SANDIA REPORT
SAND2007-6301
Unlimited Release
Printed October 2007

Implementing Wide Baseline Matching Algorithms on a Graphics Processing Unit

Daniel S. Myers, Antonio I. Gonzales, Fredrick H. Rothganger, and Kurt W. Larson

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94AL85000.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd.
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online order: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2007-6301
Unlimited Release
Printed October 2007

Implementing Wide Baseline Matching Algorithms on a Graphics Processing Unit

Daniel S. Myers, Antonio I. Gonzales, Fredrick H. Rothganger, and Kurt W. Larson
Advanced Information Systems
Sandia National Laboratories
P.O Box 5800
Albuquerque, NM 87185-0576

Abstract

Wide baseline matching is the state of the art for object recognition and image registration problems in computer vision. Though effective, the computational expense of these algorithms limits their application to many real-world problems. The performance of wide baseline matching algorithms may be improved by using a graphical processing unit as a fast multithreaded co-processor. In this paper, we present an implementation of the difference of Gaussian feature extractor, based on the CUDA system of GPU programming developed by NVIDIA, and implemented on their hardware. For a 2000x2000 pixel image, the GPU-based method executes nearly thirteen times faster than a comparable CPU-based method, with no significant loss of accuracy.

This page intentionally left blank.

Contents

1. Introduction 9

2. Wide Baseline Matching 9

3. Difference of Gaussian Feature Extraction 10

4. CUDA Architecture 11

5. Processing Images 12

6. GPU-DoG Algorithm 13

7. Experimental Results 15

8. Future Work 20

9. References 21

This page intentionally left blank.

Figures

Figure 1. DoG feature selection: Gaussian pyramid, DoG pyramid, peak discovery 10
Figure 2. Location and scale of features found by DoG feature extractor 11
Figure 3. Image *C* is tiled with blocks of constant size 13
Figure 4. Repeatability for the three algorithms 16
Figure 5. Satellite image of Florida used for timing analysis 17
Figure 6. Execution time vs. Image size 18
Figure 7. Per method execution times for 8500 and 8800 cards 20

Tables

Table 1. Timing data (in seconds) for GPU-DoG and FL on the 8500 and 8800 machines 17
Table 2. GPU vs. CPU speedup for both machines 18
Table 3. Per method execution times and speedup: 8800 vs. 8500 19

This page intentionally left blank.

1. Introduction

Wide baseline matching (WBM), based on abstract local features, is the state of the art paradigm in computer vision for object recognition and image registration. While highly effective across a wide array of problems, these methods tend to be very computationally expensive. Recently a multitude of work has emerged using the graphics processing unit (GPU) for scientific computing. Specifically, a new technology called CUDA (compute unified device architecture), available on NVIDIA GPUs, provides a C programming interface that greatly simplifies implementing software on the GPU.

In this paper, we present a CUDA implementation of the difference of Gaussian (DoG) feature selector used in WBM. We will give a brief overview of WBM, DoG feature selection, and the CUDA architecture. Then we will detail our CUDA implementation of DoG and compare the performance on two different NVIDIA GPUs to that of an existing CPU based implementation. Finally, we will discuss further performance enhancements and the implementation of the rest of the WBM algorithm.

2. Wide Baseline Matching

Wide baseline matching is a paradigm of algorithms which find matching features among a set of images and solve for the geometric transformations between them. These algorithms have applications in object recognition, image registration, robot navigation, and many other image processing problems. While varying widely in design, WBM algorithms share three essential steps:

1. *Feature Extraction* – Individual features are selected within the image that are mathematically stable and can be reliably extracted from multiple images of the same target even with variations in scale and illumination intensity. These features tend to be abstract and based on object attributes such as variations in gradients or areas of common contrast.

2. *Descriptor Creation* – The local area around each extracted feature is described by a feature vector, called a descriptor. Descriptors are generated in a way that removes the effects of rotation and illumination variation.

3. *Geometric Matching* – Candidate matching features are chosen by comparing the similarity between their descriptors. The locations of a set of candidate matches are then geometrically tested to determine if they share a common transformation. Combining appearance (descriptors) and geometry based matching leads to very low false positive rates.

For the purpose of this paper, we focus on the WBM algorithm presented by David Lowe in 2004 [1]. Specifically, we work with the DoG feature extraction algorithm, detailed in the next section.

