) assigned to a processor has to be balanced. Since the intensity range assigned to a processor determines the amount of data it will be processing, we have to vary the intensity range assigned to a processor for load balancing. We describe below the technique we have used for load balancing.

1) Calculate the histogram of the reference image
2) Find the total number of voxels that have to be processed
3) Find the average number of voxels to be processed by a processor Avg Voxels = Total/Total Number of Processors
4) Starting at the first processor, assign to it intensities that it will process while keeping track of the total number of voxels assigned to it. (Total[1]). When Total[1] > Avg Voxels, start assigning voxels to the next processor but distribute the remaining voxels to be processed among the remaining processors.
5) (New) Avg Voxels = Remaining Voxels/Remaining Processors
6) Now assign intensities to the next processor and keep track of the total number of voxels assigned to that processor. (Total[i]) when Total[i] >Avg Voxels then go to step 5 until all the voxels have been assigned to all the processors.



$$H(A,B)=\sum_{i=0}^{255}\sum_{j=0}^{j=255}p(i,j)\log(p(i,j))$$ (Equation for computing Joint Entropy)

$$H(A)=-\sum_{i=0}^{255}p_A(i)\log p_A(i)$$ (Equation for computing Entropy of Image A)

$$H(B)=-\sum_{j=0}^{255}p_B(j)\log p_B(j)$$ (Equation for computing Entropy of Image B)

Fig: 4.1 Mutual Histogram on each processor after processing using the bit-slicing algorithm on a cluster with four processors. Note how the mutual histogram is divided based on intensities of the reference image. The probability densities are calculated as shown in the figure. Formulas for computing entropy values from the probability densities are also shown.

Part of the computation for mutual information calculation can be carried out on each processor after the mutual histogram computation is completed. From Fig. 4.1 we see that $p_A(i)$ (and hence part of H(A)) and $p_{A,B}(i,j)$ (and hence part of H(A,B)) could be computed if we knew Total (the total number of voxels processed during this iteration of the registration process). Each processor knows only $Total_i$ which is the total

voxels processed on that processor. This necessitates the need for communication between processors to sum up $Total_i$ to get Total. However, we derive below that we do not need to communicate $Total_i$ between processors to carry out the computations we need to perform. We also point out that we cannot do similar computations for H(B) since we do not have $p_B(j)$. However we can reduce it as shown in equation (3).

$$MI = H(A) + H(B) - H(A,B)$$

A: Reference Image

B: Floating Image

$H(A)$: Entropy of Image A

$H(B)$: Entropy of Image B

$H(A,B)$: Joint Entropy of Image A and B

$$H(A,B) = -\sum_{i=0}^{255} \sum_{j=0}^{j=255} p(i,j)\log(p(i,j))$$

$$= -\sum_{i=0}^{255} \sum_{j=0}^{255} \frac{h(i,j)}{Total}\log\frac{h(i,j)}{Total}$$

$$\because\ p(i,j) = \frac{h(i,j)}{Total}$$

$$= -\frac{1}{Total}\left[\sum_{i=0}^{255} \sum_{j=0}^{j=255} h(i,j)\left[\log h(i,j) - \log Total\right]\right]$$

$$= -\frac{1}{Total}\left[\sum_{i=0}^{255} \sum_{j=0}^{j=255} h(i,j)\left[\log h(i,j)\right]\right] + \frac{1}{Total}\left[\sum_{i=0}^{255} \sum_{j=0}^{j=255} h(i,j)\log Total\right]$$

$$= -\frac{1}{Total}\left[\sum_{i=0}^{255}\ \sum_{j=0}^{j=255} h(i,j)[\log h(i,j)]\right] + \frac{\log Total}{Total}\left[\sum_{i=0}^{255}\ \sum_{j=0}^{j=255} h(i,j)\right]$$

$$= -\frac{1}{Total}\left[\sum_{i=0}^{255}\ \sum_{j=0}^{j=255} h(i,j)[\log h(i,j)]\right] + \log Total$$

$$\because \left[\sum_{i=0}^{255}\ \sum_{j=0}^{j=255} h(i,j)\right] = \text{Total}$$

$$= -\frac{1}{Total}\left[\begin{array}{l}\sum_{i=0}^{node1}\ \sum_{j=0}^{j=255} h(i,j)[\log h(i,j)] + \sum_{i=node1+1}^{node2}\ \sum_{j=0}^{j=255} h(i,j)[\log h(i,j)] + ... \\ + ... + \sum_{i=i=nodeN+1+1}^{255}\ \sum_{j=0}^{j=255} h(i,j)[\log h(i,j)]\end{array}\right]$$

$$+ \log Total$$

$$= -\frac{1}{Total}[m_1 + m_2 + ... + m_N] + \log Total \quad \text{- (1)}$$

N: The number of processors

Equation 1 shows that the joint entropy can be computed by evaluating m1,m2..m_N on each processor and then summing them up and applying total to compute $H(A,B)$.

$H(A)$ can also be computed in a similar manner as $H(A,B)$ in equation 1.

$$H(A) = -\sum_{i=0}^{255} p_A(i) \log p_A(i)$$

$$= -\sum_{i=0}^{255}\left[\sum_{j=0}^{j=255} \frac{h(i,j)}{Total}\left(\log\left(\sum_{j=0}^{j=255} \frac{h(i,j)}{Total}\right)\right)\right]$$

$$\because p_A(i) = \sum_{j=0}^{255} \frac{h(i,j)}{Total}$$

$$= - \frac{1}{Total} \sum_{i=0}^{255} \left[\sum_{j=0}^{j=255} h(i,j) \left(\log \left(\sum_{j=0}^{j=255} h(i,j) \right) - \log Total \right) \right]$$

$$= - \frac{1}{Total} \sum_{i=0}^{255} \left[\sum_{j=0}^{j=255} h(i,j) \left(\log \left(\sum_{j=0}^{j=255} h(i,j) \right) \right) \right] + \frac{1}{Total} \left[\sum_{i=0}^{255} \sum_{j=0}^{j=255} h(i,j) \log Total \right]$$

$$= - \frac{1}{Total} \sum_{i=0}^{255} \left[\sum_{j=0}^{j=255} h(i,j) \left(\log \left(\sum_{j=0}^{j=255} h(i,j) \right) \right) \right] + \frac{\log Total}{Total} \left[\sum_{i=0}^{255} \sum_{j=0}^{j=255} h(i,j) \right]$$

$$= - \frac{1}{Total} \sum_{i=0}^{255} \left[\sum_{j=0}^{j=255} h(i,j) \left(\log \left(\sum_{j=0}^{j=255} h(i,j) \right) \right) \right] + \frac{\log Total}{Total} \left[\sum_{i=0}^{255} \sum_{j=0}^{j=255} h(i,j) \right]$$

$$= - \frac{1}{Total} \sum_{i=0}^{255} \left[\sum_{j=0}^{j=255} h(i,j) \left(\log \left(\sum_{j=0}^{j=255} h(i,j) \right) \right) \right] + \log Total$$

$$\because \left[\sum_{i=0}^{255} \sum_{j=0}^{j=255} h(i,j) \right] = \text{Total}$$

$$= - \frac{1}{Total} \left[\begin{array}{l} \sum_{i=0}^{node1} \left[\sum_{j=0}^{j=255} h(i,j) \left(\log \left(\sum_{j=0}^{j=255} h(i,j) \right) \right) \right] + \sum_{i=node1+1}^{node2} \left[\sum_{j=0}^{j=255} h(i,j) \left(\log \left(\sum_{j=0}^{j=255} h(i,j) \right) \right) \right] + ... \\ .... + \sum_{i=nodeN-1+1}^{NodeN} \left[\sum_{j=0}^{j=255} h(i,j) \left(\log \left(\sum_{j=0}^{j=255} h(i,j) \right) \right) \right] \end{array} \right]$$

$$+ \log Total$$

$$= - \frac{1}{Total} \left[e1 + e2 + ... + e_N \right] + \log Total \qquad \text{- (2)}$$

