Friedrich-Alexander-Universität Erlangen-Nürnberg

Multimedia Communications and Signal Processing

Prof. Dr.-Ing. André Kaup

Master Thesis

Efficiency Improvement and Evaluation of Parallel Deformation Estimation in Cardiac MRI Data

Mengyi Li

March 2013

Supervisor: Dipl.-Ing. Andreas Weinlich

Erklärung

Ich versichere, dass ich die vorliegende Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe, und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

_____ _____

Ort, Datum Unterschrift

Contents

Abstract

The increasing usage of multidimensional imaging modalities such as cardiac magnetic resonance imaging (MRI) in medical examinations lead to a challenge of how to deal with the uncompressed high-resolution data.

In this thesis, the method of image compression by using motion estimation and compensation are introduced. And 2D implementation is developed both for CPU and GPU in order to have a more efficient 2D image processor. Also some improvements are applied on the existed Cuda 3D implementation. Such as a new storage of sparse matrix to save memory, and a new algorithm of Levenberg-Marquardt algorithm to search for the optimization.

Abbreviations

BMC	Block-based Motion Compensation
CG	Conjugate Gradient
CITK	Cuda Insight Toolkit
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
GNA	Gauss-Newton algorithm
GPU	Cuda Insight Toolkit
ITK	Insight Segmentation and Registration Toolkit
LGF	Lanczos-Gradienten-Filter
LMA	Levenberg Marquardt algorithm
MSE	Mean Square Error
MRI	Magnetic Resonance Imaging
PCG	Preconditioned Conjugate Gradient method
SSE	Sum of Squared Errors

Chapter 1

Introduction

With the development of medical technology, usage of multidimensional imaging modalities such as cardiac magnetic resonance imaging (MRI) in medical examinations has increased dramatically. However the significant amount of uncompressed high-resolution data requires lots of storage and transmission time, which leads to low efficiency since fewer images can be transmitted, so that an appreciate method of data reduction makes sense to improve both storage and speed issues. One method to facilitate image sequence coding is motion estimation and compensation, which exploits redundancies between images. Since the muscle tissue only expands or contracts, which means mostly tissue deformation and little translational displacements, deformable motion estimation and compensation is used instead of translational motion used in conventional video coding. The latter is the method to measure motion compensation using by Block-based Motion Compensation (BMC), which results in higher MSE comparing with the deformable motion compensation and has so-called $blockingeffects$ caused by the high intensity gradients at the block boundaries [WAH].

At the moment, there are two implementations according to the algorithm given by [WAH]. One is a 2-D implementation in Matlab and the other is 3-D in Nvidia CUDA, which is a programming language achieving the parallel computation on GPU (Graphics Processing Unit). Although a 2-D image data can be handled regarded as a special

case of 3-D by setting the z-dimension to zero, it is not as efficient in terms of speed and memory storage as a special implementation only for the 2-D image data. In this thesis, another implementation for the GPU using Nvidia CUDA was done especially for the fast 2-D case, which is sufficient for much applications. In order to have a direct and fair comparison of parallel execution speed, the 2-D case was also implemented on CPU (Central Processing Unit) using C++. Here, Insight Segmentation and Registration Toolkit (ITK) was used as a toolkit to provide already implemented function for common demands, while its extension for General-purpose computing on GPU, CUDA Insight Toolkit (CITK), was used for CUDA programming.

Another main task of this thesis is to improve and evaluate the efficiency of the given 3-D parallel deformation estimation algorithm in terms of computation time and memory requirement. On the base of the existing algorithm, Levenberg-Marquardt optimization method will be used instead of Gauss-Newton algorithm for larger step sizes and thus faster convergence, and an adapted memory-efficient handling of sparse matrices will be used to save memory so that more values in the downsampled potential field can by computed.

Chapter 2

Fundamental Concepts

In this chapter, the fundamental concept of two important issues is introduced. One is Compute Unified Device Architecture(CUDA) and the other is Insight Segmentation and Registration Toolkit (ITK) and its extension Cuda Insight Toolkit (CITK) for CUDA.

2.1 Compute Unified Device Architecture

Compute Unified Device Architecture is a a parallel programming model and a software environment for parallel computing created by NVIDIA company. It is implemented by the GPUs that they produce. The main difference between CPU and GPU architectures is that a CPU has large cache and strong control unit so that they use all resource to execute a single thread of sequential instructions, while on the contrary, GPUs have much more transistors to process data arrays instead of several sequential computing threads so that they are designed for highly parallel computation with lots of arithmetic operations. Figure 2.1 shows how much room is occupied by various circuits in CPUs and GPUs. On GPUs, it seems to split into lots of small blocks and each thread do its own instruction (for most of the time, the instructions are the same for every thread). The usage of CUDA shows the high performance when doing the parallel computation. It does not need such a large cache as that on CPU as each

thread just small cache for itself [Nvi].

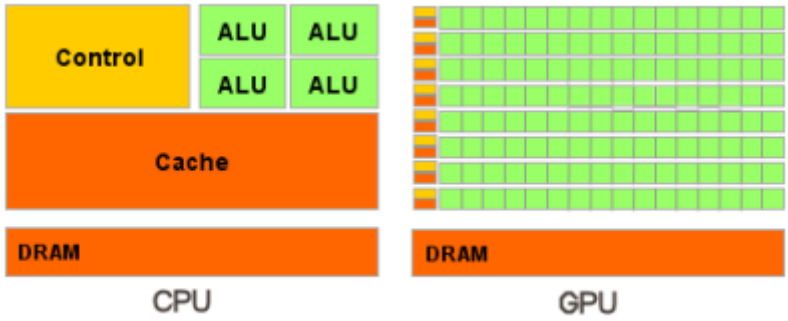


Figure 2.1: The GPU Devotes More Transistors to Data Processing [Nvi]

Besides, as Figure 2.2 showed, which is published by NVIDIA of CPU/GPU performance growth for the last years, the conclusion comes out that the `float` (single precision) type is fast in CUDA and should be used as often as possible.

CUDA does parallel computing, when each of the processors executes the same code with different data in parallel. The GPU is a computing device, co-processor (device) for a CPU (host), has its own memory and contains a lot of threads in parallel. A kernel is a GPU function executed by threads. Typically one kernel is running at a time. It is executed on the GPU (the device), and must be specifically defined with the keyword `__global__`. The number of CUDA threadblocks and the number of threads within a thread block are specified in the angle brackets `<<<...>>>` when a kernel function is called. Both threadblocks and threads can be set as one-dimensional, two-dimensional or three-dimensional. Each thread block is processed by a multiprocessor and 1,024 threads can be used per block, which is the maximal number of threads in a threadblock. It looks like Figure 2.3. Threadblocks are composed of small groups with 32 threads each called *warps*. It's minimum volume of data, which execute the same instruction at the same time.[NVI12].

Figure 2.4 illustrates the memory types. Some memory types used in the CUDA code

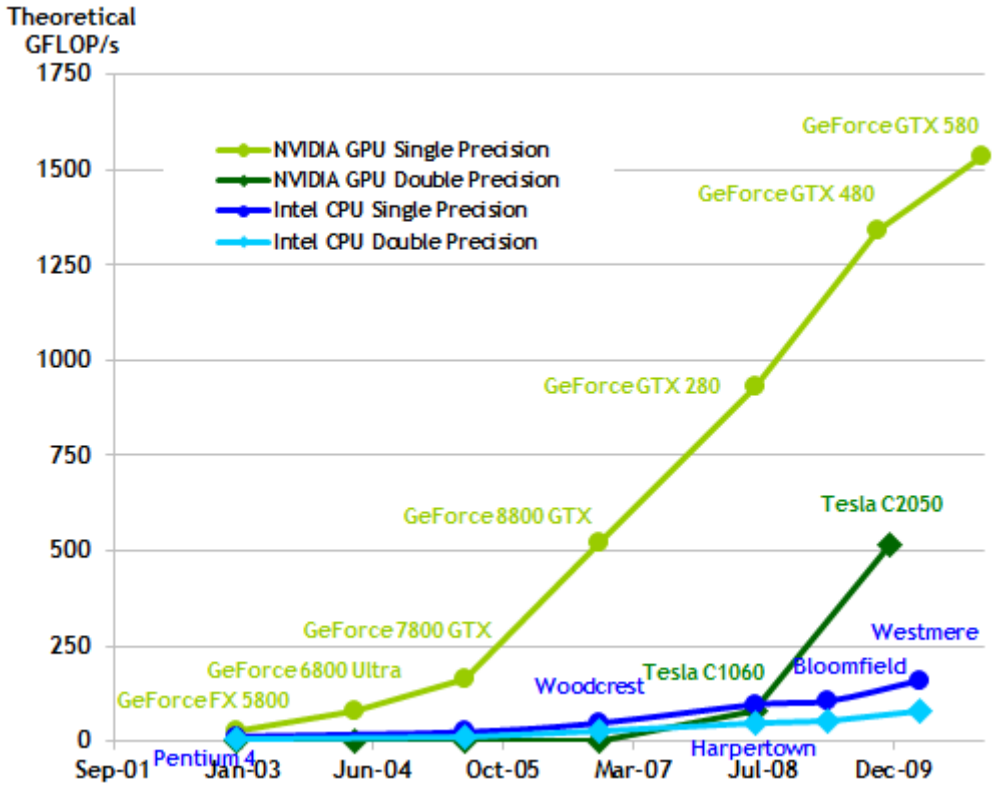


Figure 2.2: Floating-Point Operations per Second for the CPU and GPU [NVI12]

described in next chapter are introduced. First are registers and the shared memory. There are quite a lot of registers per processor, up to 1024. Access to these registers is very fast, they can store 32-bit integer or floating-point numbers. Shared memory is between 16 and 48KB memory shared between all processors in a multiprocessor. It is fast memory, just like registers. This memory makes threads access to other threads. Those two fast memories, however, are very small compared to the global memories. Global memory is the largest memory available to all multiprocessors in a GPU. It's relatively slow in comparison with shared memory. Texture memory is available for reading to all multiprocessors. Data are fetched by texture units in a GPU, so the data can be interpolated linearly by hardware implementation. [Nvi]