3. Difference of Gaussian Feature Extraction

The DoG feature extractor searches for locations in an image which are scale invariant. To achieve scale invariance, features must be found in the image across all possible scale changes (scale space). The Gaussian function is used as a kernel to search through scale space through convolution with an input image. The Gaussian function, *G*, is defined as follows:

$$G(x, y, \sigma) = \frac{1}{2\pi\sigma^2} e^{\frac{-(x^2+y^2)}{2\sigma^2}}$$

where *x, y* is the position, relative to a pixel, and σ is standard deviation. Lowe proposed that stable features across scales can be found by locating peaks in the differences between convolutions of an image with a varying Gaussian kernel [1]. Each of these difference images is called a *difference of Gaussians*.

Peaks are identified by creating a set of Gaussian convolutions of an image at regular intervals of σ. The resulting ordered set of images is called a Gaussian pyramid. DoGs are produced for each pair of adjacent images in the Gaussian pyramid, creating a pyramid of DoGs. Starting with the second image in the DoG pyramid, each pixel is compared with the eight neighboring pixels in the same image and the nine corresponding pixels in the DoG images above and below it in the pyramid. If the pixel value is either the maximum or minimum value, it is a potential peak. Figure 1 displays the feature selection process.

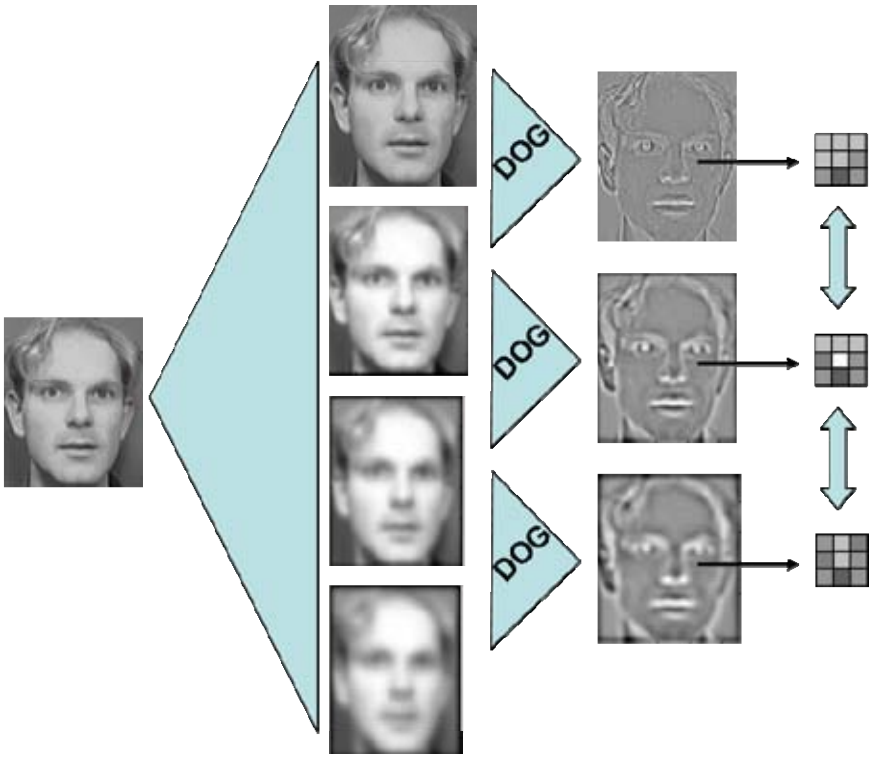

Figure 1: DoG feature selection: Gaussian pyramid, DoG pyramid, peak discovery

To increase the sampling of scales, the base image is down-sampled to half its size and the same process is repeated. Each time this processing is repeated it is called an octave. The number of octaves created is usually logarithmic with respect to the size of the original image.

The set of potential scale-invariant features are examined to ensure they are well defined. First, potential features with very low intensity values are discarded. These features are unstable with respect to lighting changes. Second, DoGs create strong responses along edges and many of these points are poorly defined and sensitive to noise. A measure of principle curvature is used to discard these poorly defined potential features. Finally, a 3D quadratic fitting function is used to interpolate the sub-pixel and sub-scale location of the remaining features. At this point, feature selection is complete. Figure 2 shows an image with its selected features. Each feature sits at the center of the circle, with the radius representing the scale at which the feature was detected.