Equation 2 shows that the entropy of the reference image can be computed by evaluating e1,e2..e_N on each processor and then summing them up and applying total to compute $H(A)$.

$$H(B) = -\sum_{j=0}^{255} p_B(j) \log p_B(j) \qquad \text{- (3)}$$

$$p_B(j) = \sum_{i=0}^{255} \frac{h(i, j)}{Total}$$

$$= \frac{1}{Total}\left[\sum_{i=0}^{Node1} h(i, j) + \sum_{i=Node1+1}^{Node2} h(i, j) + \ldots + \sum_{i=NodeN-1+1}^{NodeN} h(i, j) \right]$$

$$= \frac{1}{Total}\left[l_1(j) + l_2(j) + \ldots l_N(j) \right] \qquad \text{- (4)}$$

Equation 4 indicates that the probability density of each intensity (j) can be computed by evaluating $l_1(j)$, $l_2(j)$, .., $l_N(j)$, on each processor and then summing them up and applying equation 4 to compute $p_B(j)$.

We summarize the steps involved in computing MI using the bit slicing idea:

1) Pre-process the reference image and assign voxels to each processor in the cluster based on the intensities that each processor will be processing. The number of voxels processed by each processor need to be load balanced and thus the number of intensities of the reference image that each processor processes will need to be assigned dynamically as per the algorithm we defined earlier
2) Start the registration process
3) For each iteration of the registration process, the mutual histogram is computed and as shown in Fig. 4.2, we get a part of the mutual histogram on each processor

4) MI computation follows from equations 1, 2 and 3. Each processor computes m_k which is necessary for computing H(A,B), e_k which is necessary for computing H(A) and $l_k(j)_{j=0}^{255}$ which is necessary for computing $p_B(j)_{j=0}^{j=255}$

5) Finally all processors communicate to one another to sum

   i. Total_i (the total voxels processed on each processor) to get Total

   ii. m_k and use equation (1) to get H(A,B)

   iii. e_k and use equation (2) to get H(A)

   iv. $l_k(j)_{j=0}^{255}$ to get $p_B(j)_{j=0}^{j=255}$ using equation (4) and finally use equation (3) to get H(B)

6) MI is computed as H(A)+H(B)-H(A,B)

Results

	#1	#2	#3	#4
Execution Time (s)				
Single CPU	104	136	301	89
8 CPU image subdivision	21	26	55	18
8 CPU bit-slicing	20	26	44	17
16 CPU image subdivision	15	19	43	12
16 CPU bit-slicing	11	14	26	9
32 CPU image subdivion	11	15	30	10
32 CPU bit-slicing	7	9	18	6

Table 4.1 Execution time for rigid registration of three 256x256x256 CT images shown for various configurations of the cluster with and without bit-slicing algorithm

	#1	#2	#3	#4
Single CPU Translation (Tx,Ty,Tz) Rotation(Rx,Ry,Rz)	17.82, 13.19,14.37 -4.03, -0.98, -0.47	5.52 36.57, 16.42 0.58, -1.19, -5.28	-7.79, -19.38, -6.85 6.65, 7.19, 12.57	2.43, 0.0081,6.97 -2.72, 1.50, -3.91
8 CPU Bit Slicing Translation (Tx,Ty,Tz) Rotation(Rx,Ry,Rz)	17.82, 13.19,14.37 -4.03, -0.98, -0.47	5.52 36.57, 16.42 0.58, -1.19, -5.28	-7.79, -19.38, -6.85 6.65, 7.19, 12.57	2.43, 0.0081,6.97 -2.72, 1.50, -3.91
16 CPU Bit-slicing Translation (Tx,Ty,Tz) Rotation(Rx,Ry,Rz)	17.82, 13.19,14.37 -4.03, -0.98, -0.47	5.52 36.57, 16.42 0.58, -1.19, -5.28	-7.79, -19.38, -6.85 6.65, 7.19, 12.57	2.43, 0.0081,6.97 -2.72, 1.50, -3.91
32 CPU Bit-slicing Translation (Tx,Ty,Tz) Rotation(Rx,Ry,Rz)	17.82, 13.19,14.37 -4.03, -0.98, -0.47	5.52 36.57, 16.42 0.58, -1.19, -5.28	-7.79, -19.38, -6.85 6.65, 7.19, 12.57	2.43, 0.0081,6.97 -2.72, 1.50, -3.91

Table 4.2 Accuracy of rigid registration using bit-slicing algorithm as compared to the single CPU version. The transformation parameters recovered by the bit-slicing algorithm is the same as that of the single CPU version.

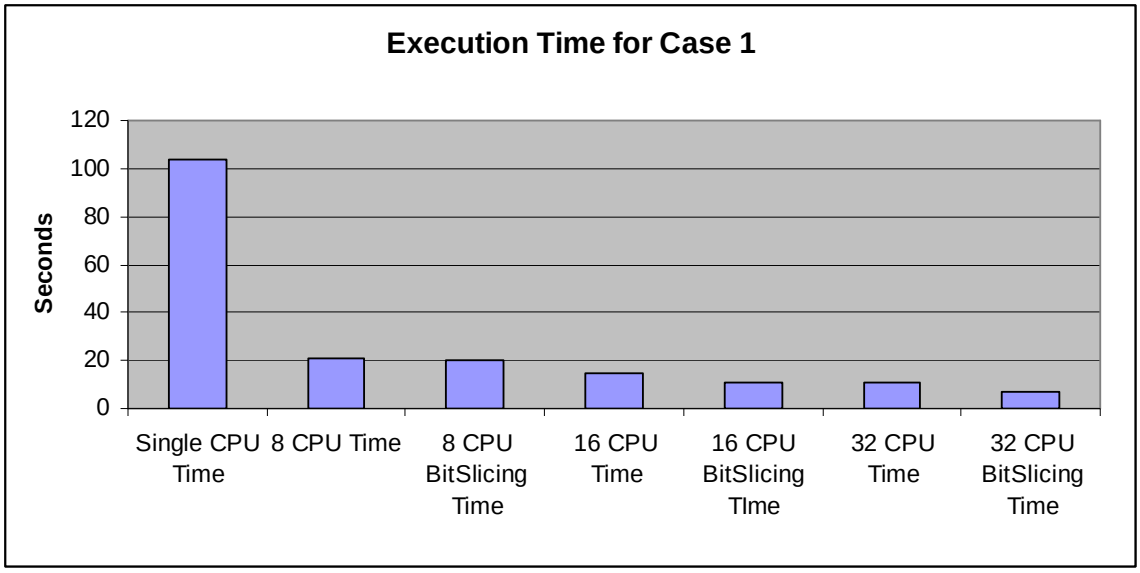


Fig 4.2 Execution time for rigid registration of the first set of 256x256x256 CT images shown for various configurations of the cluster with and without bit-slicing algorithm compared with the single CPU version

