ABSTRACT

With the decline of single-core processor computers and the shift to multi-core CPUs and the extensive use of graphics processing units we see an industry that is constantly revolutionized. In the field of computing, processing power holds the key to solving complex problems that are performance intensive. The NVIDIA CUDA (Computer Unified Device Architecture) provides the tools necessary to solve problems in parallel computing at relatively low costs with a performance potential similar to supercomputing clusters. GPUs have outpaced standard CPUs in floating-point performance in recent years and have become easy, if not easier, to program than multi-core CPUs. Presented herein is a framework for converting traditional CPU-based template matching algorithms into a GPU-based, which can be generalized and adapted to other CPU-intensive digital image processing algorithms. We can use the highly scalable GPU architecture to improve matching algorithms and image processing functions which are usually computationally intensive depending on the intended applications. From previous case studies in performance and the lack of an open source library style for image processing on the GPUs, due to the relatively new nature of image processing on the GPU, we see a great benefit in creating a set of functions that could be available for anyone interested in research and teaching.

An experimental approach for evaluating the GPU optimized template matching algorithm was taken. A small dataset of different sized images were created for comparison between CPU and GPU runtimes. The results show a drastic reduction in runtime compared to a CPU implementation while retaining accuracy.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 CPU Processing and Graphics Processing

In the late 1980s and early 1990s, the growth in popularity of graphically driven operating systems such as Microsoft Windows helped to create a market for a new type of processor [1]. Graphics Processing Units offered the operating system the assistance to display content without putting a strain on the CPU and memory. While in the early stages they were used for basic 2D acceleration, 3D graphics made their way quickly into government applications, scientific research, and cinema just to name a few. It was not until the release of the Open Graphics (OpenGL) library by Silicon Graphics in the 1990s that a huge market for consumer applications began to rise with the development of video games and the growth of companies such as NVIDIA and ATI Technologies.

The NVIDIA CUDA (Computer Unified Device Architecture) provides the tools necessary to solve problems in parallel computing at relatively low costs with a performance potential similar to supercomputing clusters. GPUs have outpaced standard CPUs in floating-point performance in recent years and have become easy, if not easier, to program than multi-core CPUs. However, it is still difficult to implement a template algorithm which is highly efficient and still able to respond to the parameters of position invariance, rotation and scale. Real time applications such as product inspection, surveillance, and face recognition rely heavily on fast and accurate template matching routines which have to take into a count every parameter.

Using MATLAB under the Linux environment, we developed a basic template matching algorithm and quantified the results of the benchmark experiments. The

Pyramid matching implementation and a hybrid approach using the Fourier Transform in conjunction with a spatial implementation can be implemented in the future based on the results from a basic iterative match. Parallelizing the basic accumulator array matching algorithm will provide a base result for other implementations of template matching and the experience to identify which parts of the code can be chosen to parallelize. The MEX and PTX file interfaces together with CUDA C facilitates the development of parallel computing algorithms. The GPU cluster available at the Department of Computer Science at the University of Texas at Brownsville will serve as the basis of this implementation. This GPU cluster is comprised of four NVIDIA Tesla C1060 graphics cards each with 240 cores, two Quad-Core Intel Xeon E5520 processors and 32GB of DDR3 RAM reaching speeds of 4 Teraflop/s; which is more than ideal for the development of Image Processing functions and other parallel applications.

NVIDIA's release of the GeForce 3 series in 2001 represents the most important breakthrough in GPU technology. The GeForce 3 series was the computing industry's first chip to implement Microsoft's then-new DirectX 8.0 standard. This standard required that compliant hardware contain both programmable vertex and programmable pixel shading stages. For the first time, developers had some control over the exact computations that would be performed on their GPUs [1]. Researchers looking for general purpose computation took notice of this great opportunity and began using programmable shaders using OpenGL and DirectX APIs to accelerate their computations. Initially, the pixel shader data displayed as colors represented results of the particular application which was being developed.

