Performance Analysis of Accelerated Image Registration Using GPGPU

Peter Bui
University of Notre Dame
pbui@cse.nd.edu

Jay Brockman
University of Notre Dame
jbb@cse.nd.edu

ABSTRACT

This paper presents a performance analysis of an accelerated 2-D rigid image registration implementation that employs the Compute Unified Device Architecture (CUDA) programming environment to take advantage of the parallel processing capabilities of NVIDIA's Tesla C870 GPU. We explain the underlying structure of the GPU implementation and compare its performance and accuracy against a fast CPU-based implementation. Our experimental results demonstrate that our GPU version is capable of up to 90× speedup with bilinear interpolation and 30× speedup with bicubic interpolation while maintaining a high level of accuracy. This compares favorably to recent image registration studies, but it also indicates that our implementation only reaches about 70% of theorectical peak performance. To analyze our results, we utilize profiling data to identify some of the underlying limitations of CUDA that prohibit peak performance. At the end, we emphasize the need to manage memory resources carefully to fully utilize the GPU and obtain maximum speedup.

Categories and Subject Descriptors

I.4.3 [**Image Processing and Computer Vision**]: Enhancement—*Registration*

Keywords

GPGPU, CUDA, image registration, performance analysis

1. INTRODUCTION

Image registration is the process of aligning two or more images in order to determine the point-by-point correspondence among a set of images [3]. This technique is used in a variety of applications such as remote sensing, map updating, weather forecasting, and computer vision to provide an integrated view of the image data [17]. In the medical field, image registration is used in clinical tasks such as diagnosis, radiotherapy, and image-guided surgery. Because of the widespread use of this procedure and the immense computational workload required, image registration has been a popular target for acceleration in recent studies.

One form of image registration is rigid registration, which primarily deals with global image alignment and shifting. This is in contrast to elastic registration which can account for changes in local morphology over time [12]. In this investigation, we focus on 2-D rigid image registration where the objective of the registration procedure is to find a transformation to apply to a *source* image that best aligns it with a *target* image (or visa versa). To accomplish this task, the following pipeline is commonly used:

1. **Image Generation:** Generate a *temporary* image using the *source* image and a transformation estimate.
2. **Similarity Measurement:** Compare this *temporary* image to the *target* image to test for similarity.
3. **Optimization:** Update the transformation estimate using the similarity measurement as a cost function in an optimization algorithm.

This process is repeated until an acceptable level of tolerance (high similarity between the *target* and temporary images) is achieved. The resulting transformation data from this procedure maps the relationship between the *target* and *source* and is used for integrated analysis.

In most registration implementations, the image generation and similarity measurement, are the most computationally intensive aspects of the registration process since they involve performing convolutions or computations on each image pixel. For this study, we accelerate a 2-D rigid image registration application by implementing these components using NVIDIA's CUDA programming environment to take advantage of the parallel processing capabilities of the NVIDIA Tesla C870 GPU. The intrinsic data parallelism in image registration makes the GPGPU (general-purpose computing on graphics processing units) approach a suitable platform for acceleration.

This paper examines the process of porting the CPU implementation to the CUDA framework and analyzes the resulting performance gains along with any potential pitfalls. Although a sizable amount of speedup is obtained, the implementation still falls short of the maximum potential speed increase. Our study focuses on understanding the limitations of CUDA that inhibit maximum performance and examines various methods to overcome some of these obstacles encountered in developing the image registration application.

GPGPU 2009, Washington, DC USA

The remainder of this paper is as follows: The next section covers related work involving accelerating image registration using GPGPU. After this, we provide a brief overview of the CUDA programming model and then we present our CUDA implementation of the image registration algorithm, while examining potential programming traps and highlighting key practices to achieve good speedup. Next, we compare the CUDA implementation's performance and accuracy versus the CPU version and use profiling data to analyze the experimental results. Finally, we discuss the general lessons learned from the project and possible improvements to the CUDA programming environment to obtain further performance and productivity gains.

2. PREVIOUS WORK

The growing popularity and use of graphics processing units in scientific computing has been summarized by Owens, et al. in their overview of GPU computing [11]. They note that due to the increasing amount of memory bandwidth and computational horsepower on graphics processing units, GPUs substantially outpace their CPU counterparts, thus making them attractive acceleration platforms. For instance, the NVIDIA programming guide [9] states that current GPUs such as NVIDIA's Tesla C870 have up to 128 concurrent processors capable of over 300+ giga-floating point operations per second (in aggregate) while providing high memory bandwidth (80+ GB/s) to data stored locally on the graphics units. According to Fung and Mann [2], these attributes make GPUs attractive platforms for accelerating image processing applications such as image registration which exhibit data-level parallelism and have high computational costs.