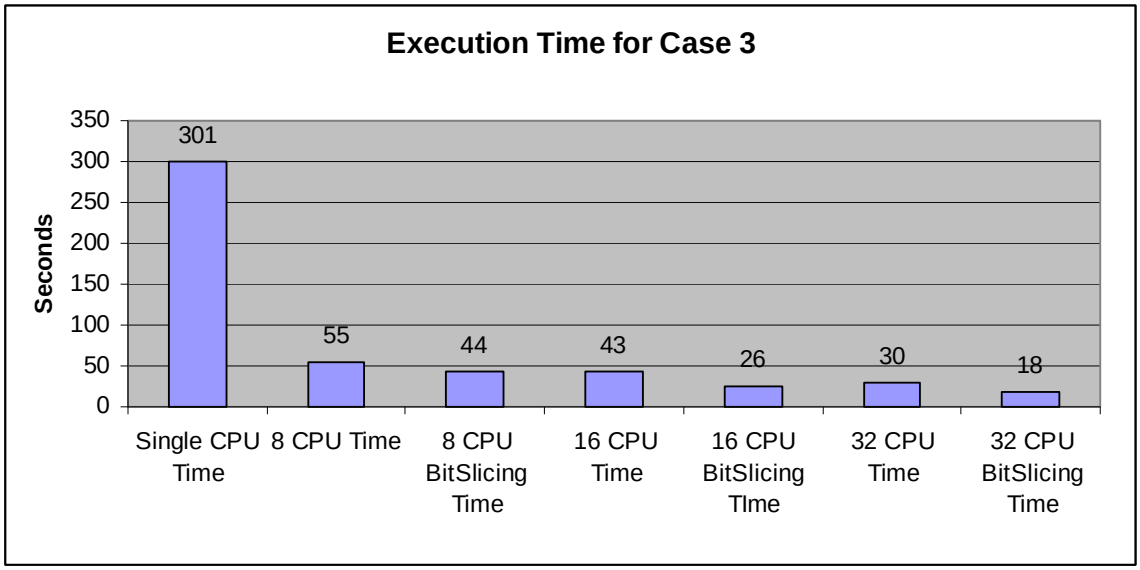


Fig 4.3 Execution time for rigid registration of the third set of 256x256x256 CT images shown for various configurations of the cluster with and without bit-slicing algorithm compared with the single CPU version

