

Parallelization of Mutual-Information based Registration in the ITK Toolkit using CUDA and TBB

Abstract

General purpose computation on GPUs (GPGPU) has been gaining high popularity in many fields of research in recent years. However, not many studies have been reported in medical image processing. Medical image processing can be one of the most time critical research areas because a fast processing is a key success factor during surgical operations.

In this paper, we parallelized one of the registration algorithms from ITK (Insight Toolkit from National Library of Medicine) on the GPU and CPU. We implemented CUDA and TBB versions of a mutual-information based registration application, which can be directly used by other users. We implemented four different versions of GPU code and one TBB code. Our results showed that the best optimized GPU code can achieve 14.61x speedup in the kernel itself and up to 5.91X speedup on the entire application. The TBB version of the code also shows 5.28X speedup in the kernel and 3.43X speedup for the application using a 8-core processor.

1. Introduction

The computing power of GPUs has been increasing rapidly over past few years. NVidia's GTX280 architecture [1] provides 933 Gflop/s with 240 cores, while Intel's next generation processor will support more than 900 Gflop/s. To take advantage of these high computing power, more and more programmers are interested in using GPUs for other computing applications. Furthermore, with the introduction of a new programming language for GPUs, such as CUDA [15], porting of highly-parallel sequential codes into the GPU has started from various areas of research and computationally-intensive applications.

Not much GPGPU work is evident in the areas related to Medical Image processing. However, there is no doubt that medical image processing area is one of the most time critical and important areas, especially during the surgical operations. Considering the high ratio of GPU performance to the costs, the potential roles of GPUs in the *actual* operative rooms in the hospitals is not unrealistic.

In order to carry out the GPGPU movement into the medical field and to generate a high impact from parallelization, ITK (Insight Toolkit from National Library of Medicine) is chosen for its completeness as an application and the

large audience of users worldwide. The ITK Toolkit can be divided into two fields; segmentation and registration. Segmentation is the process of identifying and classifying an image data. For medical images, this segmentation process refers to locating tumors and other pathologies. On the other hand, registration is the process of aligning between two images of multiple dimensions to combine the useful information. Two 3D images can be different in their scale of the pixel, and the orientation with respect to each other (e.g., CT¹ vs. PET²).

In our work, a highly used mutual-information based image registration application called MultiResMIRegistration is picked for the parallelization on the GPU and on the CPU using Intel's TBB library [10]. The details of the application are discussed in the subsequent sections.

We implemented four different versions of GPU which vary in their optimization levels. We also compared the GPU results with the CPU multithreaded version using TBB library on an 8-core machine. 14.61X speedup is achieved only for the kernel, and the corresponding application speedup of 5.91X is achieved for this registration application in the ITK Toolkit on the GPU. The TBB version provides 5.28X speedup for the kernel itself and 3.43X speedup for the same application.

We found out two major difficulties in parallelizing ITK using CUDA. First, it is the highly modularized implementation of the ITK Toolkit where each function is processing only a small subset of data. Secondly, it is the highly object-oriented programming environment where each function is frequently defined in other classes with virtual function calls and inheritances. Both increase the complexity of the programming and reduce the benefit of using the GPU.

This is the first work which directly parallelized the application from the *complete* toolkit in use. This has several advantages compared to parallelizing a kernel only containing just the targeted algorithm since from our results the users can directly benefit from the performance improvement. This work sets the first step towards started GPGPU movement for the ITK Toolkit [22].

2. Background

ITK (Insight Segmentation and Registration Toolkit) [22] is an open-source software system that employs leading-edge segmentation and registration algorithms in two, three and more dimensions. The concepts of registration and segmentation are discussed in details in the following paragraphs. ITK is powerful in processing many kinds of images where most file types and multiple dimensions of images (i.e., 3-dimension and more) are supported. Not only familiar image processing algorithms are provided, but ITK is equipped with several algorithms which can

¹Computer Computed tomography (CT)

²Positron emission tomography (PET)

process clinically valuable operations between two 3D images which can vary in the scale of the pixel (i.e., CT and MRI) and alignment (i.e., one image is rotated compared to other). The ITK toolkit is open-source which supports cross-platform by using CMake [4] to manage the compilation process. It provides a wrapping process (Cable [3]) to interface between C++ and programming languages such as Tcl, Java, and Python.

2.1. Segmentation

The ITK Toolkit can be divided into two fields; segmentation and registration. Segmentation is the process of identifying and classifying data found in a digitally sampled representation. In other words, an image is partitioned into multiple regions for a better analysis of the image. Segmentation in the medical imaging area provides many benefits such as locating tumors and other pathologies. ITK provides several segmentation algorithms such as region-growing [9]. For region-growing, the desired point in an image is selected and the region associated with the point iteratively grows by comparing the intensity of the neighboring pixels.

2.2. Registration

Registration [8] is the process of determining a transformation that maps points from one image to points in the other image. Images can be multi-dimensional and they can vary in the pixel scale. For example, a typical CT image has a pixel size in the order of 1 millimeter, while a typical PET image is in the order of 5 millimeters to 1 centimeter [9]. For this reason, a naive image mapping will not simply work well at the pixel granularity of images. Thus, prior to registration, the pixel values are converted into the *actual* spatial coordinates by using the pixel size information provided from the user. Then, the registration process is carried out in the actual space without considering the scaling factor associated between two images.

