Multimodal Image Registration Using GPU Parallel Computing Technology

by

Xiaolu Sun

Supervisor: Professor Justin W.L. Wan

A research paper presented to the University of Waterloo in fulfillment of the project requirement for the degree of Master of Mathematics in Computational Mathematics

Waterloo, Ontario, Canada, 2010

Copyright Xiaolu Sun 2010

I hereby declare that I am the sole author of this research paper. This is a true copy of the research paper, including any required final revisions, as accepted by my examiners.

Acknowledgements

I would like to thank Professor Justin W.L. Wan for all of his guidance with this project. I would also like to thank Tiantian Bian for his great help.

Table of Contents

List of Figures

List of Tables

1 Introduction

This research project studies the parallel computing technique offered by the graphics processing unit (GPU), and uses it to accelerate the computation of image registration. Image registration is a process that aligns two images so that the point in one image corresponds to the same anatomical point in the other. It is a key part in the medical imaging analysis. Medical images are often taken at different time and places, resulting in varying frame of references for the same part of the human body in the images. As shown in Figure 1-1, a patient's brain appears at the upper part in one image, whereas the same brain appears at the lower right part in the other image that is taken at a different time. To better detect the change of the patient's brain over time, it is necessary to move one image to align with the other so that the differences between the two images can be easily identified.

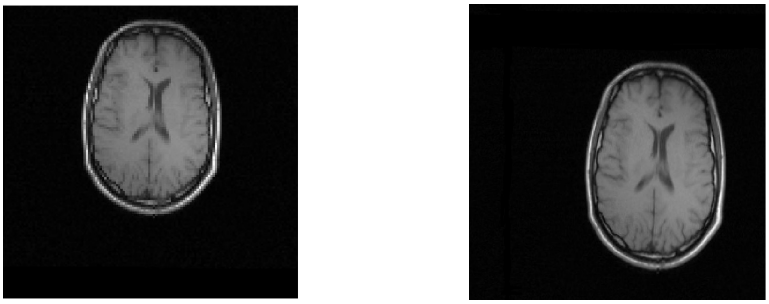


Figure 1-1 Images of brain need to be aligned. The brain is located at the top part of the image (Left) and located at the bottom right of the image (Right) from the Retrospective Image Registration Evaluation project.

The first challenge in image registration is that the modalities of the medical images could be different. Modality is the type of equipment used to acquire images of the body. Take the image of the brain as an example again, the X-ray computed tomography (CT) [1] helps to detect the fracture along the contour of the brain skull while the magnetic resonance imaging (MRI) [2] is more useful to provide greater contrast between the different soft tissues within the brain skull. Sometimes using the same equipment but with different parameters could produce multi-model images as well, such as T1-weighted MRI, T2-weighted MRI and proton density (PD)-weighted MRI [3]; see Figure 1-2. The intensities between two images of different modalities are quite different so some common measure used in monomodal image registration cannot be used here. This project considers mutual information as the similarity measure for image registration of

multimodal images [4]. The mutual information and joint entropy are further explained in Section 2.1.

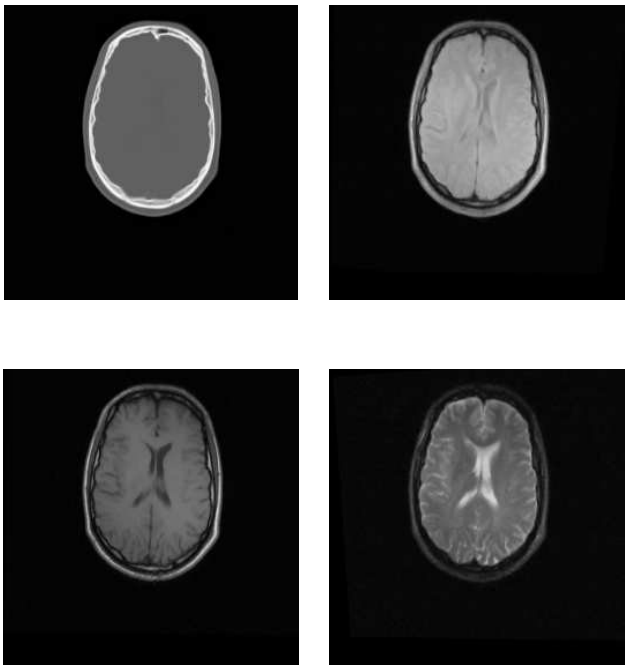


Figure 1-2 Medical Images of various modalities: CT (top left), T1-weighted MRI (bottom left), T2-weighted MRI (top right) and PD-weighted MRI (bottom right) from the Retrospective Image Registration Evaluation project.

Another challenge in image registration is that the size of medical images is usually large and hence the computational time for the registration process is long. This is especially not desired when the 3- dimensional images (3D images) of certain organ of a patient are created constantly and need to be registered within a very short span of time, such as at the brain surgery where the surgeon has to keep track of the patient's brain in the real-time manner. The size of the data contained in the 3D image is particularly large. For example, a 3D image of resolution 256×256×64 has more than 4 million voxels. If the computation in the registration process is performed on one voxel at a time in a sequential fashion, it will take a substantial amount of time.

In order to reduce the computational time in image registration, this project tries to perform the computation in parallel using the general purpose graphics processing unit (GPGPU) [5]. GPGPU is the graphics processing unit used to perform general scientific computation. It has substantially more processing units than a CPU. Thus it is able to perform parallel computation with every unit working at the same time. This project, therefore, transforms most of the sequential computation in image registration to parallel so that it becomes well-suited for the GPU computing. The Compute Unified Device

Architecture (CUDA), the parallel computing architecture developed by NVIDIA, is used in this project to perform those parallel computations on NVIDIA's supercomputing device Tesla C1060 (the GPGPU).

This paper is organized as follows: Section 2 provides the background information on mutual information and joint entropy (Section 2.1), the interpolation and partial volume registration models (Section 2.2 and Section 2.3), and the parallel computing architecture CUDA (Section 2.4). Section 3 explains the implementation of the interpolation and partial volume models to register 2D (Section 3.1 and 3.2) and 3D (Section 3.3) images using CUDA. Section 4 presents the numerical results where the performance gain using GPU is measured against using the CPU. Section 5 is the conclusion of this paper.

2 Background

Image registration is a fundamental task in medical imaging analysis. By aligning two images, medical practitioner can conveniently combine the information contained in the individual medical images, thus facilitating the diagnosis of the patients. In many cases, however, the medical images that need to be aligned are created by different sensors, such as CT and MRI. The multimodal images are more difficult to register than the monomodal images because the same part of the image may have different intensity values for different modalities and the simple sum of squared difference measure is not applicable. Numerous registration methods have been proposed by the researchers such as stereotactic frame-based registration [6], anatomical point landmark-based registration [7] and surface-based registration [8]. However, stereotactic frame-based registration although precise is cumbersome to implement. If landmarks are found by experts, then computation is small but method is laborious. Surface-based registration is not suitable for multimodal images [9]. This project employs the intensity-based registration method since it is accurate and well suited to register multimodal images.

The following Section 2.1 explains several key concepts in the intensity-based image registration, followed by the Sections 2.2 and 2.3 discussing the interpolation model and partial volume model and then Section 2.4 introduces the background of parallel computing technology which will be used to implement the intensity-based image registration.

2.1 Fundamental concepts in image registration

In image registration, one image F, called the template image, has to be transformed to align with the other fixed image G, called the target image. The transformation could be rigid, affine, or curved. Rigid transformation includes translation and rotation only; affine transformation considers upward or downward scaling; and curved transformation allows translation and rotation while also permits mapping straight lines to curves [10]. This project focuses on rigid transformation, which is defined by T_{β}, where β is the transformation parameter that includes the translations in x and y directions and the degree in rotation. The image registration process is to find T_{β} so that some similarity measure is optimized. In this project, the measure is chosen to be the mutual information *MI*. Thus image registration can be formulated as an optimization problem:

$$\max_{\beta} MI\left(F\left(T_{\beta}\right), G\right) \tag{1}$$

where $F(T_\beta)$ is the template image transformed by the rigid parameter β.

2.1.1 Individual and joint histogram

Before going into details of the mutual information based registration method, the concepts of the individual and joint histogram of an image need to be first explained. Histogram of an image is the display of the frequency counts of the image intensity values. The number of bins will decide how many equally spaced intervals exist within the range of the intensity values. For example, suppose image A is of resolution 4×4 with the intensity value ranging from 0 to 7 and the number of bins is 2. Then the intensity values 0 to 3 are grouped into bin index 1 and the intensity values 4 to 7 are grouped into bin index 2. Image A and its bin index are shown in Figure 2-1. In this example, the frequency count of bin index 1 is four and bin index 2 is twelve. The individual histogram, or simply histogram, of image A is shown in Figure 2-2.

0	0	0	0
0	8	2	0
0	7	5	6
0	2	3	2

0	0	0	0
0	1	0	0
0	1	1	1
0	0	0	0

Figure 2-1 Image A with intensity values (left) and its corresponding bin index at each pixel (right).

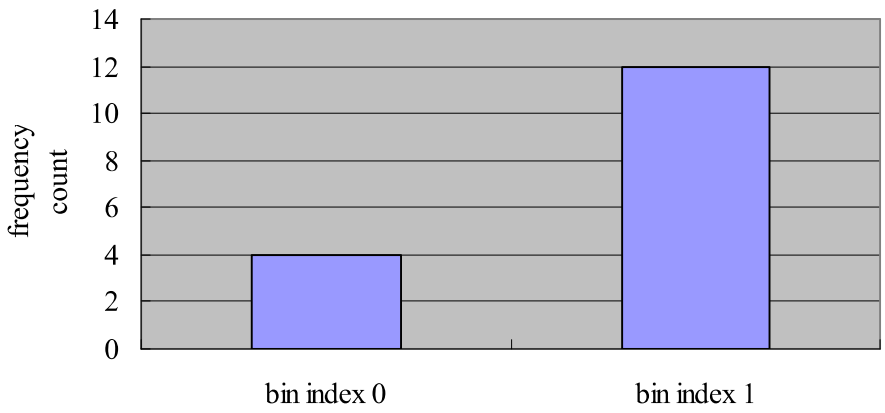


Figure 2-2 Individual histogram of the image A

Suppose we have a second image B of the same resolution as shown in Figure 2-3. The joint histogram between images A and B is defined such that the bin indices of image A and B at the same location form pairs and the number of times of these pairs found is counted. For example, the top left pixel of image A and B would form a bin index pair (0, 1), where 0 is the bin index from image A and 1 is the bin index at the same location from image B. We continue to count the pairs formed in this way throughout both images. Finally, the bin index pair (0,1) appears 4 times, (1,0) appears 0 times, (0,0) appears 8 times, and (1, 1) appears 4 times. In order to represent this result in a systematic way, we create a 2 by 2 histogram matrix (2 is the number of bins) and treat the bin index of the image A as the row index of the joint histogram matrix and bin index of the image B as the column index of the joint histogram matrix. Thus, the bin index pairs, (0,1), (1,0), (0,0) and (1,1), are the location in this histogram matrix. Incrementing the histogram matrix by 1 for every pair of the bin indices will eventually produce the joint histogram matrix, as shown in Figure 2-4.

4	0	0	7
3	6	2	0
4	7	4	6
1	2	7	2

1	0	0	1
0	1	0	0
1	1	1	1
0	0	1	0

Figure 2-3 Image B with intensity values (left) and its corresponding bin index at each pixel (right).

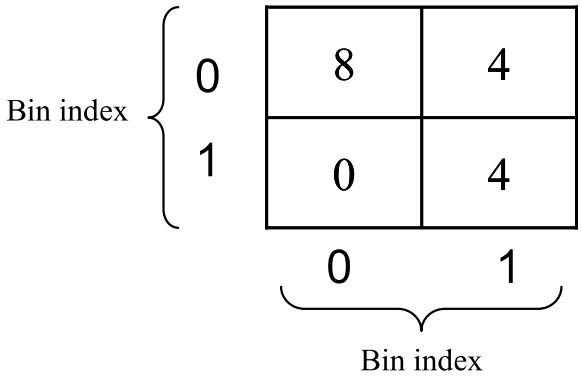


Figure 2-4 Joint histogram matrix of image A and B.

2.1.2 Individual and joint probability density distribution

The individual and joint probability density distributions are constructed by normalizing the histograms defined in the previous section. More precisely, the probability density distribution for image A is that the bin index 1 accounts for 4/16 and the bin index 0 accounts for 12/16 of all the data. Similarly, there are 16 bin index pair in the joint histogram of images A and B, out of which bin index pair (0,1) accounts for 4/16, (1,0) accounts for 0/16, (0,0) accounts for 8/16, and (1, 1) accounts for 4/16. The resulting joint probability distribution matrix is shown in Figure 2-5.