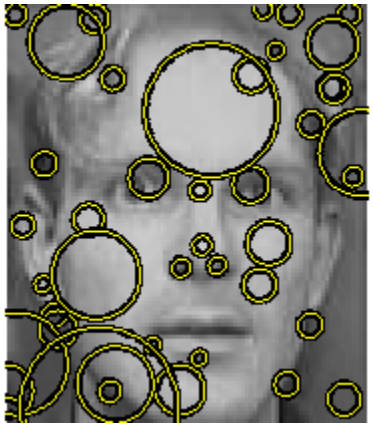

Figure 2: Location and scale of features found by DoG feature extractor

After extraction each feature is assigned a characteristic orientation and a descriptor is built for the local area location around the feature. These steps are not part of our GPU implementation and, therefore, are not described further in this paper. For a detailed description, see [1].

4. CUDA Architecture

CUDA is an application programming interface (API) that enables general purpose computation on NVIDIA GPUs. The CUDA interface uses the C programming language, making scientific computing on GPUs more accessible than previous OpenGL based technologies. Currently, the CUDA SDK is only available for NVIDIA's G80 family of GPUs, but NVIDIA promises that the technology will be supported on all its future hardware.

The CUDA programming model uses the GPU as a coprocessor, capable of supporting the execution of applications on the host CPU. Functions executed on the GPU – called *kernels* – are called from programs running on the host CPU. The GPU maintains its own DRAM, and values can be exchanged between the device and host memories with calls to special memory-management functions in the CUDA API.

As a coprocessor, the GPU executes a large number of threads in parallel. Much of the challenge of CUDA programming revolves around finding implementations of algorithms that take advantage of this parallelism. CUDA groups its threads into *thread blocks*. Each block is a collection of threads that share access to a block of device memory. This allows threads within a block to cooperate and solve problems more efficiently.

Conceptually, blocks are organized into a grid of two or three dimensions, with each block having unique indices. Likewise, the threads within a block are also grouped into a grid, with their own indices. The combination of block and thread indices uniquely identifies each thread in an executing kernel program. The dimensions of the block grid and thread grid are specified when the kernel function is invoked, as follows:

```
dim3 threads(16, 16);
dim3 blocks(12, 12);
sampleKernelGPU<<<blocks, threads>>>();
```

This example creates an 12x12 grid of blocks, with each block having a 16x16 grid of threads. CUDA allows a maximum of 512 threads per block. There can be at most $2^{16}x2^{16}x2^{16}$ blocks. NVIDIA recommends using at least twice as many blocks as there are processors on the device to reduce idle time during execution.

When a kernel program is invoked, all the created threads execute the kernel in parallel. Threads may access values stored in global device memory, and threads within the same block have access to shared memory, but the threads may execute in arbitrary order.

5. Processing Images

The block-based structure provides a convenient way of performing kernel operations on images. This section explains the basic concepts used in the GPU-DoG image processing kernels. There are many other ways of implementing image operations in CUDA, but this method is arguably the simplest and illustrates many of the practical details of CUDA programming.

We assign one dedicated thread to each pixel in the image. Each thread executes the kernel program to process its pixel. When all threads have executed, every pixel in the image will be successfully processed. Because there is an upper limit of 512 threads per block, this requires multiple blocks for any image with more than 512 pixels. Let each block be a square grid of threads, and all blocks the same size. Since the size of the image is known and the block dimensions are constant, we can compute the size of the block grid required to tile the image and create one thread per pixel. Figure 3 gives a graphical depiction of this approach.

The kernel program is written so that each thread can effectively process its pixel in parallel, without any information from other threads. First, the kernel program uses CUDA API calls to determine the indices of the currently executing thread and the indices of its block. Combined with the known constant block dimensions, these indices identify the pixel location in the image corresponding to the current thread. The thread

then performs processing for its pixel and stores the results in the appropriate location in an output image. It is important to remember that all threads execute the same kernel program in parallel. Therefore, the kernel must be written so that all threads will execute correctly, regardless of their position in the image tiling.

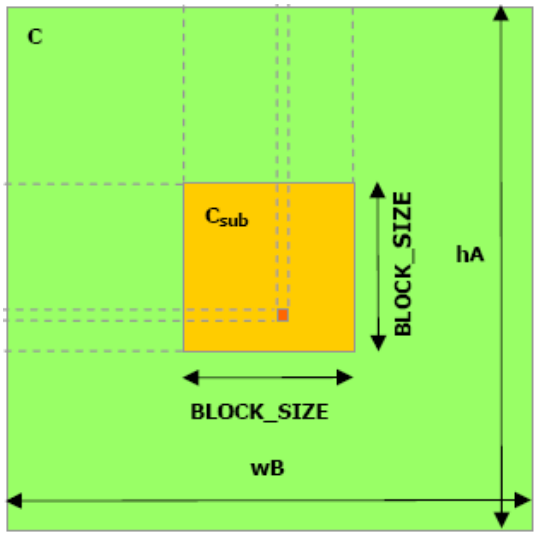

Figure 3: Image *C* is tiled with blocks of constant size [2]