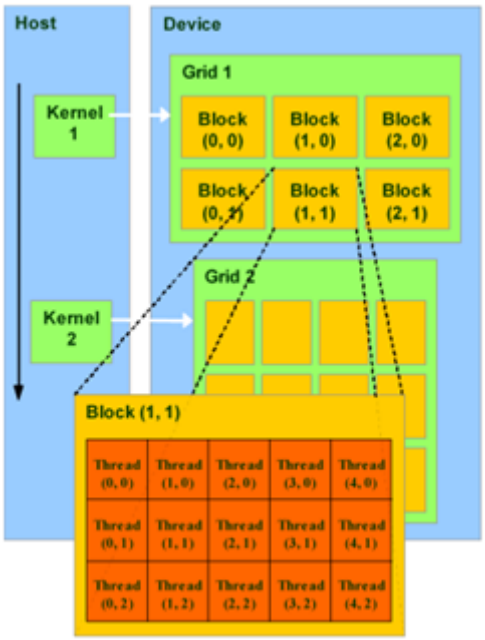


Figure 2.3: Grid of Thread Blocks [Nvi]

2.2 Insight Segmentation and Registration Toolkit

Insight Segmentation and Registration Toolkit(ITK) is an open-source, object-oriented software library for image processing, segmentation, and registration. It is implemented in C++, and it uses the CMake (cross-platform make) build environment to manage the configuration process. CMake is an operating system and compiler independent build process that produces native build files appropriate to the operation system and compiler that it is run with. This thesis is done on Unix, where CMake produces makefiles. In addition, developers can create software using a variety of programming languages, since there is an automated wrapping process generating interfaces between C++ and interpreted programming languages such as Java and Python. ITK's C++ implementation style is referred to as generic programming (i.e., using templated code). Such C++ templating means that the code is highly efficient, and that many software problems are discovered at compile-time, rather than at run-time during program execution. [ISNC05]

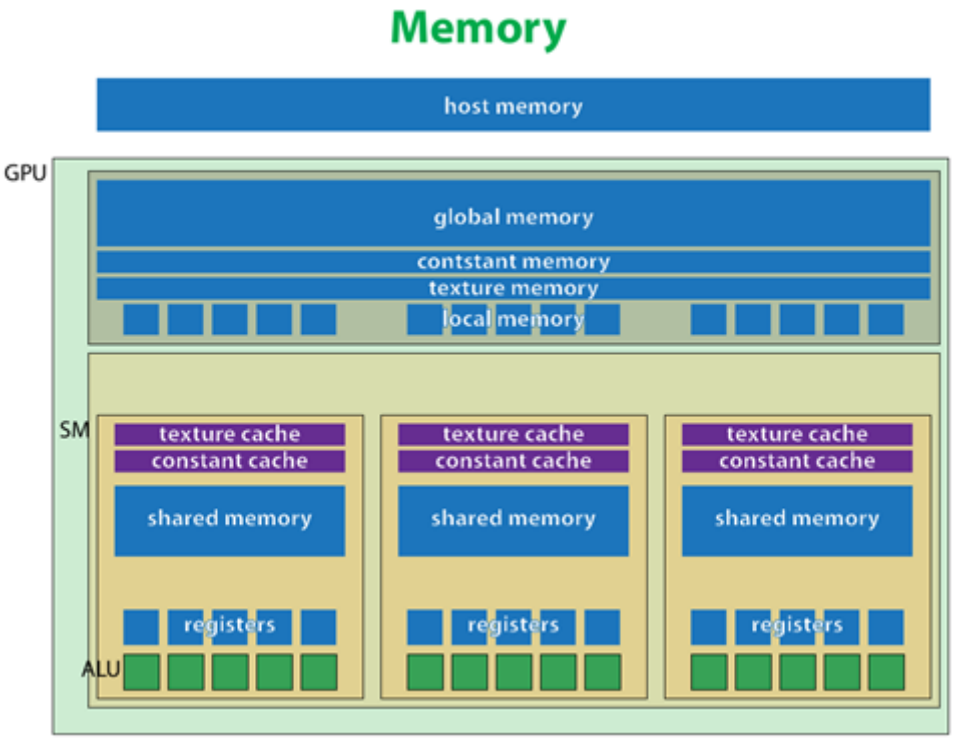


Figure 2.4: CUDA memory model [Nvi]

2.3 Cuda Insight Toolkit

Cuda Insight Toolkit (CITK) is an extension to the ITK architecture to support general-purpose computing on GPU. This is useful, since the GPU cannot access the host memory, and copying memory from the host to the device is a very slow process.

CITK makes it possible to process images on the graphics card by modifying the ITK architecture, so as to provide the input image on the device, whilst performing minimal host to device memory transfers. The results from the CITK filter development show anywhere from a 5x speedup (from AddImageFilter) to a 800x speedup (from MeanImageFilter) [cit]. The performance of CITK varies slightly based on the hardware and drastically based on its application.

It is not difficult for the existing users of ITK to enable ITK to use GPU technology.

What needed is only to acquire a CUDA enabled graphics card, extract the code and rebuild ITK in order to see the benefits of this technology. CITK follows the ITK style requirements and all CITK class names mimic their ITK counterparts with a CUDA prefix. It maintains all existing ITK features and provides the developer with access to the input image already on the device. This input image is referenced by a pointer and can be used within a CUDA kernel to execute the required algorithm. CITK also allows the developer to pass a pointer to the processed image on either device memory or host memory.

Chapter 3

Algorithm and Implementation

In this chapter, first of all, the algorithm is introduced in more details. And the second and the third part are the discussion about the C++ and CUDA implementation respectively.

3.1 Algorithms

The main goal is to find a set of parameters representing the deformation of previous image to estimate current image, and the idea of the existing implementation is based on the following formula:

$$\hat{p} = \underset{\hat{p}}{\operatorname{argmin}} \left(\sum_{\mathbf{x} \in \Omega} (s_2(\mathbf{x}) - s_1(\mathbf{x} + \nabla(\Lambda * \hat{p}_u)(\mathbf{x})))^2 \right). \tag{3.1}$$

Here, $s_1(\mathbf{x})$ is the previous image function for the position $\mathbf{x}$, while $s_2(\mathbf{x})$ is the current image. $\hat{p}$ is the parameters to be found, and $\hat{p}_u$ is the up-sampled version of $\hat{p}$, which is interpolated by the 2-D Lanczos filter Λ:

$$\Lambda(\mathbf{x}) = \begin{cases} \operatorname{sinc}(\frac{x}{d})\operatorname{sinc}(\frac{x}{3d})\operatorname{sinc}(\frac{y}{d})\operatorname{sinc}(\frac{y}{3d}) & \max(|x|, |y|) < 3 \\ 0 & \text{otherwise} \end{cases}. \tag{3.2}$$

Here d represents the number of pixels between two succeeding parameters in the po-

tential field. More about the formula is introduced in [WAH]. In order to improve the efficiency of the giving algorithm and implementation, two algorithms are applied in the implementations introduced below.

First is Levenberg-Marquardt optimization. According to Equation(3.1), the MSE (Mean Square Error) between s_2 and the deformation of s_1 should be minimized, which is a non-linear least squares minimization problem. It should be solved by an iterative method. In the existiong implementations, Gauss-Newton algorithm is applied to solve this problem. However, for each iteration, the step length of Gauss-Newton is large, which means, once the direction of next step is wrong, the large step length will lead to far away from the correct position. So it will take more iterations later to go back to the correct direction, which will increase the number of the iterations. Besides, this algorithm may converge slowly or not at all if initial guess is far from minimum. On the other hand, gradient descent algorithm is also a method to find a minimum in a function. It convergents well at the beginning of the iterations, but is linear and very slow in the final convergence.

Levenberg-Marquardt combines the advantages of gradient descent and Gauss-Newton algorithm, whose step $\mathbf{h}_{lm}$ is defined as:

$$(J^T J + \mu I)\mathbf{h}_{lm} = -J^T \mathbf{f} \quad \text{with } \mu \geq 0. \tag{3.3}$$

Here, $\mathbf{f}$ stands for the residuals of s_2 and the deformation of s_1 in Equation(3.1), that is, $\mathbf{f} = s_2(\mathbf{x}) - s_1(\mathbf{x} + \nabla(\Lambda * \hat{p}_u)(\mathbf{x}))$ in this case. And J is the Jacobian matrix, which contains all first-order partial derivatives of $\mathbf{f}$. The difference between Levenberg-Marquardt and Gauss-Newton algorithm is that there is an additional parameter μ in the step calculation of Levenberg-Marquardt method. Large value of μ makes the step much like a gradient descent step, while very small value makes it much like a Gauss-Newton step. For each iteration, if it is far from the minimizer, then μ will be

increased to take more of a gradient descent direction and reduce the step size; if it is closer to the minimizer, then μ will be decreased to take more of a Gauss-Newton step[KM01].