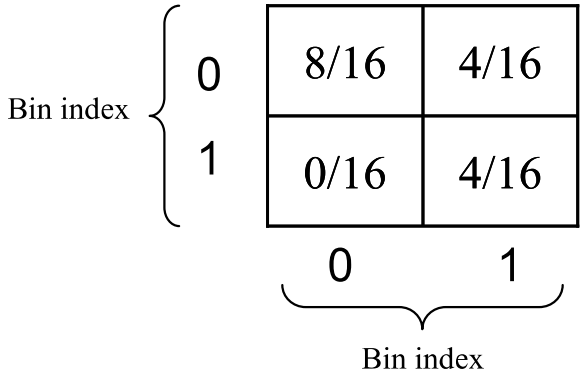


Figure 2-5 Joint probability density matrix of image *A* and *B*.

2.1.3 Mutual Information

Mutual information (MI) is a fundamental concept in information theory that measures the mutual dependence of the two random variables [11]. Treating the image intensities as random variables, the intensity-based method uses MI to measure the

similarity of two images being registered. MI can be thought as the information in image A which is also contained in image B, or vice versa. When such information that contained in both image reaches the maximum, the two images are successfully registered. MI is defined as [9]:

$$I(A,B) = H(A) + H(B) - H(A,B) \tag{2}$$

where I is the mutual information of the two images A and B, $H(A)$ and $H(B)$ are the Shannon entropy of images A and B, respectively, and $H(A,B)$ is their joint entropy. Entropy measures the dispersion of the image [9]. As the two images are misaligned the joint entropy $H(A,B)$ will increase resulting in a smaller mutual information $I(A,B)$ based on equation (2). The maximum mutual information hence is found when the joint entropy $H(A,B)$ is minimized. The entropies are defined as follows [9]:

$$H(A) = -\sum_{a} P_A(a) \log P_A(a) \tag{3}$$

$$H(B) = -\sum_{b} P_B(b) \log P_B(b) \tag{4}$$

$$H(A,B) = -\sum_{a,b} P_{AB}(a,b) \log P_{AB}(a,b) \tag{5}$$

where a and b are the image intensities, P_A and P_B are the probability density distributions of images A and B, and P_{AB} is the joint probability density distribution of the two images.

2.2 Interpolation model

During image registration, the target image G is fixed, and the template image F is translated and/or rotated to align with the target image. Since the joint histogram is constituted by the bin index pairs of the target and template images at the same pixel locations and the target image is fixed, the intensity values of the template image are interpolated at the pixel locations using the intensity values of the template image after transformed to the new positions [12]. The interpolation could be bilinear or trilinear, depending on the images being registered are 2-dimensional or 3-dimensional, respectively.

The bilinear interpolation for the 2-dimensional case is illustrated in Figure 2-6 below. $I(x_i-\triangle_i, y_j-\triangle_j)$, $I(x_i-\triangle_i+1, y_j-\triangle_j)$, $I(x_i-\triangle_i+1, y_j-\triangle_j-1)$, $I(x_i-\triangle_i, y_j-\triangle_j-1)$ are the intensity values of the template image F after transformation, where i and j are the pixel locations. $I(x_i, y_j)$ is the new interpolated image intensity after transformation at (x_i, y_j). It is

computed by adding the contributing weights as follows:

$$w_{00} = \Delta_i \times \Delta_j \tag{6}$$

$$w_{10} = \Delta_i \times (1 - \Delta_j) \tag{7}$$

$$w_{01} = (1 - \Delta_i) \times \Delta_j \tag{8}$$

$$w_{11} = (1 - \Delta_i) \times (1 - \Delta_j) \tag{9}$$

$$I_{(x_i, y_j)} = w_{00} \times I_{(x_i - \Delta_i + 1, y_j - \Delta_j - 1)} + w_{01} \times I_{(x_i - \Delta_i, y_j - \Delta_j - 1)} + w_{10} \times I_{(x_i - \Delta_i + 1, y_j - \Delta_j)} + w_{11} \times I_{(x_i - \Delta_i, y_j - \Delta_j)} \tag{10}$$

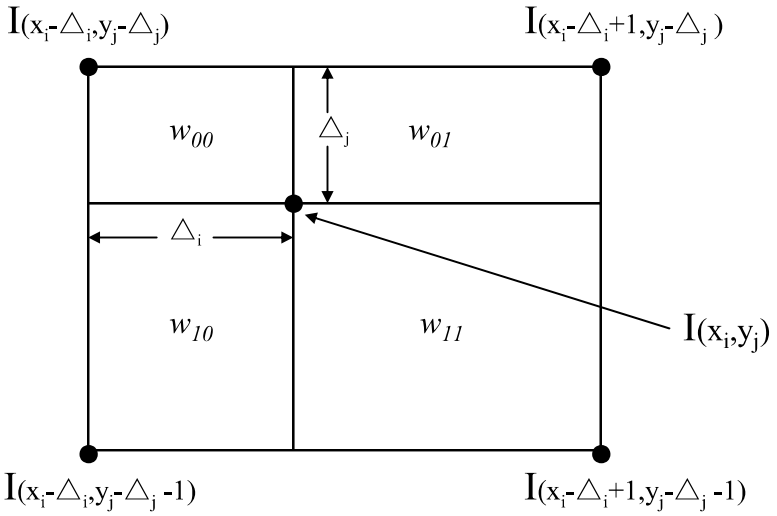


Figure 2-6 Bilinear interpolation of template image F with translation △i and △j.

By equations (3), (4), and (5), to compute the entropies of the images, the probability distributions P_A, P_B and P_{AB} have to be computed based on the corresponding histograms. We divide the intensity value of both the transformed template image F and the target image G into the same number of bins, making the square shaped histogram matrix of size bins×bins. When constructing the joint histogram matrix, the bin index for the intensity value of the template image F, denoted by *bin_index_f,* is thought as the row index of the joint histogram matrix, and the bin index for the intensity values of the target image G, denoted by *bin_index_g* , is considered as the column index of the histogram matrix.

The elements in the histogram matrix, denoted by $h_{f,g,}$ are obtained by forming the pair (*bin_index_f, bin_index_g*) and incrementing the histogram matrix by 1 at each such

location (*bin_index_f, bin_index_g*) found:

$$h_{f,g} = h_{f,g} + 1 \quad \text{at location } (bin\_index\_f,\ bin\_index\_g) \tag{11}$$

The joint probability density matrix P_{FG} is completed by dividing $h_{f,g}$ by the total number of pairs. The individual probability distributions for each image, P_F and P_G, are found by summing over the rows and columns of the joint probability density matrix, respectively.

2.3 Partial volume model

The difference between the partial volume and interpolation model is that the intensity value of the template image F is not interpolated after transformation, but remains the same [9]. Thus the partial volume model constructs the joint histogram matrix differently. Assume one pixel of the template image F is translated by △i and △j in the x and y direction respectively as shown in Figure 2-7 and the bin index of that pixel is *bin_index_f*. Its nearest four neighboring pixels on the target image G and their bin indices are *bin_index_g_1, bin_index_g_2, bin_index_g_3, and bin_index_g_4,* also shown in Figure 2-7. Similarly, the bin index of the template image F and the target image G are the row and column index of the histogram matrix, respectively. Next, we increment the histogram matrix at (*bin_index_f, bin_index_g_1*) by weight $w_{11,}$ at (*bin_index_f, bin_index_g_2*) by weight $w_{10,}$ at (*bin_index_f, bin_index_g_3*) by weight w_{01}, and at (*bin_index_f, bin_index_g_4*) by weight $w_{11,}$ according to equations (12), (13), (14) and (15). Note that in the partial volume model, we increment the histogram by fractional weights rather than the integral 1. The same procedure is repeated for all the pixels of transformed template image F, until the joint histogram is obtained.

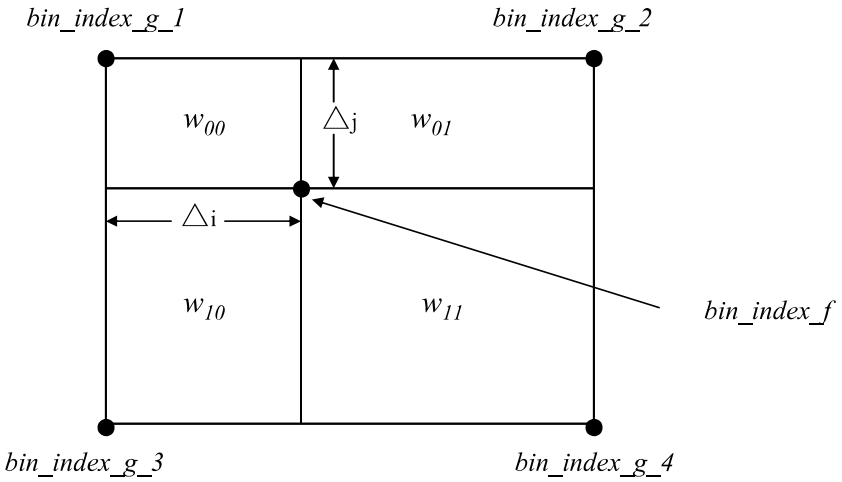


Figure 2-7 Joint histogram construction in the partial volume model

$$h_{f,g} = h_{f,g} + w_{11} \quad \text{at location } (bin\_index\_f,\ bin\_index\_g\_1) \tag{12}$$

$$h_{f,g} = h_{f,g} + w_{10} \quad \text{at location } (bin\_index\_f,\ bin\_index\_g\_2) \tag{13}$$

$$h_{f,g} = h_{f,g} + w_{01} \quad \text{at location } (bin\_index\_f,\ bin\_index\_g\_3) \tag{14}$$

$$h_{f,g} = h_{f,g} + w_{00} \quad \text{at location } (bin\_index\_f,\ bin\_index\_g\_4) \tag{15}$$

The subsequent individual and joint probability density distribution, and entropy and mutual information are obtained in the same way as in the interpolation model.

2.4 Parallel computing technology of GPU

The image registration process is often time-consuming, especially for the 3-dimensional image registration where the computation becomes exceedingly intensive. This project employs graphics processing unit (GPU) to accelerate the image registration process. GPU devotes substantially more hardware resources to data computation than data caching and flow control. Therefore, the multithread and multi-processing core GPU is well-suited for compute-intensive and parallel computation. For instance, the NVIDIA Tesla C1060 GPU has 240 processing cores and could reach 933GFLOP/s in single precision calculations [14], while the Intel i7-950 has 4 cores with peak performance of 42.56GFLOP/s in double precision calculations [15]. It is worth noting that while GPU is highly efficient in single precision calculations; its performance is not equally impressive in double precision calculations. As an example, the Tesla C1060 peak performance is only 78GFLOP/s in double precision calculations [14]. In our project, only single

precision calculation is required thus we could fully utilize the computing power of the GPU. The background information contained in this section are from NVIDIA CUDA Programming Guide [16].

2.4.1 CUDA Programming Model

The Compute Unified Device Architecture (CUDA) is the parallel computing platform developed by NVIDIA that enables GPU to carry out various numerical applications. CUDA allows programmers to use 'C with NVIDIA extensions (or CUDA C)' as the programming language. A typical CUDA program can be viewed as the collaboration of the CPU and the GPU. It is composed of the host code, usually the regular C code, executed sequentially on the CPU and the device code, mainly the kernel functions, executed on the GPU. This is illustrated in Figure 2-8.

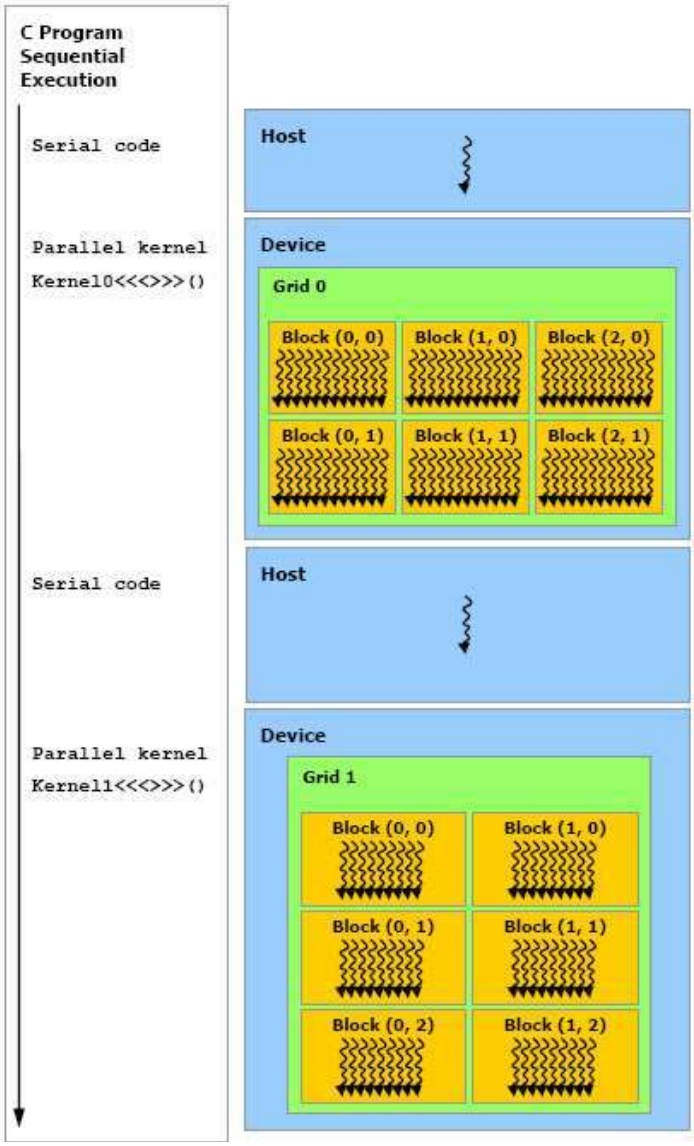


Figure 2-8 Execution of a typical CUDA program from NVIDIA's website.