In the past, applications using the GPGPU approach for application acceleration have employed the method of mapping their computations onto the shader and vertex processors of the graphics unit. This is done by programming in various shading languages such as Cg, GLSL, or HLSL, along with graphics libraries such as OpenGL or DirectX and loading those programs directly into the graphics rendering pipeline [4]. Ino et al. used this technique to present a fast rigid registration method in their OpenGL-based implementation [6] that demonstrated between 5.0× and 9.6× speedup while maintaining a tolerable level of accuracy. Kubias et al. were able to post similar results in their implementation of rigid image registration [7] with the added twist that they implemented eight similarity measurements on the GPU and used these in aggregate to optimize the registration. Both of these studies took advantage of not only the parallel processing abilities of the GPU, but also the hardware supported bilinear filtering provided by textures. Of course, this technique requires programmers to manipulate their algorithms and data structures to fit the graphics programming model, which is not always straightforward.

The introduction of NVIDIA's CUDA, however, allows programmers to avoid using graphics programming libraries and instead work in a stream processing environment [9]. In this framework the vertex or fragment processors in the graphics pipeline are unified and abstracted as a set of programmable concurrent stream processors. Rather than use shading languages, programmers use a C/C++-like programming environment with common programming constructs such as arrays, pointers, and variables to program the GPU. This increase in user programmability allows developers without a deep graphics programming background to quickly and effectively take advantage of the parallel processing abilities of the graphics processing unit. Recently, Sugiura et al. [14] have taken advantage of this programming environment to accelerate rigid image registration used in bronchoscope tracking by a factor of 16× with moderate levels of accuracy. Likewise, Muyan-Özçelik et al. use CUDA to implement fast deformable (elastic) image registration with a factor of 55× speedup [8]. They also provide an exhaustive and detailed accounting of their implementation and experimental results.

In this study, we implement a 2-D rigid image registration system similar to Ino [6] and Kubias [7] using the CUDA programming environment. We use a pyramid-based algorithm outlined by Thévenaz [15], but we omit cubic spline representations and use the simplex optimizer function from the GNU Scientific Library [1] rather than the proposed Marquardt-Levenberg algorithm. Unlike previous studies, we implement the more computationally intensive bicubic interpolation scheme in addition to the hardware supported bilinear interpolation method in order to increase registration accuracy which is needed in some applications such as motion tracking. Furthermore, we follow the methodology of Muyan-Özçelik [8] in providing a thorough analysis of the implementation of our algorithm, noting possible CUDA traps and how to optimize for maximum speedup. To aid in this analysis we examine the efficiency of our implementation and profiling data to identify limitations of the CUDA platform. At the end, we review the obstacles and issues present in the CUDA framework and propose potential means of addressing them.

3. METHOD

Before examining the details of the image registration implementation, it is first necessary to provide a brief overview of the CUDA programming model.

3.1 CUDA Programming Model

In CUDA, the programmable units of the GPU expose a single-program multiple-data (SPMD) programming model. The user passes a program to the device in the form of a kernel. The GPU in turn processes many elements (threads) in parallel using the specified kernel on each thread. This is similar to the traditional single instruction, multiple data (SIMD) model provided by Intel's SSE and MMX CPU instructions. CUDA, however, allows for limited branching within the kernel, permitting different elements to take divergent code paths and provides a much broader instruction set. These threads are organized into *warps* or groups of 32 parallel scalar threads where each *warp* executes one kernel instruction at a time.

The user programmable kernels are executed by an array of concurrent stream processors (SPs) which support 32-bit integer and single-precision floating point operations. These SPs are clustered together in groups of 8 to form a single stream multi-processor (SM) core with each SP executing one thread. The SPs within a SM share a local store referred to as shared memory. Collections of *warps* are known as *thread blocks* and these threads all run on the same MP and share a part of the local store. The number of warps in such a *thread block* is defined by the user when calling the kernel.

In addition to the shared memory, the CUDA programming environment exposes a limited memory hierarchy in the form of registers, constant memory, global memory, and

textures. The register file is the fastest level in this on-chip hierarchy but only supports a limited amount of space (32-64KB). After this, shared memory is provided which is slower and smaller (16KB). Constant memory is a subset of device memory (64KB) that is shared by all the SPs and cannot be modified at run-time by the device. Other device memory is generally grouped as global memory which permits both read and write operations from all threads, but is uncached and has long latencies. The amount of global memory available depends on the specific hardware configuration of the GPU. Texture memory is also a subset of the device memory similar to the constant memory in that it is read-only on the device. However, unlike global memory, it provides cached reads and thus is faster. Moreover, it allows addressing through a specialized texture unit which allows for filtering and clamping.

Accelerating an application using CUDA, then, requires identifying program bottlenecks and mapping into this programming model. This involves careful structuring of the user defined kernels to ensure that all the SPs are executing at peak performance and strict management of data access patterns to fit into the limits imposed by the memory hierarchy. For data intensive programs such as image registration, this latter issue of memory management is key to obtaining good speedup.