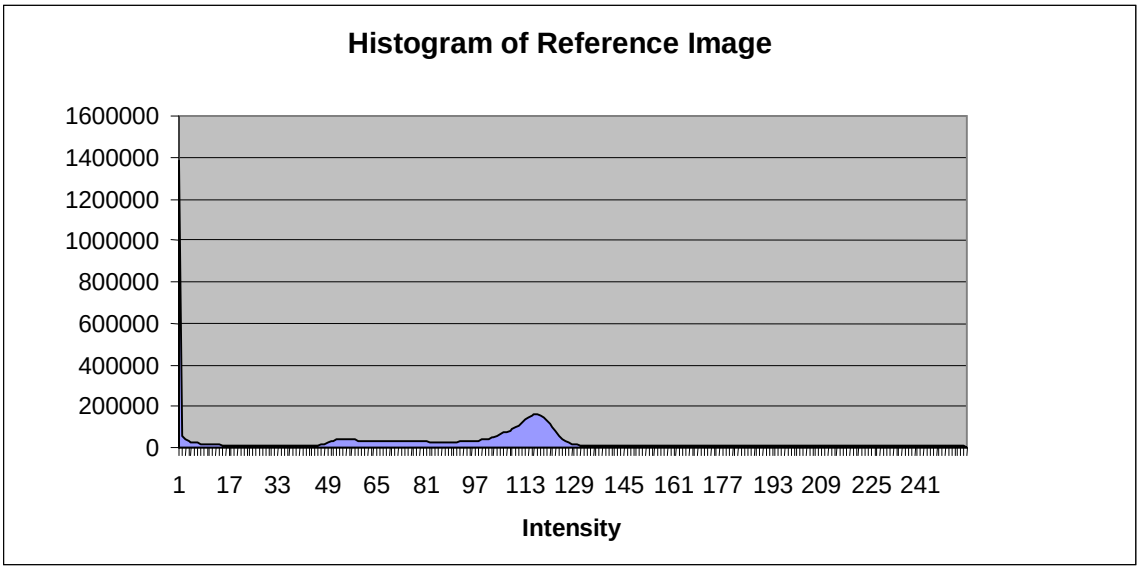


Fig 4.4 Histogram of the reference image from image set 3

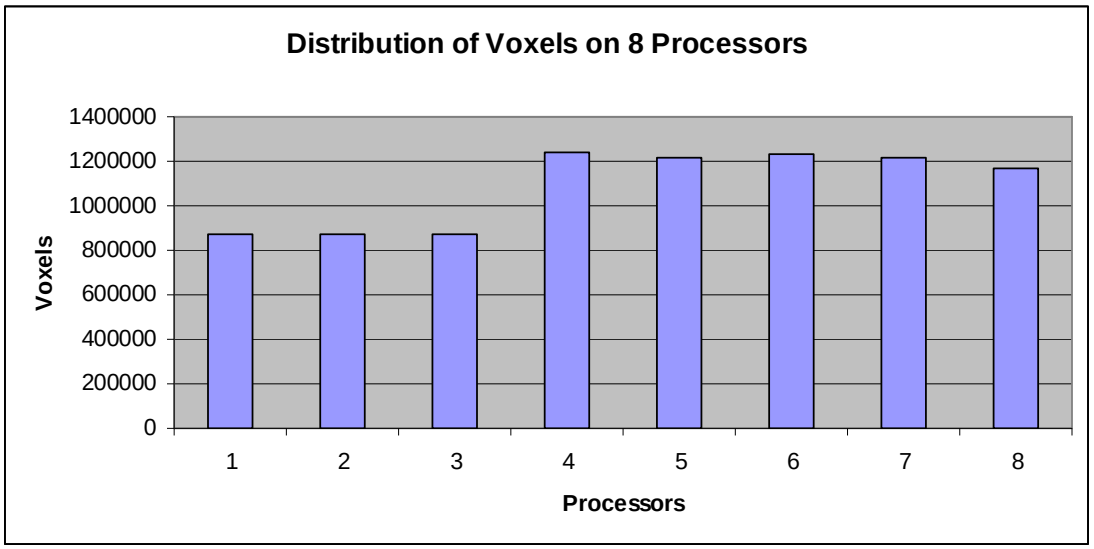


Fig 4.5 Distribution of voxels for 8 processors

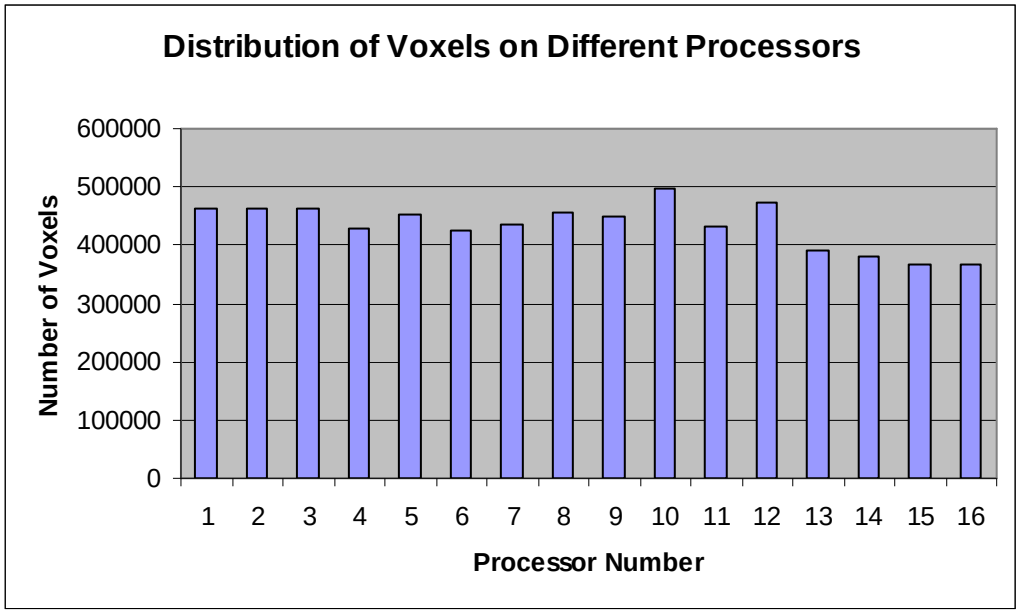


Fig 4.6 Distribution of voxels on 16 processors by the load balancing algorithm

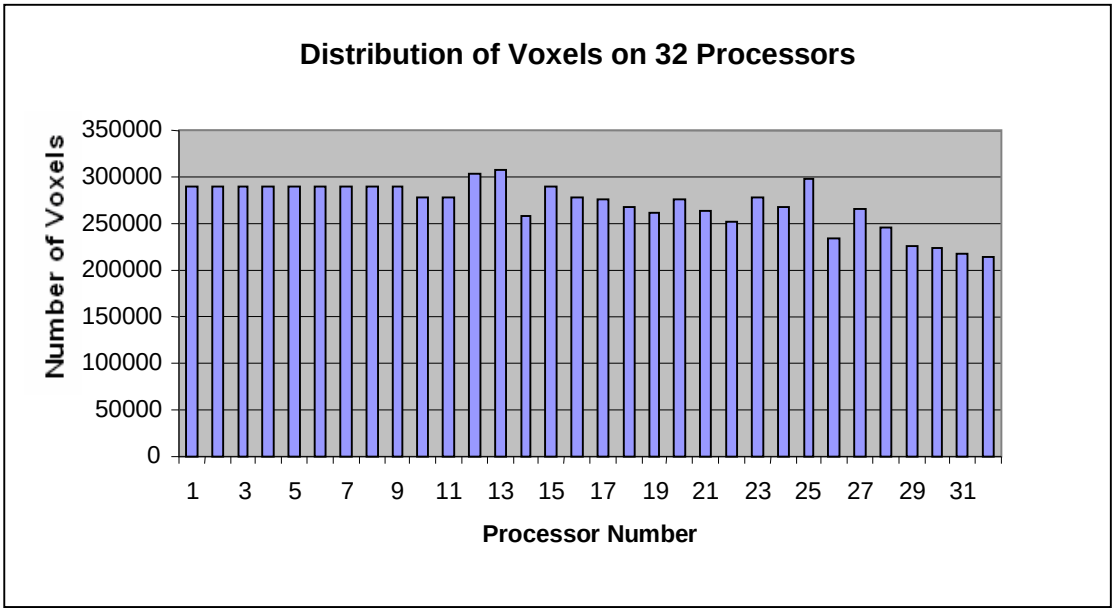


Fig 4.7 Distribution of voxels on 32 processors by the load balancing algorithm

	Histogram Computation Time(s)	Communication Time(s)	Percentage Comm. Time to Tot. Time(s)
MI Computation by Image Subdivion			
8 CPU	0.305947	0.029223	8.6
16 CPU	0.211320	0.030716	11
32 CPU	0.119552	0.035948	23
MI Computation by Bit-slicing			
8 CPU	0.313878	0.000615	0.2
16 CPU	0.160779	0.000630	0.39
32 CPU	0.092174	0.000653	0.7

Table 4.3: Histogram computation time and the associated communication time for different cluster configurations.

Discussion

Table 4.2 indicates that registration using the bit-slicing algorithm for computing the mutual information recovers the same transformation parameters as the rigid registration process on a single processor.

From the comparisons of execution time in Table 4.1 we see that the bit-slicing algorithm runs more efficiently for 32 processors than for eight processors. The execution time for bit-slicing algorithm for 8 processors does not improve significantly from the execution time of the image subdivision based algorithm for eight processors. This is because the bit-slicing algorithm mainly speeds up the communication time required for transferring results across the network. When there

are only eight processors the communication time is not a big factor of the processing time as seen in Table 4.3. In the image subdivision based algorithm for computing mutual information, the image is divided equally between processors and hence no load balancing is done to ensure that equal amount of foreground voxels are being assigned to each processor. Better load balancing could be done in the image subdivision based approach by dividing the image based on the foreground pixels but a preprocessing stage would be required and demarcation of regions to process on each processor in the cluster would need to be communicated. However the bit-slicing algorithm balances the load on the cluster by assigning only foreground voxels of the reference image by the previously mentioned load balancing algorithm. As a result there is an improvement in execution time of the bit-slicing algorithm on eight processors due to contributions from speed up in communication and better load balancing. If the load balancing algorithm is not able to balance the load perfectly then the performance of the bit-slicing algorithm can be severely hampered.

We see from Table 4.3 that with increasing number of processors, the communication time becomes a larger factor in the processing time. Consequently the bit-slicing algorithm performs better because it reduces the communication time. We have run the bit-slicing algorithm only for 32 processors but we see that with increasing number of processors the bit-slicing algorithm is able to perform much better than the image subdivision based algorithm for computing mutual information. However there will be a stage at which increasing the number of processors will not improve the overall speed up because the communication time will become more significant.

Chapter 5: MI calculation on GPU using Bit-Slicing Idea

Introduction and Related Work

We present an implementation of mutual information calculation on an NVIDIA GPU. GPUs have been increasing in computation power rapidly as they has evolved into a highly parallel, multithreaded, multicore processor with high memory bandwidth [9]. Compute intensive processing which involves data parallel processing is well suited for implementation on the GPU. We use the compute unified device architecture (CUDA) for programming the GPU. CUDA is a programming model that allows the programmer to utilize the parallelism offered by the GPU while limiting the learning curve as it is an extension of the C programming language. However current implementations for MI calculation on the GPU have been limited by the shared memory size of the GPU. In [8], we see a 64-bin implementation (six bit intensity values of a histogram which is far too low for any practical image registration purpose. In [10], an approximate method is proposed for MI calculation which reduces these inefficiencies. However, they are limited to byte data types and 10000 bins (100x100) for the mutual histogram, in addition to calculating only an approximate MI value. The byte data type does not allow them to use PV interpolation as it required float data type for the mutual histogram. However an accurate implementation of MI computation needs to have PV interpolation and in cases 16384 bins (128x128) mutual histogram. The bit- slicing idea we proposed for the cluster maps well to the GPU and this implementation is the first that is able to calculate MI for 8 bit images (65,536 bins) using float data types for the mutual histogram and allows us to use PV interpolation. As seen in the cluster

implementation arithmetic computations for entropy can be parallelized without additional overhead with this new idea.

GPU Architecture

This GPU is based on the TESLA architecture by NVIDIA which provides a platform for both graphics and general purpose parallel computing applications. Figure 5.1 shows the Tesla Architecture which consists of a scalable array of streaming multiprocessors. A multiprocessor consists of eight scalar processors two special function units for transcendental operations, an instruction unit and shared memory. Access to the shared memory is fast (four clock cycles) but access to device memory is much slower (400 to 600 clock cycles). . In CUDA threads are scheduled in groups of 32 known as warp. A block having 128 threads will have four groups of 32 or in other words four warps in the block. Threads in a half warp (half the threads of the warp) can access the global memory together if the address accessed by each thread in the half warp is aligned. If the addresses accessed by the threads in the half warp are not aligned then the resulting access to global memory will be sequential. NVIDA GPUs with compute capability 1.0 do not provide atomic updates (when two or more threads try to update the same memory location at the same time, only one thread will update the location if atomic updates are not supported)or support double (i.e., double-precision floating point) data types.