2.4.1.1 CUDA kernel function

In CUDA, the kernel function is a C-like function executed on the GPU by many CUDA threads simultaneously. It has the declaration specifier '**__global__**'. When the kernel is called, the programmer needs to specify the number of CUDA threads active for this kernel function, using the **<<<…>>>** execution configuration syntax. The major difference between a GPU CUDA kernel function and the standard CPU C function is that the computations contained in the kernel function are simultaneously executed by multiple threads in parallel whereas the computations in the CPU function are

sequentially executed.

2.4.1.2 CUDA thread hierarchy

Each thread that executes the kernel function needs to have an ID to decide what data it will work on. In order to achieve this, CUDA is able to allow programmer to group multiple threads into blocks and further group multiple blocks into grids. The block can be 1-dimensional, 2-dimensional, or 3 dimensional, while the grid could only be either 1-dimensional or 2-dimensional. Their relation is illustrated in Figure 2-9.

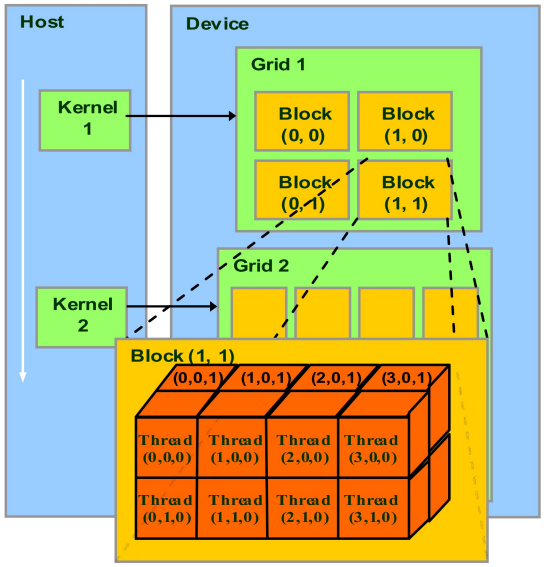


Figure 2-9 CUDA thread, block and grid from NVIDIA's website.

The block size and grid size specified in <<<…>>> during the kernel function call have special CUDA type **dim3**. CUDA provides the built in variables **threadIdx.x**, **threadIdx.y**, **threadIdx.z**, for thread index in the x, y, and z directions, and **blockIdx.x**, and **blockIdx.y** for block index in the x and y directions. The block dimension is also given by the built-in variables **blockDim.x**, **blockDim.y**, and **BlockDim.z**. As an illustration, the following code specifies the number of thread, block and grid when invoking the kernel function:

```
dim3 gridsize (2,2);
dim3 blocksize(4,4,4);
kernel_function_name<<<gridsize, blocksize>>>(…);
```

Here, the grid is 2-dimensional with 2×2 blocks and each block is 3-dimensional with 4×4×4 threads. The **blockDim.x** and **blockDim.y** are both 2. The thread ID, **idx**, in each block can be calculated as follows:

$$idx = threadIdx.x + threadIdx.y \times blockDim.x + threadIdx.z \times blockDim.x \times blockDim.y \quad (16)$$

2.4.1.3 CUDA Memory Hierarchy

There are various types of memory specified in CUDA that can be accessed by threads during the execution of kernel functions. The global memory is the memory that can be accessed by all the blocks and grids while executing different kernel functions. The shared memory can only be accessed by the threads in one block and it has the same lifetime as that block. The memory is usually lost when the current kernel function is finished. Accessing the shared memory is much faster than accessing the global memory thus the program will become more efficient when the data is brought into shared memory. The shared memory and global memory is illustrated in Figure 2-10.

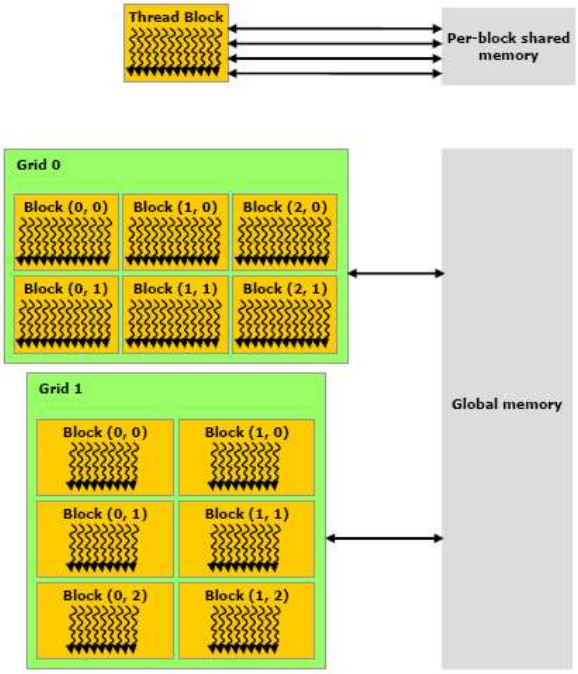


Figure 2-10 Global Memory and Shared Memory from NVIDIA's website

Another important memory type is the texture memory. It is a special type of GPU memory that makes texture data readily available to the rendering process. By passing the image (intensities) into the texture memory and then fetching the image back to the global memory, the image is automatically linearly interpolated. This project uses this feature to achieve better speed-up compared with regular image interpolation method.

Before using the texture memory, its texture reference needs to be set appropriately. It is declared as a structure in CUDA C as follows:

```
texture<Type, Dim, ReadMode> texRef;
```

In our project, **Type** is defined as float since we only deal with single-precision floating point calculation. **Dim** is set to 2 and 3 for the 2-dimensional and 3-dimensional images, respectively. **ReadMode** is chosen to be **cudaReadModeElementType**, which copies the original image pixel/voxel coordinates into the texture memory without normalizing it to [0.0 1.0].

After the declaration, the texture reference attributes need to be set for the **texRef**. They are **texRef.addressMode**, **texRef.filterMode**, **texRef.normalized**, and **channelDesc**, which are explained as follows:

texRef.addressMode specifies the boundary condition. It is set as **cudaAddressModeClamp** in our project. All the out-bound coordinates are clamped. For instance, if the coordinates is from 0 to 127, the value below 0 is set to 0 and the value above 127 is set to 127.

texRef.FilterMode specifies the interpolation method. We set it to **cudaFilterModeLinear**. If the texture dimension is 2, it is bilinear interpolation, while if the texture dimension is 3, it is trilinear interpolation.

texRef.normalized specifies whether the coordinates are normalized. In our project, we use original image coordinates thus it is set as false.

channelDesc is set to **cudaChannelFormatFloat** as the data feed into the texture memory is float type.

The detailed usage of texture memory in 2D and 3D image interpolation is covered in Section 3.1.

3 GPU Implementation

As discussed in the previous sections, transforming the sequential computation into parallel computation results in substantial increase in efficiency for the image registration process. This chapter explains the technique of using CUDA to invoke multiple threads on GPU to carry out the computation simultaneously for the 2D and 3D interpolation model and the 2D partial volume model in the image registration.

Both the interpolation and partial volume model use the steepest descent to optimize the mutual information measure by finding the minimum of the joint entropy. In our program, the steepest descent algorithm is executed in the sequential fashion on the host side (CPU) while the mutual information is computed in the parallel fashion on the device side (GPU). Since steepest descent method is one of the commonly used algorithms in intensity-based image registration and it runs on the host side, it is not the focus of the project thus the discussion of it is skipped in this chapter. Instead, the following sections are devoted to the parallel computation of mutual information and joint entropy on the GPU. The parallel implementation of the 2D interpolation model is first presented in Section 3.1, followed by the 2D partial volume model in Section 3.2 and the 3D interpolation model in Section 3.3.

3.1 2D Interpolation Model

To compute the mutual information and the joint entropy, the template image is first transformed through interpolation. Next, the joint histogram and joint probability density matrix are constructed between the transformed template image and the target image. Finally, the entropy and mutual information are computed based on the joint probability density matrix. Below we focus on the parallel implementation of these steps using the GPU.

3.1.1 Image Interpolation

When interpolating the template image F at the new coordinates, new intensity values are found through interpolation. We assume that the images have wide enough margins (the dark regions around the brain as shown in Figure 1-2) so that during the interpolation it is not likely that the useful part of image will go out of boundary. The

texture memory of GPU is used to perform this task by exploiting the feature that the data fetched from the texture memory is automatically interpolated. This involves first passing the image to the texture memory and fetching the interpolated data from it. Unfortunately, such direct memory copy to the texture memory is not allowed in CUDA. The image F has to be initially copied into an opaque CUDA memory of type CUDA Array, and then this CUDA Array needs to be bound with the texture memory with appropriate texture memory references. To fetch the data, built-in CUDA function **tex2D** is invoked with the new coordinates. The output of **tex2D** is the interpolated image on the new coordinates and it is copied to the pre-allocated GPU global memory denoted by **B_o**.

3.1.2 2D CUDA array allocation

The following code declares **cuArray** of the type CUDA array:

```
cudaChannelFormatDesc  channelDesc = cudaCreateChannelDesc<float>();
cudaArray *cuArray =0;
cudaMallocArray(&cuArray, &channelDesc, f.width, f.height);
```

The data type the array holds is set by the **cudaChannelFormatDesc** to float. Then the **cuArray** is allocated by calling **cudaMallocArray** with specified channel format and size. The array's width is equal to the width of the template image F, **f.width**, and the array's height is equal to the height of image F, **f.height**.

3.1.3 Memory copy to CUDA array

Although the image is 2-dimensional, it is always stored in the 1D linear memory in the program by reading the intensity value from the top left to the bottom right. Copying the image F from the 1D linear global memory to the CUDA array, **cuArray**, is achieved by

```
cudaMemcpyToArray(cuArray ,0 ,0, d_f.elements,
f.width*f.height*sizeof(float), cudaMemcpyDeviceToDevice);
```

where **d_f.elements** is the float type pointer on GPU global memory holding the intensity values of image F before transformation. (Note that the letter "**d**" in **d_f.elements** indicates that the image is now stored on the **d**evice) **f.width*f.height*sizeof(float)** is the size in bytes the memory transferred and **cudaMemcpyDeviceToDevice** indicates that the transfer is from device to device, since the **d_f.elements** and **cuArray** are both sitting

on the device side (GPU).

3.1.4 Binding CUDA array to texture memory

After the pre-transformed template image F is copied into the **cuArray**, we set the texture memory reference and bind the **cuArray** with it. This is done by the following code:

```
/* ----- 2-Dimensional float type texture declaration -----*/
texture<float,2, cudaReadModeElementType> texRef;
/* ----- texRef setup ------*/
// boundary are clamped in x direction
texRef.addressMode[0] = cudaAddressModeClamp;
// boundary are clamped in y direction
texRef.addressMode[1] =cudaAddressModeClamp;
// bilinear interpolation
texRef.filterMode     = cudaFilterModeLinear;
// use original coordinates, without normalizing to [0,1]
 texRef.normalized    = false;
/* ----- Bind the array to the texture -----*/
cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc<float>();
cudaBindTextureToArray(texRef,cuArray, channelDesc);
```

cudaBindTextureToArray binds the CUDA array with the texture memory using the specified texture reference, **texRef**, and the channel format, **channelDesc**. The boundary conditions along both directions are clamped and the interpolation is set to bilinear.

3.1.5 Interpolating by calling tex2D

After allocating the CUDA array, **cuArray**, and also binding it with the texture memory, we are ready to call the built-in texture function **tex2D** alongside the new coordinates. The new texture coordinates **u** and **v** are calculated based on the rotation angle and the horizontal and vertical translation, **shift_x**, and **shift_y**. It is important to note that the data is stored at the “center” of the texture coordinates, thus the new texture coordinates need to be offset by 0.5 pixels in both horizontal and vertical directions. For example, if the original coordinates **x** is translated horizontally by an amount of **shift_x**, the new texture coordinate is:

```
u = x + shift_x + 0.5;
```

Next, copying the output of **tex2D** to the pre-allocated global memory **B_o** is done by:

```
B_o[…] = tex2D(texRef,u,v);
```

3.1.6 Joint probability density matrix construction

The joint probability density matrix, denoted by P_{FG}, between the transformed image F and the target image G is obtained by dividing the elements in the joint histogram matrix, denoted by *A*, by the summation of all elements in *A*. This is done by first transforming the intensity value of both images to index of bins. The index of bins of the template image F, denoted by *bin_index_f*, is also equal to the row index of the histogram matrix *A*, and the index of bins of the target image G, denoted by *bin_index_g*, is equal to the column index of the histogram matrix *A*. We then increment the element located at (*bin_index_f*, *bin_index_g*) in *A* by 1. However, similar to the way the 2D image is stored on the GPU, the matrix *A* is stored as a 1-dimensional linear memory. The 2-dimensional address (*bin_index_f*, *bin_index_g*) is hence transformed into the 1-dimensional linear memory address using the following formula:

$$\textit{1-dimensional addess in } A = \textit{bin\_index\_f} \times \textit{width of } A + \textit{bin\_index\_g} \quad (17)$$

Therefore, instead of counting how many times the pairs (*bin_index_f*, *bin_index_g*) appear, we count the appearance of the corresponding 1-D linear memory addresses. Once the histogram *A* is constructed, each element in *A* is divided by the sum of all the entries in *A*, resulting the joint probability density matrix.