Second is PCG (Preconditioned Conjugate Gradient method). In order to solve the linear system, and find the step $\mathbf{h}_{lm}$ in Equation(3.3), PCG method is applied. Although for the linear system, a direct method, such as Gaussian elimination with pivoting, can be used and give an accurate solution, it does not perform well for solving a large number of equations in a system, particularly when the coefficient matrix is a spare matrix because it is computationally expensive. For an n $\times$ n positive definite linear system, Gaussian elimination method with pivoting needs n steps to find the solution, while PCG only needs about $\sqrt{n}$ steps.[KM01]

Preconditioned conjugate gradient method is an extension of conjugate gradient method (CG), which is an iterative algorithm to solve a linear system. CG method is very useful for the large sparse systems, but its speed is determined by the condition number of the coefficient matrix: if the condition number is large, the speed of convergence will be slow. A preconditioner is used in that case to reduce the condition number. CG method with a preconditioner is so-called PCG. It substitutes the linear equations $P^{-1}(A\mathbf{x} - \mathbf{b}) = 0$ for $A\mathbf{x} - \mathbf{b} = 0$ (the system of linear equations to be solved) with a preconditioner P. One of the simplest selections of P, which is also applied in the implementations to be introduced below, is the Jacobi (or diagonal) preconditioner: $P = diag(A)$, with the assumption that $A_{ii} \neq 0$, $\forall i$[pre].

3.2 CPU Implementation

Although the implementation of the algorithm in Matlab has been already given, the implementation in C++ is still necessary. This is because Matlab and C++ are much different. Matlab is interpreter, which is excellent for implementing mathematical

modeling in various algorithms. And it is easy and understandable dealing with those algorithm due to a few build-in functions. Therefore, Matlab is often used to implement the first version of a new algorithm, when result is more important than the speed and memory usage. The good thing is that it is efficient about the matrix operation, which is very common in image processing. However, it considers less about the memory storage and stores lots of large-size matrices in a code. On the other hand, C++ is a programming language, and it considers more about the memory storage and computation time. It is very slow to read random elements in a matrix, especially in a large-size matrix, but is fast to use a loop function, such as the `for` loop statement. And the memory storage and the execution speed are considered, which are important way to improve an existing algorithm to be efficient. In addition, the most important thing is that, implementation on CPU with C++ is a relatively direct comparison of parallel execution speed with that on GPU with Nvidia CUDA, even though there is some inevitable difference between these two ways of achievement regarding to the difference in essence.

Besides those common difference between Matlab and C++ programming, there are still three challenges for the implementation of Equation(3.1) in C++. First is the sparse matrix format. Unlike storing the complete Jacobian matrix J in Matlab, it is not stored in the C++ code due to its large size. For example, if the size of the input image is 512×512, which is the common resolution of a cardiac image, and the size of the parameter grid is 24×24, then the size of the Jacobian matrix would be $512^2 \times 24^2$. Therefore, not all elements of the Jacobian matrix J but elements of one row are calculated at one time, and multiplies with the transposed of itself and with the residuals respectively, adding the result into the right position of matrix $J^T J$ or vector $J^T \mathbf{f}$, as Equation(3.3) shows. Furthermore, the number of non-zero elements in one row of J is much smaller than the width of the matrix (24×24). As showed in Equation (3.2), a three-lobes 2-D Lanczos filter is used to interpolate the potential field, so the value of each pixel is determined by the 6 parameters surrounding in 1-D,

and 36 in 2-D case. In other words, in each row of J, which is corresponding to each pixel of the image, only 36 non-zero elements are calculated and stored each time.

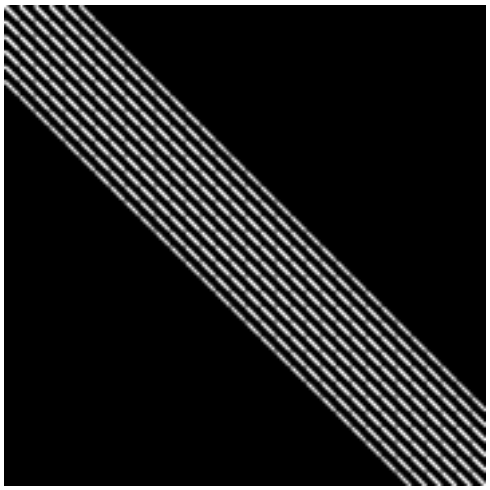


Figure 3.1: Structure of Sparse Matrix J^TJ [Bae12]

However the large size of matrix J^TJ has a storage problem. Since the complete matrix will be used as a part of coefficient matrix for PCG algorithm to find the step $\mathbf{h}_{lm}$ of Levenberg-Marquardt method (see Equation(3.3)), it does not work to calculate part of the matrix each time like the way done with J. If J has the size $512^2 \times 24^2$ as the examples given above, and then the size of J^TJ is $24^2 \times 24^2$, denoted by A. The structure of J^TJ is showed as Figure 3.1. The white parts stand for non-zero elements, while the black parts for zeros. The number of zeros is much larger than that of non-zero elements, so it is a sparse matrix, and it saves lots of memory if only the non-zero elements is stored. As mentioned above, in one dimension, the value of a certain pixel is determined by 6 surrounding parameter, and each column of J represents one parameter. For each parameter, it multiplies and is multiplies by the other 5 corresponding parameters as well as itself (only once), so it comes to 11 results as elements adding into one row of J^TJ. In two dimension, this is the same case, and for each row (also for each parameter) there are 11^2 elements. So the number of non-zero entries of J^TJ is $24^2 \times 11^2$ (the area of the white parts in Figure 3.1). As those 11

elements (121 elements in 2-D case) are symmetrical, only half of the entries need to be stored, so the size of matrix J^TJ is reduced to $24^2 \times 61$, and the way to transform that $24^2 \times 24^2$ sparse matrix (denoted by I) into this $24^2 \times 61$ matrix (denoted by II) will be discussed in section 3.3.

Besides the implementation in C++ with the same algorithm as Matlab, using Gauss-Newton method to solve the non-linear least squares minimization problem showed as Equation (3.1), there is another implementation in C++ applied Levenberg Marquardt method to solve that problem, which is the second challenge. The general algorithm is from [KM01]. Different from Gauss-Newton method, here the first calculation of matrix J^TJ and vector $J^T\mathbf{f}$ is at the beginning of the code before the iteration loop in order to initialize the parameter μ (see Equation(3.3)). Then in the interation loop, μ is updated each time to adjust the algorithm either closer to Gauss-Newton or closer to gradient descent method, as discussed in section 3.1. The result will be evaluated and the comparison with Gauss-Newton will be discussed in chapter four.

The third challenge for C++ implementation is the solver for sparse systems of linear equations. In the Matlab code, no iterative method is used to solve the system of linear equations $A\mathbf{x} - \mathbf{b} = 0$, but just do the "slash" operation. However in the C++ implementation, an efficient iterative method, PCG method, is applied, as mentioned in section 3.1. The function `CG` (see appendix A.1) is a linear equation solver specifically for this sparse matrix format, since its first parameter is a matrix with size $24^2 \times 61$, as discussed before. In this function, the first step is to find the position in matrix II for each element in matrix I, and then do the general PCG algorithm[BF08]. In addition, the last paramter μ is used to make the function compatible for both Gauss-Newton algorithm and Levenberg-Marquardt algorithm. μ is an additional parameter for Levenberg-Marquardt (Equation (3.3)), and it can be just initialized by 0 passed to the function when it is called as Gauss-Newton is applied.

3.3 GPU Implementation

The existing implementation in Nvidia CUDA is a 3-D version using Gauss-Newton algorithm (GNA) to solve Equation (3.1). In this thesis, two versions of 2-D implementaion in Nvidia CUDA are achieved. One applies GNA, the same as the 3-D version does, and the other implementation applies Levenberg Marquardt algorithm (LMA) to improve the algorithm by reducing computation time. In addition, those 2-D version use different sparse matrix format from what the 3-D version does, which improves the algorithm by saving memory storage. In the rest part of this section, the 2-D implementation with LMA is describes in details, since it is more complicated and quite different from GNA which is already introduced in the 3-D implementation in [Bae12].

The CUDA code is composed of two parts: host code and device code. The host code is something like the main function in a general CPU code, which runs on CPU, while the device code is something like the general function to be called when parallel computation need to be done, which runs on GPU. The code starts from the host code.

The first part is the initialization part. Global variables, pointers and the number of threads and threadblocks used for each kernel function as well as some parameters of LMA are initialized in this part.

The second part is the preprocessing part. First is to split the input image into small parts to achieve parallelization, as Figure 3.2 shows. The small square is called *microblock*, and the original input image (the light blue part) is expanded to a suitable size if necessary, so that all microblocks have same size. The padded area (the pink part) depends both on the size of the image and the number of downsampled potential field, in order to make sure that the potential field is in the middle of the original image. For example, if the size of the input image is 512×512 and the size of the

downsampled potential field is 24×24, then the microblock size is

$$ceil((512 \times 512)/(24 \times 24)) = 22 \times 22. \tag{3.4}$$

Each side of the padded part is equal to

$$(22 \times 24 - 512)/2 = 8 \tag{3.5}$$

The red point in each microblock is the downsampled potential field to be estimated and updated, while the dark blue points are used to pad the downsampled potential field in order to guarantee that one microblock contains one parameter, which are not calculated and updated.