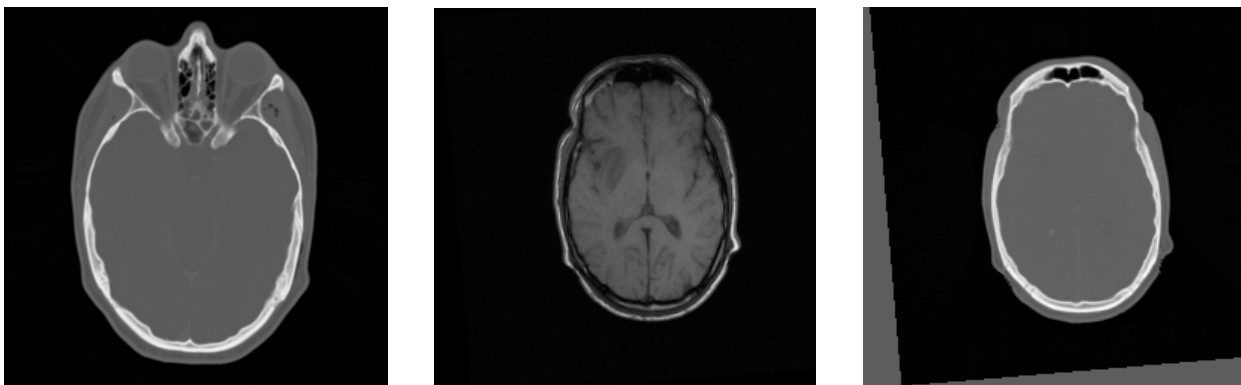


Figure 1. Registration between CT and MRI 3D images by using MultiResMIRegistration Algorithm [13]

Figure 1 illustrates a registration process on the two 3D input images. The first image is called *moving* image, while the second image is *fixed* image. Note that those images not only differ in the number of pixels, but differ in

pixel scales as well (i.e., CT, MRI). When registration is carried out between these two images, the output containing the information of how the image should be rotated and translated is produced.³ Then, the images can be aligned and the information from both images is obtained as shown in the third image. Transformation and combined information obtained from the registration process have several benefits in surgical operative environments where multiple imaging devices produce different types of outputs.

2.3. Registration Framework in ITK

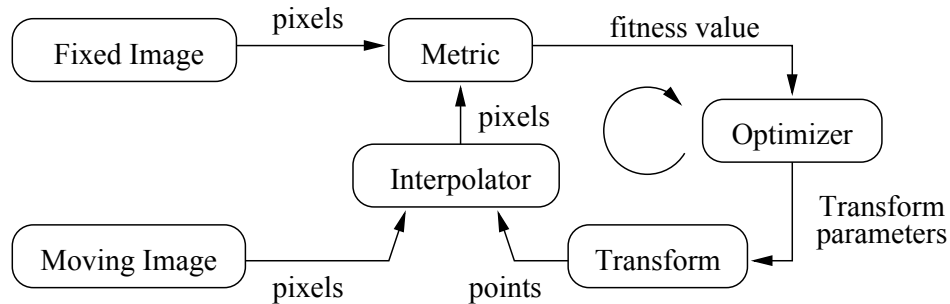


Figure 2. Registration Framework [9]

Figure 2 shows the high-level view of the registration process which is represented by six blocks. A structural modular approach [6] is chosen by ITK such that programmers can choose a desired algorithm or technique for each block. Two blocks on the left-side represent the input images which are moving and fixed. *Metric* represents the quantitative criterion which specifies how well the moving image has transformed to fit the fixed image [9]. Several algorithms available for the *Metric* in the ITK toolkit include (1) mean Squares, (2) pattern intensity, and (3) mutual Information. *Interpolator* is used to evaluate image intensities at non-grid positions (i.e., when an image moves or rotates, a computed pixel location may fall between grids) which is a common technique in the image processing field. Since a value of the *Metric* represents how well two images are aligned, the purpose of the *Optimizer* block is to search for the parameters of the *Transform* which optimizes the *Metric*. Given the parameters from *Optimizer*, the moving image is transformed by the *Transform* block. Some of the optimizers available in the ITK toolkit include (1) gradient descent, (2) conjugate gradient, and (3) regular step gradient descent. Detailed explanation of the metric and optimizers can be referenced by [8][9].

2.4. Viola-Wells Mutual-Information Metric

For the *Metric* block in Figure 2, Mutual-information based metric based on the work by Viola and Wells [23] is chosen for our work. One primary reason is that the naive mean-squares metric between two images that differ

³Mutual-information based registration is in the category of rigid registration. Output contains a rotate and transform matrix

in the modality simply will not work. However, by using the statistical information between image intensities as random variables, this metric is able to assess the images from a higher level. It is a very general and powerful statistical metric that can work on the images that differ in the resolution and modality. However, since this metric is calculating the criterion stochastically, the values are too noisy to work with a simple, less computationally intensive *Optimizer* [9]. Thus, the optimizer frequently used with the mutual-information based metric is called *GradientDescentOptimizer* which adjusts the transform parameters (i.e., values of output matrix) in the direction of the gradient of the metric. This comes with a highly increased computation cost from the metric block. Because it needs to provide not only the metric values as criteria, but the derivative values of the metric values for the optimizer as well. Additionally, how much the optimizer updates the transform parameters depends on the user specified learning-rate values. If the specified learning-rate values are too small, registration will not work between images that are too far off. However, if the learning-rate values are too large, registration will not produce parameters that match the images in fine detail. Due to this nature of mutual-information based registration, the registration process is first carried out with high learning-rate values. Then, the same registration process is repeated again with lower learning-rate values for the fine-control. For our experiment, this process is repeated five times (i.e., No. of Multi-Resolutions in Table 2) with decreasing learning-rate values in series. For each learning-rate, the optimizer updates the parameters for the *iterations*⁴ that the user specifies. Thus, the computation cost of this Mutual-Information based registration is not only increased due to the additional derivative calculation, but also from the repeated sets of the experiment.

3. GPU Implementation

3.1. Profiling MultiResMIRegistration

By executing the MultiResMIRegistration application in the ITK Toolkit which has the Mutual-Information based registration and GradientDescentOptimizer mentioned in the previous section, the profile data obtained by GNU Gprof 2.17.50.0.6 version [7] shows that the `GetValueAndDerivative()` function takes up about 11.31% of the total execution time. However, this profiling data is not accurate since Gprof does not profile multithreaded applications correctly under certain kernels such as Linux [21] (i.e., Gprof only profiles the main thread). Manual profiling shows that the `GetValueAndDerivative()` function is invoked for every *iteration* discussed in Section 2.4. For the BW_2.5K input where the number of iterations is 2500, the total invoked number of the function is equivalent to 12500 after multiplying by the number of Multiresolutions in Table 2. A single invocation of this function takes about 2100 microseconds, and the *accumulated* execution time of this function over the entire execution is 26.81 seconds. Since the total application execution time is 32.59 seconds, the percentage of this function

⁴Iterations, No. Multi-Resolutions are specified in the Table 2

execution time to the application execution time is 82.3% (26.81 / 32.59). This function takes a big portion of the total execution time, because it is invoked for every iteration to provide both the metric and the derivative value to the optimizer. When the Amdahl's law is applied to find the upper bound of the speedup, the best achievable speedup is around 5.63x (32.59 / (32.59 - 26.81)).