For easy exposition, we assume that the resolutions of the template and target images are both 4×4 with their intensity value ranging from 0 to 7. The number of bins is chosen to be 2. The intensity values of the two images are shown below in Figure 3-1:

0	4	3	7
3	6	5	3
5	1	0	2
7	4	7	6

2	0	1	6
3	5	0	2
3	4	3	7
4	0	1	0

Figure 3-1 The intensity values of the template image F (left) and the target image G(right)

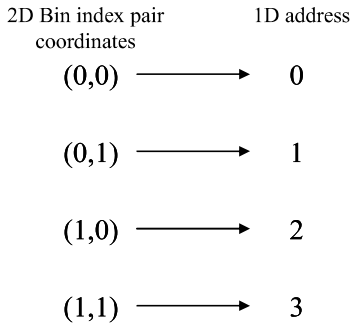
Since each image has 16 pixels, we invoke 16 threads. Each thread accesses one pixel of the image F, and converts it to the bin index simultaneously. The same for the target image G. Since the number of bins is 2, the intensity value will be divided into two levels, 0 and 1, with intensities from 0 to 3 grouped into bin index 0 and intensities from 4 to 7 grouped into bin index 1. The results are shown in Figure 3-2 below.

0	1	0	1
0	1	1	0
1	0	0	0
1	1	1	1

0	0	0	1
0	1	0	0
0	1	0	1
1	0	0	0

Figure 3-2 Bin index for the template image F (left) and the bin index for the target image G (right)

The bins indices of the template image F and the target image G give the row and column indices of the histogram matrix *A*, respectively. Since the matrix *A* is stored in the 1D linear memory, it is necessary to convert the 2D row-column coordinates into 1D memory address. To do so, we invoke 16 threads again, with each thread taking one bin index of the image F and one bin index at the corresponding location in the image G, and then computing the 1D address using formula (17). The matrix so created, denoted by *1D-address-matrix*, is shown in Figure 3-3 below.



0	2	0	3
0	3	2	0
2	1	0	1
3	2	2	2

Figure 3-3 Mapping from 2D bin coordinates to 1D-address-matrix

Each element in the *1D-address-matrix* represents a bin index pair between the template image F and the target image G. Also, the same element represents the address in the linear memory of the histogram *A*. The value of the histogram *A* at one address is determined by the number of times that address shows up in the *1D-address-matrix*. As

shown in Figure 3-3, address “0” shows up five times, address “1” shows up two times, address “2” shows up six times, and address “3” shows up three times. The histogram *A*, therefore, should look like the following in Figure 3-4:

Histogram *A*

Address	Contents
0	5
1	2
2	6
3	3

Figure 3-4 Histogram *A* stored in 1D linear memory

In order to generate such a histogram, ideally, we could invoke 16 concurrent threads, with each thread simultaneously reading one element from the *1D-address-matrix* (Figure 3-3) and then incrementing *A* at the corresponding location by 1. Unfortunately, this causes multiple threads, for instance the threads reading the address 0, to attempt to update the same location in the memory at the same time, resulting in data contention. This is not permitted in the GPU. To avoid such data contention, a large matrix denoted by *L_A* is created. The number of its rows is equal to the size of 1-D linear memory of the histogram *A* and the number of its columns is two times the size of the width of the *1D-address-matrix*. Hence, in our example *L_A* has 4 rows and 8 columns. Each column is treated as one copy of the histogram *A*. Next, we employ 8 concurrent threads to read the first two rows (each row has 4 elements in our example) of the *1D-address-matrix* and let each thread access one individual column of the matrix *L_A* and then increment the corresponding location by 1. There are 8 threads and 8 columns, with every thread going to the different columns, it is guaranteed they will not write to the same memory location at the same time thus data contention is avoided.

The same threads, then, will move to the next two rows of the *1D-address-matrix* and increment the *L_A* again at the corresponding locations in the same fashion until the whole *1D-address-matrix* is covered. In our example, the 8 concurrent threads need two iterations in total to cover the whole *1D-address-matrix*. The resulted *L_A* after each iteration is illustrated in Figure 3-5 and Figure 3-6.

Address in 1-D linear memory of histogram A								
0	1	0	1	0	1	0	0	1
1	0	0	0	0	0	0	0	0
2	0	1	0	0	0	0	1	0
3	0	0	0	1	0	1	0	0

Figure 3-5 *L_A* after the first iteration

Address in 1-D linear memory of histogram A								
0	1	0	2	0	1	0	0	1
1	0	1	0	1	0	0	0	0
2	1	1	0	0	0	1	2	1
3	0	0	0	1	1	1	0	0

Figure 3-6 *L_A* after the second iteration

3.1.7 Parallel Reduction

Each column of *L_A* can be viewed as one copy of the linear memory for the histogram *A* that contains only partial results. Furthermore, *L_A*'s row index coincides with the 1D linear memory address for the histogram *A*. Therefore, summing each row of *L_A* will produce the histogram *A*. Although the amount of computation to sum all the rows of *L_A* in the given example seems little, it may become formidable for the real size medical images. The technique of parallel reduction is hence used to compute the sum of each row simultaneously to improve efficiency.

This is done by creating multiple blocks with each block covering one row of *L_A*. The number of threads in each block is equal to the number of elements in each row of *L_A*. Our example requires four blocks with each block having eight threads, since *L_A* has four rows with eight elements in each row. Mirroring those four rows of *L_A*, four arrays are allocated in the shared memory (the number of array in shared memory is always equal to the number of blocks) and each row of *L_A* is copied into one of those

arrays. Faster memory access is thus possible as the data is now sitting on the shared memory. Next, the eight threads in each block get the eight elements in the corresponding array. Then summation is performed in pairs: The 5^{th} thread is added to the 1^{st} thread, 6^{th} thread is added to the 2^{nd} thread, 7^{th} thread is added to the 3^{rd} thread, and 8^{th} thread is added to the 4^{th} thread. Now the first four elements in each array contain the sum of the four pairs while the last four elements in the array bear no interest for us. We continue to reduce such row by adding the 3^{rd} thread to the 1^{st} thread and 4^{th} thread to the 2^{nd} thread, resulting in a virtually 2-element array, abandoning the other 6 elements. In the final round of reduction, the 2^{nd} thread is added to the 1^{st} thread and the sum of the 8-element array is now stored in the first thread (or the first address) in the array. The complexity of the parallel reduction for each row of the *L_A* is therefore $\log_2 N$, where N is the number of the elements in the row. Compared with the ordinary summation with complexity N-1, the benefit of parallel reduction is evident when N is large.

This parallel reduction is carried out in each row of *L_A* at the same time. In the end all the 1^{st} elements in every array are passed into the pre-allocated 1-D array in the global memory and this is the histogram matrix *A*. Since parallel reduction is performed at every row simultaneously, the efficiency is further improved on top of the reduced complexity.

After the joint histogram matrix *A* is obtained, we can easily calculate the probability density P_{FG} by dividing *A* by the sum of the entries in *A*. In the interpolation model, the sum of *A* is always equal to the number of pixels in the image. For instance, if the image is 512×512, the sum of *A* is 512^2. In our example, the sum is 16 since the image size is 4×4 and this can be confirmed by adding all the entries in the histogram *A* as shown in Figure 3-4. We could perform the division after *A* is obtained, or we could merge such division into the final round of parallel reduction. The latter is chosen since accessing the shared memory is much faster than accessing the global memory. The parallel reduction in each block, or each row of *L_A*, and the resulted probability density matrix is illustrated in Figure 3-7, Figure 3-8, Figure 3-9 and Figure 3-10.

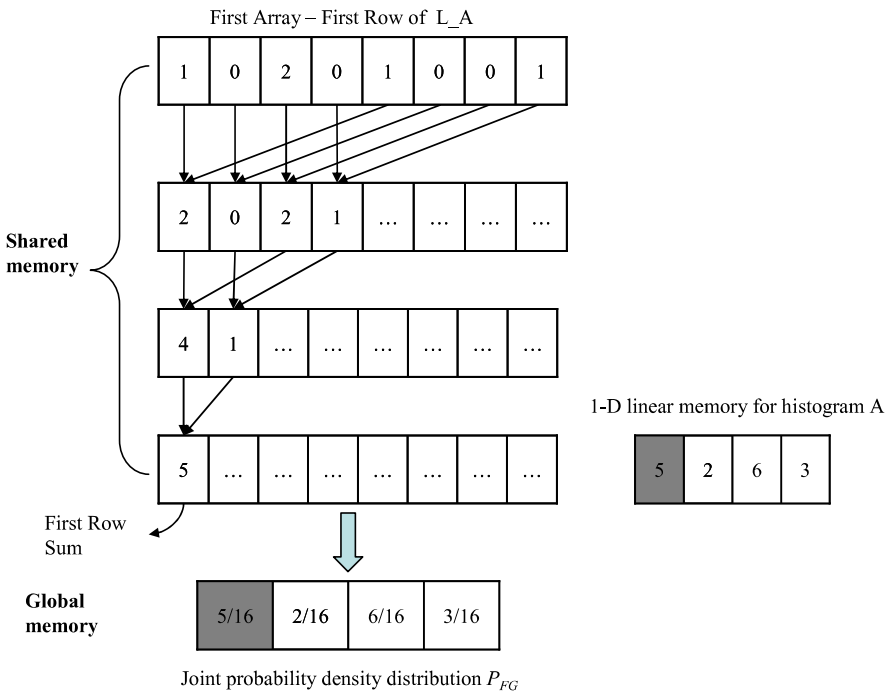


Figure 3-7 Parallel reduction of the first row of L_A to obtain the frequency count and probability density at the first address in the histogram A and the joint probability density P_{FG}

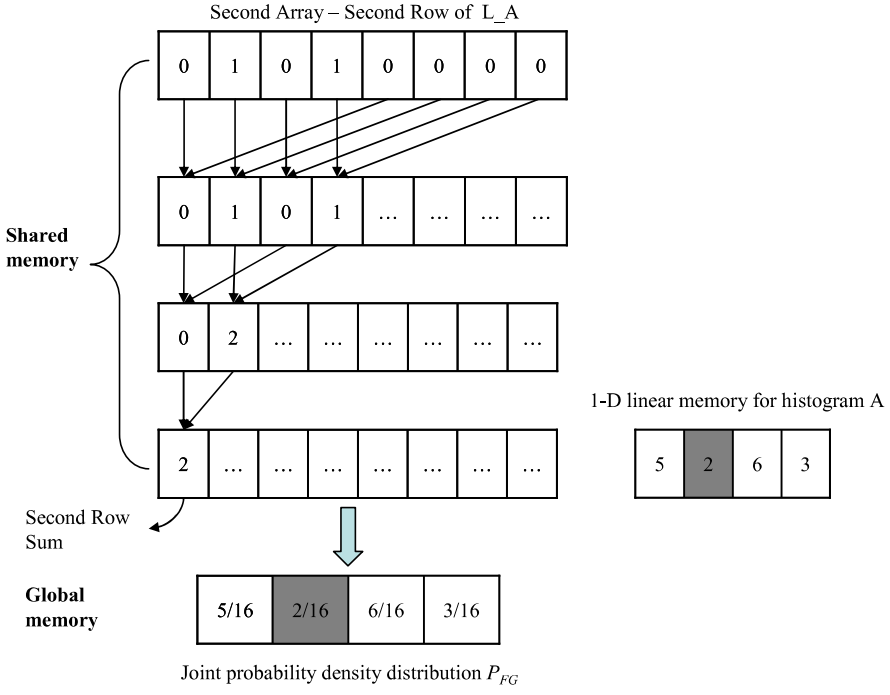


Figure 3-8 Parallel reduction of the second row of L_A to obtain the frequency count and probability density at the second address in the histogram A and the joint probability density P_{FG}.