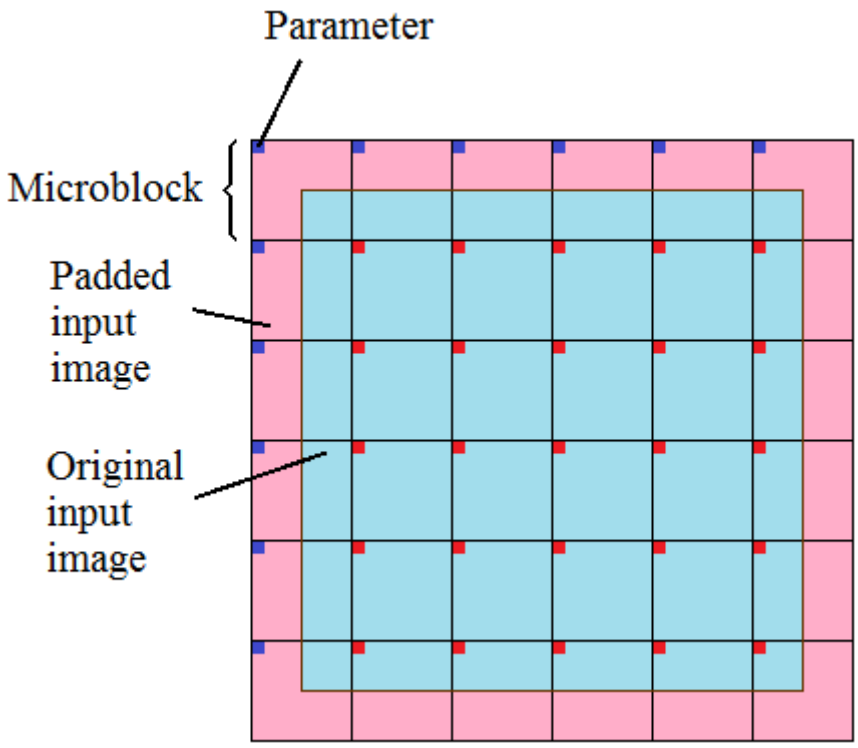


Figure 3.2: Microblocks with Parameters

Second of preprocessing part is to copy the input data to GPU. Firstly, a cuda function `cudaMallocArray` is applied to allocate two CUDA arrays in the GPU for two expanded input image. And then using the cuda function `cudaMemcpyToArray` to copy

an array from the memory area to the CUDA array, i.e., the image is copied into the array on GPU. Finally, cuda function `cudaBindTextureToArray` binds the CUDA array to the texture reference so that pixel value is available to access on GPU. [Bae12] gives some advantage of function `cudaBindTextureToArray`. Besides the input data, the memory of the output data (downsampled potential field) is also allocated and initialized with the cuda function `cudaMalloc3D` and `cudaMemset3D` respectively. Another way of memory allocation is for the input and output data of the PCG algorithm for linear equation solver, and it is done with cusp, which is a library for sparse linear algebra and graph computations on CUDA. Vectors can be defined with a given cusp function `cusp::array1d` and the corresponding pointers with a given thrust function `thrust::raw_pointer_cast`[cus].

Third of this part is Lanczos-Gradienten-Filter (LGF) pattern calculation, see the discussion in section 3.2. Taking the convenience of the program into account, there is a little change of Equation (3.1). The gradient is just applied directly to the Lanczos filter, with which to interpolate the parameters grid, and thus obtain the motion vector, which is the same way to calculation of LGF as 3-D version does. For more clear, a formula is given below:

$$\nabla(\Lambda * \hat{p}_u)(\mathbf{x}) = \nabla\Lambda * \nabla\hat{p}_u(\mathbf{x}) \tag{3.6}$$

The left is a part extract from Equation (3.1). Due to the fact that the positions of the parameters is exactly on the zero-crossings of the sinc function, the value of $\hat{p}$ is not changed when it is interpolated, i.e., $\nabla\hat{p}_u(\mathbf{x}) = \hat{p}_u(\mathbf{x})$. So Equation (3.6) modifies as

$$\nabla(\Lambda * \hat{p}_u)(\mathbf{x}) = \nabla\Lambda * \hat{p}_u(\mathbf{x}) \tag{3.7}$$

Since these patterns only have to be created once, they are created at the beginning

in the host code and then loaded along with the other required data to the GPU.

Fourth of the preprocessing part is to calculate the initial matrix $J^T J$ and the initial vector $JT\mathbf{f}$ of Equation (3.3), as mentioned in section 3.2. More information about the algorithm of LMA can be found in [KM01]. So the kernel function `TDCudaMedMotionEstimationImag` (See Appendix A.3) is called. A kernel function runs on GPU, and consists of several threadblocks. Each threadblock contains maximum 1024 threads. Each thread excutes the same code, and stores the variables or arrays either in shared memory or in global memory, depending on the number of data. The threads in one threadblock shares the data in the shared memory. The output of this function is the sparse matrix $J^T J$ (II), which is denoted in section 3.2 with the size $24^2 \times 61$. This function consists of four steps, whose basic idea is just like that in 3-D version.

Step One: calculating the prediction of the image to be estimated. Since the kernel code runs through every threadblock, the number of threads used in one threadblock is exactly the number of pixels to be calculated in one microblock, which means all the values of pixels are calculated parallel. The motion estimation of each pixel is computed in this step, and it is the key to obtain prediction of the previous image. As mentioned in Section 3.1, the motion estimation is equal to the gradient of filtered potential field with Lanczos filter, which is also the potential field filtered by the gradient of Lanczos filter. Here it is a discrete problem, so summing up the multiplication of the discrete pixel values and their corresponding filter values is the convolution of the filter and the previous image. These motion vectors are stored in the first two components in a `float3`, a kind of vector types, which has three components of each pixel: the motion estimation in x direction, the motion estimation in y direction and the pixel value of predicted image.

In host code, the alignment requirement of a vector type is equal to the alignment requirement of its base type. However this is not always the case in device code. For

instance, `float3` needs 12 bytes in host code (the base type `float` needs 4 bytes), but only 4 bytes is required for `float3` in device code[NVI12]. Pixel values of the predicted image stored in the third component of `float3` is equal to the sum of the pixel values in the previous image and the motion estimation in each direction. This vector is allocated on shared memory space of each thread block, specified by the variable type qualifier `__shared__` before the definition of vector. As mentioned in chapter2, shared memory is much faster than global memory, however it has the lifetime of the block and it is only accessible from all the threads within the block. Here in our case, the motion estimation and the prediction pixel values are only accessed in a threadblock, therefore the disadvantage is not a problem, and the fast access speed should be exploited. In addition, the is `float3` also declared as an external array, specified by `extern`. The size of the array is determined at launch time, i.e., when calling the `__global__` kernel function. It is given by the execution configuration, which is specified by inserting an expression of the form `<<<Dg, Db, Ns, S>>>` between the function name and the parenthesized argument list. Here `Ns` defines the size of the external array, which is of type `size_t` and specifies the number of bytes in shared memory that is dynamically allocated per block for this call in addition to the statically allocated memory. At the end of this step, an intrinsic function `__syncthreads` is called to synchronize all threads in the kernel, which acts as a barrier making all threads wait until all threads are ready to proceed.

Step Two: calculating the Jacobi matrix J and the residual between the predicted image and the real image. In kernel function, the computation within each threadblock is considered. Correspondingly, each microblock of the whole image is considered respectively. The Jacobi matrix has a large size and it is not required to store the whole matrix, since it is only used to obtain the multiplication matrix A and residual vector b for the later linear system solver computation, showed as Figure 3.1. They are stored in the shared memory because of the matrix multiplication. For each pixel, only 36 surrounding parameters determine its value, which are derived from the LGF pattern

(6 parameters in width and 6 parameters in height). Figure 3.4 illustrates the sub-matrix and its multiplication. The width of J is the number of relevant parameters for every pixel, which is 36, and the height is the number of pixels in one microblock, which is equal to the square of the size of a microblock. Although the whole microblock should be stored together, the size is still too large for the shared memory limited to 48Kbytes. Taking the input image with size 512×512 and the parameter grid with size 24×24 as an example, the size of a microblock is 22×22, and product is 17424, which needs 68Kbytes. So it has to be broken into smaller pieces with multiple times, and as a result, 16 rows of Jacobi matrix (i.e., 576 values) are computed in parallel, considering the maximum number of threads in one threadblock. However 32 rows are stored in shared memory at the same time for the later matrix multiplication computation. It is because that warps execute threads in groups of 32 parallel threads, and a warp executes one common instruction at a time, so full efficiency is realized when all 32 threads of a warp agree on their execution path, while different warps execute independently. Besides, the residual vector is also stored in the shared memory since the results of all the threads will be summed up.

Step Three: It is also achieved in the smaller potion instead of the whole microblock. It shares a large `for` loop with step two, and calculates the multiplication of 32 rows of sub Jacobi matrix and its transpose in Figure 3.4 in each loop, putting them into the right position of matrix A. That means, with the restriction of the capacity of shared memory, as mentioned before, the multiplication is done immediately after the calculation of 32 rows in sub Jacobi matrix. Consequently matrix A is filled up, when the `for` loop is terminated. On the other hand, these two steps do not share the same number of threads being used, because in step two what is considered is the elements in Jacobi matrix, while in step three is elements in matrix A. Since matrix A is symmetrical to the main diagonal, only entries of upper triangular need to be stored. In order to match the arrangement of threads in a threadblock better, the matrix A is transformed into a smaller matrix with lower height. Figure 3.5 shows the new position of elements.