3.2 Image Registration Algorithm

The image registration algorithm used in this study follows the general process outlined in the introduction:

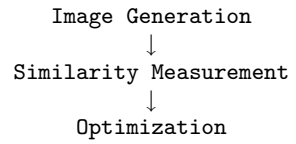

Figure 1: Registration Pipeline

For the image generation component, we allow for the normal set of affine transformations: rotation, translation, and scaling in both the X and Y direction. The image generator uses the transformation data (usually in the form of a matrix) to transform or map the *source* image into the *temporary* image. To compute the intensity values in the *temporary* image at these new coordinates produced by the mapping, we use bicubic interpolation. This involves computing 5 convolutions on a 4×4 neighborhood of pixel values (4 for each row, and then 1 along the first column). This is done instead of the more common bilinear interpolation because former produces more accurate intensities than the latter, albeit at the cost of more computation.

To measure similarity, we use the mean squared error (MSE) metric defined by following formula:

$$MSE = \frac{\sum \sum (Temporary(x,y) - Target(x,y))^2}{NumberOfPixels} \quad (1)$$

where *Temporary(x, y)* and *Target(x, y)* are intensity values in the *temporary* and *target* images at coordinates *(x, y)*, and *NumberOfPixels* is the total size of one image.

As mentioned earlier, we utilize the Nelder-Mead Simplex multi-dimensional minimizer from the GNU Scientific Library as the optimization algorithm in our image registration implementation. This function uses the MSE as the cost function to update the transformation estimate in order to find the set of rotation, translation, and scaling transformations that produces the least amount of error.

To algorithmically improve registration time, we employ a pyramid scheme to provide a coarse-to-fine grain registration process. This is done by taking the *source* and *target* images and reducing them using a mean filter into quarter-sized images for each successive pyramid level. The whole registration process is performed on each level of the pyramid, starting with the smallest level and working up to the full sized image. After this final level, we will have the transformation matrix. This pyramid technique allows for faster image alignment as detailed by Thévenaz [15].

3.3 Implementation and Test Environment

The image registration algorithm was first implemented in *C* on a host system featuring a Intel Quad-Core Q6700 2.66 GHz CPU and 8.0 GB of RAM and running Ubuntu Linux 8.04 (kernel 2.6.20). The graphics unit on the system is the NVIDIA Tesla C870 which has 128 stream processors and 1.5 GB of on-chip memory [9]. To compile the program codes we used GCC 4.1.2 with an optimization level of 3 (i.e. `-O3`) and NVIDIA's CUDA 1.1 development kit. Table 1 contains a complete listing of the images we used to test our codes. For each of these images, we registered these source images against target images that contained various amounts of rotations, translations, and scaling.

Image	Dimensions (Pixels)
lenna	512×512
ndbuntu	768×768
halo	1024×1024
jump	1536×1536
victory	2048×2048
crabnebula	3072×3072

Table 1: List of test images

3.4 CUDA Algorithm

After profiling the CPU version, it was observed that the image generation and similarity measurement components composed $95-99\%$ of the run-time and thus were prime targets for acceleration. Since both procedures involve scanning the image pixels and performing the same computation for each element there is a high level of data parallelism. This means that both components easily map into the CUDA programming model, making the GPU a suitable platform for accelerating these functions.

Moving the image generation and similarity measurement to the GPU produces the pipeline in Figure 2. As can be seen in this figure, the image generation (represented by the `transform` and `interpolate` blocks) and similarity measurement (`calculate error`) components are moved to the GPU. After our first complete CUDA implementation, we also implemented the pyramid on the GPU, since the computations performed by the pyramid component map well to the architecture. The optimization part remains on the CPU as it does not affect the run-time that much (less than $1-5\%$ of the total run-time). This is because the optimization function is just a short computation with no data parallelism and thus unsuitable for accelerating on the graphics unit.