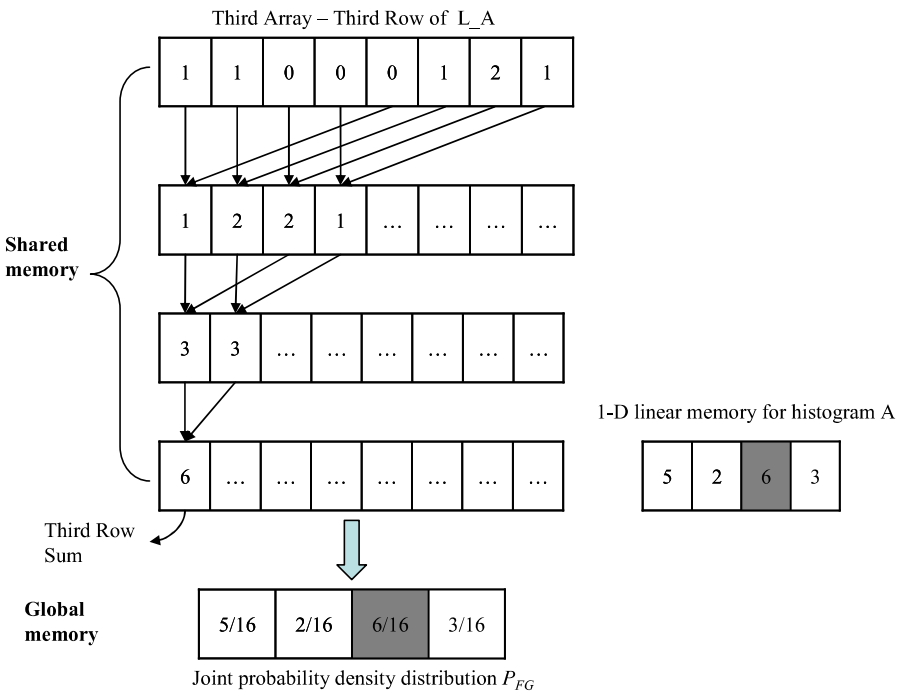


Figure 3-9 Parallel reduction of the third row of L_A to obtain the frequency count and probability density at the third address in the histogram A and the joint probability density P_{FG}.

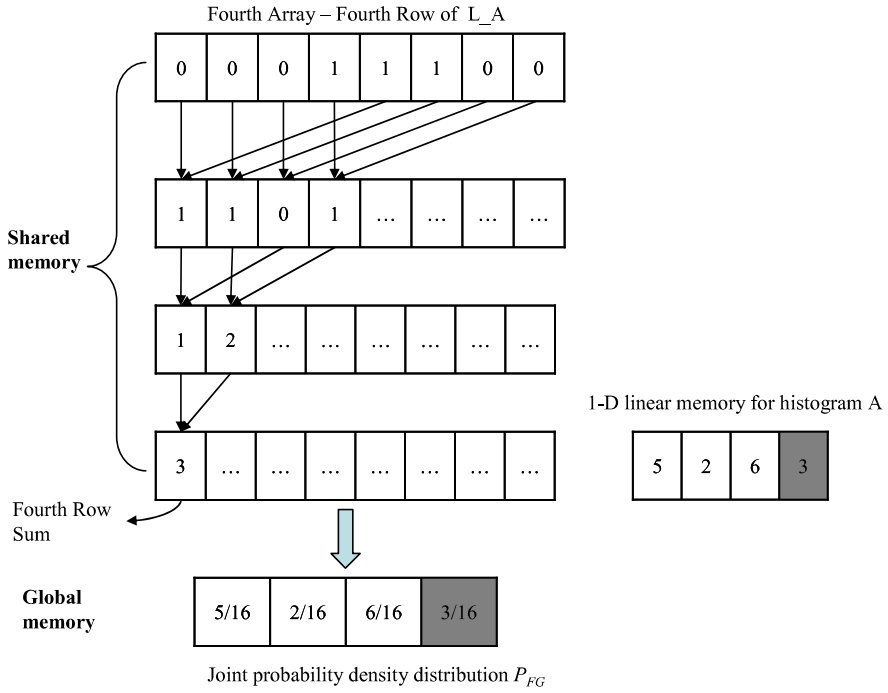


Figure 3-10 Parallel reduction of the fourth row of L_A to obtain the frequency count and probability density at the fourth address in the histogram A and the joint probability density P_{FG}.

Note that when processing the image of size 512×512, the row size of L_A is 1024 but this is over the limit of the maximum 512 threads allowed in one block. The remedy for this problem is to set the number of threads in one block to the maximum 512 and let the threads take the first 512 elements in each row of L_A and add onto the next 512 elements of the same row and then perform the parallel reduction.

3.1.8 Entropy calculation

After the joint probability density distribution P_{FG} is obtained, we are ready to compute the entropies and finally, the mutual information. Although P_{FG} is stored in a 1D memory, it is considered as a 2-D matrix during the entropy calculation as illustrated in Figure 3-11. We sum over its rows and columns to compute the individual probability density distribution for image F and G, P_F and P_G, respectively. The row and column sum are obtained by the parallel reduction in the similar fashion as discussed in Section 3.1.7.

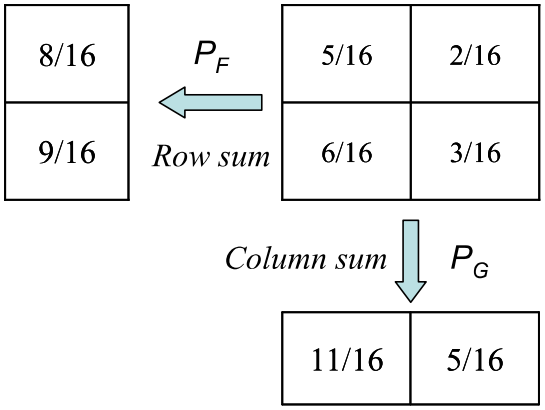


Figure 3-11 Calculate the row and column sum to obtain the individual probability density P_F and P_G

The individual entropy H_F, H_G and joint entropy H_{FG} are calculated directly using equations (3), (4), and (5), respectively, on the host side (CPU). The mutual information is then computed according to equation (2) on the host side as well.

3.1.9 General 2D interpolation model

Although the image interpolation, the construction of the joint probability density matrix, and the computation of the entropy and mutual information are illustrated using the example with resolution 4×4 and the number of bins 2, the implementation of real

medical imageswith resolution 128×128, 256×256, and 512×512 and number of bins 32 follow exactly the same pattern. For real size images and large number of bins, the computation is intensive and the real benefit of GPU parallel computation should manifest. For instance, the 512×512 image has 262144 pixels, the size of the histogram A is 32×32 and L_A is 1024×262144 with approximately 2.68435×10^8 elements. The computational time for the bilinear interpolation, histogram creation, and row/column summation is extensive, should they were carried out in the sequential fashion using the CPU. The desired efficiency gain is hence achieved by using GPU to perform millions of computation with concurrent blocks and threads.

3.2 2D Partial Volume Model

Comparing the 2D partial volume model with the 2D interpolation model, the entropy calculation is the same once the histogram matrix A is obtained. However, in the partial volume model, the intensity value after image transformation is not explicitly interpolated, but combined into the construction of the histogram matrix A and the joint probability density matrix P_{FG}. The example used for the 2D interpolated model is repeated here to illustrate such difference and the CUDA implementation.

First the bin index for both the template image F and the target image G are obtained, in the same fashion as in the 2D interpolation model. The resulted matrices for the two images are shown in Figure 3-2 in Section 3.1.6. Next, suppose the template image F is translated by 0.5 pixel in both the x and y directions. The target image G and the transformed template image F are illustrated in Figure 3-12. Note that the pixels or their intensity values/bin indices are represented by the solid and empty circles in Figure 3-12 .

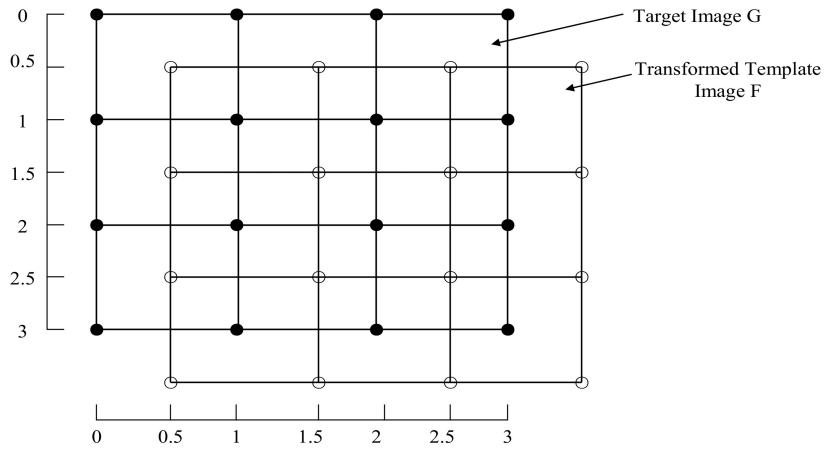


Figure 3-12 Target image G and the transformed template image F with pixel coordinates in x and y direction

In the transformed template image, the new coordinates of the pixels are computed by invoking 16 threads with each adding 0.5 on the corresponding pixel's old coordinates. As highlighted in Figure 3-13, after the transformation, some pixels of the template image F fall outside the region of the target image. Those pixels with their neighboring pixels are eliminated in the following computation. In order to identify those out-of-bound pixels and their neighboring pixels, we let 16 threads read the transformed image coordinates, x and y, and check simultaneously the floor(x), floor(y), x+1 and y+1 all fall in the range [0,3], otherwise those threads will be deactivated and not participate in the following calculation.

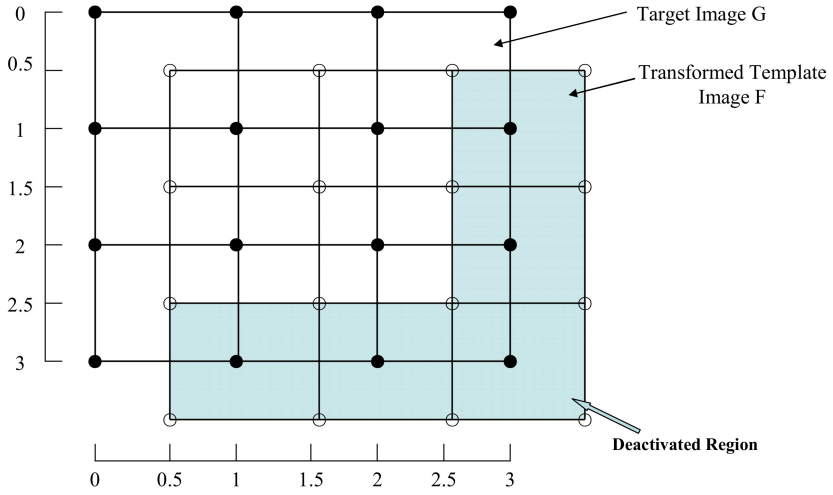


Figure 3-13 Target image G and the transformed template image F with highlighted region indicating the out-of-bound portion of image F where CUDA threads are deactivated.

Next, following the procedure outlined in Section 2.3, for every pixel of the template image falling in the non-deactivated region, four neighboring pixels of the target images are found. They are (*bin_index_f, bin_index_g_1*), (*bin_index_f, bin_index_g_2*), (*bin_index_f, bin_index_g_3*) and (*bin_index_f, bin_index_g_4*) (see Figure 2-7 in Section 2.3). Similar to the 2D interpolation model, the bin index of the transformed image F is the row index of the histogram matrix *A* and the bin index of the target image G is the column index of the histogram matrix *A*. The locations (*bin_index_f, bin_index_g_1*), (*bin_index_f, bin_index_g_2*), (*bin_index_f, bin_index_g_3*), and (*bin_index_f, bin_index_g_4*) are then converted simultaneously to the 1D address of the histogram *A*, by the non-deactivated threads, resulting *1D-address-matrix_1* for (*bin_index_f, bin_index_g_1*), *1D-address-matrix_2* for (*bin_index_f, bin_index_g_2*), *1D-address-matrix_3* for (*bin_index_f, bin_index_g_3*), and *1D-address-matrix_4* for (*bin_index_f, bin_index_g_4*). The same non-deactivated threads also calculate the weights w_{00}, w_{01}, w_{10}, and w_{11} using equations (6), (7), (8), and (9) in Section 2.2.

Once the four 1D address matrices and weights are obtained, following equation (12) in Section 2.3, the large matrix *L_A* is incremented by the *1D-address-matrix_1* in the same manner as in the interpolation model but the increment is the fractional weight w_{00} not integral 1. *L_A* is then updated by *1D-address-matrix_2* using equation (13). The same procedure is repeated for *1D-address-matrix_3* and *1D-address-matrix_4*, following equation (14) and (15).

The parallel reduction is then used to obtain the final histogram matrix *A* and the joint probability density matrix P_{FG}. The remaining calculation of the entropy and mutual information will be the same as in the interpolation model.

Although the procedure is illustrated using the 4×4 images, real size image follow the same procedure with the variance that more threads and blocks are used. For example, with a 128×128 image we invoke 16 blocks with each block 256 threads. The 2-dimensional blocks and grids are declared and included in the kernel function as follows:

```
Dim3 Block(16,16);
Dim3 Grid(2,2);
Somekernelfunction<<<Grid,Block>>>(…);
```

3.3 3D Interpolation Model

In the 3D interpolation model, we work on the 3D images of dimension 256×256×64. Similar to the 2D model, the template and target images are registered by minimizing the joint entropy using the steepest descent algorithm. The construction of the joint probability density matrix and the entropy calculation are the same as in the 2D interpolation model.