It was not until the release of NVIDIA’s GeForce 8800 GTX series in 2006 that researchers could actually program general-purpose applications by using the newly implemented Compute Unified Device Architecture (CUDA). For the first time ever, researchers now had an Application Programmable Interface (API) that allowed access to the arithmetic logic unit. With this new hardware and API, researchers could develop highly scalable applications at very low costs compared to supercomputing clusters. Anybody that has access to a CUDA compatible graphics card can now begin developing software using the CUDA architecture.

```
#include<iostream>
__global__ void add( int a, int b, int *c){  //__global__ is a keyword to execute on device
*c = a+b;
}
int main(void){
        int c;
        int *dev_c;
        cudaMalloc((void**)&dev_c,sizeof(int));

        add<<<1,1>>>(2,7,dev_c);
        cudaMemcpy(&c,dev_c,sizeof(int),cudaMemcpyDeviceToHost);
        printf(“2+7 = %d\n”,c);
        cudaFree(dev_c);
        return 0;
}
```

Figure 1.1 - Sample CUDA C program [1]

1.2 NVIDIA CUDA Architecture

The NVIDIA CUDA parallel architecture provides the tools necessary to achieve supercomputing speeds at a price comparable to an everyday workstation by the use of GPUs. Unlike CPUs, GPUs have a high parallel throughput architecture that emphasizes on executing many concurrent threads slowly, rather than executing a single thread very quickly [1]. GPUs have outpaced CPUs in floating-point performance in recent years and have become as easy, if not easier to program than a supercomputing cluster.

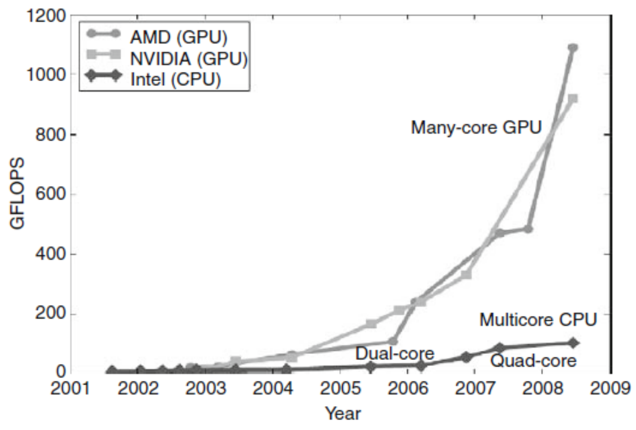


Figure 1.2 – Difference in performance between CPU and GPU [2]

The architecture of a modern GPU is organized into an array of highly threaded streaming multiprocessors in which two SMs form a building block. Each SM has a number of streaming processors that share control logic and instruction cache [1]. For example, each Tesla C1060 graphics card holds up to 4GB of graphics double data rate RAM referred to as global memory. This DDR RAM differs in CPU DRAM in which they are essentially the frame buffer memory that is used for graphics [1]. Different device memories are available such as shared memory, constant memory and texture memory.

Shared memory resides on the physical GPU as opposed to off-chip DRAM which makes for lower latency. Although this is an exploitable asset, shared memory is only 16KB of size. The use of constant memory reduces required memory bandwidth but only 64KB are available. Texture memory has a cache of 8KB, but benefits from having higher bandwidth by reducing memory requests to off-chip DRAM [1]. These memories have to be taken into account when programming exploitable pieces code. Some applications might benefit from using other memories instead of global.

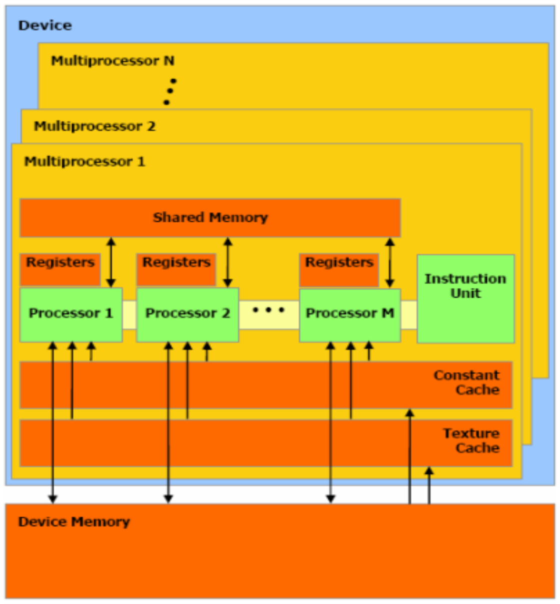


Figure 1.3 – NVIDIA GPU architecture [2]