6. GPU-DoG Algorithm

This section covers the major parts of our GPU-DoG implementation. Our implementation has the following major steps:

1) Allocation of device memory
2) Gaussian blurring of images
3) Difference of Gaussians computation
4) Threshold masking
5) Comparison of DoG images to find interest points
6) Refinement of interest point locations
7) Down-sampling and preparation for next octave

Though the basic steps of the algorithm are the same as other DoG implementations, we have made changes to take advantage of GPU processing. In particular, graphics hardware excels at rapid parallel execution of relatively short programs. This leads us to divide our algorithm into several small kernels, and execute them sequentially over the entire image, as opposed to performing in-depth processing on one pixel at a time.

The most unique aspect of our algorithm is the use of multiple masking operations to streamline execution. As discussed in section 3, there are several tests in the algorithm to reject pixels that cannot be features: threshold, curvature, and neighborhood comparison. In order to reduce the number of unused threads on the GPU, we maintain a list of working pixels and process only those that have passed previous tests. After each new test, the list of working pixels is filtered to remove any rejected points. This change

greatly accelerates execution on the GPU by ensuring that every thread is working on a pixel that still has the potential to be a true feature.

Before executing any other steps, we first allocate memory on the GPU for all intermediate copies of the image required during execution. This is done with the `createSpaceOnDevice()` function, which takes image height and width as inputs, and returns a pointer. These pointers refer to locations in GPU memory, and are passed to kernel functions. The allocation step is done once at the beginning of execution.

After allocation is complete, Gaussian blurring of the images is performed using the `convolutionSeperable()` method. This is an NVIDIA program for implementing efficient separable convolutions on the GPU. A full explanation of the algorithm is beyond the scope of this paper. Consult the documentation for further information [3]. This algorithm requires a Gaussian kernel of fixed width as an input. This prevents us from using variable-width kernels during the blurring process, which can affect the quality of the final interest points. In future implementations we will modify this program to take advantage of variable-width kernels.

The DoG images are produced using the kernel defined in `imageDifference_kernel.cu`. This function takes two blurred images as input and creates one thread per image pixel location. Each thread computes the difference between the input images at its assigned pixel location and stores the result in the output. The resulting DoG images are stored in the pre-allocated device memory. The host keeps a pointer to the location of each DoG image. Interest point extraction begins after the blurred and DoG images have been computed.

As discussed above, we use a series of masking operations to identify pixels that cannot possibly be interest points. In later steps, these pixels are ignored. The first mask is based on a simple threshold check, and computed by `threshold_kernel.cu`. One thread is created per pixel in the DoG image. Each thread checks the pixel value against a threshold. The output of this kernel is a binary image, where pixels that pass the threshold test are assigned true and all other pixels are false.

To simplify computations with the mask we use a special data structure: the `locationArray`. The `locationArray` stores the locations of possible interest points in the image. When the location of a possible interest point is found, its index (calculated using the one-dimensional representation of the image) is stored in the `locationArray`. To populate the `locationArray`, the threshold mask is read back to the CPU from the GPU and examined. The indices of pixels with a true mask value are stored in the `locationArray` in sequential order. A counter records the total number of true values found in the mask image. In general, transferring data between the GPU and CPU is an expensive operation and should be avoided, but in this case, the operation executes more efficiently on the CPU.

The `locationArray` is used to simplify the next step: difference of Gaussian comparison. Three DoG images are input to `comparison_kernel.cu`: the central image and the two

DoG images above and below it in the pyramid. One thread is created for each element of `locationArray`. This thread compares the pixel value at the stored location to its eight neighbors in the central image and to the nine neighbors in both the upper and lower DoG images. If the pixel value is greater than or less than all of its neighbors, the point is a potential feature. The curvature at the pixel is computed in order to reject inherently unstable interest points positioned on edges. If a pixel value passes both tests, its position remains stored in the `locationArray`. If it fails either test, its position in the array is set to false signifying that the pixel is no longer a candidate interest point. After the comparison is complete, the `locationArray` is read back to the CPU, and rebuilt so the only the surviving candidate locations remain. The count is updated to reflect any changes in the number of candidate points.

The final step is point refinement, performed by `refinePos_kernel.cu`. This kernel accepts the final `locationArray` and the three difference of Gaussian images as input, and creates one thread for each interest point location in the `locationArray`. The position refinement method solves a linear system to determine the sub-pixel location of the interest point peak. The refined location and interpolated peak value are returned.