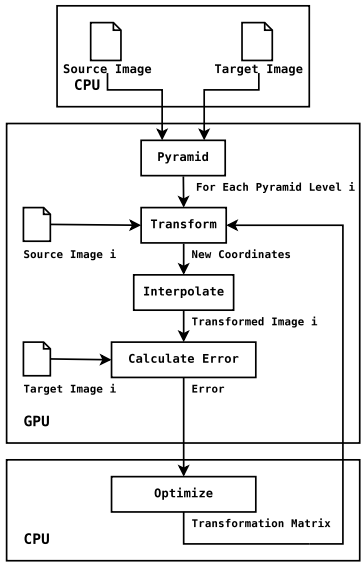

Figure 2: CUDA Registration Pipeline

3.5 CUDA Implementation

As noted in previous studies [6, 7, 8], two keys to achieving good performance on a GPU is **(1)** to minimize the number of kernel calls and **(2)** to limit the number and size of data transfers to and from the host (CPU) and the device (GPU). To accomplish this, we moved the pyramid component to the GPU and generated the different pyramid levels on the device. This allowed us to store all the images on the GPU and removed the need to transfer whole images to and from the host and device. Additionally, we implemented the image generation and part of the similarity measurement all in one kernel rather than two separate modules. This means we do not have to pass image data between kernels and only need to invoke one kernel per optimization iteration, thus reducing costly memory transfers costs and avoiding having to scan through the image twice (once for generation, and again for similarity measurement).

```
1  __global__ void
2  bilinear_kernel(float* dp, size_t dstride,
3                  uint rows, uint columns)
4  {
5  /* Thread column, row */
6  int ic = threadIdx.x;
7  int ir = threadIdx.y;
8
9  /* Target column, row */
10 int tc = (blockIdx.x*blockDim.x)+ic;
11 int tr = (blockIdx.y*blockDim.y)+ir;
12
13 /* Map source column, row using matrix */
14 float sc = (matrix_a(ConstantMatrix)*tc) +
15           (matrix_b(ConstantMatrix)*tr) +
16            matrix_e(ConstantMatrix);
17 float sr = (matrix_c(ConstantMatrix)*tc) +
18           (matrix_d(ConstantMatrix)*tr) +
19            matrix_f(ConstantMatrix);
20
21 /* Compute error, store in local memory */
22 local_element(ic,ir)  = get_src_texel(sc,sr);
23 local_element(ic,ir) -= get_tgt_texel(tc,tr);
24 local_element(ic,ir) *= local_element(ic,ir);
25 __syncthreads();
26
27 /* If first thread, compute partial mse */
28 if ((ic + ir) == 0) {
29     float sum = 0.0;
30     for (ir = 0; ir < BLOCK_SIZE; ir++)
31         for (ic = 0; ic < BLOCK_SIZE; ic++)
32             sum += local_element(ic,ir);
33     dp[blockIdx.y*dstride+blockIdx.x] = sum;
34 }
35 }
```

Listing 1: Bilinear CUDA Kernel

To examine how we implemented this single kernel, the CUDA bilinear kernel is shown in Listing 1. Each pixel/thread executes the following:

- `Lines 06 - 11`: Calculate the location of current thread with respect to the local thread block and the *temporary* image.
- `Lines 14 - 19`: Compute the coordinates that map the *temporary* image and the *source* image using the transformation matrix stored in constant memory.
- `Lines 22 - 25`: Get the pixel intensity for the current pixel location by performing a texture fetch and store it in a local shared memory array. Then compute the squared error and save it in the same location.
- `Lines 28 - 34`: If this is the first thread in the block, then compute the partial sum of mean squared errors and save it to the destination array in global memory.

For the bicubic kernel, the first 19 lines are the same. After line 25, we decompose the coordinates `sc` and `sr` into their integer and fractional parts. Then, we replace line 22 with calls to the cubic convolution function defined in the CUDA device for a 4×4 neighborhood as explained earlier. Once the intensity is computed, we then continue with the rest of kernel by calculating the squared error and if appropriate, the partial sum for the block. These kernels fit into the complete CUDA image registration pipeline in the following manner:

1. **Pyramid Construction:** Both the *source* and *target* images are transferred to the GPU and a pyramid of images is generated using parallel reduction algorithm.
2. **Registration:** For each level of the pyramid, starting with the smallest level (i.e. most coarse image), do the following:
   (a) **CUDA Kernel:** After each thread has executed the kernel as explained above, the partial sum of mean squared errors for all the thread blocks are stored in global memory.
   (b) **Similarity Measurement:** The host will download the partial sums created by the CUDA kernel and complete the summation and division to produce the mean squared error.
   (c) **Optimization:** The MSE computed in the previous step is used by the optimization function to update the transformation matrix.

Once the registration for a particular pyramid level is complete, use the computed transformation estimate for the registration of the next pyramid level. After we register the final pyramid level (i.e. the full sized images), we have found our transformation data and return that to the user.

There are a few key ideas to note in this implementation. First, we only the send the *source* and *target* images once to the GPU where they are stored as CUDA arrays and bound to texture references. A pyramid kernel is called on each image for every pyramid level and the output is stored on the GPU. The use of textures is important because they provide the CUDA threads cached reads to our image data. Moreover, using the texture memory gives us hardware support for bilinear interpolation as shown in Listing 1 and automatic clamping. This negates the need to explicitly check the computed intensities for overflow and underflow.

Second, originally there were explicit instructions to load the transformation matrix into a local shared memory array used by all the threads in the block. This was important because the kernel received the transformation matrix as a pointer argument rather than explicit arguments, which means that the values are stored in global memory rather than registers in the local frame. To compute the mapping, each thread would have been forced to fetch the same 6 floating point numbers for each kernel execution if this prefetching was not done. Not only would this have lead to contention on the memory bus amongst the threads, but it would also mean that each thread was performing memory fetches from the slow uncached global memory. To prevent this, the technique of locally caching frequently used data was employed. Storing the transformation matrix in the shared memory allows all the threads within a block to fetch the data from the much faster shared memory space. This idea was also applied to the storing of the pixel intensities. Rather than write to global memory, each thread saves its computed intensity to a local shared array, which is later summed by the first thread of each group.

In a later revision of our implementation, we replaced the shared transformation matrix array with a constant memory array. As noted earlier, constant memory provides limited cached read-only memory to the threads. Rather than manually loading the transformation matrix in, we can simply read the values in from constant memory. This simplified the code as shown in Listing 1 and provided slightly better performance than manually pre-fetching the matrix (since we no longer needed to synchronize the threads after loading the data).