3.2. Code Structure

Since the `GetValueAndDerivative()` function takes a significant portion of the total execution time, this function is the target of parallelization. The code structure of this function is illustrated in Figure 3. There are three for-loops in the function including one outer for-loop, and two inner for-loops. This nested for-loop structure calculates both the metric value and the derivative value for one function invocation. `aiter` and `biter` are the C++ iterators on the portion of the image. The first inner for-loop calculates metric values and the average of those metric values for the second inner loop, which then computes the derivative values for the optimizer. There is one function call between two inner for-loops which is used towards a derivative calculation. The numbers of iterations for both inner and outer loops are dynamically found for all the inputs sets, which are 50s for both loops.

```
289: void MutualInformationImageToImageMetric::GetValueAndDerivative()
...   {
...     ...
339:   for( biter = m_SampleB.begin(); biter != bend; ++biter)
...     {
...
346:       for( aiter = m_SampleA.begin(); aiter != aend; ++aiter)
...         {
...           }
...
380:       this->CalculateDerivatives((*biter).FixedImagePointValue, derivB);
...
384:       for(aiter = m_SampleA.begin(),aditer = sampleADerivatives.begin(); aiter != aend; ++aiter,++aditer)
...         {
...           }
...
414:   } //end outer for-loop
...   ...
... }
```

Figure 3. Code structures in the `GetValueAndDerivative` method

3.3. Naive method

Implementing the `GetValueAndDerivative` function in CUDA requires parallelization of either the outer for-loop, or the individual parallelization of the two inner for-loops. The function call that is between the two inner for-loops is found to be difficult to parallelize since it invokes several other hierarchical and virtual function calls defined in different classes. Therefore, two separate CUDA kernels are written for each inner loop (Line 346,

384) for the first GPU version. The functions which were originally inside the inner-loops are also implemented in CUDA. The number of for-loop iterations is dynamically found by calling the `size()` function from the following objects: `m.SampleA` and `m.SampleB`. The difficulty with the `aditer` iterator is that it traverses a structure which varies in the number of elements. For this case, the maximum fixed number of elements is used for allocating the two-dimensional data structure on the GPU. The performance of parallelizing the inner for-loops is discussed in Section 5.

3.4. Higher level parallelization

Not only the workload⁵ is small from the parallelization of the inner for-loops, but the overhead of calling two CUDA kernel calls for each outer for-loop iteration is quite high. Because for each CUDA kernel invocation, the GPU memory is allocated and the data from the CPU is copied to the GPU DRAM memory. Hence, this implementation of parallelizing the two inner-loops is very inefficient. As a result, the parallelized execution time on the GPU is significantly worse than the original CPU version. Therefore, in order to increase the workload for the GPU, and to reduce the number of kernel calls, the *outer* for-loop has to be parallelized.

We found out that, the function call that is in between two inner loops is *independent* from the outer loop code. The reordering process of the independent function to facilitate the parallelization process is illustrated in Figure 4. Note that this optimization can be applied to the original version without using the TBB and the GPU.

```

void MutualInformationImageToImageMetric::GetValueAndDerivative()
{
    ...

    for( biter = m_SampleB.begin(); biter != bend; ++biter )
    {
        for( aiter = m_SampleA.begin(); aiter != aend; ++aiter )
        {
            ...
        }

        this->CalculateDerivatives( (*biter).FixedImagePointValue, derivB );

        for( aiter = m_SampleA.begin(), aditer = sampleADerivatives.begin(); aiter != aend; ++aiter, ++aditer )
        {
            ...
        }
    }

    ...
}

```

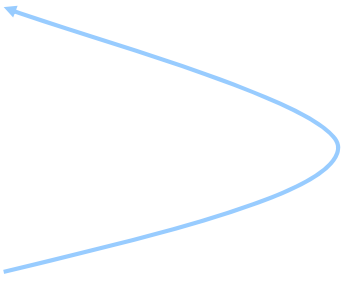


Figure 4. Moving the independent function to the outside for-loop

⁵The amount of computed work per a GPU kernel invocation

As a naive GPU implementation, each iteration of the outer loop is assigned to each GPU thread. This implementation results in only one CUDA block which does not make a full use of the computational power of the GPU. However, the main significance of this outer loop parallelization is that the parallelization task on the GPU is facilitated by reordering the independent function call. This mechanism increases the GPU workload per invocation in comparison to the inner loop parallelization version, and reduces the number of GPU kernel invocations by half.

For parallelizing the outer loop, all the data structures accessed are individually allocated on the GPU, and the contents of those structures are copied to the GPU DRAM memory by calling `cudaMemcpy` API. The function calls resided inside the loop structure are converted to CUDA code by appropriately inlining the mathematical operations. Figure 5 provides the CUDA call interface (i.e., wrapper function call⁶) which shows several data structures involved in the computation of the metric and derivative value. The data types including `pt_aiter_derivative` and `pt_biter_derivative` are pre-computed as the result of the algorithmic change mentioned previously. The 2-dimensional data types which vary in the number of elements are appropriately created.

The performance result from this GPU parallelization of the outer loop is indicated by *OuterLoop* in Table 3.

```
gpu_loop(float *pt_aiter_FixedImageValue,
         float *pt_aiter_MovingImageValue,
         int num_aiter_elements,
         float *pt_biter_FixedImageValue,
         float *pt_biter_MovingImageValue,
         int num_biter_elements,
         float m_FixedImageStandardDeviation,
         float m_MovingImageStandardDeviation,
         float m_MinProbability,
         float *pt_output_values,
         float *pt_aiter_derivative,
         float *pt_biter_derivative,
         int DIM_DERI);
```

Figure 5. The original function call interface for the GPU

3.5. Optimizations

In order to achieve a better performance, it is necessary to reduce the effective latency of memory operations. One way is to reduce the number of memory requests by changing the algorithm. An alternative method is to reduce the effective latency by using caches available on the GPU [11, 20]. In the context of CUDA-enabled NVidia GPUs, the software managed cache (i.e., shared memory) is useful if the values are reused. However, if the data is not reused and the location of the data accessed is irregular, using the shared memory can not effectively improve the performance. Furthermore, using caches (software-managed cache) increases the complexity of programming.