The only difference is at the interpolation phase, where 3D texture memory is used rather than the 2D texture memory and rotation is not considered. The technique using the 3D and 2D texture memory differ and thus justifying the detailed explanation. In preparation of the interpolation using the 3D texture memory, the 3-dimensional image has to be transformed and stored in a 1-dimensional linear memory. As shown in Figure 3-14, the cube representing the 3D image with 256 pixels in width and height and 64 pixels in depth is sliced into 64 layers along its depth and transformed into a 2D matrix by placing one layer below another. Reading and storing this 2D matrix one at a time from top left to the bottom right, it is converted into the 1D linear memory for the texture memory interpolation.

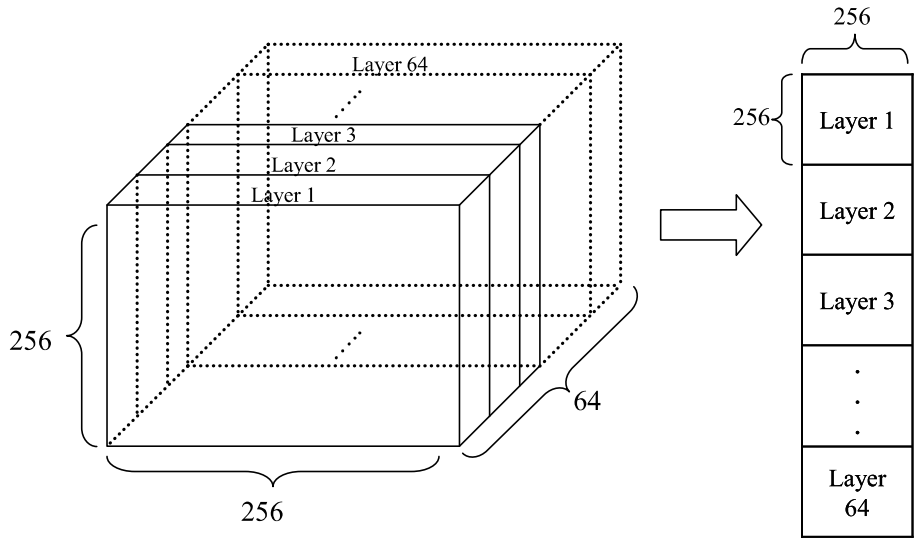


Figure 3-14 The transformation of image from 3D to 2D.

The major steps in 3D texture memory interpolation are similar to the 2D texture memory though it is greatly more complicated. They include allocating a 3D CUDA array, copying the image from the 1D linear memory into the 3D CUDA array, setting appropriate texture memory reference, binding 3D CUDA array with the texture memory, and finally interpolating the image by calling built-in CUDA function **tex3D** with new coordinates. The following sections follow this order. 3-dimensional CUDA texture memory is learned by consulting the material in [17].

3.3.1 3D CUDA array

The structure **volumeSize** is created to indicate that the dimension of the array that will be allocated. **volumeSize** has the same dimension as the 3D image.

```
cudaExtent const volumeSize = {width, height, depth};
```

The 3D CUDA array, **d_volumeArray**, is then allocated with the indicated **volumeSize** and the data contained in the array is of float type, which is set by **channelDesc**.

```
cudaArray *d_volumeArray = NULL;
cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc<float>();
cudaMalloc3DArray(&d_volumeArray, & channelDesc, volumeSize);
```

3.3.2 Copy data to 3D Array

Copying the 3D image from the 1D global memory to the 3D CUDA array is cumbersome. There are multiple parameters, contained in **copyParams** need to be set before the memory copy. They are **copyParams.srcPtr**, **copyParams.dstArray**, **copyParams.extent**, and **copyParams.kind**.

copyParams.srcPtr contains the pointer **hin** pointing to the source of the memory copy, which is the 1D linear global memory for the image. Also, it indicates that the pitch in byte of the allocated memory is **volumeSize.width*sizeof(float)** and the number of elements along the width and height of such memory are **volumeSize.width** and **volumeSize.height**, respectively.

copyParams.dstArray sets the destination array, which is the allocated 3D CUDA array, **d_volumeArray**. **copyParams.extent** indicates the complete dimension of the 3D array and the related size of the ensuing memory copy. **copyParams.kind** sets the direction of the copy. It is set to **cudaMemcpyDeviceToDevice** as both the 1D linear global memory containing the image and the **d_volumeArray** both sit on the GPU. The CUDA code is given as follows:

```
cudaMemcpy3DParms  copyParams = {0};

copyParams.srcPtr =
make_cudaPitchedPtr((void**)hin,volumeSize.width*sizeof(float),
volumeSize.width, volumeSize.height);

copyParams.dstArray  =  d_volumeArray;
copyParams.extent   =  make_cudaExtent(width,height,depth);
copyParams.kind     =  cudaMemcpyDeviceToDevice;

// copy data to the 3D CUDA Array
cudaMemcpy3D(&copyParams);
```

3.3.3 Texture reference and memory binding

Setting the texture reference and binding the 3D CUDA array to the texture memory are similar to the 2D case. The interpolation is trillinear since it is in the 3D space. The **filterMode** is set to **cudaFilterModeLinear**. The boundary condition, in x, y, and z direction, represented by **addressMode[0]**, **addressMode[1]** and **addressMode[2]**, are

set to **cudaAddressModeClamp**. The code is given below:

```
// set texture parameters
texRef.normalized = false;              // actual texture coordinates
texRef.filterMode    = cudaFilterModeLinear; // trilinear interpolation
texRef.addressMode[0] = cudaAddressModeClamp;
texRef.addressMode[1] = cudaAddressModeClamp;
texRef.addressMode[2] = cudaAddressModeClamp;

// bind array to 3D texture
cudaBindTextureToArray(texRef,d_volumeArray,channelDesc);
```

3.3.4 Interpolation by calling tex3D

Once the 3D CUDA array is bound with the texture memory, new texture coordinates **u**, **v**, and **w** are generated given the image translation **shift_x**, **shift_y**, **shift_z**, in the x, y and z direction, respectively. Since the data is stored at the "center" of the texture coordinates, the new texture coordinates have to be further offset by 0.5 pixel to ensure the correct interpolation results. Passing the new texture coordinates alongside the texture reference **texRef** to the built-in CUDA function **tex3D**, the results fetched from it will be the desired interpolated data and they are transferred to the pre-allocated memory, **B_o**. Denoting **x0**, **y0** and **z0** the original coordinates, the code for calculating and then offsetting the new texture coordinates, calling the **tex3D** function and finally copying the results to pre-allocated memory is given below:

```
int idx = x0 + y0*width + z0 * width * height;
float u = x0+0.5f-shift_x;
float v = y0+0.5f-shift_y;
float w = z0+0.5f-shift_z;
B_o[idx] = tex3D(texRef,u,v,w);
```

4 Numerical Results

Three experiments are conducted in this project to register medical images of different sizes and modalities, using the GPU parallel computation techniques outlined in Section 2 and Section 3. The computational time is compared with the time when performing the same task using CPU only, where the computation is sequential. Also, the various aspects that affect the efficiency gain using the GPU are discussed in this section.

The medical images considered in this project are 8-bit grey-scale images with intensity value ranging from 0 to 255. They are obtained from retrospective image registration evaluation project (RIRE) [18]. They are of various modalities, either from different sensors: magnetic resonance image (MRI) and X-ray computed tomography (CT), or from the same sensor but with different parameters: T1-weighted magnetic resonance imaging (MRI) and T2-weighted MRI. For the 2D image registration, the following image resolutions are used: 64×64, 128×128, 256×256, and 512×512; and for the 3D case the image resolution is 256×256×64.

The GPU and CPU used in this project are Tesla C1060 and Intel i7-950, respectively. Their specifications are listed in Table 4-1 and Table 4-2 below. The RAM in the workstation is 8G DDR3.

# of Streaming Processor Cores	240
Frequency of processor cores	1.3 GHz
Single Precision floating point performance (peak)	933
Floating Point Precision	IEEE 754 single & double
Total Dedicated Memory	4G DDR3
Memory Speed	800MHz
Software Development Tools	C-based CUDA

Table 4-1 GPU Tesla C1060 Specifications

# of Cores	4
# of Threads	8
Clock Speed	3.06 GHz
Intel® Smart Cache	8 MB

Table 4-2 CPU i7-950 Specifications

4.1 Experiment 1

The first experiment is to register 2D multimodal images with varying sizes using the interpolation and partial volume model. The process is first carried out by the CUDA program employing the GPU parallel computation technique and then compared with the standard C program run by CPU only. The modality is T1-weighted MRI for the target image and T2-weighted MRI for the template image. The image resolution varies from 64×64, 128×128, 256×256, to 512×512. They are shown below in Figure 4-1, Figure 4-2, Figure 4-3 and Figure 4-4.

Figure 4-1 Target image (Left, size 64×64, T1-weighted MRI) and Template Image (Right, size 64×64, T2-weighted MRI)

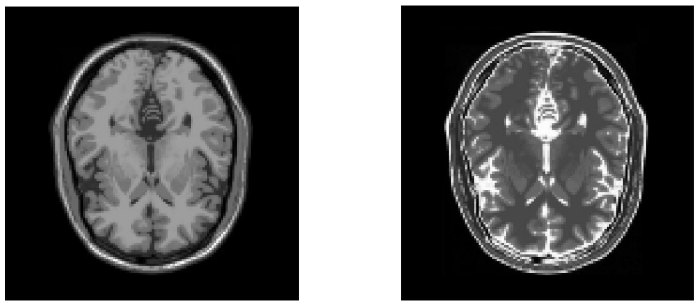


Figure 4-2 Target image (Left, size 128×128, T1-weighted MRI) and template image (Right, size 128×128, T2-weighted MRI)

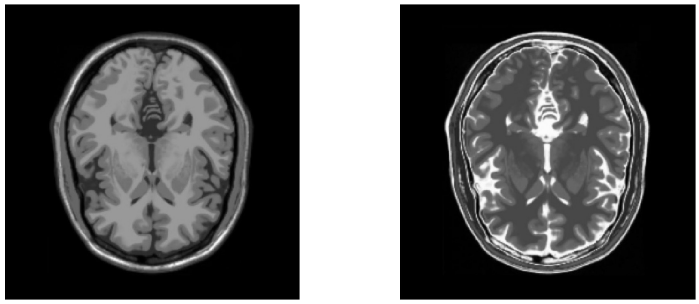


Figure 4-3 Target image (Left, size 256×256, T1-weighted MRI) and Template Image (Right, size 256×256, T2-weighted MRI)

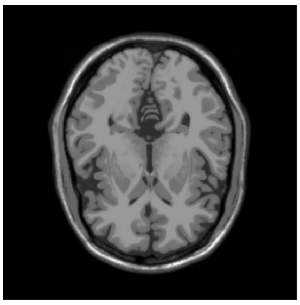
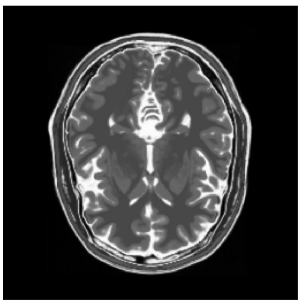



Figure 4-4 Target image (Left, size 512×512, T1 weighted MRI) and Template Image (Right, size 512×512, T2 weighted MRI)

Since the whole image registration process is dominated by minimizing the joint entropy, the computational time of this calculation is of major interest to this project. Thus time used for comparison is the time to compute the joint entropy and mutual information **only once** based on the new translation and rotation parameters. Furthermore, due to the restriction that GPU is only able to perform single precision floating computation efficiently, the variables and calculation in the CPU program are all of type "float" to ensure fair time comparison. Table 4-3 and Table 4-4 summarize the computational time for the 2D interpolation and partial volume model using CUDA program and C program with various image sizes. Figure 4-5 and Figure 4-6 show the plots of the GPU and CPU time.

	Computational Time (milliseconds)		Speed-up
Image Resolution	CUDA (GPU)	C (CPU)	
64×64	0.2	0.725	3.63
128×128	0.275	2.95	10.73
256×256	0.50	11.78	23.55
512×512	0.825	50.40	61.09

Table 4-3 Comparison of the GPU and CPU computational time for the 2D interpolation model. The speed-up is the CPU time over the GPU time.

	Computational Time (milliseconds)		Speed-up
Image Resolution	CUDA (GPU)	C (CPU)	
64×64	0.325	0.50	1.54
128×128	0.55	1.75	3.18
256×256	1.175	7.025	5.98
512×512	2.40	28.875	12.03

Table 4-4 Comparison of the GPU and CPU computational time for the 2D partial volume model. The speed-up is the CPU time over the GPU time.