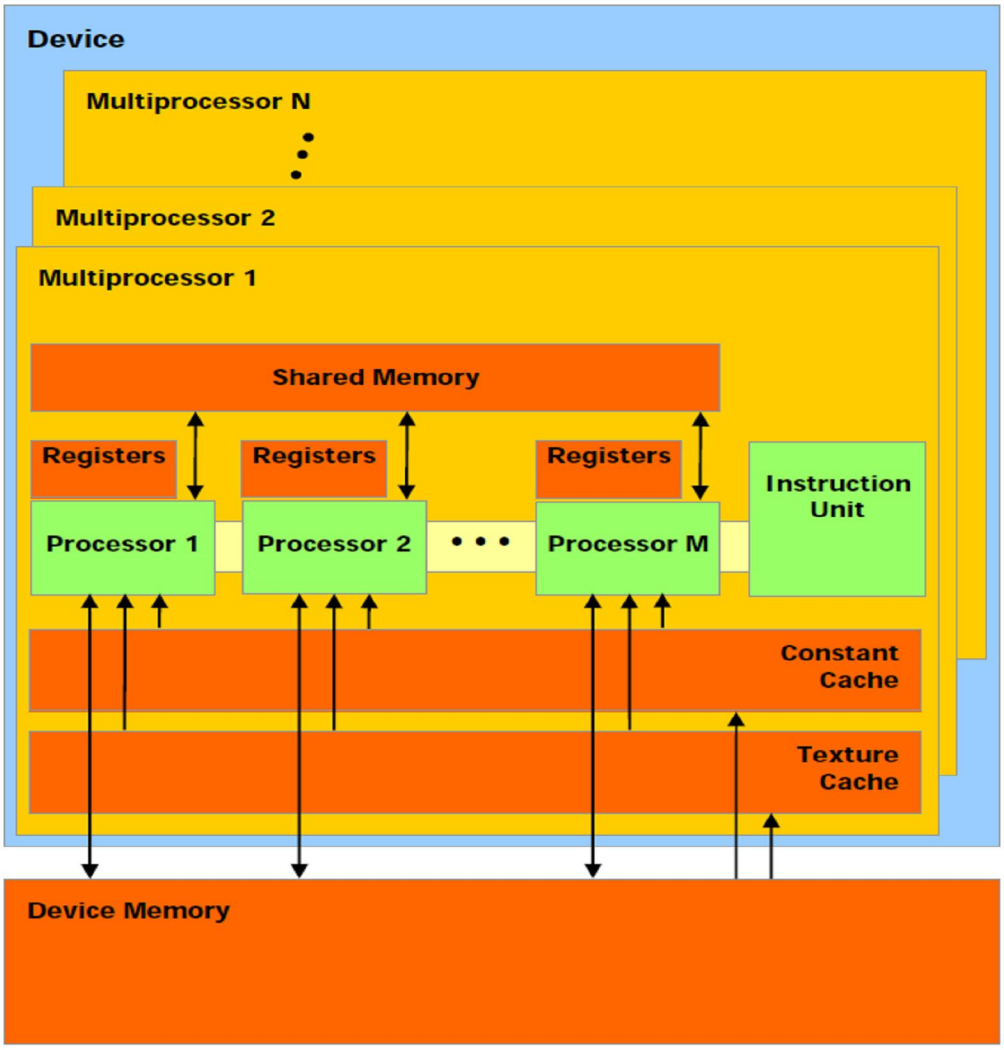


Fig. 5.1 Tesla Architecture (Courtesy [9])

CUDA abstracts the program to be run as a kernel. A kernel is composed of a Grid and blocks. The number of blocks and threads are set by the programmer and this is known as the execution configuration. Each block is composed of several threads. This is shown in the Fig 5.2. Each block of the grid runs on a streaming multiprocessor with the threads in a block running concurrently. Each thread is mapped to a scalar processor core and it has its own instruction address and register

state which allows each thread to run a different program, though not efficiently. As mentioned earlier the multiprocessor schedules threads in groups of 32, called warps. A warp executes a common instruction at a time so maximum efficiency is obtained when the code in a warp does not diverge. This architecture is known as Single instruction Multiple Thread (SIMT)

Threads within a block can be synchronized. It is not possible to synchronize between threads of different blocks. As a programmer, this limits some of the options like waiting for results from other blocks. We handle this by copying results from each block into device memory and then using another kernel call (programs are written as kernels in CUDA) to sum the results in device memory. The GPU scheduler waits for all the blocks to finish processing before another kernel is allowed to execute. We achieve synchronization between blocks in this manner.

The steps necessary to implement a CUDA program involve the following:

1) Have the host computer allocate memory on the GPU
2) Copy data to be processed into the GPU device memory
3) Set the execution configuration and invoke the kernel
4) Once execution is over, copy the processed data back to the host

We have used an NVIDIA 8800 GTX GPU with 16 streaming multiprocessors and 8 scalar processor cores per multiprocessor, which is of compute capability 1.0 (the compute capability indicates whether some feature such as atomic updates or double

precision operations are supported) and uses CUDA 2.0 programming model. It has 16 KB shared memory per multiprocessor and 768 MB global memory.

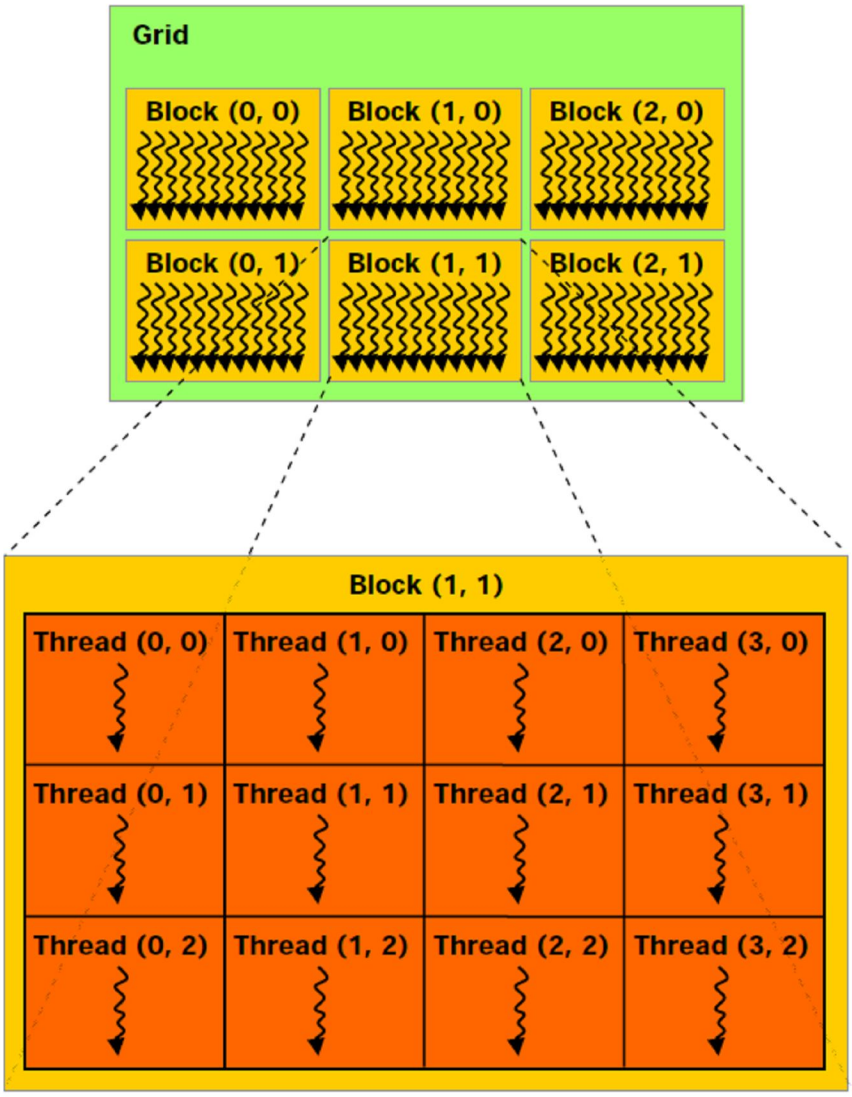


Fig. 5.2 CUDA Programming Model (Courtesy [9])

Implementation Details

The implementation on the GPU is similar to that on the cluster with some key differences noted in the next paragraph. Here a block plays the role of the processor in the cluster. The reference image is pre-processed to assign intensities to blocks.