⁶Wrapper function is defined in a separate .cu file. Actual CUDA kernel is invoked from this function.

Table 1. Data access using an offset

Data Offset	Pointer
0 - 49	aiterFixedImageValue
50 - 99	aiterMovingImageValue
100 - 149	biterFixedImageValue
150 - 199	biterMovingImageValue
200 - 549	aiter derivative
550 - 899	biter derivative

3.5.1. Constant Memory When it is not applicable to use the software-managed shared memory due to irregular indexing of data that is accessed only once, the hardware managed constant cache [16] can be used instead. 64KB constant cache is available for 8800GTX (8KB cache per SM) [20]. In the `GetValueAndDerivative` function, the read-only derivative values indexed by *Aiter* iterator in Figure 3 are chosen as candidates. They are copied to the constant cache before the GPU kernel invocation. Depending on the patterns of the data accesses, the latency of a load is reduced when it is a cache hit. Hence, using the constant memory is better than naively accessing the global memory. The GPU implementation which uses the constant cache for the derivative values is indicated by *OuterLoopConst* in Table 3.

3.5.2. Reducing the number of DMA operations Figure 5 shows several data structures indicated by the floating data type pointers. The previous GPU implementations such as *OuterLoop* or *OuterLoopConst* allocates separate data structures, and a separate `cudaMemcpy` API is called for each data structure. However, this mechanism requires extra driver overhead for each `cudaMemcpy` operation. Thus if it is applicable, it is better to create only one data structure and have only one memory copy operation. The desired data elements are accessed by using an appropriate offset in the GPU kernel as illustrated in Table 1. Note that in Figure 6, all the input data structures (excluding the data that were copied to the constant cache) are combined into the `data_struct` pointer.

```
int gpu_loop(float *data_struct,      // Combined data structure
            int data_struct_size,
            float *pt_output_values,
            int output_size,
            int num_aiter_elements,
            int num_biter_elements,
            float m_FixedImageStandardDeviation,
            float m_MovingImageStandardDeviation,
            float m_MinProbability,
            int DIM_DERI,
            int begin_iter,
            int end_iter)
```

Figure 6. The modified function call interface for the GPU

3.5.3. Work granularity change Since our baseline processor is a multithreaded processor, the memory latency can be hidden by switching to a different available thread (or a group of threads in CUDA) after issuing memory requests for one thread (or a group of threads). To overlap as many as memory requests, it is essential to generate as many threads as possible[20], so that it is more probabilistic to have available groups of threads waiting for an execution. However, unlike typical data intensive applications that require many threads, the registration algorithm that we are parallelizing only has 50 iterations for the outer loop (The inner loops also have only 50 iterations). When each GPU thread is assigned to each iteration of the outer loop, the maximum number of threads to be invoked on the GPU is only 50. The consequence is that only one SM out of 16 SMs is utilized for the algorithm in 8800GTX GPU.

To overcome this under-utilization problem and to increase the number of threads, the granularity of the work per thread has to be changed. Rather than assigning each iteration of the outer loop to each thread, each CUDA block takes each iteration of the outer loop, and each thread processes each iteration of the inner loop. After that a synchronization instruction is issued between the two inner loops to run a reduction algorithm safely. For a faster execution, the shared memory is used during the reduction. By taking this approach, the number of threads is dramatically increased from 50 threads to 2500 threads. This can reduce the memory latency penalty also. Furthermore, it is more scalable. Because in the previous approach, only one block is used and the number of threads inside a block can not go over 512 (512 is the limitation imposed by G80 architecture/CUDA programming). The GPU implementation which uses this new algorithm, the constant memory, and combining the data transfers is indicated by *OuterLoop_Opti* in Table 2.

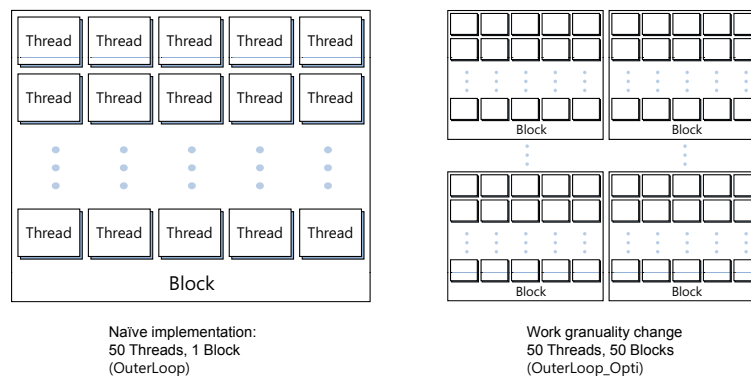


Figure 7. Work granularity change that generates more number of threads

4. Methodology

For the ITK Toolkit, version 3.10.1 released in 2008 is used for the experiment. Nvidia QuadroFX5600 [2] GPU and NVCC version of 2.1 are used to produce the results. All the libraries and applications in the ITK Toolkit