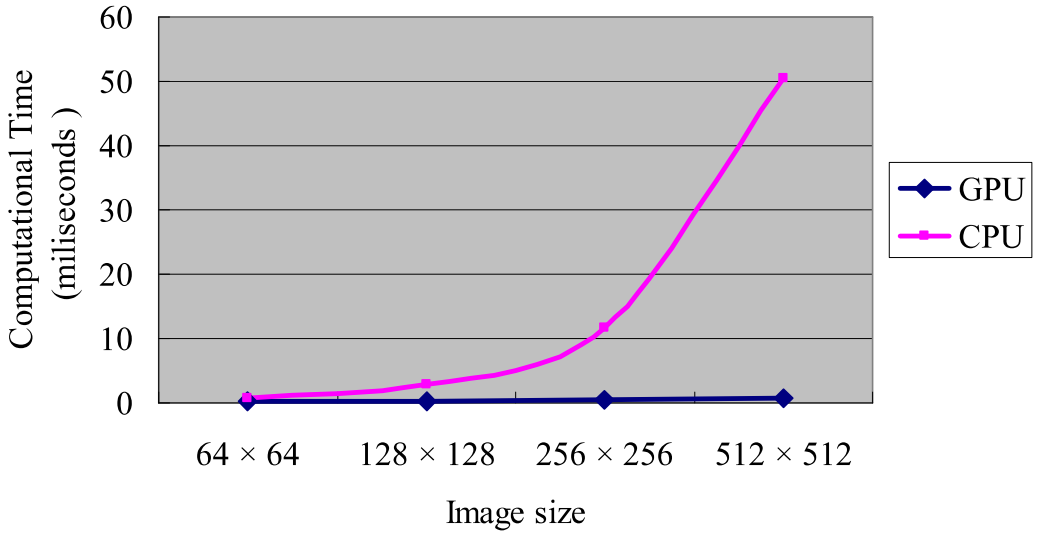


Figure 4-5 Comparison of GPU and CPU computational time for 2D interpolation model

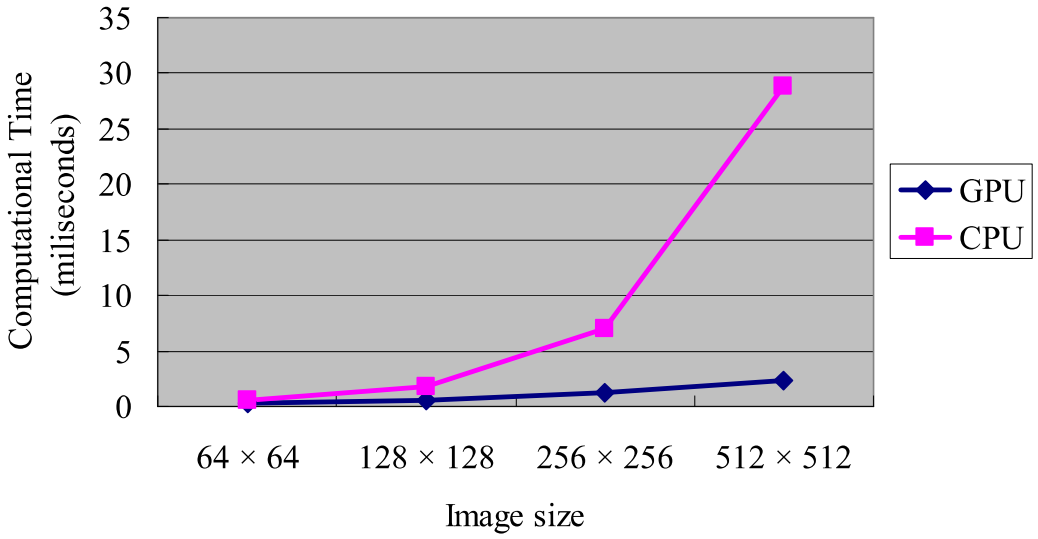


Figure 4-6 Comparison of GPU and CPU computational time for 2D partial volume model

Several observations are drawn from the above tables and figures. First, as the image size increases, the time consumption increases dramatically for the CPU while remains relatively flat for the GPU, as shown in Figure 4-5 and Figure 4-6. This is because when the image size grows, GPU could simply employ more concurrent threads to perform the computation simultaneously at each pixel. Whereas the standard C program running under CPU has to finish the computation at one pixel before moving to the next. Thus as the image size doubles, triples or even quadruples, so does the computational time.

Second, larger speed-up ratio is gained with the growing image size, thanks to the scalability of the GPU through invoking more threads. For instance, for the 64×64 image size the speed-up is 3.63 and 1.54 for the interpolation and partial volume models. It reaches 61.09 and 12.03 respectively when the image size became 512×512.

Third, better speed-up ratio is obtained for the interpolation model than for the partial volume model as indicated in Table 4-3 and Table 4-4. This is due to the fact that when constructing the joint histogram the partial volume model needs to update the matrix *L_A* four times by adding one fractional weight each time (see Section 3.2, 2D Partial Volume Model) while the interpolation model just needs to update the *L_A* once (see Section 3.1, 2D Interpolation Model). The updating of *L_A* process is quite time consuming because it is not a completely parallel process. It processes only two rows of the *1D_address_matrix* at one time (see Section 3.1.6). As an example, the 512×512 image needs 256 such iterations due to the 512 rows of the *1D_address_matrix*. Furthermore, GPU prefers ordered memory access where, say, the first thread accesses the 1st element of the array; the second thread accesses the 2nd element of the array and so on. However, this is not the case when updating *L_A*, where the thread goes is determined by the *1D_address_matrix*. Suppose the 1st thread reads the first element of *1D_address_matrix* with value 5; then the thread has to go to the 5th element in the *L_A* other than the 1st element. When the memory access is entangled in this way, the speed of the process slows.

4.2 Experiment 2

In this experiment, the modalities of the two images differ from the experiment 1. The modality of the target image is X-ray computed tomography (CT) while the template image is T1-weighted MRI. The pairs of target and template images are of resolutions 64×64, 128×128, 256×256, to 512×512, as shown in Figure 4-7, Figure 4-8, Figure 4-9 and Figure 4-10.

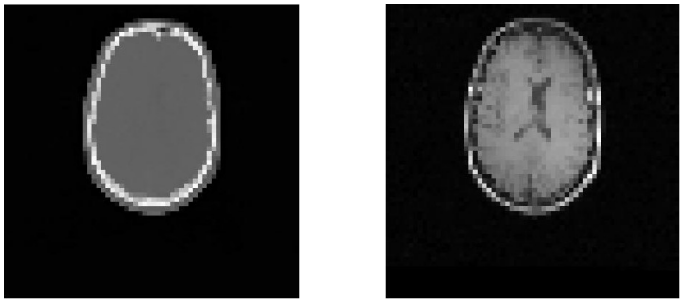


Figure 4-7 Target image (Left, size 64×64, CT) and Template Image (Right, size 64×64, T1-weighted MRI)

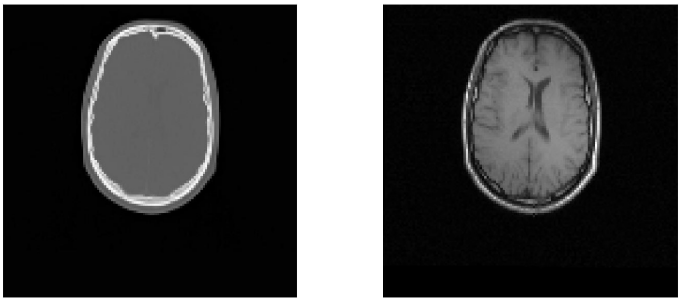


Figure 4-8 Target image (Left, size 128×128, CT) and Template Image (Right, size 128×128, T1-weighted MRI)

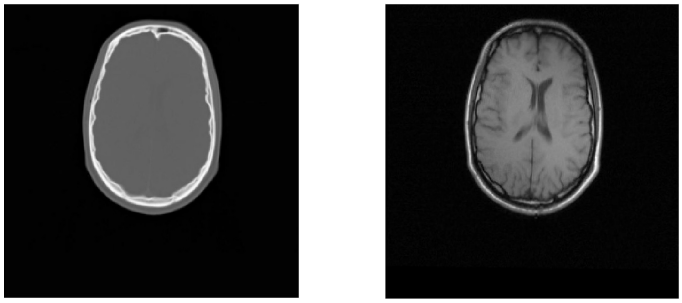


Figure 4-9 Target image (Left, size 256×256, CT) and Template Image (Right, size 256×256, T1-weighted MRI)

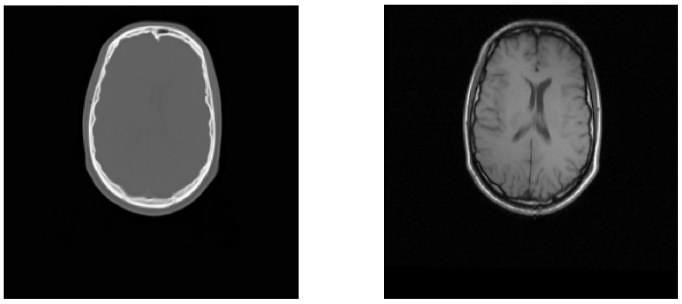


Figure 4-10 Target image (Left, size 512×512, CT) and Template Image (Right, size 512×512, T1-weighted MRI)

Similar to the experiment 1, the four pairs of images are registered using the GPU and CPU, respectively. The computational time is summarized in Table 4-5 and Table 4-6. Figure 4-11 and Figure 4-12 show the plots of the GPU and CPU time.

	Computational time (milliseconds)		Speed-up
Image Resolution	CUDA (GPU)	C (CPU)	
64×64	0.20	0.58	2.90
128×128	0.28	2.68	9.57
256×256	0.48	10.73	22.35
512×512	0.80	47.88	59.85

Table 4-5 Comparison of GPU and CPU computational time for the 2D interpolation model.

	Computational time(milliseconds)		Speed-up
Image Resolution	CUDA (GPU)	C (CPU)	
64×64	0.33	0.48	1.45
128×128	0.53	1.70	3.21
256×256	1.05	6.80	6.48
512×512	2.25	27.13	12.06

Table 4-6 Comparison of GPU and CPU computational time for the 2D partial volume model.

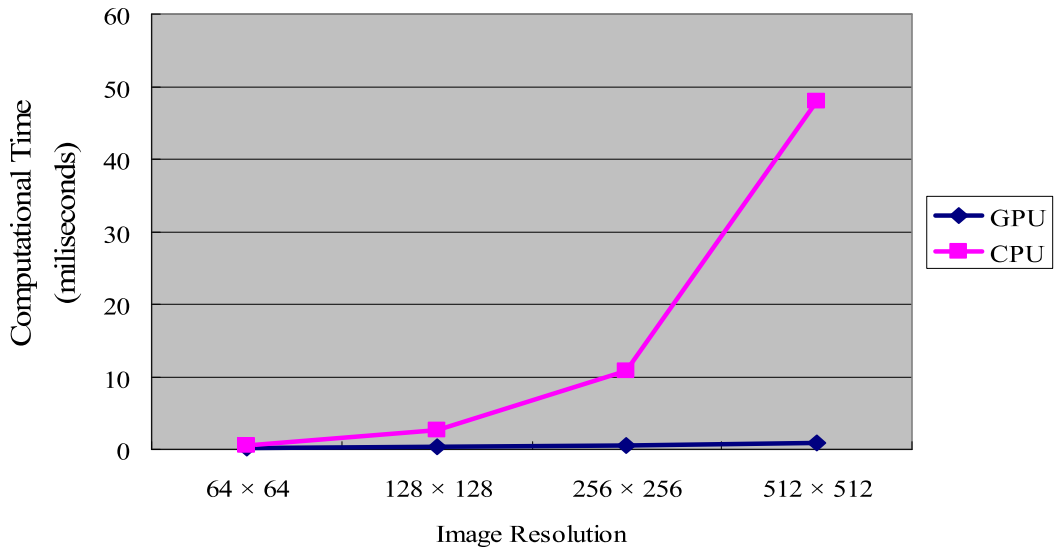


Figure 4-11 Comparison of GPU and CPU computational time for the 2D interpolation model

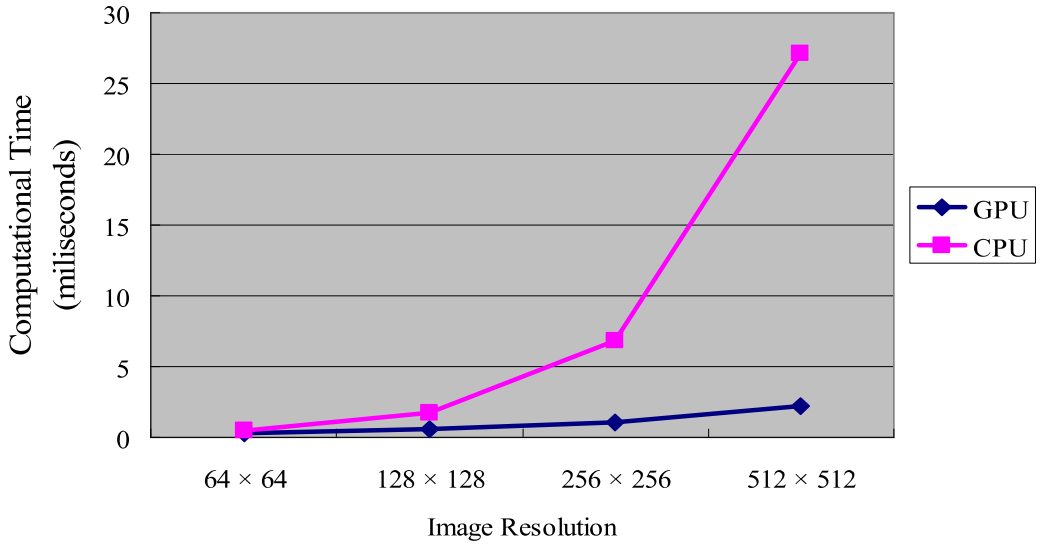


Figure 4-12 Comparison of GPU and CPU computational time for the 2D partial volume model

From the above tables and figures several similar observations can be drawn. First, as the image size increases the computational time increases. However, the slope of the increase is much steeper for the CPU while it remains generally flat for the GPU, as indicated in Figure 4-11 and Figure 4-12. The benefit of deploying more concurrent threads when image size grows manifests here again. Second, similar to the experiment 1 the larger speed-up ratio is obtained when image size is larger due to the scalability of the GPU. Third, in the experiment 2 the speed-up ratio is still higher for the interpolation model than for the partial volume model as indicated in Table 4-5 and Table 4-6, for the same reason explained in the experiment 1. Fourth, the computational time and the speed-up to a great extent remain the same when the modalities of the images being registered are different from experiment 1.