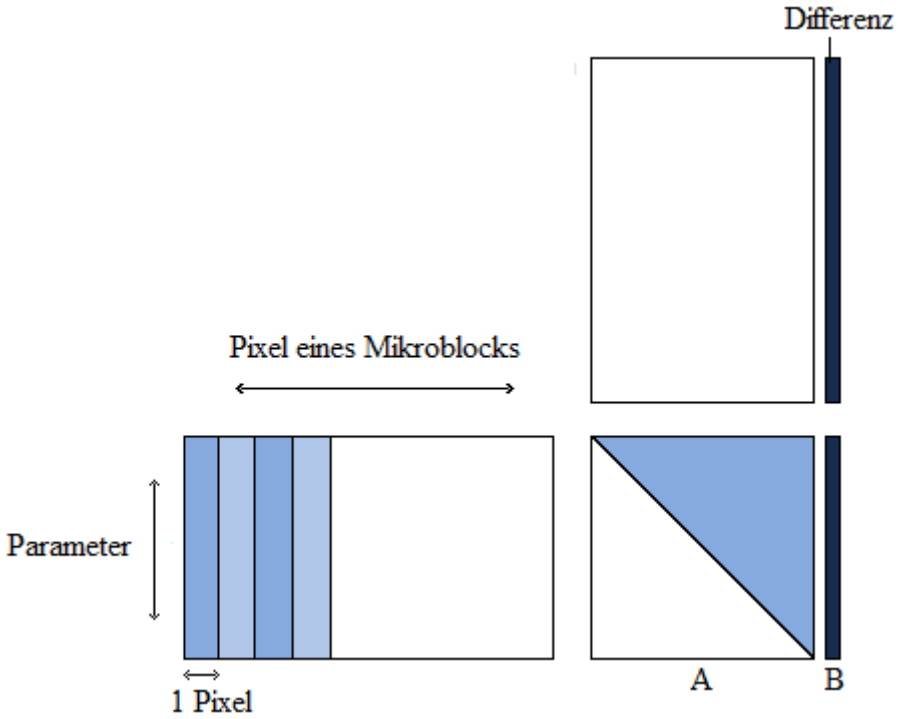


Figure 3.3: Multiplication in a Microblock[Bae12]

The whole sub matrix A is 36×36 (the width of Jacobi matrix is 36), and an additional vector b represents the result of the transpose of Jacobi matrix multiplies residual matrix (the difference between the motion compensation image and the real image), the same as the vector b showed in Figure 3.4. The largematrix A is now 36×37. After the transformation, the size of new matrix A is 18×39. Thus totally 702 threads need to be used in this step. Those 32 rows in Jacobi matrix derived from step two multiply with its transposed matrix, and each element of the result matrix is added into the right position in matrix A. Also vector b is calculated at the same time, as an additional column of matrix A in the right. Besides, the MSE is calculated here as well, by adding up the square of all elements in residual vector, and stored in a separate variable.

Step Four: Copying to global memory. As the sub matrix is available, the fourth step is to copy these values into the entire square matrix in global memory and the length of the side is equal to the number of parameters. In this step, we also consider about

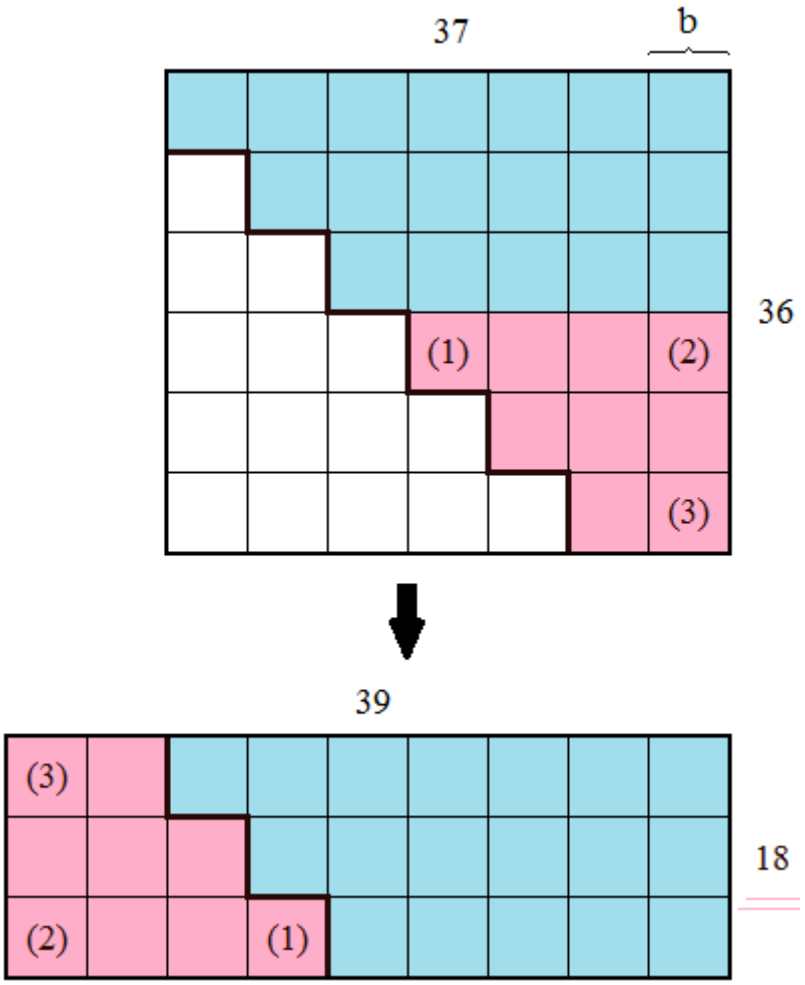


Figure 3.4: Matrix A Transformation

the elements in the sub matrix, and the way to copy them into global memory, so the number of threads being used is also 702. Since the register copy the data to the global memory automatically, it is incontrollable of the exact time when each thread copies, which leads to conflict of accessing the same variable in global with large chance. It can be avoided by using the atomic function `atomicAdd()`. This function guarantees that each thread performs the operation without interference from other threads, which means no other thread can access this variable address on global memory until the operation is complete. The entire vector b is also needed and copied into global memory for the later `cg` algorithm. In addition, the summation of MSE from all microblock is required to be stored into global memory as well for the stop condition of `while`

loop. Another issue to be considered in this step is the large size of entire matrix A, as mentioned in section 3.2. And also with the solution applied in CPU implementation, only the matrix with size $24^2 \times 61$ is stored instead of the original one with $24^2 \times 24^2$.

Fifth of the preprocessing part is to find the maximum value among the diagonal of matrix $J^T J$ (I), and use it to initilize μ in Equation (3.3).

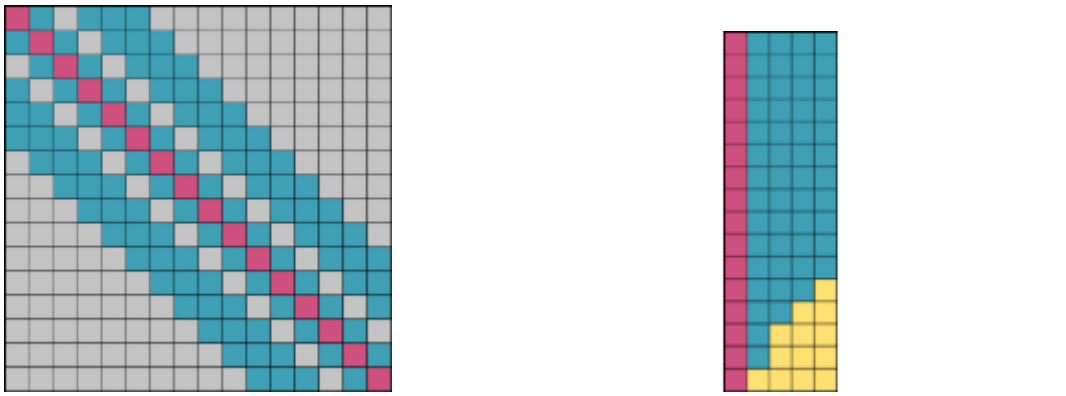
The third part of this CUDA code is the processing part. Here starts the iteration of LMA, and it is achieved in a *while* statement. The stop condition is that either the number of iterations reaches the maximum, which determined by the user when executing the program, or the latest MSE (Mean Square Error) is larger than the previous one, which indicates that the algorithm is no longer convergent.

Step One: solving system of linear equations with PCG solver. In order to use the exited CG function in cusp library, one of the matrix format supported by cusp library should be used. However no matter what standard matrix format needs at least one transformation from the generated matrix A into suitable format. Every non-zero entries needs, for example COO (Coordinate matrix format), three values to be represented, and for CSR (Compressed Sparse Row matrix format), more than two values is required, and the other three takes even more[cus]. However, Cusp permits users to define their own matrix formats without converting into one of Cusp's formats. It can be done with a user-defined linear operator, which takes in a vector x and compute the result $y = A \times x$. Appendix B.1 is the implementation of this linear operator. User is required to define class `mySparseMatrix`. It is the derived class from `linear_operator`, inheriting members from the base class, as well as defining its own members, like constructor and overloaded call operator. Firstly constructor is defined and initializes the data members of the class with an arbitrary sparse matrix format defined by the user. The parameter of the constructor is the matrix itself instead of a pointer to the matrix. And then the function-call operator is overloaded and it

takes two parameters: x and y of the linear equation $A \times x = y$. In the template, the operator function is just an example for copying two vectors, but not the necessary one.