Third, we implemented two image generation kernels: the first one uses the built-in bilinear filtering provided by the texture map, and the second one uses our own implementation of bicubic interpolation. This was done because there are some applications that require more accuracy than what bilinear interpolation can provide, and so bicubic interpolation is presented in a separate kernel as described above.

Finally, it must be noted that we only compute partial sums of the squared error on the GPU and finish the similarity measurement on the CPU. This was mainly done for simplicity of implementation. Initially, we implemented a parallel prefix version of the mean squared error calculation, but perhaps due to poor implementation on our part the calculations accumulated too much error and thus produced poor results (i.e. low accuracy). To limit our exposure to error and due to ease of implementation, we only computed partial sums on the GPU and performed the rest of the computation on the CPU.

4. RESULTS

For our experiment, we compared the performance and accuracy of the CUDA implementation against our fast CPU version on the data set described earlier. We ran our tests 15 times across a period of about a week and the results are summarized in Table 2. In the proceeding tables and graphs, *CUDA 0* refers to the image registration using the bilinear kernel while *CUDA 1* is the one using the bicubic kernel.

lenna	Version	Run-time	Speedup	PSNR
	CPU	5.19	1.00	56.32
	CUDA 0	0.69	7.48	51.60
	CUDA 1	0.80	6.50	55.43
ndbuntu	Version	Run-time	Speedup	PSNR
	CPU	12.49	1.00	56.35
	CUDA 0	0.74	16.80	47.05
	CUDA 1	0.98	12.64	51.27
halo	Version	Run-time	Speedup	PSNR
	CPU	23.73	1.00	44.19
	CUDA 0	0.83	28.40	31.70
	CUDA 1	1.30	18.25	34.60
jump	Version	Run-time	Speedup	PSNR
	CPU	48.09	1.00	55.86
	CUDA 0	1.04	46.41	47.92
	CUDA 1	2.07	23.22	48.34
victory	Version	Run-time	Speedup	PSNR
	CPU	92.50	1.00	52.83
	CUDA 0	1.36	67.89	42.31
	CUDA 1	3.16	29.17	42.85
crabnebula	Version	Run-time	Speedup	PSNR
	CPU	205.31	1.00	54.26
	CUDA 0	2.24	91.47	44.86
	CUDA 1	6.24	32.92	45.80

Table 2: Summary of experimental results for each image, with the run-time in seconds

4.1 Performance

Figure 3 shows the average execution time for each registration on all the test images which are displayed along the X axis in sorted order (by size). It is clear from this chart that the running time of the CPU version is much higher than that of the CUDA versions. Moreover, this gap appears widen as the size of the input images grows. Additionally, the running time for the bicubic version is slightly higher than the bilinear, manifesting that it is slower than its simpler counterpart.

The graph in Figure 4 presents the speedup of the CUDA versions over the CPU implementation. The bilinear implementation was able to achieve between 7.5× - 91.5× speedup while the bicubic implementation was able to obtain between 6.5× - 33× speedup. This difference in speedup is as expected as bicubic interpolation is not only more computationally complex than bilinear filtering, but it also requires 16 memory fetches while bilinear only needs 4. The

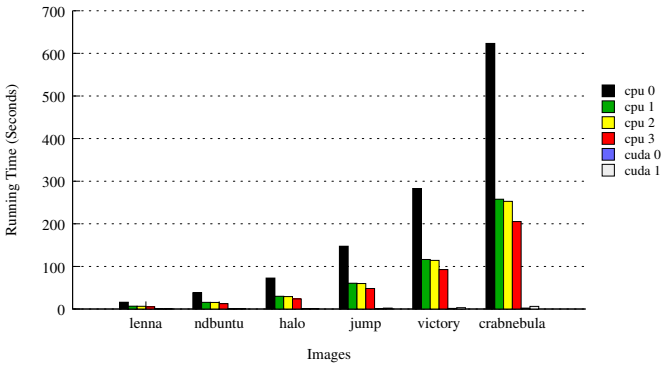

Figure 3: Average running time for all implementations for each test image set

additional memory fetches, in effect, stall the threads in the bicubic kernel, increasing the running time. This is readily apparent in Figure 5 which shows the efficiency (i.e. how well we are using the CUDA device). As demonstrated in this graph, because of the increased memory accesses the bicubic kernel is much less efficient than the bilinear implementation.

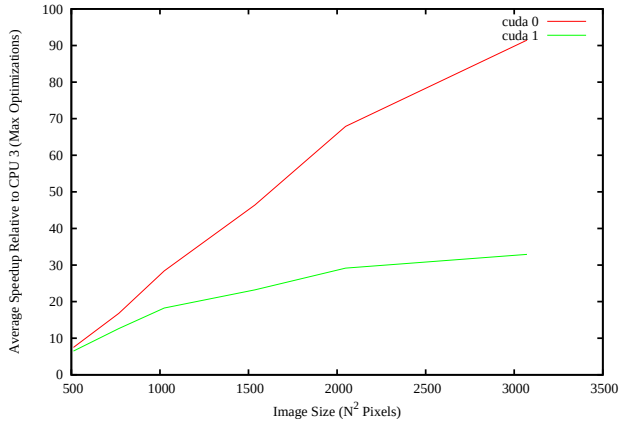

Figure 4: Average speedup over CPU version as image size increases

These graphs also reveal that the bilinear version scales well as image size increases while the bicubic version levels off much more quickly. Once again, this is mainly due to the increase in memory accesses leading to less efficient kernels.