Table 2. Inputs to MultiResMIRegistration

Input Name	3D Fixed Volume	3D Moving Volume	No. Multi-Resolutions	No. Iterations	Source
BrainWeb_10.0K	T1 (6.78MB) [181x217x180]	T2 (6.78MB) [181x217x180]	5	10000	ITK Toolkit
BrainWeb_5.0K	T1 (6.78MB) [181x217x180]	T2 (6.78MB) [181x217x180]	5	5000	
BrainWeb_2.5K	T1 (6.78MB) [181x217x180]	T2 (6.78MB) [181x217x180]	5	2500	
CTtoMP_102_10.0K	CT (24.50MB) [512x512x49]	MP (16.00MB) [256x256x128]	5	10000	Patient 102 [5]
CTtoMP_102_5K	CT (24.50MB) [512x512x49]	MP (16.00MB) [256x256x128]	5	5000	
CTtoMP_102_2.5K	CT (24.50MB) [512x512x49]	MP (16.00MB) [256x256x128]	5	2500	
CTtoMP_109_10.0K	CT (20.50MB) [512x512x41]	MP (16.00MB) [256x256x128]	5	10000	Patient 109
CTtoMP_109_5K	CT (20.50MB) [512x512x41]	MP (16.00MB) [256x256x128]	5	5000	
CTtoMP_109_2.5K	CT (20.50MB) [512x512x41]	MP (16.00MB) [256x256x128]	5	2500	
T1toPD_009_10.0K	T1 (3.00MB) [256x256x24]	PD (3.00MB) [256x256x24]	5	10000	Patient 009
T1toPD_009_5.0K	T1 (3.00MB) [256x256x24]	PD (3.00MB) [256x256x24]	5	5000	
T1toPD_009_2.5K	T1 (3.00MB) [256x256x24]	PD (3.00MB) [256x256x24]	5	2500	

are compiled with *Release* version (-O3), and GPU codes are also compiled with -O3. For comparison to GPU performance, a CPU multithreaded version using Intel’s TBB [10] library (Version 2.1) is implemented and executed on a 8-core machine. The machine setting is as follows, 2 socket 1.87 GHz Quad-core Intel Xeon (Total 8 Cores), 4MB L2-cache, 8GB RAM and QuadroFX5600 GPU. There are two input source files in each experiment shown in Table 2. BrainWeb input is included in the ITK Toolkit. In order to obtain more data sets which vary in the file sizes and in the types of images, data sets from *Retrospective Image Registration Evaluation Project* [5] are used in the experiment. Table 2 shows the actual data from the patients indicated by the patient number. Types of images are shown in the second and third columns indicated by Fixed and Moving volume. Different types of images indicated by T1, T2, PD, and MP are generated by setting different timing parameters of the MRI scanner. For those configuration details on the images, these sources can be referenced [18, 12].

Table 3 shows the lists of the CPU and GPU implementations. *Original* implies the original ITK Toolkit code without any modifications. This is not necessarily a serial execution since some portions of ITK code are already parallelized [9]. In *InnerLoop* implementation, each inner loop is separately parallelized by CUDA while the outer loop code is running on the CPU. In *OuterLoop*, the outer loop is parallelized by CUDA after making algorithmic changes to the code. This is a pure parallelization without using any optimizations. In *OuterLoop_Const*, read-only data such as derivative values are copied to the constant cache on the GPU. In *OuterLoop_Opti*, there are three optimizations: (1) The granularity of the thread is changed to generate more number of threads. (2) The number of DMA transfers between CPU and GPU is reduced. (3) The constant memory is used. The *kernel_lowerbound* function is inserted to measure the GPU kernel overhead. The detailed descriptions on these different GPU implementations are discussed in Section 3.

Table 3. Experiment Configuration

Implementation	Details
Original	Original ITK execution on 8 Core machine
InnerLoop	Parallelized two inner-loops in two separate GPU Kernels
OuterLoop	Parallelized outer for-loop after making algorithmic change
OuterLoop_Const	Const memory is used for read-only values
OuterLoop_Opti	Work granularity change for GPU thread. Further optimized by reducing number of loads and reduction.
Kernel_lowerbound	GPU kernel implement ion commented out. Shows kernel overhead in other GPU implementations

5. Results

Figure 8 shows the average execution time of a single invocation of the `GetValueAndDerivative()` function. The value for the CPU and GPU is obtained by dividing the cumulated execution time of the function by the total number of invocations. Note that the total number of invocations is obtained by multiplying the number of Multi-Resolutions and the number of Iterations in Table 2. Figure 8 shows that the CPU execution time is 1527 microseconds, but it only takes about 104 microseconds on the GPU, resulting in 14.61x speedup.

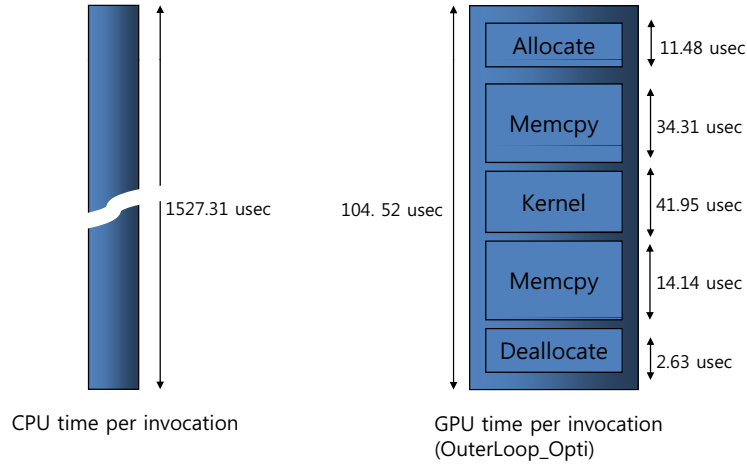


Figure 8. The averaged outer-loop execution time per `GetValueAndDerivative()`

This result from Figure 8 implies that the benefit of running on the GPU increases as the number of GPU kernel invocation increases. Figure 9 shows the application execution time on the various inputs specified in Table 2. The first observation is that as the number of invocations increases for a given input (i.e., 2.5K, 5K, and 10K), the execution time on both GPU and TBB implementations decreases.

For the GPU implementations, the execution time decreases further as more optimizations are applied. *Outer_Const* which uses the constant memory shows a negligible improvement compared to the naive GPU implementation. However, *Outer_Opti* shows an average of 41.5% performance improvement over the *Outer_Const* version, approaching within 7% of the *Kernel_Lowerbound*. The GPU implementation indicated by *InnerLoop* is not included in the result figures due to the very severe performance degradation of more than 800%.