After processing all the DoG images for an octave, we use down-sampling by a factor of two to reduce the work required at the next octave. The final image of the Gaussian pyramid from the last octave is down-sampled to become the first image of the Gaussian pyramid in the next. The final two DoG images from the last octave are down-sampled and become the first two DoG images in the next. The down-sampling kernel, called `decimate_kernel.cu`, creates one thread for each pixel (x, y) in the down-sampled image. The thread obtains the pixel value for the down-sampled image from the pixel at position ($2x$, $2y$) in the original image. As described earlier, these steps (except for memory allocation) are repeated until a number of octaves, logarithmic to the size of the base image, have been processed.

7. Experimental Results

We tested the performance of our GPU-DoG implementation in two ways. First, we used a standard test for repeatability to show that the algorithm produces correct features. Secondly, we measured execution time to quantify performance improvement over a common CPU based implementation.

For our first test, we evaluated the repeatability of points produced by GPU-DoG. Given two images of the same scene with a known homography between them, repeatability measures an algorithm's ability to find the same features in both images. High repeatability is a desirable characteristic for any WBM technique. We use the test to verify that GPU-DoG's results are both correct and also competitive with other versions of DoG feature extraction.

We tested GPU-DoG against the CPU-based DoG feature extraction implementation contained in the FL image processing library, written by Fred Rothganger (Org. 6341). FL has proven effective at solving many image processing problems at Sandia. To evaluate the repeatability of extracted feature points, we used Krystian Mikolajczyk's

region detector performance evaluation software, the standard test for algorithms of this type [4].

The software takes six views of the same image with progressive projective rotations of 20% to 60%. The feature extractor is run on each image and all the extracted points from each rotated image are transformed into the base image. Each feature is assigned a circle denoting its local neighborhood, based on the scale where it was extracted. The repeatability measurement is calculated as the percentage of points from different images that lie in the same location and have a neighborhood overlap of at least 60%. If an algorithm has high repeatability, its results are robust in the presence of scene transformations. Figure 4 summarizes the results.

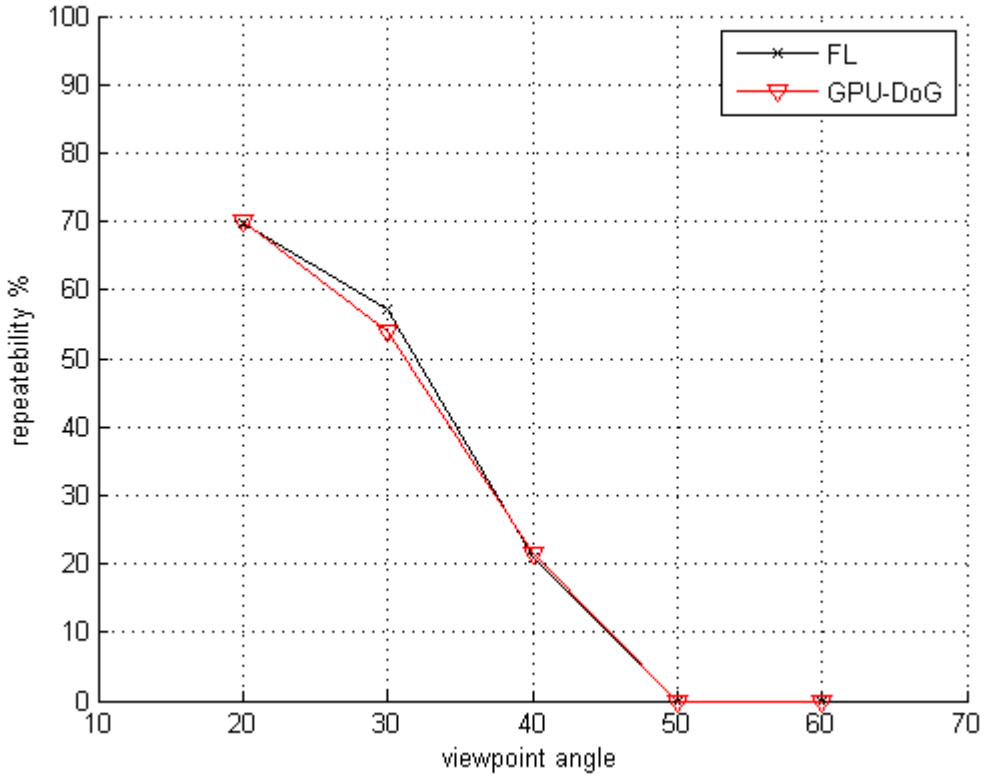

Figure 4: Repeatability for the two algorithms

FL slightly outperforms GPU-DoG on repeatability, though the results are reasonable, and demonstrate that GPU-DoG is producing correct results. The difference in repeatability between GPU-DoG and FL is acceptable given the timing results presented below. We believe that FL's superior performance is due to the use of variable-width Gaussian kernels; GPU-DoG uses a fixed-width kernel. We plan to include variable-width kernels in future implementations of GPU-DoG.