4.2 Accuracy

To check the quality of the image registrations we calculated the mean square error between the image produced by the registration and the actual *target* image. The MSE was then used to compute the Peak-Signal-to-Noise Ratio (PSNR). As explained by Obukhov and Kharlamov [10], the PSNR is commonly used to evaluate image reconstruction quality. Generally values between 30, 50 denote high fidelity, while anything above 50 says the results are nearly identical. The PSNR was computed using the following formula:

$$PSNR = 10 * log_{10}(\frac{MaxIntensity^2}{MSE}) \quad (2)$$

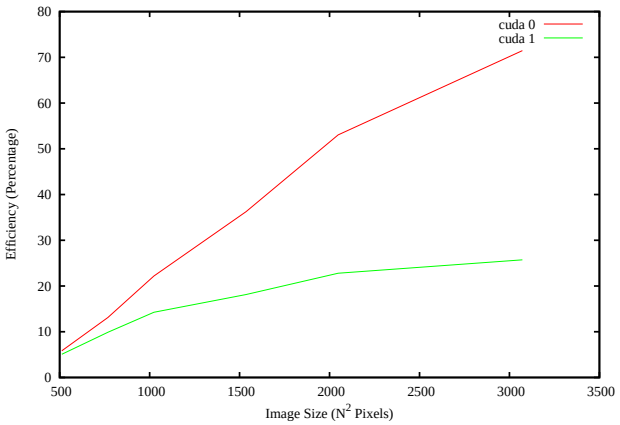

Figure 5: Average efficiency as image size increases

where *MaxIntensity* is the maximum pixel intensity value. Since we used portable grayscale pixmaps to store our image data, this value was 255.

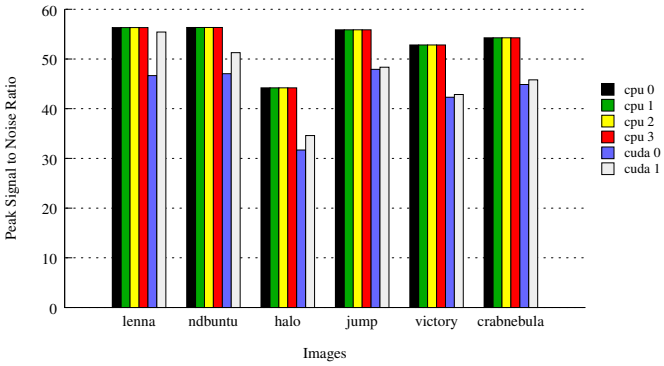

Figure 6: Average Peak-Signal-to-Noise Ratio for All Images

The chart in Figure 6 reveals that across the board the CPU version had a higher PSNR than the CUDA implementations. These means that it had higher accuracy and produced the best registration. However, the CUDA versions are still relatively accurate for the most part. As expected, the bicubic interpolation kernel produced a higher PSNR than the bilinear kernel. Overall all of the implementations produced PSNRs in the range $35 - 55$, which denotes high similarity to the actual *target* image. This discrepancy between the CPU and CUDA versions is most likely due to the non-standard floating point implementation in the GPU which leads to subtle but possibly damaging errors [10] [5].

4.3 Profiling

In order to understand what parts of the pipeline the CUDA implementations were spending their time in, we profiled each test run by timing each relevant function call. Figures 7 and 8 show the break down of the functions and their percentage of the running time for the lenna and crabnebula images respectively.

The lenna image is the smallest in our data set and had a total running time of 0.7 - 0.8 seconds for both CUDA versions. As can been seen in Figure 7, the vast majority of the

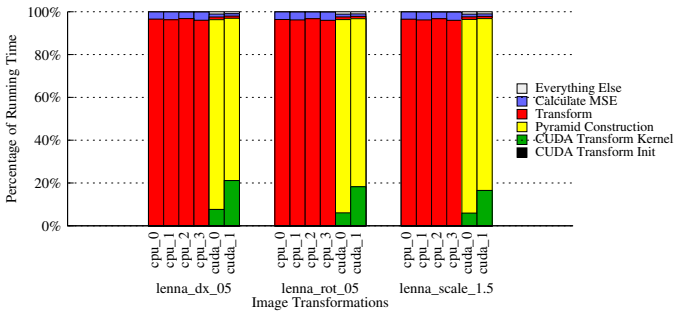

Figure 7: Function running time percentage for lenna

time is spent in the pyramid construction. This means it is spending most of its time waiting for the CUDA device to startup. Since this number is unpublished, we performed a startup micro-benchmark and found that the CUDA device takes about $0.5 - 0.6$ seconds to initialize before any execution on the GPU can occur. Since the total run-time for lenna is only 0.7 - 0.8 seconds, this 0.5 - 0.6 second startup cost completely dominates the running time, distorts the run-time cost of the pyramid component, and prevents an overall larger speedup in the application.