In this overloaded call operator function, three pointers are created to A, x and y respectively. Since the matrix-vector computation is costly in time, it is done in the GPU with the kernel function `CudaMatrixMultiplication` (See Appendix A.4). The number of threads is set to be width of matrix A, that is 61 in our case, and the number of threadblocks is set to be the height of matrix A, that is 24^2 for the parameters grid is 24^2. Figure 3.6 shows the small example about the relationship with original sparse matrix and the transformed one. The gray part in (a) is the zero entries, which is not be stored in (b). The diagonal (red part) is stored in the first column in (b). The rest (blue part) is stored half amount, since they are diagonally symmetric, and only the part in the upper triangular is stored in (b). Besides, the yellow part in (b) does not exist in (a) and set to zero. The main idea of this kernel function is to find the right position in (a) for each element in (b), and then the element multiplies corresponding element in vector x, adding the result into the right position in vector y. There are three situations: The first is the element in blue part in (b), which has two positions in (a), so that it has two results obtained from the multiplication with two elements respectively in vector x and adds them into the corresponding position in vector y. Secondly is the diagonal part, which only has one position in (a), and adds once into y. The third situation is the yellow part in (b), which has no position in (a), and certainly, they do not need any calculation. After calling the kernel function, the class definition is complete.

When PCG algorithm is applied, the preconditioner is also need to be calculated. According to *cusp*, the parameter required to gain the preconditioner is also the same situation as the sparse matrix format. Therefore, the similar class is defined, as Appendix B.2 showed.



(a) Full Matrix A (b) Transformated Matrix A

Figure 3.5: Matrix A Transformation

In the host code again, the objects of those two classes are defined and initialized by the sparse matrix obtained from step four with the size $24^2 \times 61$. When `cg` function is called, the parameter *monitor* is calculated as an optional one. It allows users to define their own maximum iterations and the relative tolerance of cg algorithm, while the default convergence criteria is used without this parameter. There are three classes given by Cusp to calculate the monitor. The most basic one is `default_monitor`. `Verbose_monitor` and `convergence_monitor` are the derived classes from `default_monitor`, and display some additional information of the solver status during iteration and report a summary when iteration has stopped. `Verbose_monitor` gives the status for each iteration like the iteration number and the residual norm after each iteration, while `convergence_monitor` gives the global status like a final residual and some convergence factors. And then `cg` function is called. It is defined by the user to apply this function with or without monitor or preconditioner, and since the function is overloaded, it is convenient to switch among them by using different parameters. Besides

several other iterative solver for linear system are available, such as `bicgstab` using Biconjugate Gradient Stabilized method, and `gmres` using Generalized Minimal Residual method for non-symmetric linear system [cus].

Step Two: After solving the linear system, the parameters grid can be updated by the solution obtained in previous step. The kernel function `CudaSolToOutKernel` (See Appendix A.5) is called to calculate the euclidean norm of the downsampled potential field, and then to weight the solution vector with *stepsize*, adding the elements to the corresponding parameters. These updated parameters is used to renew the potential field.

Step Three: updating parameter μ. Firstly, step of LMA (see Equation 3.3), i.e., the solution of PCG is evaluated. If it satisfies certain condition regarding to the euclidean norm calculated in the previous step (more details see [KM01]), then the while loop stops; otherwise renew the potential field. And then Kernel function `TDCudaMedMotionEstimationImageFilterKernel` (See Appendix A.3) is called again to calculate and update SSE (sum of squared errors). After that, the approximation is evaluated. If it is close to the minimizer, then μ will be decreased, otherwise μ will be increased, as discussed in section 3.1.

The fourth part of this CUDA code is the postprocessing part. When either current MSE is larger than previous MSE or the iteration times reaches the maximum, the `while` is stopped. The result of parameters grid is copy to CPU and be used as an input in the next motion compensation part, which is to reconstruct the image with the previous one and the given parameters grid (see Appendix A.6). In that compensation function, the fourth kernel is used. The method of motion compensation is almost the same as the method to obtain the predicted image in the first step in motion estimation function. Consequently, a reconstructed image is available, and the results will be comparison and analysis in the next chapter.

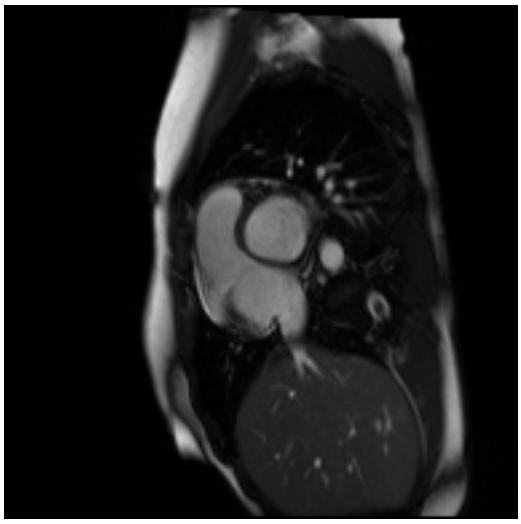
Not only the 2-D implementation in CUDA applied LMA, but also the existing 3-D implementation is modified by substituting LMA for GNA. the results also will be discussed in next chapter.

Chapter 4

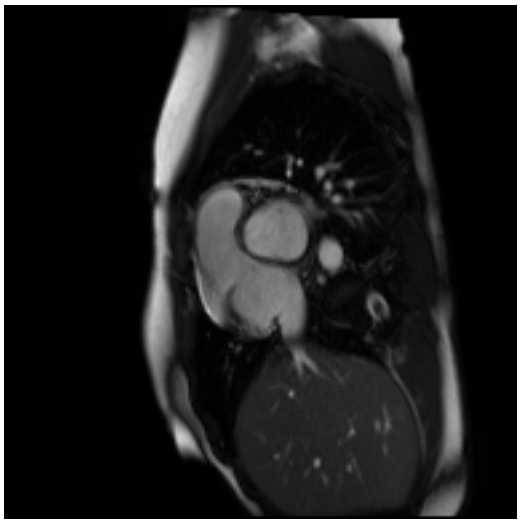
Test and Evaluation

In this section, results of the CUDA 2D implementation (on GPU), the CUDA 3D implementation (on GPU) [Bae12], C implementation (on CPU) and MATLAB implementation (on CPU) [WAH] are showed and compared. For the CUDA implementation, the used GPU device is an Nvidia GeForce GTX 480. All these four implementation are in 2-D mode. First, a pair of CT data with image size 512×512 is taken as the input data. Besides, MRI data is also of great interest. Therefore another pair of MRI data is used too, in order to evaluate the performance of these implementations on MRI data. The input MRI data is with image size 192×192, since MRI data with image size 512×512 is not available yet. The MRI data is shown in Figure 4.1.

The motion estimation was carried out between the two images of each pair of data. The output data for each implementation contains the lattice parameters and the motion-compensated image, which are shown in Figure 4.2. Besides, the difference between the original images and the predicted images for 2-D implementations in CUDA are also calculated and shown in Figure 4.2. From these error images we can see, they are mostly consisted of the high-frequency part of the original image, which can be considered that the predicted images are lack of high-frequency component. That is corresponding to the method of [WAH].

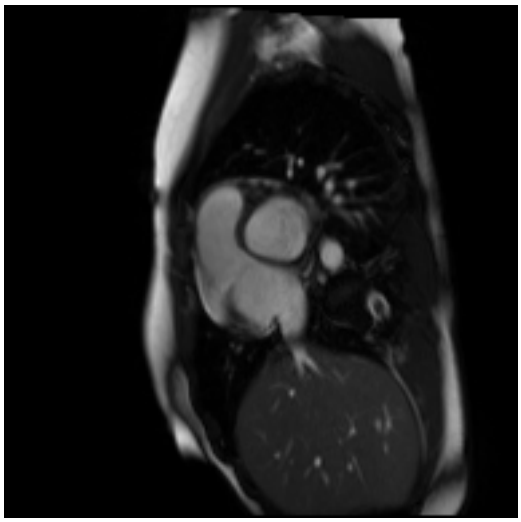


(a) Previous Image

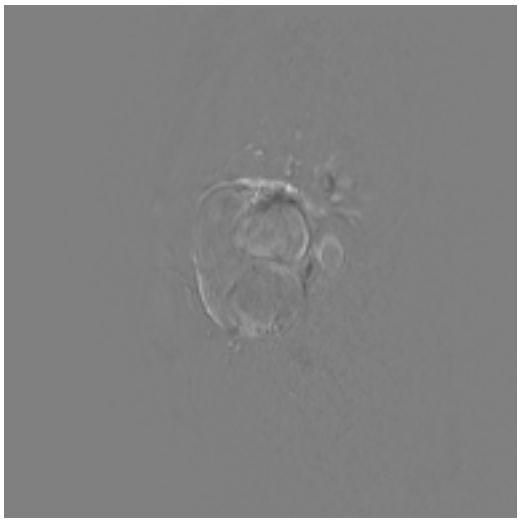


(b) Current Image

Figure 4.1: Input data from HEART_MRI



(a) Reconstructed Image



(b) Difference Image

Figure 4.2: Reconstructed Image and Difference Image

4.1 Numerical Comparison

In this section, the results are shown, compared and analyzed numerically, which contains the run time and MSE under different algorithms and for different image sizes for 2-D mode.