After processing, each block will have a part of the final mutual histogram. This was the case with each processor in the cluster. This is shown in Fig. 5.3. The MI value can then be computed making use of the same equations we presented for the cluster implementation.

Each block in the GPU can have many threads to do the computation assigned to it. We maintain the mutual histogram in the shared memory as updates are faster to the shared memory and it also allows us to do the arithmetic computations for entropy calculation faster. The problem with previous implementations was that a partial mutual histogram which is the size of the final mutual histogram had to be stored in the shared memory. Since current GPUs have only 16Kb of shared memory, these implementations were limited in the number of bins (10,000) of the mutual histogram and also the data type for the storage of the frequency of each bin was limited to an unsigned char. That meant they were unable to implement PV interpolation because they have to make a trade of between the number of bins and the size of the data type of each bin. As the bit-slicing algorithm allows us to store only part of the final mutual histogram on each block, we can use float data type for each memory location of the mutual histogram without having to sacrifice the number of bins we are using. . We show results for 8 bit intensities for 256x256 mutual histogram. We are also using PV interpolation. Due to the shared memory size we are limited to the number of intensities (i.e., bins) of the mutual histogram we can store. We assign a maximum of 10 intensities to each block which necessitates the need for 10Kb of shared memory (1Kb per intensity of the reference image assigned to the block). As we are

limited to 10 intensities for each block we have to have at least 26 blocks to cover 256 intensity levels. After processing we have a part of the mutual histogram on each block. This is similar to having a part of the mutual histogram on each processor in the cluster. This is shown in Fig 5.3. We then do arithmetic operations to find the MI value. Each block does the same processing that each processor in the cluster did. In the cluster, after processing, data has to be communicated between processors through the network. In the GPU, we store the processed data in the global memory, which is then processed by a single block to find the final MI value. We give a summary of the whole process which gives a better picture of the processing involved.

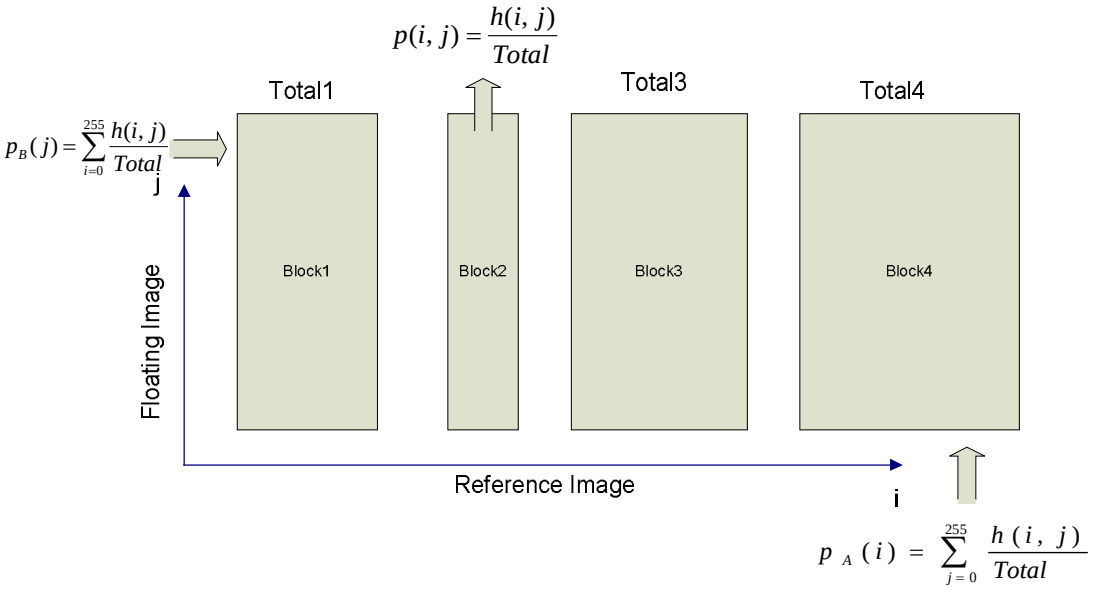


Fig. 5.3 Mutual Histogram divided between four blocks

The similarity between the cluster implementation ends with the above discussion. We now go on to the nuances of the GPU implementation. Each block has more than one thread running simultaneously. On the NVIDIA 8800 GTX GPU atomic updates

are not supported. Hence we cannot have more than one thread from a warp making simultaneous updates to the same memory location which in our case is the mutual histogram stored in the shared memory of each block (threads from a different warp do not update the same location at the same time). As a consequence we have to limit each thread in a warp to updating only one column of the histogram assigned to that block and that limits the number of threads running in that block. If we were to follow the naïve approach, we assign to each block an equal number of intensities to process. However, as each thread from the warp does processing on a single intensity of the reference image, we could have a load imbalance resulting in slower processing time as the other blocks will have to wait for the block whose thread has many voxels to process.

Hence we do load balancing to ensure each thread of a block processes about the same number of voxels. This is done by allowing for an intensity to be split so another thread can work on different voxels but of the same intensity. We allocate a 2D array of size 256x10 (256 rows and 10 columns) of type float (10KB) in the shared memory. When an intensity is split, the threads that do processing for the split intensity have to have a separate column of the 2D array in the shared memory assigned to it. The split intensities are then combined by summing up the columns processed by the corresponding threads. If only one intensity is assigned to a block, then its split 10 times and processed by 10 threads each working on its own column of memory (1KB). The load balancing algorithm assigns an intensity to a block, splits it and compares the voxels being processing per thread with the average voxels per

thread for the entire image. If its lower than the average voxels per thread then it assigns another intensity to the block. (The steps of the load balancing algorithm are given in detail later on) Now there are two intensities assigned to the block. We split all the intensities assigned to the block equally. All the intensities need not be split equally but splitting it equally makes it easier to indicate how many times the intensities have been split for a block. The intensities assigned to a block are split such that the total columns in the 2D array required by the threads are still less than 10. We thus allow the two intensities to split five times each so that each intensity is covered by 5 threads and hence by 5 columns of the 2D array. Fig 5.4 shows the case for 3 intensities. Our constraint is that after splitting we can only have a maximum of 10 columns of the 2D array in the shared memory being used by the threads. Another constraint is that we split all the intensities equally. Hence three intensities can be split up only 3 times and nine columns of the 2D array are used. If there are six intensities assigned to a block then it isn't split up as all the intensities need to be split equally as per our implementation and that would result in the need of 12 columns of the 2D array in the shared memory...

We show results for 40 blocks. We could have used 26 blocks but then 25 of the 26 blocks would need to process 10 intensities each. By using 40 blocks we allow more flexibility in grouping together intensities without being forced to assign many intensities to a block. This allows us to split the intensities which have many voxels.