4.3 Experiment 3

In this experiment 3D images of resolution 256×256×64 are registered using the 3D interpolation model. The modality of the template and the target image is T1-weighted MRI and T2-weighted MRI, respectively. Layer 2, 14, 20 and 30 of the 3D images are shown in Figure 4-13 and Figure 4-14.

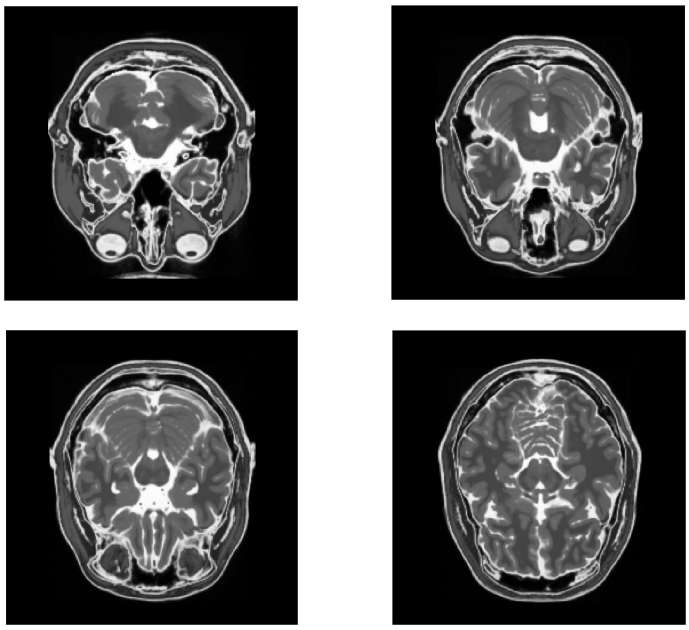


Figure 4-13 Characteristic layers of the 3D target image of modality T2-weighted MRI

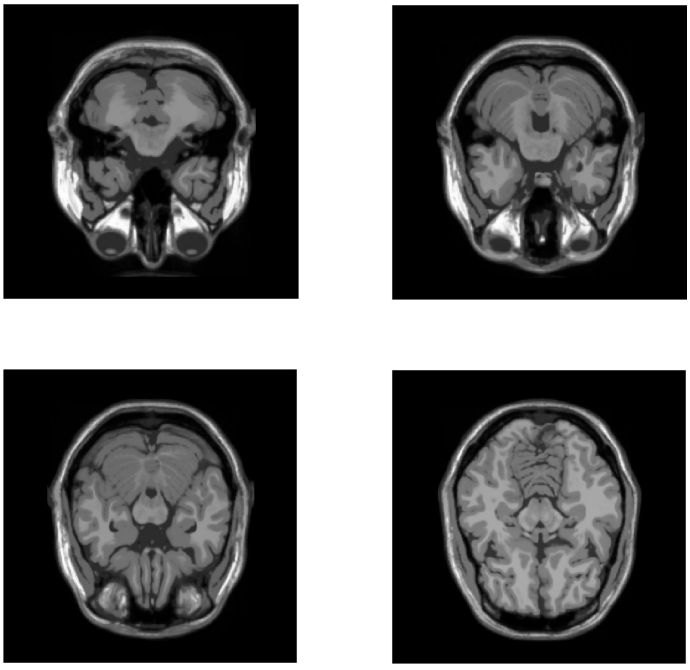


Figure 4-14 Characteristic layers of the 3D template image of modality T1-weighted MRI

The computational time for the 3D image registration is much longer than the 2D cases. In our experiment the time needed to compute the joint entropy and mutual information once is 1742.7 milliseconds on CPU and 24.3 milliseconds on GPU. As a comparison, the computational time for the 512 ×512 image in the 2D interpolation

model is 50.4 milliseconds and 0.83 milliseconds using CPU and GPU, respectively. This is due to the fact that the number of pixels of the 3D image exceeds any of the 2D images in the experiment 1 and 2, resulting larger amount of computation.

The speed-up, 71.72 times (Table 4-7), in experiment 3 is the highest among all three experiments. But comparing with the speed-up 61.09 times in the case of 512 ×512 image in the 2D interpolation model, it is not a substantial gain in efficiency, even though the 3D image has 16 times the pixels of the 512 ×512 image. There are two explanations for this phenomenon. First, the size of the image, 64 megabytes, is large. This causes all the arrays, such as the bin index array, and the coordinates array, used in the program to have similar or even bigger size. The large memory consumption might have reached the limit of the GPU and slows down the process. Second, recall that the procedure of updating the large matrix *L_A* is not perfectly parallel. We only process two rows of the *1D_address_matrix* once. In the 3D case, the size of the *1D_address_matrix* is much larger which demands more iterations. Therefore the amount of time needed for updating *L_A* does not remain flat but increases with the size of the image. Table 4-7 summarizes the results of experiment 3 and Figure 4-15 incorporates the speed-up result with the ones from 2D interpolation model.

	Computational time(milliseconds)		Speed-up
Image Resolution	CUDA (GPU)	C (CPU)	
256×256×64	24.3	1742.7	71.12

Table 4-7 Comparison of GPU and CPU computational time for the 3D interpolation model

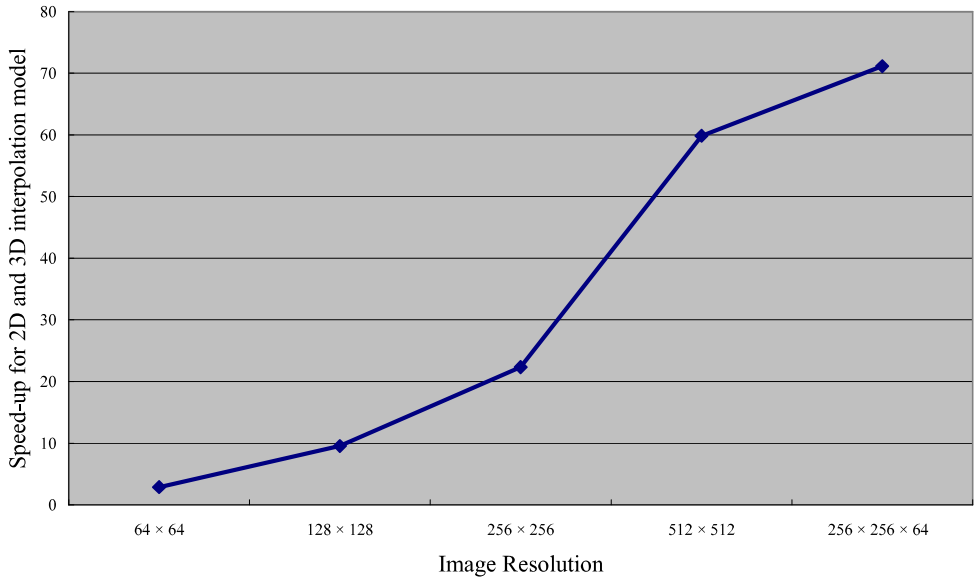


Figure 4-15 Speed-up for the 2D and 3D interpolation model

5 Conclusion

This project has addressed the challenge of the intensive computational time for registering multimodal 2D and 3D medical images. The modalities used are T1-weighted MRI, T2-weighted MRI, and CT. To perform the image registration, two models, interpolation and partial volume model, are used where mutual information is used as the measure. Their computation are transformed from sequential to parallel so that GPU could perform the computation on every pixel or voxel at the same time, in the parallel manner. The GPU computational time is compared with the CPU time where all computational is performed in a sequential way.

When calculating the mutual information and joint entropy using the parallel computing architecture, CUDA, several of its key technologies are used, such as the thread-block hierarchy, shared and texture memory and the parallel reduction, to ensure the image interpolation, joint histogram construction and entropy calculation are done in a parallel and efficient way. The highest speed-up is 61.09 for the 2D interpolation model and 12.06 times for the partial volume model. For the 3D interpolation model the speed-up is achieved at 71.12, the best among all the experiments. It is thus found that as more pixels or voxels are involved in the computation, the performance gain using GPU is higher due to the scalability of the GPU where more threads are invoked to perform the calculation.

Possible future work includes 3D image registration using the 3D partial volume model and exploring deeper into the CUDA technology to discover more intricacy such as the bank conflict in the shared memory and the actual execution of thread wrap in the CUDA thread block, in order to further improve the GPU efficiency in the image registration problem.

6 Reference

[1] S. C. Bushong, *Computed Tomography*. NY:McGraw-Hill Medical, 2000.

[2] C. Westbrook, C. K. Roth, and J. Talbot, *MRI in Practice*, 3rd edition, Turin: Wiley-Blackwell, 2005.

[3] M. I. Miller, G. E. Christensen, Y. A. Amit, and U. Grenander, In *Mathematical Textbook of Deformable Neuroanatomies*, Vol. 90, Medical Sciences, National Academy of Sciences.1993, pp. 11944–11948.

[4] P. Viola and W. M. Wells III, "Alignment by Maximization of Mutual Information," *International Journal of Computer Vision*, vol.24, pp. 137-154, 1997.

[5] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*, Addison-Wesley Professional,2010.

[6] L. Lemieux and R. Jagoe, "Effect of fiducial marker localization on stereotactic target coordinate calculation in CT slices and radiographs." *Phys. Med. Biol.*, vol.39, pp. 1915–1928, 1994.

[7] A. C. Evans, S. Marrett, J. Torrescorzo, S. Ku and L. Collins, "MRI–PET correlation in three dimensions using a volume of interest (VOI) atlas." *J. Cerebral Blood Flow Metabolism*, vol. 11, A69–A78, 1991.

[8] Y. Ge, C. R. Maurer, and J. M. Fitzpatrick, "Surface-based 3-D image registration using the iterative closest point algorithm with a closest point transform," *Medical Imaging: Image Processing*, Vol. 2710, pp. 358–367, 1996.

[9] F. Maes, A. Collignon, D. Vandermeulen, G. Marchal, P. Suetens, "Multimodality Image Registration by Maximization of Mutual Information," *IEEE Transactions on Medical Imaging,* vol. 16, No.2, 1997.

[10] J. B. A. Maintz and M. A. Viergever, "A survey of medical image registration," *Medical Image Analysis*, vol.2 pp. 1-36, 1998.

[11] K. Kuczynski and P. Mikofajczak, "Information theory based medical images processing," *Opto-Electronics Review*, vol.11, pp. 253-259, 2003.

[12] Lin Xu, Justin Wan, and Tiantian Bian, "A continuous mutual information model for multimodality image registration."

[13] J. P. W. Pluim, J. B. A. Maintz and M. A. Viergever, "Mutual information based registration of medical images: a survey," *IEEE Transcations on Medical Imaging*, vol. xx, No. Y, 2003.

[14] NVIDIA Corporation (2010). *NVIDIA Tesla C1060 Specifications* [Online].Available: http://www.nvidia.com/object/product_tesla_c1060_us.html

[15] Intel Corporation (2010). *Intel® Core™ i7-950 Processor Specifications* [Online]. Available: http://ark.intel.com/Product.aspx?id=37150

[16] NVIDIA Corporation (2010). *NVIDIA CUDA Programming Guide,* version 3.0, [Online]. Available: http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_ProgrammingGuide.pdf

[17] G. Dalley, IAP 2009 CUDA at MIT [Online] Available: http://sites.google.com/site/cudaiap2009/

[18] Retrospective Image Registration Evaluation Project [Online] Available: http://www.insight-journal.org/rire/