Firstly, it shows the comparison between different algorithms. Table shows below the average run time under Gauss-Newton method and Levenberg-Marquardt method for the CT data with image size 512 × 512. The lattice parameters here is 24 × 24 for the 2-D case. It can be seen that the CUDA 2D implementation spent less run time under both algorithms compared to the other three implementations. Besides, from the comparison between Gauss-Newton algorithm and Levenberg-Marquardt algorithm, it can be seen that for each iteration in these three implementations, the run time for LMA. But the total run time of Levenberg-Marquardt method was less than Gauss-Newton, because it used less iterations based on its two added conditions of termination.

	Matlab (GNA)	Matlab (LMA)	C++ (GNA)	C++ (LMA)	CUDA-2D (GNA)	CUDA-2D (LMA)
Run time per iteration	3.457s	-	6.6044s	6.4616s	0.138s	0.1245s
Total run time	345.714s	-	660.44s	646.16s	13.610s	12.451s

Table blow shows the mse (mean squared error) for these four implementations with image size 512 × 512 and lattice parameters 24 × 24. Mse stands for the quality of the prediction. From this table, it can be seen that, mse showed not quite much difference between GNA and LMA. That's because 100 iterations were not enough for LMA to reach a certain low mse, where the gradient descent algorithm could apply.

	Matlab (GNA)	Matlab (LMA)	C++ (GNA)	C++ (LMA)	CUDA-2D (GNA)	CUDA-2D (LMA)
Run time per iteration	40009	-	40001	40024	42259	42253
Total run time	27842	-	27831	27920	26019	26674

Next, the MRI data is used for test. Table below shows the average run time under Gauss-Newton method and Levenberg-Marquardt method for the MRI data with image size 192 × 192. The lattice parameters here is 12 × 12 for the 2-D case. Similar as the CT data, it can be seen that the Cuda 2D implementation is still the fastest. And also the LMA cost less run time compared with GNA.

	Matlab (GNA)	C++ (GNA)	CUDA-2D (LMA)
Run time per iteration	0.6020s	0.7389s	0.036
Total run time	39.1294s	48.03s	0.604

Table below shows the mse (mean squared error) for MRI data with image size 192 × 192 and lattice parameters 12 × 12. From this table, it can be seen that, mse are nearly the same, which mean both two algorithms can reach a certain quality if iterations are enough.

	Matlab (GNA)	C++ (GNA)	CUDA-2D (LMA)
Original mse	1614	1614	1606
Final mse	1381	1381	1381

4.2 Convergence

In this section, the convergences are shown. Figure 5.4 shows the convergences of different algorithms for the CT data with image size 512 × 512. And Figure 5.5 shows the convergences of different algorithms for the MRI data with image size 192 × 192.

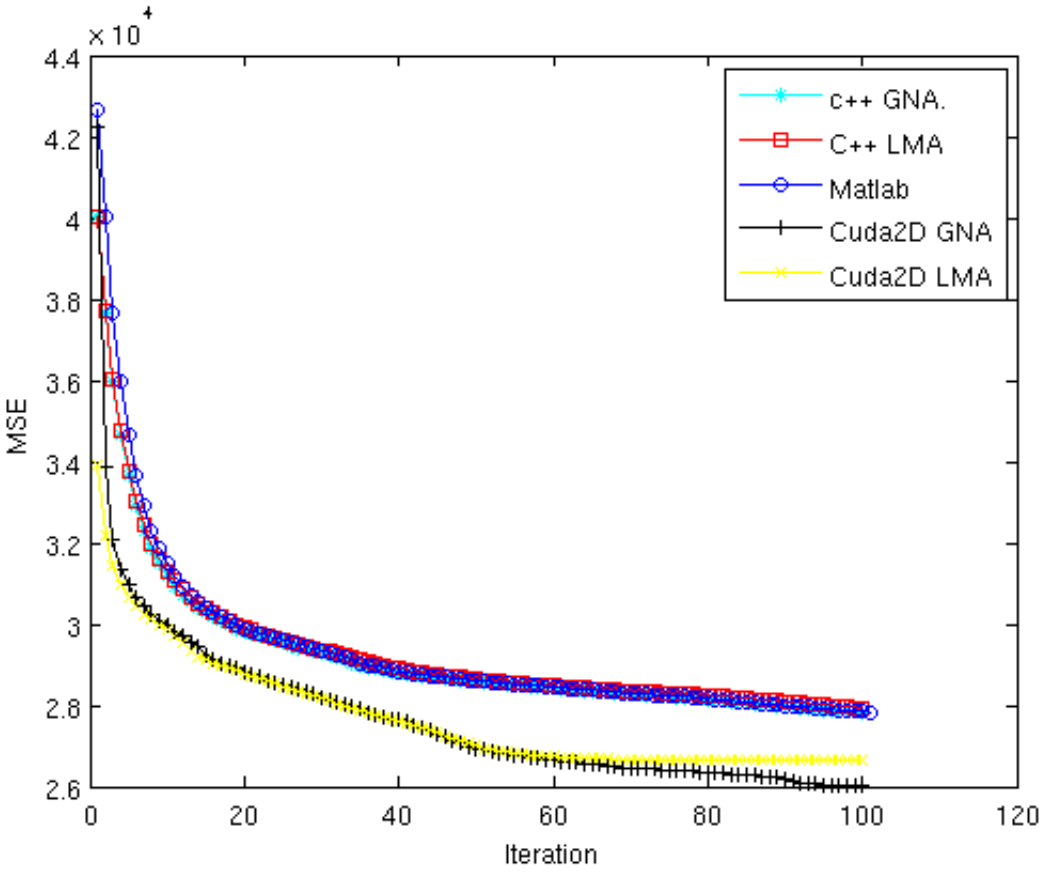


Figure 4.3: Convergences of CT Data

From these two figure, it can be seen that the Cuda 2D implementation has the better convergence. But the difference between GNA and LMA is not so obvious.

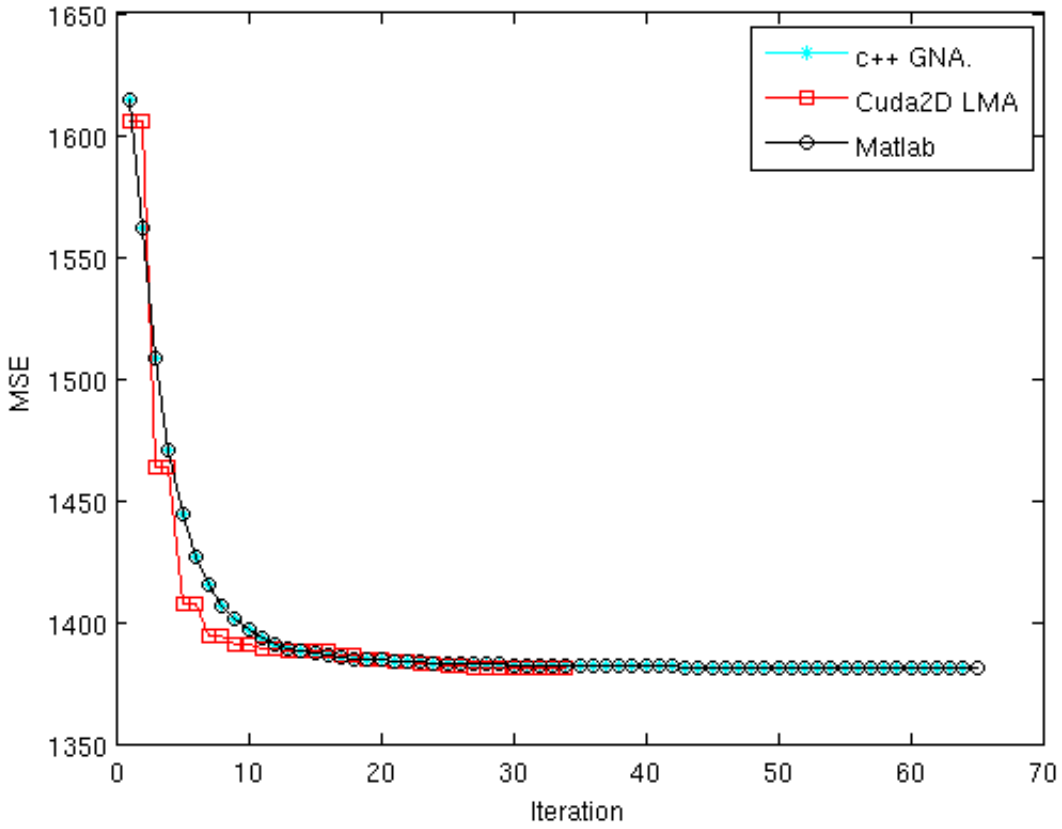


Figure 4.4: Convergences of CT Data

4.3 Memory Storage

In Cuda 2D implementation, the usage of memory is also considered as a performance of the method. For the Cuda 3D implementation, it use the COO and CSR matrix as the storage of the sparse matrix for CG algorithm. However, these two matrix are quite large, and the Cuda 3D implementation normally cost 164MB during the estimation period. Unlike the Cuda 3D implementation, the Cuda 2D implementation use a new compressed matrix to store the sparse matrix, as discussed in Chapter 3. According to the statistical result, this kind of storage costs 72MB during the estimation, which is an improvement compared with the COO and CSR standard for this problem.