CUDA schedules 32 threads at a time. This is known as a warp. As we do not have atomic updates, we will make active only a part of the 32 threads for processing the data for the mutual histogram. The number of threads made active in the warp is dependent on the number of intensities assigned to the block by the load balancing algorithm and also the number of times those intensities have been split. The other threads are left idle. Each thread in the warp is assigned a column of the 2D array allocated in the shared memory. Threads from a different warp are however allowed to update to the same column of the 2D array in the shared memory. We run four warps so we have four threads updating the same column in memory. As threads in the same warp are not allowed to update to the same location we do not have a problem of different threads updating to the same location. Also as threads from different warps are scheduled separately, we again do not have threads from different warps trying to update the same location..

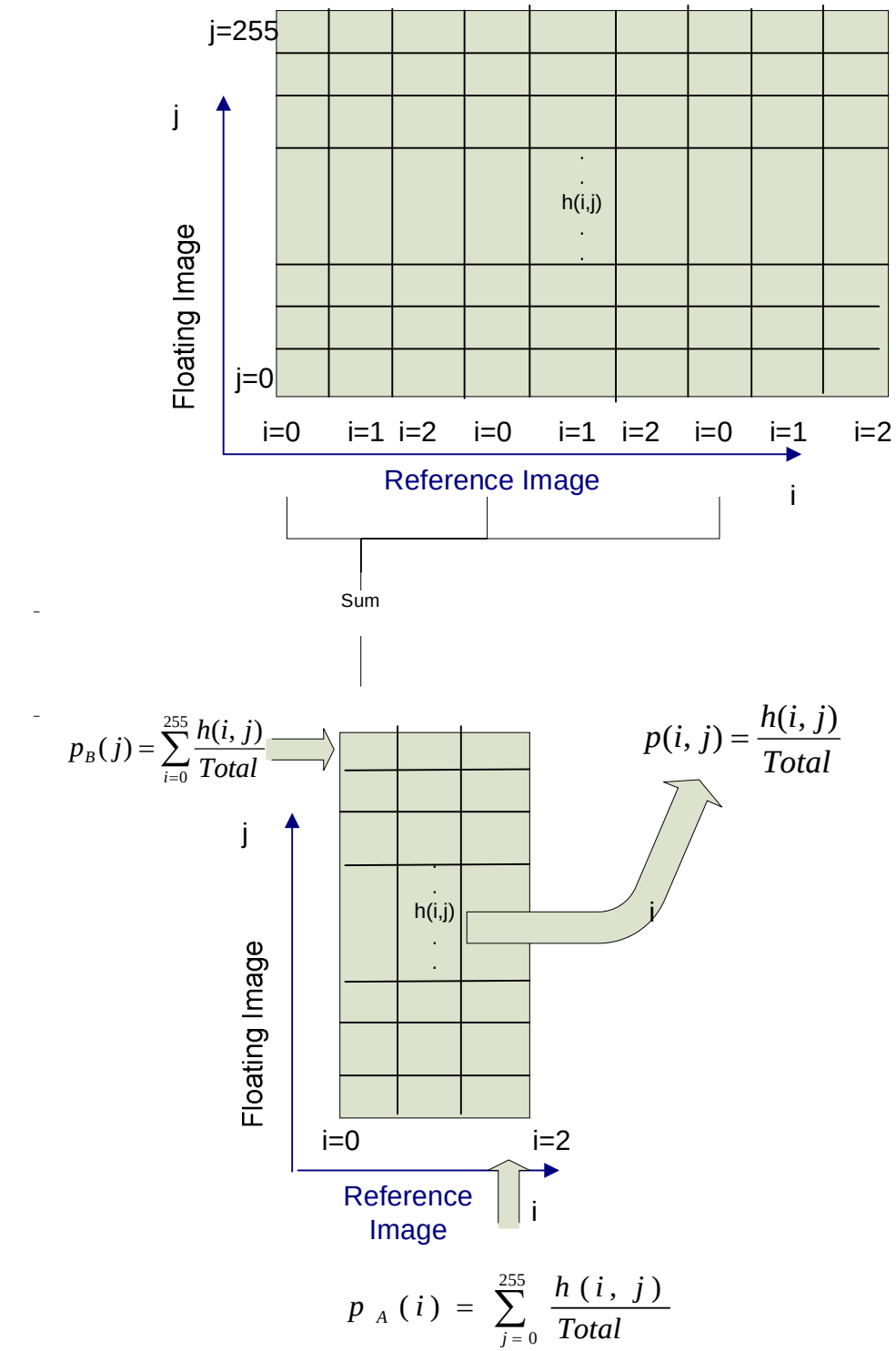


Fig 5.4 Mutual Histogram assigned to a block is further split up to have more threads active in a block

We use the following algorithm for assigning intensities to each block.

1) Calculate the histogram of the reference image
2) Find the total number of voxels that have to be processed
3) Find the average number of voxels to be processed by a thread in a block Avg Voxels=Total/(Total Number of Blocks x Threads Per Block)
4) Starting at the first block, assign to it intensities that it will process while keeping track of the maximum of the number of voxels of the intensity levels assigned to the block (MaxVox). After each intensity is assigned to the block compute the Voxels per thread for that block (VoxperThread[i]). This is computed using MaxVox/Ntimes where Ntimes is the number of times we can split the currently assigned intensity levels so that it is still below 10. For example when intensity levels is 3, Ntimes is 3 since we can split 3 intensity levels 3 times and still be below 10. When voxels per thread for the block (VoxperThread [i]) > Avg Voxels, we start assigning intensity levels to the next block but distribute the remaining voxels to be processed among the remaining threads. However, if there are more intensities left to be assigned than there are remaining threads then we do not move on to assign intensities to the next block but we keep on assigning intensities to the current block till we are left with equal number of remaining intensities and threads.
5) Repeat Step 4 until all the intensity levels have been assigned to all the blocks

We summarize the steps involved in computing MI using the bit slicing idea on the GPU

1) Pre-process the reference image and assign voxels to each thread in a block on the GPU based on the intensities that each block will be processing. The number of voxels processed by each thread needs to be load balanced and thus the number of intensities of the reference image that each block processes will need to be assigned dynamically as per the algorithm we defined earlier
2) Start the registration process
3) For each iteration of the registration process, the mutual histogram is computed and as shown in Fig. 5.e, we get a part of the mutual histogram on each block
4) MI computation follows from equations 1, 2 and 3 of chapter 3. Each block computes m_k which is necessary for computing H(A,B), e_k which is necessary for computing H(A) and $l_k(j)_{j=0}^{255}$ which is necessary for computing $p_B(j)_{j=0}^{j=255}$
5) Each block stores the following in global memory. Here 'k' denotes the block number
    i. $Total_k$
    ii. m_k
    iii. e_k
    iv. $l_k(j)_{j=0}^{255}$
6) Another kernel call sums up the results stored in global as described below:
    i. $Total_k$ to get Total

ii. m_k and uses equation (1) to get H(A,B)

iii. e_k and uses equation (2) to get H(A)

iv. $l_k(j)_{j=0}^{255}$ to get $p_B(j)_{j=0}^{j=255}$ using equation (4) and finally uses equation (3) to get H(B)

7) MI is computed as H(A)+H(B)-H(A,B)

Results

	#1	#2	#3
Single CPU Time (s)	119	151	349
GPU Time (s)	19.8	26	53
Speedup	6	5.9	6.58

Table 5.1 Comparison of CPU and GPU execution times for rigid registration using mutual information computed from mutual histogram having floating point counters.