For our second test, we compared execution times of GPU-DoG and FL. We evaluated the execution time using a satellite image of Florida (shown in Figure 5) scaled to five different sizes: 250x250, 500x500, 1000x1000, 1500x1500, and 2000x2000.

Figure 5: The satellite image of Florida used for timing analysis

The timing tests used two different machines. The first is equipped with an NVIDIA 8500 GTS video graphics card (2 processors), a 3.2 GHz Pentium D CPU, and 3.5 GB RAM. The second machine has a higher-end NVIDIA 8800 Ultra (16 processors), dual Xeon 3.73GHz CPUs, and 3GB of RAM. Each of the 8800 Ultra's processors is 1.5 times as fast as the processors on the 8500 GTS. The timing results for each image, machine, and algorithm (in seconds) are summarized in Table 1 and Figure 6.

Image Size	CPU: Pentium D 3.20 GHz	CPU: Dual Xeon 3.73 GHz	CUDA: GeForce 8500 GTS	CUDA: GeForce 8800 Ultra
250	0.0744	0.0640	0.0774	0.0578
500	0.2972	0.2560	0.1063	0.0717
1000	1.1932	1.0388	0.2398	0.1251
1500	2.6802	2.3401	0.5048	0.2194
2000	4.7776	4.1547	0.7390	0.3221

Table 1: Timing data (in seconds) for GPU-DoG and FL on the 8500 and 8800 machines