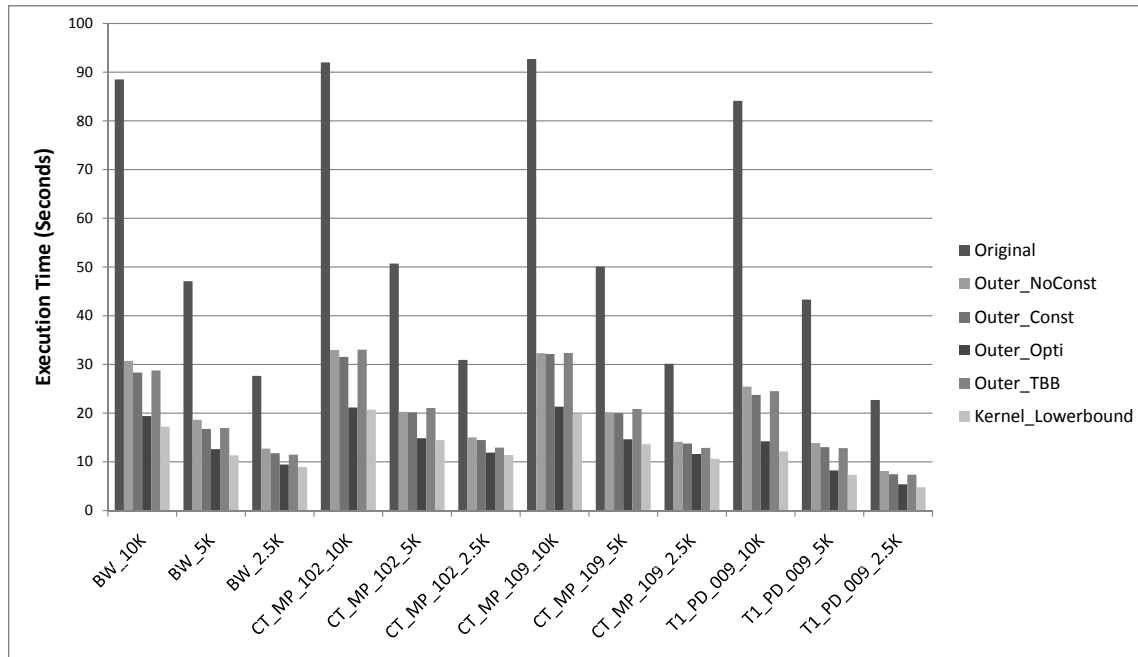


Figure 9. Application Execution times. Note that Innerloop data is not shown due to severe slowdown

In comparison to 14.61x speedup on the nested loops parallelized by the GPU, the effect on the total application speedup ranges from 3 to about 6 times as shown in Figure 10. The best speedup of 5.91x is obtained by T1_PD_009_10K input ⁷. The GPU speedup is 42% faster than the TBB implementation on the 8-core machine on average. Figure 11 shows the GPU speedup of the loops in the `GetValueAndDerivative()` function. The best speedup of 14.749x on the loop is achieved by T1_PD_009_10K input, where this input results in the best speedup on the application as well. The TBB speedup across the inputs for a kernel itself is 5.13X and for the application is 3.43X.

For an error analysis, each value of the transform matrix and the offset matrix ⁸ generated by the original CPU version is compared with all the GPU implementations. *GPU output* in Table 4 shows the output element which produces the largest deviation from the correct output. Note that this value is selected after comparing across all the GPU implementations (i.e., NoConst, Const, Opti). *Correct CPU output* shows the corresponding correct output value, and *absolute value error* shows the absolute difference between the correct value and the GPU output. The results show that all the errors are less than 0.05 across all the input values.

⁷Note that the upperbound of 5.63X in Section 3.1 is obtained by using BrainWeb.2.5K input

⁸Registration outputs: Transform matrix rotates a moving image. Offset matrix moves a moving image in xyz directions

Table 4. The largest selected GPU Errors for 5K (No.Iterations) input

Input	BrainWeb_5.0K	CTtoMP_102_5K	CTtoMP_109_5K	T1toPD_009_5.0K
GPU output	0.14079	29.5355	52.9688	0.000111
Correct CPU output	0.14093	29.5393	52.9299	0.000112
Absolute value error	0.00014	0.0038	0.0389	0.000001

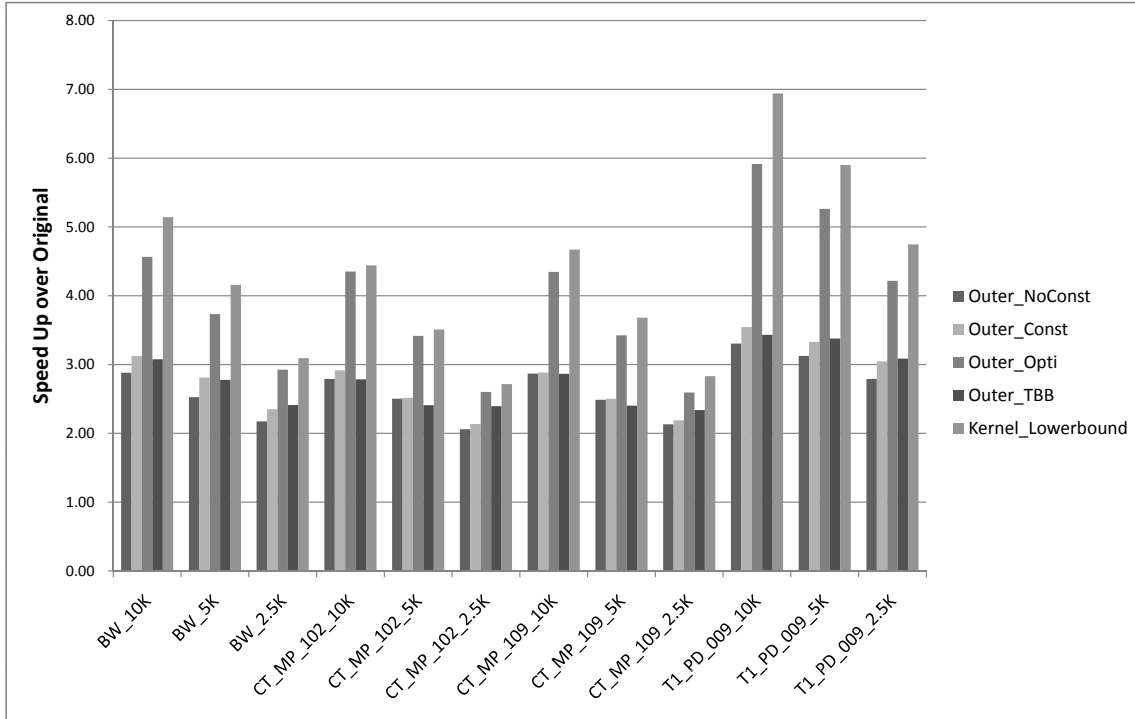


Figure 10. Speedup over the Original implementation. Note that Innerloop data is not shown due to severe slowdown

6. Discussions

6.1. Object-oriented code

ITK source code is modularized in functionality whenever applicable. It is based on the template programming to incorporate different kinds of image inputs. Any function within the ITK toolkit is likely to invoke other methods which are defined in other classes. The difficult decision arises when porting the code into another platform such as GPUs. Because the amount of work per function call is not large enough to gain an advantage since additional overhead exists from DMA data transfers. This type of problem makes a parallelization implementation even more challenging.