	#1	#2	#3
Single CPU (mm) Translation (Tx,Ty,Tz) Rotation(Rx,Ry,Rz)	 5.52, 36.57, 16.42 0.58, -1.19, -5.28	 17.82,1 3.19, 14.37 -4.03, -0.98, -0.47	 -7.68, -19.42, -6.64 6.76, 7.24, 12.69
GPU (mm) Translation (Tx,Ty,Tz) Rotation(Rx,Ry,Rz)	 5.52, 36.57, 16.42 0.58, -1.19, -5.28	 17.82,1 3.19, 14.37 -4.03, -0.98, -0.47	 -7.68, -19.42, -6.64 6.76, 7.24, 12.69

Table 5.2 indicates that registration using the bit-slicing algorithm for computing the mutual information on the GPU recovers the same transformation parameters as the rigid registration process on a single processor.

Discussion

We have been able to implement computation of mutual information from the mutual histogram using PV interpolation. This has been made possible by the bit-slicing algorithm which allows us to maintain only a part of the mutual histogram in the

shared memory of each block. We see from Table 5.2 that our implementation recovers the same transformation as the CPU version of the rigid registration algorithm. We however were limited to floating data type as the 8800 GTX GPU does not support double data types. We see from Table 5.1 that we get a speedup of approximately 6. In [10], they were able to report a speedup of the order of four but then they were limited to 10000 bins and could not implement PV interpolation. However, our implementation is for 65536 bins. An avenue worth exploring is to try this implementation on a GPU that offers atomic updates on the hardware. An implementation with double data types for the mutual histogram can also be explored on a supporting GPU.

Chapter 6: Conclusions

We have seen that the 32 processor cluster is able to reduce the time of registering two 256x256x256 CT images from 2 hours to 7.5 minutes. We get a speedup of approximately 16. We had indicated that a dynamic algorithm for assigning subvolumes to idle processors in the cluster will help improve speedup by better load balancing.

By dividing the image based on its histogram rather than a spatial subdivision we showed how the bit-slicing algorithm allows speeding up of mutual information computation by reducing the communication time. The bit-slicing algorithm is useful for scaling up the computation of mutual histogram to many processors. We see that for 32 processors in the cluster, the communication time is only 0.7% of the total processing time. However, for computation of mutual information by image subdivision the communication time is 23% of the computation time for 32 processors. With increasing processors the communication will become much more significant in the latter case. The bit-slicing algorithm, on the other hand, will allow the number of processors to be scaled up much higher. Taking the bit-slicing algorithm onto a cluster with many more processors is one avenue that can be explored.

An application of the bit-slicing algorithm on the GPU has also been demonstrated. While previous investigators were unable to implement mutual information computation with PV interpolation for sufficient number of bins, the bit-slicing algorithm has made this possible for us. Our implementation has given a speedup of

six but has been limited by the lack of hardware support for atomic updates. Investigation of the bit-slicing algorithm on another GPU board with hardware support of atomic updates will be necessary to judge how much speedup can be obtained. The bit-slicing algorithm can also be used to implement computation of mutual information from mutual histogram having memory locations of type double on a supporting GPU board.

Bibliography

[1] F. Maes, A. Collignon, D. Vandermeulen, G. Marchal, and P. Suetens, Multimodality image registration by maximization of mutual information, IEEE Trans. Med. Imag., vol. 16, no. 2, pp. 187–198, Apr.1997.

[2] W.M. Wells, P. Viola, H. Atsumi, S. Nakajima, R. Kikinis, Multi-modal volume registration by maximization of mutual information, Medical Image Analysis 1 (1996) 35-51

[3] 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.

[4] V.Walimbe and R. Shekhar, "Automatic elastic image registration by interpolation of 3-D rotations and translations from discrete rigid-body transformations," Med. Image Anal., vol. 10, pp. 899–914, 2006.

[5] D. Rueckert, L.I. Sonoda, C. Hayes, D.L.G. Hill, M.O. Leach, D.J. Hawkes, Nonrigid registration using free-form deformations: application to breast MR images, IEEE Trans. Med. Imaging 18(8) (1999) 712–721.

[6] O.Dandekar and R. Shekhar, FPGA-Accelerated Deformable Image Registration for Improved Target-Delineation During CT-Guided Interventions, IEEE Transactions on Biomedical Circuits and Systems, vol. 1, no. 2, June 2007

[7] F. Ino, K. Ooyama, and H. Kenichi, A data distributed parallel algorithm for non-rigid image registration, Parallel Computing, vol. 31, pp.19–43, 2005

[8] Victor Podlozhnyuk, 64-bin Histogram, http://developer.download.nvidia.com/compute/cuda/sdk/website/projects/histogram64/doc/histogram64.pdf

[9] NVIDIA CUDA Programming Guide, ver 1.1, http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf

[10] R. Shams and N. Barnes. Speeding up mutual information computation using Nvidia CUDA hardware, Digital Image Computing Techniques and Applications, pages 555–560, 2007

[11] R. Shams and R. A. Kennedy, Efficient histogram algorithms for NVIDIA CUDA compatible devices, Proc. Int. Conf. on Signal Processing and Communications Systems (ICSPCS), Gold Coast, Australia, Dec. 2007, pp. 418-422

[12] MPICH2 User's Guide, ver 1.0.5, http://www.mcs.anl.gov/research/projects/mpich2/documentation/files/mpich2-doc-user.pdf

[13] Message Passing Interface (MPI), SP Parallel Programming Workshop, http://www.mhpcc.edu/training/workshop/mpi/main.html

[14] C. Studholme, D. L. G. Hill, and D. J. Hawkes, “An overlap invariant entropy measure of 3D medical image alignment,”Pattern Recognit., vol. 32, no. 1, pp. 71–86, 1998

[15] A. Collignon, F. Maes, D. Delaere, D. Vandermeulen, P. Seutens, and G. Mar.al, Automated multimodality image registration using information theory, in Information Processing Medical Imaging: Proc. 14th Int. Conf. IPMI’95, 1995, pp. 263–274

[16] Krucker, J.F., LeCarpentier, G.L., LeCarpentier, G.L., Fowlkes, J.B., Carson, P.L., 2002, Rapid elastic image registration for 3-D ultrasound, IEEE Trans. Med. Imaging 21, 1384–1394

[17] A. Collignon, D. Vandermeulen, P. Suetens, and G. Marchal, "3D multimodality medical image registration using feature space clustering," in Proc. 1st Int. Conf. Computer Vision, Virtual Reality and Robotics in Medicine; Lecture Notes in Computer Science 905, N. Ayache, Ed. New York: Springer-Verlag, Apr. 1995, pp. 195–204.

[18] J. B. West, J. M. Fitzpatrick, M. Y. Wang, B. M. Dawant, C. R. Maurer, Jr., 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.. 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, Jr., M. Pollack, C. A. Pelizzari, R. A. Robb, D. Hanson, and R. P. Woods, "Comparison and evaluation of retrospective intermodality image registration techniques," J. Comput. Assisted Tomogr., vol. 21, pp. 554–566, 1997

[19] Yuping Lin, Medioni G, Mutual information computation and maximization using GPU, Computer Vision and Pattern Recognition Workshops, 2008. CVPR Workshops 2008. IEEE Computer Society Conference on 23-28 June 2008 Page(s):1 -6

[20] S. Warfield, F. Jolesz, R. Kikinis, A high performance computing approach to the registration of medical imaging data, Parallel Computing 24 (9/10) (1998) 1345–1368

[21] C. Studholme, D. L. G. Hill, and D. J. Hawkes, “An overlap invariant entropy measure of 3D medical image alignment,” Pattern Recognition, vol. 32, no. 1, pp. 71 86, 1999

[22] Pluim, J.P., Maintz, J.B.A., 2000. Interpolation artefacts in mutual information based image registration. Comput. Vis. Image Understanding 77, 211–232

[23] Victor Podlozhnyuk, Histogram Calculation in CUDA, http://developer.download.nvidia.com/compute/cuda/sdk/website/projects/histogram256/doc/histogram.pdf