Chapter 5

Conclusion

According to the algorithm about deformable motion compensation and the 2-D implementation in Matlab given by [WAH] and the 3-D implementation in Nvidia CUDA by [Bae12], the slice-based motion estimation and compensation for 2-D medical image data is implemented and improved in this thesis. Levenberg Marquardt algorithm is applied for both 2-D and 3-D implementations in C++ or in CUDA to enhance Gauss-Newton algorithm for larger step sizes and less iterations. And a special sparse matrix format are established to save memory storage for our special case to store large matrices for further solver for sparse system of linear equations using PCG method, and skips the COO and CSR format in CUDA implementaion. Besides, MRI data are tested as well as CT data.

Although it is fast for 2-D implementation on GPU, there is still some modification can be done to speed up. Firstly, in the user-defined linear operator class for sparse matrix format transformation, now the entire matrix is passed as the argument into the member function. However if only the pointer points to the matrix that is passes into the function, half of the memory will be saved. The same as the class for preconditioner generation. Secondly, a multi-resolution approach can be developed that first finds the motion in a downsampled data set and then uses it to find the motion in the high resolution faster.

Appendix A

Functions calling

A.1 Call the PCG Function

```
template <typename T>
void CG(T A[][61], T *b, const unsigned int rowsize, double err,
        int paramszX, T *x, float mu)
```

A.2 Call the Motion Estimation Function

```
template <typename T, typename S>
void
TDCudaMedMotionEstimationImageFilterKernelFunction(const T* input1,
const T* input2, S* output, unsigned int input_width,
unsigned int input_height, unsigned int output_width,
unsigned int output_height, unsigned int max_iter, float stepsize)
```

A.3 Call the Kernel-1 Function

```
template <typename T, typename S>
__global__ void TDCudaMedMotionEstimationImageFilterKernel
(cudaPitchedPtr outPtr, cudaExtent outExtent, cudaPitchedPtr patternX,
cudaPitchedPtr patternY, T* bufPtr, T* rhsPtr, unsigned int blksizeX,
unsigned int blksizeY,
float stepsize, T* mse)
```

A.4 Call the Kernel-2 Function

```
__global__ void CudaMatrixMultiplication (const float* buf, const float* x,
float* y, int paramszX, int paramszY, size_t col_num, float diag)
```

A.5 Call the Kernel-3 Function

```
template <typename T, typename S>
__global__ void CudaSolToOutKernel(cudaPitchedPtr outPtr, T* solPtr,
float stepsize, T* solnorm, T* paranorm)
```

A.6 Call the Motion Compensation Function

```
template <typename T, typename S>
void
TDCudaMedMotionCompensationImageFilterKernelFunction(const T* input1,
const S* input2, T* output, unsigned int image_width,
unsigned int image_height, unsigned int param_width, unsigned int param_height,
```

```
float stepsize)
```

A.7 Call the Kernel-4 Function

```
 template <typename T, typename S>
__global__ void TDCudaMedMotionCompensationImageFilterKernel(cudaPitchedPtr paramPt
cudaPitchedPtr outPtr, cudaPitchedPtr patternX,cudaPitchedPtr patternY,
unsigned int blksizeX, unsigned int blksizeY, float stepsize)
```

Appendix B

User-Defined Linear Operators

B.1 mySparseMatrix.h

```
/*=================================================
User defined class for sparse matrix
=================================================*/



#pragma once

#include <cusp/detail/config.h>
#include <cuda.h>
#include <cusp/format.h>
#include <cusp/blas.h>
#include <cusp/exception.h>
#include <cusp/detail/matrix_base.h>
#include <cusp/linear_operator.h>
#include <cusp/array2d.h>
```

```
__global__ void CudaMatrixMultiplication (const float* A, const float* x,
         float* y, int paramszX, int paramszY, size_t Apitch, float diag);

namespace cusp
{
  class mySparseMatrix : public cusp::linear_operator<float,
                                                       cusp::device_memory>
  {
    public:
    typedef cusp::linear_operator<float,cusp::device_memory> Parent;

    int paramszX, paramszY;

    cusp::array1d<float, cusp::device_memory> Abuf;
    size_t APitch;
    float diag;

    // constructor
    template <typename MatrixType>
    mySparseMatrix(int bufparamszX, int bufparamszY, MatrixType& buf,
                   size_t bufPitch, float bufdiag)
      : Parent(bufparamszX*bufparamszY, bufparamszX*bufparamszY) {
          paramszX = bufparamszX;
          paramszY = bufparamszY;
          Abuf = buf;
          APitch = bufPitch;
  diag = bufdiag;
      }
```

```
    // linear operator y = A * x
    template <typename VectorType1, typename VectorType2>
    void operator()(const VectorType1& x, VectorType2& y) const
    {
      const float* x_ptr = thrust::raw_pointer_cast(&x[0]);
      float* y_ptr = thrust::raw_pointer_cast(&y[0]);
      const float* A_ptr = thrust::raw_pointer_cast(&Abuf[0]);

      dim3 threadsPerBlock_K3C(paramszX * paramszY);
      dim3 numBlocks_K3C(APitch);

      CudaMatrixMultiplication<<<numBlocks_K3C, threadsPerBlock_K3C>>>
                (A_ptr, x_ptr, y_ptr, paramszX, paramszY, APitch, diag);
    }
  };

} // end namespace cusp
```

B.2 myPreconditioner.h

```
/*=====================================================
User defined class for preconditioner
=====================================================*/



#pragma once

#include <cusp/detail/config.h>
#include <cuda.h>
#include <cusp/format.h>
#include <cusp/blas.h>
#include <cusp/exception.h>
#include <cusp/detail/matrix_base.h>
#include <cusp/linear_operator.h>

__global__ void CudaMatrixMultiplication (const float* A, const float* x,
        float* y, int paramszX, int paramszY, size_t Apitch, float diag);

namespace cusp
{
class myPreconditioner : public cusp::linear_operator<float,
                                                cusp::device_memory>
{
    public:
    typedef cusp::linear_operator<float,cusp::device_memory> Parent;

    int paramszX, paramszY;
```

```
cusp::array1d<float, cusp::device_memory> Mbuf;
size_t MPitch;
float diag;

// constructor
template <typename MatrixType>
myPreconditioner(int bufparamszX, int bufparamszY,
                 MatrixType& buf, size_t bufPitch, float bufdiag)
    : Parent(bufparamszX*bufparamszY, bufparamszX*bufparamszY) {
        paramszX = bufparamszX;
        paramszY = bufparamszY;
        Mbuf = buf;
        MPitch = bufPitch;
diag = bufdiag;
    }

// linear operator y = A * x
template <typename VectorType1, typename VectorType2>
void operator()(const VectorType1& x, VectorType2& y) const
{
    const float* x_ptr = thrust::raw_pointer_cast(&x[0]);
    float* y_ptr = thrust::raw_pointer_cast(&y[0]);
    const float* M_ptr = thrust::raw_pointer_cast(&Mbuf[0]);

    dim3 threadsPerBlock_K3C(paramszX * paramszY);
    dim3 numBlocks_K3C(MPitch);

    CudaMatrixMultiplication<<<numBlocks_K3C, threadsPerBlock_K3C>>>
```

```
                (M_ptr, x_ptr, y_ptr, paramszX, paramszY, MPitch, diag);
    }
 }; // myPreconditioner

} // end namespace cusp
```

List of Figures

Bibliography

[Bae12] Baetz, Michel: Entwurf und Implementierung einer voxelbasierten Bewegungsschaetzung fuer 4-D-Medizindaten in Nvidia CUDA. (March 2012)

[BF08] Burden, Richard L. ; Faires, J. D.: *Numerical Analysis*. Thomson Learning, Inc., Augest 2008

[cit] *Cuda Insight Toolkit (CITK)*. `https://code.google.com/p/cuda-insight-toolkit/`. – [Online: accessed March 31, 2013]

[cus] cusp: *cusplibrary*. `https://github.com/cusplibrary/cusplibrary`. – [Online: accessed March 22, 2013]

[ISNC05] Ibanez, Luis ; Schroeder, Will ; Ng, Lydia ; Cates, Josh: The ITK Software Guide Second Edition Updated for ITK version 2.4. (November 21, 2005)

[KM01] K. Madsen, O. T. H.B. Nielsen N. H.B. Nielsen: *Methods for Non-Linear Least Squares Problems (2nd Edition)*. Informatics and Mathematical Modelling, Technical University of Denmark, April 2001

[Nvi] *Nvidia CUDA: preview-BeHardware*. `http://www.behardware.com/art/imprimer/659/`. – [Online: accessed March 31, 2013]

[NVI12] NVIDIA: NVIDIA CUDA C Programming Guide version 4.2. (April 16, 2012)

[pre] *Preconditioner*. http://en.wikipedia.org/wiki/Preconditioner. – [Online: accessed March 31, 2013]

[WAH] Weinlich, Andreas ; Amon, Peter ; Hutter, Andreas: Representation of Deformable Motion for Compression of Dynamic Cardiac Image Data.