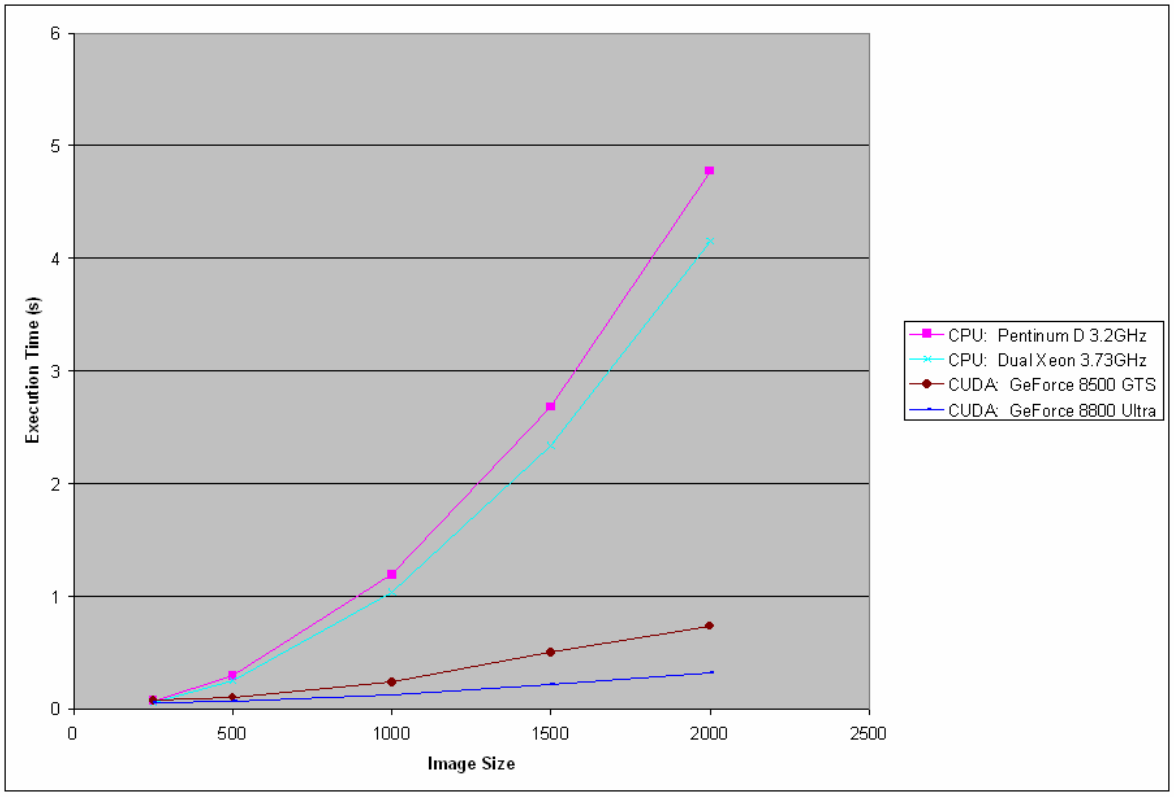

Figure 6: Execution Time vs. Image Size

For the 250x250 image, there is little difference between the different methods and machines; in fact, FL is slightly faster. This is due to the fixed overhead cost of copying data to the GPU and allocating device memory. As image size increases, this start-up cost becomes insignificant and GPU-DoG performance significantly exceeds that of FL.

Table 2 shows the speedup obtained by switching from the CPU-based FL algorithm to GPU-DoG on each of the two machines.

Image Size	Speedup: GeForce 8500 GTS vs. Pentium D 3.2 GHz	Speedup: GeForce 8800 Ultra vs. Dual Xeon 3.73 GHz
250	0.9615	1.1064
500	2.7951	3.5720
1000	4.9765	8.3031
1500	5.3092	10.6637
2000	6.4645	12.9005

Table 2: GPU vs. CPU speedup for both machines

Though the execution time of GPU-DoG is clearly superior to FL, we can also compare the performance of the two NVIDIA graphics cards. The 8800 Ultra card has 16 processors, each 1.5 times as fast as the 2 in the 8500 GTS processor, so we expect

computational power roughly equal to twelve 8500 GTS GPUs. If our implementation is efficient, we expect GPU-DoG to execute approximately twelve times as fast on the 8800 Ultra.

From the raw times in Table 1, the 8800 Ultra is approximately 2.3 times faster than the 8500 GTS, far below our prediction. Table 3 shows a breakdown of execution times for each GPU-DoG step, and the speedup gained by using the faster 8800 card. Note that "Non-GPU Time" includes all time required for memory allocation, data transfer, and CPU control code.

Operation	GeForce 8500 GTS	GeForce 8800 Ultra	Speedup: GeForce 8800 Ultra
Convolution	0.2943	0.0259	11.3483
Refinement	0.0113	0.0052	2.1718
Compare	0.0289	0.0037	7.7535
Decimate	0.0239	0.0036	6.6428
Threshold	0.0209	0.0021	9.9764
Difference	0.0399	0.0040	9.9529
Non-GPU Time	0.3198	0.2775	1.1524

Table 3: Per method execution times and speedup: 8500 vs. 8800

For the first six tasks, which are performed on the GPU, the 8800 has a cumulative time of .0446 seconds. The 8500 performs the same tasks in .4193 seconds, giving a speedup ratio of 9.4, much closer to the expected speedup of 12. Based on this analysis, we can conclude that the 8800 Ultra GPU is significantly faster than the 8500 GTS when performing GPU tasks, but the overall performance improvement is lessened by factors unrelated to the processing speed of the GPU. Most of the non-GPU time is taken up by I/O operations: reading and writing data to and from the card. Because this cost is not improved by changing to the 8800 card, it limits the overall speedup, despite the fact that GPU tasks execute significantly faster on the higher-end card. Figure 7 illustrates this fact.