Another issue which prevents achieving a better performance is related to many invocations of a small function body. When these functions are parallelized on the GPU, the performance improvement is not significant because the driver overhead and extra time associated with DMA data transfers build up for each GPU invocation. In comparison, these extra overhead does not exist if all the function call is combined and only one invocation exists. Unfortunately, due to the highly-modularized implementation style in ITK, there are high number of invocations for each function.

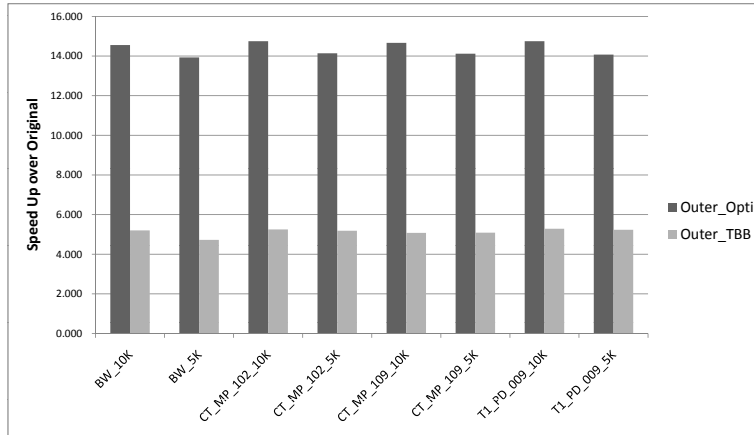


Figure 11. Outerloop Speedup over Original (Kernel)

To overcome this performance degradation factor, parallelization on the codes that are in the higher-level is necessary.

6.2. More optimizations

The speedup of the GPU can be further improved by skipping redundant allocation and deallocation between each GPU kernel invocation. One way to implement this mechanism is to store the global pointer that points to the GPU memory between invocations of the kernel. We plan to conduct this study in our future work.

7. Related Work

There have been several recent works which parallelized filters and applications in ITK. The work by Ohara and Yeo [17] implemented a mutual-information based registration algorithm on the Cell Broadband Engine processor [19]. Parallelization with the Cell SIMD instructions produced the speedup of 4.5x. However, that speedup did not include the file I/O time. On contrast, we developed the parallelization algorithm on the original, publically available ITK source-code release [22] thereby our GPU implementation being directly applied to the ITK toolkit used by the community users.

Muyan and Owens implemented a deformable registration algorithm on the GPU [14]. But the algorithm parallelized in their work is in the other category of the registration (Not rigid registration), and the source code parallelized is in C-code containing only the algorithm. In comparison, to the best of our knowledge, our work is the first work which directly parallelizes the registration algorithm from the application suite in use, and optimized the GPU implementation despite highly object-oriented C++ code environment, and small computation work per each function call due to the highly-modularized style of implementation. Other related works include parallelization of the filters by Jeong [24]. In his work, basic ITK image filters such as mean, gaussian, median, and anisotropic diffusion filters are parallelized by CUDA, and the speedups reported range from 25x to 140x. These improved filters can enhance the performance indirectly, especially the algorithms in the segmentation category.

8. Conclusions

This paper parallelized the mutual-information based registration application from a widely used medical imaging toolkit called ITK (Insight Toolkit). This work is significant for two reasons. One is that this work parallelized the registration algorithm from the complete software in use, not only the registration algorithm kernel itself. The outcome of the work could be direct applicability and have an high impact to the community users worldwide. Secondly, this work extracted out the performance on the GPU from such highly modularized, object-oriented, and template-based source code. ITK source code is modularized so highly that any given work is divided into many different function calls, and the style is very object-oriented that those function calls are often virtual function calls and defined in different classes with different levels of inheritances. Under these circumstances, this work implemented a portion of a highly used Mutual-Information based registration application on the GPU using CUDA, achieving 14.61x speedup on the kernel and 5.91x speedup on the application. We also show that using TBB the speedup of the kernel itself is 5.28x and the speedup of the application is 3.43x. This work contributes to one of the most time critical surgical operative environments as well as a faster 3D image processing and analysis by the associated users.

9. References

- [1] GeForce GTX280. <http://www.nvidia.com/object/geforcefamily.html>.
- [2] Quadro FX5600. http://www.nvidia.com/object/quadro_fx_5600_4600.html.
- [3] CABLE. <http://public.kitware.com/Cable/HTML/Index.html>.
- [4] CMAKE. Cross-platform system for build automation. <http://www.cmake.org/>.
- [5] Dr. Michael Fitzpatrick. The Retrospective Image Registration, National Institutes of Health. <http://insight-journal.org/rire/>.
- [6] Erich Gamma, Richard Helm, Raph Johnson, and John Vlissides. Design Patterns. Addison Wesley, 1995.
- [7] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A call graph execution profiler. In *In SIGPLAN 82: Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, 1982.
- [8] L. Ibanez, L. Ng, J. Gee, and S. Aylward. Registration patterns: the generic framework for image registration of the insight toolkit. *Biomedical Imaging, 2002. Proceedings. 2002 IEEE International Symposium on*, pages 345–348, 2002.
- [9] L. Ibanez, W. Schroeder, L. Ng, and J. Cates. *The ITK Software Guide*. Kitware, Inc. ISBN 1-930934-15-7, <http://www.itk.org/ItkSoftwareGuide.pdf>, second edition, 2005.