Note, this micro-benchmark does not refute the claims of Volkov and Demmel [16], as they measured the cost to launch a kernel and not the startup time of the CUDA device. Before any kernel can be called, the CUDA driver must perform some configuration. This is why many of the examples in the NVIDIA software developer's kit include sections of code to "warm up" the device, masking away this startup cost from the final timing information. Our micro-benchmark measured this time to be around 0.5-0.6 seconds.

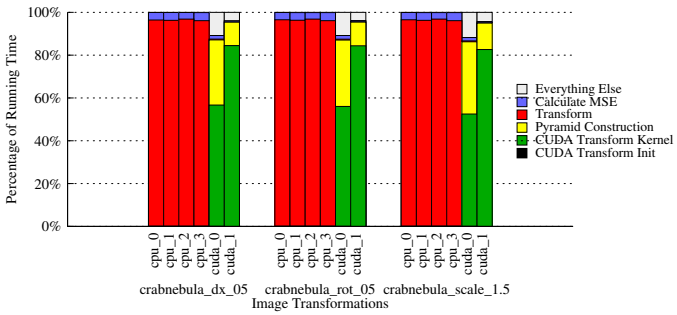

Figure 8: Function running time percentage for crabnebula

The run-time percentage chart for the largest image, the crabnebula data file, manifests a different set of issues. In this case, the transformation kernel execution dominates the running time which is desirable because this means that we are mainly occupied in the data parallel section of our code rather than the serial portion. Since the CUDA kernel dominates the running time, we are able to obtain a large amount of speedup.

However, there are a few other interesting trends in Figure 8 that need to be pointed out. First, the "Everything Else" portion of the algorithm grew in percentage relative to the lenna profiling. This portion of the code is mainly reading the image from the filesystem and storing it in host memory. Due to the large size of the image, this serial portion became a bigger factor in terms of running time percentage. Likewise, the pyramid component remains a big factor in the speed of the program. Once again, the startup cost of the CUDA device is a serious bottleneck and limits overall speedup.

The key point here is that in porting our CPU implementation to the GPU, we introduced new issues and reveal potential bottlenecks such as the startup time of the CUDA device and the increased role of serial portions of the implementation. Although a GPU core is very different from a CPU core, it is reasonable to say that since the Tesla C870 has 128 stream processors, a perfect linear speedup would mean that the maximum theoretical speedup on the CUDA device is 128×. As shown above, our code is achieving up to 90× speedup with the bilinear kernel and 33× with the bicubic, which is about 71% and 26% of the total possible speedup, respectively. As noted, the addition of unanticipated bottlenecks and sources of serialization contribute in limiting our speedup. Furthermore, the high cost of memory fetching greatly reduced the efficiency of the kernels and also prohibited performance gains.

5. FUTURE WORK

We would like to increase the performance of our implementation by moving remaining portions of the algorithm to the GPU, improving the current CUDA kernels, extending the functionality of the image registration application and investigating alternative optimization algorithms. To get a better idea of how the CUDA platform compares to the traditional OpenGL and DirectX methods, it may be prudent to implement a version of the image registration algorithm using the older GPGPU techniques and test it on the Tesla C870.

In terms of improving our current CUDA code, we may want to try to reduce the memory bottleneck in the bicubic kernel. As of this moment, our bicubic kernel performs 16 texture references to perform a precise bicubic interpolation. It is possible to take advantage of the bilinear texture hardware to perform an imprecise but fast high order interpolation with less memory access as explained by Sigg and Hadwiger [13]. Additionally, we can amortize the startup cost of the CUDA device if we considered an image registration system that worked on a stream of input images such as from a video source or a time lapsed camera. Another possible improvement would be to revisit the mean square error calculation and fully implement it with a parallel prefix algorithm, although the pay off for this seems limited as indicated by our profiling data.

Furthermore, we can further improve our implementation by using a more advanced optimizer. We mainly chose the Simplex optimizer because it was readily available in the GNU Scientific Library, allowing us to focus on the CUDA portion of the application. However, previous research studies have performed steepest descent and gradient optimizers on the GPU and these could be viable candidates for porting to CUDA. Perhaps, even the Marquardt-Levenberg algorithm would be a viable candidate for implementation. The reason to explore alternative optimization algorithms is that these more advanced optimizers can perform the minimizations in less iterations than the Simplex algorithm and thus algorithmically reduce the running time of the application.

6. CONCLUSION

In developing this application, it became quite apparent that the key to achieving the most performance from the CUDA device is to effectively manage the program's memory access patterns. Developers must minimize the amount of global memory accesses and take full advantage of the texture, constant, and shared memories. Our initial naive CUDA implementation simply passed the transformation matrix in as kernel parameter and we were only able to achieve 3× - 8× speedup on all of the test images. It was only after we reorganized our memory access pattern did we get drastic improve performance.