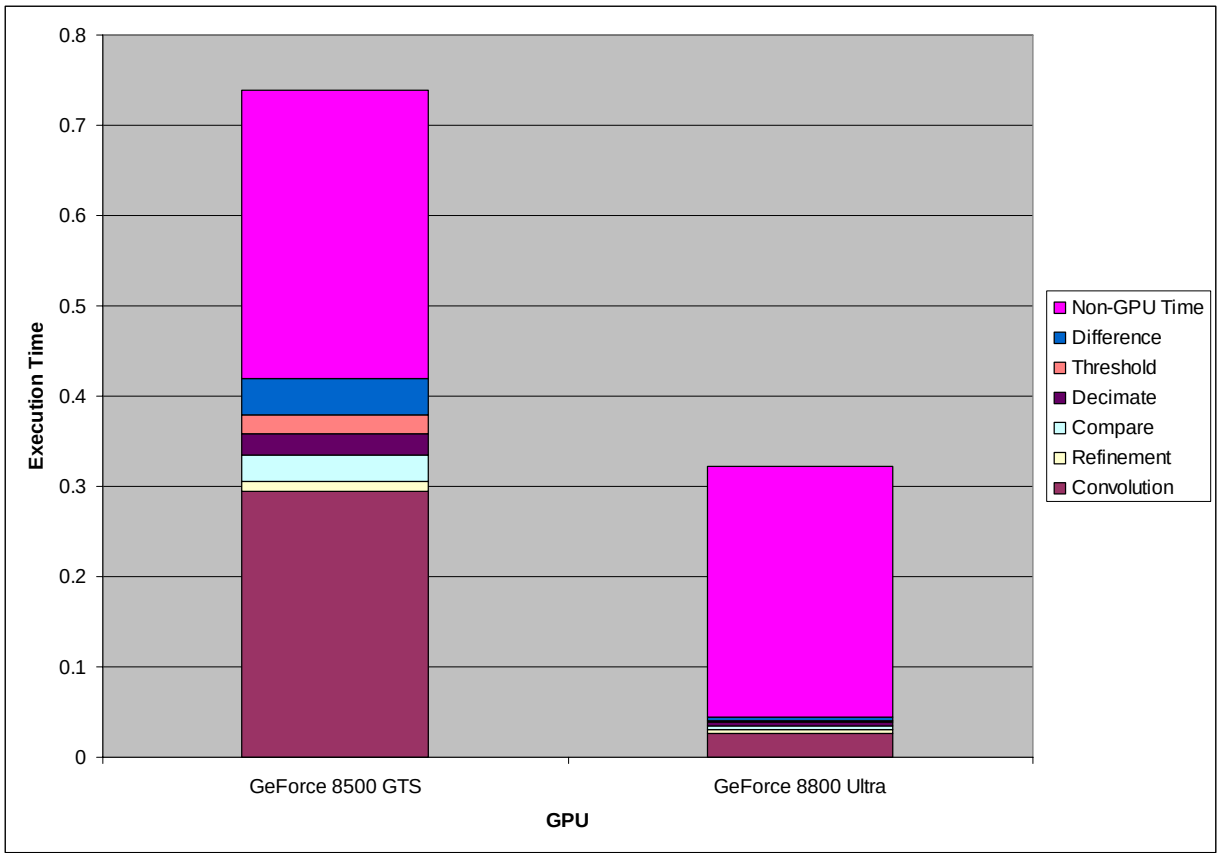

Figure 7: Per method execution times for 8500 and 8800 cards

8. Future Work

The present work shows great promise for future development. Compared to the CPU-based FL implementation, GPU-DoG produces features of competitive quality, and is significantly faster for large images. There are several possible directions for future work.

First, this test implemented only the first step of the wide baseline matching: feature extraction. Future work will focus on implementing descriptor creation and feature matching on the GPU. Second, we would like to implement some of the many other feature extraction algorithms other than DoG. In particular, we would like to develop a GPU-enabled version of the Maximally Stable Extremal Regions algorithm [5].

The combination of computational power and relatively low cost makes general-purpose GPU technology a key performance enhancer, applicable to a wide variety of scientific computing problems.

9. References

[1] Lowe, D.G. "Object recognition from local scale-invariant features". In *Proc. International Conference on Computer Vision*, pp.1150-1157, 2004.

[2] *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*. http://developer.download.nvidia.com/compute/cuda/1_0/NVIDIA_CUDA_Programming_Guide_1.0.pdf, Accessed September 24, 2007.

[3] Podlozhnyuk, V. *Image Convolution with CUDA.* http://developer.download.nvidia.com/compute/cuda/sdk/website/projects/convolutionSeparable/doc/convolutionSeparable.pdf, Accessed September 24, 2007.

[4] K. Mikolajczyk, T. Tuytelaars, C. Schmid, A. Zisserman, J. Matas, F. Schaffalitzky, T. Kadir and L. Van Gool. "A comparison of affine region detectors". In *IJCV* 65(1/2):43-72, 2005.

[5] J. Matas, O. Chum, M. Urban, and T. Pajdla. "Robust wide baseline stereo from maximally stable extremal regions". In *Proceedings of the British Machine Vision Conference*, pages 384–393, 2002.

Distribution

5	MS	0576	Daniel Myers, 5534
1		0661	Joselyne Gallegos, 5530
1		1009	Fred Rothganger, 6341
1		1243	Kurt Larson, 5534
5		1243	Antonio Gonzales, 5534
1		1243	Carol Harrison, 5534
1		1243	Steve Kempka, 5555
1		9018	Central Technical Files, 8944 (electronic copy)
1		0899	Technical Library, 9536 (electronic copy)