- [10] Intel Corporation. Intel threading building blocks. <http://threadingbuildingblocks.org/>.
- [11] Mark Harris. Optimizing CUDA. Supercomputing 2007 CUDA tutorial. <http://www.gpgpu.org/sc2007/>.
- [12] J. P. I. Mugler and J. R. Brookeman. Three-dimensional magnetization-prepared rapid gradient-echo imaging (3d mp-rage). *Magnetic Resonance in Medicine*, 15:152–157, 1990.
- [13] MultiResMIRegistration Readme file. InsightApplications-3.10.0/MultiResMIRegistration. Kitware Inc., 2008.
- [14] P. Muyan-Ozcelik, J. D. Owens, J. Xia, and S. S. Samant. Fast deformable registration on the gpu: A cuda implementation of demons. In *ICCSA '08: Proceedings of the 2008 International Conference on Computational Sciences and Its Applications*, pages 223–233, Washington, DC, USA, 2008. IEEE Computer Society.
- [15] NVIDIA Corporation. Cuda (compute unified device architecture). <http://www.nvidia.com/cuda>.
- [16] NVIDIA Corporation. CUDA Programming Guide, June 2008.
- [17] M. Ohara, H. Yeo, F. Savino, G. Iyengar, L. Gong, H. Inoue, H. Komatsu, V. Sheinin, and S. Daijavad. Accelerating mutual-information-based linear registration on the cell broadband engine processor. *Multimedia and Expo, 2007 IEEE International Conference on*, July 2007.
- [18] Olivier Cuisenaire. The physics of T1- and T2-weighted MRI. <http://www.tele.ucl.ac.be/PEOPLE/OC/these/node100.html>.
- [19] D. Pham, E. Behnen, M. Bolliger, H. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, B. Le, Y. Masubuchi, S. Posluszny, M. Riley, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, and K. Yazawa. The design methodology and implementation of a first-generation cell processor: a multi-core soc. *Custom Integrated Circuits Conference, 2005. Proceedings of the IEEE 2005*, pages 45–49.
- [20] S. Ryoo, C. I. Rodrigues, S. S. Bagsorkhi, S. S. Stone, D. B. Kirk, and W. mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82, New York, NY, USA, 2008. ACM.
- [21] Sam Hocevar. Using gprof with multithreaded applications, 2004.
- [22] The Insight Toolkit. <http://www.itk.org/>.
- [23] P. Viola and I. W. M. Wells. Alignment by maximization of mutual information. In *ICCV '95: Proceedings of the Fifth International Conference on Computer Vision*, Washington, DC, USA, 1995. IEEE Computer Society.
- [24] Won-Ki Jeong. CUDA ITK Image Filters. <http://www.cs.utah.edu/wkjeong/>.

10. Appendix

This section provides information how to integrate CUDA to the ITK Toolkit. ITK is configured by CMake [4] which is a cross-platform and open-source build system. Depending on the platform used, CMake generates makefiles for Linux, and workspaces for Visual Studio in Windows. The user can change the compilation settings only by modifying the `CMakeLists.txt` file, not by directly changing the contents of the makefile or the workspace. For integration of the CUDA code and the ITK Toolkit, one option is to modify the `CMakeLists.txt` file to include the NVCC compiler and the associated CUDA wrapper function. But the easier method for integrating is to generate a library of the GPU code, and simply link that library in the CMake compilation configuration.

A CUDA wrapper function `gpu_loop.cu` is created as a separate file, so that it can be compiled by using the NVCC compiler. The wrapper function contains all the necessary GPU API calls such as `cudaMemcpy` as well as the kernel source. To generate a static library of the GPU code, the following Linux commands are executed. Note that the generated library `libgpu.a` and the corresponding header file containing the function declaration need to be copied to the appropriate directory.

```
nvcc -c -O3 gpu_loop.cu -I/$INCLUDEPATH -L/$LIBPATH -l$LIB
ar rcs libgpu.a *.o
cp libgpu.a ~/GPU_Codes/Library
cp gpu_function.h ~/GPU_Codes/Include
```

Figure 12. Linux commands for generating static GPU library

The information of the GPU library and the corresponding header file needs to be updated in `CMakeLists.txt` as shown in Figure 13. The interface in `CMakeList.txt` is straightforward. The directory which contains the header file is added with the `INCLUDE_DIRECTORIES` command. Similarly, the directory that contains the library is added with the `LINK_DIRECTORIES` command, and the `LINK_LIBRARIES` command is used to specify the names of the library used in the application. Figure 13 shows the inclusion of the GPU library as well as the TBB library.

After changes are made to the `CMakeList.txt` file, and the GPU library is generated in the appropriate directory, simply the left task is typing the `make` command in the `MultiResMIRegistration` application folder in the ITK Toolkit. Similar approach can be used in the other applications that use the CMake build system.

```
1 PROJECT( MultiResMIRegistration )
2
3 INCLUDE_DIRECTORIES(~ /GPU_Codes/Include)
4 LINK_DIRECTORIES(~ /GPU_Codes/Library)
5 LINK_LIBRARIES(gpu)
6
7 INCLUDE_DIRECTORIES(/usr/local/cuda/include)
8 INCLUDE_DIRECTORIES(~ /NVIDIA_CUDA_SDK/common/inc)
9 LINK_DIRECTORIES(/usr/local/cuda/lib)
10 LINK_DIRECTORIES(~ /NVIDIA_CUDA_SDK/lib)
11 LINK_DIRECTORIES(~ /NVIDIA_CUDA_SDK/common/lib)
12 LINK_LIBRARIES(cuda)
13 LINK_LIBRARIES(cudart)
14 LINK_LIBRARIES(GL)
15 LINK_LIBRARIES(GLU)
16
17 INCLUDE_DIRECTORIES(~ /Library/tbb21_20080605oss/include)
18 LINK_DIRECTORIES(~ /Library/tbb21_20080605oss/build/
    linux_em64t_gcc_cc3.4.6_libc2.3.4_kernel2.6.9_release)
19 LINK_LIBRARIES(tbb tbbmalloc)
```

Figure 13. Inserted configurations in CMakeList.txt for CUDA and TBB integration