Although GPUs offer a vast amount of computational power, they still face the age old problem of the *memory wall*: meaning there are cases where the processors idle because data cannot be streamed in fast enough. In the case of image registration, due to the nature of interpolation, it is difficult to structure accesses that benefit from memory coalescing (reading a coherent block of memory with a block of threads) and thus the processors are required to wait for data to be fetched. Unfortunately, GPUs are designed for high throughput and not low latency, so these memory accesses, particularly to global memory, are quite costly. It is up to the programmer to effectively group the data access patterns and take advantage of the available memory resources.

In regards to programming in CUDA, there are a variety of things NVIDIA can do to improve the platform. For instance, a profiling tool that can provide timing information amount the memory access patterns would be quite beneficial for developers. It would allow programmers to structure their programs to take full advantage of the GPU's memory hierarchy which is vital to achieving full performance. Moreover, NVIDIA should consider exposing more of the built-in hardware through the CUDA API. For instance, although the GPU contains mipmap hardware units, this feature is only exposed through the traditional graphics programming interfaces. Likewise, the texture handling needs to be refined a bit. Currently it is not possible to have an array of texture references, nor is it possible to pass in a texture reference to a kernel thus forcing messy kludges to select the appropriate textures. These last two issues were particularly relevant to our implementation of the pyramid component where we had to implement our own parallel reduction and had to perform a conditional check to determine if we should store a *source* or *target* image.

Overall, the study presented in this paper presents a thorough performance analysis of our CUDA 2-D rigid image registration implementation. The investigation reveals some of the problems encountered in the converting the CPU implementation to the CUDA framework and identifies key techniques and practices to overcoming these obstacles. Likewise, we also examine some of the limiting factors that prevent obtaining maximum performance on the CUDA device and discuss a few recommendations on how to not only speed up the image registration application, but also to improve the CUDA platform as a whole. In the end, our CUDA implementation is able to achieve up to 90× speedup with the bilinear kernel and 33× with the bicubic version.

7. REFERENCES

[1] Free Software Foundation. GNU scientific library. `http://www.gnu.org/software/gsl/`.

[2] J. Fung and S. Mann. Using graphics devices in reverse: GPU-based image processing and computer vision. In *IEEE Int'l Conf. on Multimedia & Expo*, pages 9–12, 2008.

[3] A. A. Goshtasby. *2-D and 3-D Image Registration*. Wiley-Interscience, 2005.

[4] M. Harris. Mapping computational concepts to GPUs. In *GPU Gems 2*, pages 493–508. Addison Wesley, 2005.

[5] K. E. Hillesland and A. Lastra. GPU floating-point paranoia. In *GP2 ACM Workshop on General Purpose Computing on Graphics Processors*, page 8, 2004.

[6] F. Ino, J. Gomita, Y. Kawasaki, and K. Hagihara. A GPGPU approach for accelerating 2-d/3-d rigid registration of medical images. In *International Symposium on Parallel and Distributed Processing and Applications (ISPA)*, pages 939–950, 2006.

[7] A. Kubias, F. Deinzer, T. Feldmann, D. Paulus, B. Schreiber, and T. Brunner. 2d/3d image registration on the GPU. *Pattern Recognition and Image Analysis*, 18(3):381–389, 2008.

[8] P. Muyan-Özçelik, J. D. Owens, J. Xia, and S. S. Samant. Fast deformable registration on the GPU: A CUDA implementation of demons. In *International Conference on Computational Science and Its Applications (ICCSA)*, pages 223–233, 2008.

[9] NVIDIA. *NVIDIA CUDA Compute Unified Device Architecture, Programming Guide, Version 2.0*. NVIDIA, 2008.

[10] A. Obukhov and A. Kharlamov. Dct8x8. *NVIDIA Software Development Kit (SDK)*, 2008.

[11] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[12] W. Plishker, O. Dandekar, S. Bhattacharyya, and R. Shekhar. Towards a heterogeneous medical image registration acceleration platform. *Biomedical Circuits and Systems Conference (BIOCAS)*, pages 231–234, Nov. 2007.

[13] C. Sigg and M. Hadwiger. Fast third-order texture filtering. In *GPU Gems 2*, pages 313–317. Addison Wesley, 2005.

[14] T. Sugiura, D. Deguichi, T. Kitasaka, K. Mori, and Y. Suenaga. A method for accelerating bronchoscope tracking based on image registration by GPGPU. In *Augmented environments for Medical Imaging including Augmented Reality in Computer-aided Surgery (AMI-ARCS) 2008 (Medical Image Computing and Computer Assisted Intervention 2008)*, 2008.

[15] P. Thévenaz, U. Ruttimann, and M. Unser. A pyramid approach to subpixel registration based on intensity. *IEEE Transactions on Image Processing*, 7(1):27–41, January 1998.

[16] V. Volkov and J. W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

[17] B. Zitová and J. Flusser. Image registration methods: a survey. *Image and Vision Computing*, 21(11):977–1000, October 2003.