1.3 MATLAB Executable Files and Parallel Thread Execution Files

MATLAB Executable (MEX) files are dynamically linked subroutines from C, C++ or FORTRAN source code that, when compiled, can run from within MATLAB in the same way as MATLAB files or built-in functions [3]. Although not appropriate for all kinds of applications, MATLAB MEX-files provide a way for developing CUDA kernels in C/C++ and call them from MATLAB accelerating the application. Since we are interested in parallelization, sequential bottlenecks can be avoided by compiling and executing parts of the code in the CUDA environment. MEX-files are compiled with the compiler of choice from the MATLAB MEX setup, for example Microsoft Visual Studio in Windows or gcc in Linux.

All MEX-files in C/C++ must include a MEX header, the mexFunction gateway which is how MATLAB gains access to the DLL, the mxArray which represents a MATLAB array in C, and the provided API functions used to access data inside of mxArrays. They are also used to do memory management, and create and destroy mxArrays [3].

Parallel Thread Execution (PTX) is another alternative for writing CUDA kernel files to be used in the MATLAB environment. By compiling a CUDA kernel file of extension .cu, with the flag –ptx, we can generate a pseudo-assembly language file to be used by MATLAB. The NVIDIA graphics driver contains a compiler which translates PTX into binary code which can be run on the processing cores [11]. We are then able to evaluate the kernel in MATLAB using a built-in function in the Parallel Processing Toolbox to execute the CUDA kernel with the freedom to choose how many threads and blocks to be used in the same way we would do with a regular C/C++ CUDA implementation. This process is simpler to implement and lets us prototype an algorithm faster as only the CUDA kernel needs to be written and we don’t have to worry about the transfer of information from MATLAB to CUDA code because the MATLAB Parallel Computing Toolbox functions do this process for us.

1.4 Template Matching

Template matching in digital image processing is the simple process of matching a template to an image. It is a model-based approach in which the shape is extracted by searching for the best correlation between a known model and the pixels in an image [4]. Template matching algorithms are common in the industry for applications such as motion tracking, quality control, and pattern recognition to name a few.

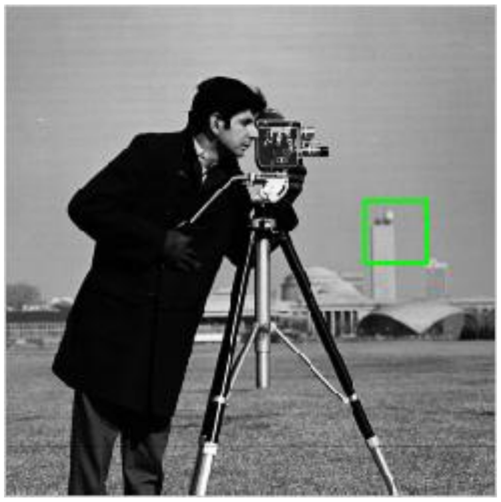


Figure 1.4 - Example of template location

A simple iterative implementation of a template matching algorithm is the use of an accumulator array. This array translates to the best matches in the whole image, giving the location and the correlation between the image and template. The computational cost of a template matching algorithm is rather expensive and increases as more parameters are added; such as rotation, position, scale and invariance. A template size of N x N and an image size m x m has a computational cost of $O(N^2m^2)$ [4].

The Fourier transform implementation of a template matching algorithm is much faster. This approach exploits the fact that a multiplication in the space domain corresponds to a convolution in the frequency domain and vice-versa [4]. The crosscorrelation is used as a multiplication to compute the match.

Another approach to matching a template is the use of a hierarchical representation such as a pyramid. By filtering the image from high resolution to low resolution we decrease the size of the image that we are matching and it becomes more efficient. A threshold level for resolution filtering decides how strong a match will eventually be; a low threshold would lead to too much computational cost and too high will miss the match. The runtime complexity of a pyramid matching algorithm is O(whn); the width, height, and points in a template [5]. Some famous pyramid implementations are Gaussian, Laplacian, and Wavelet.

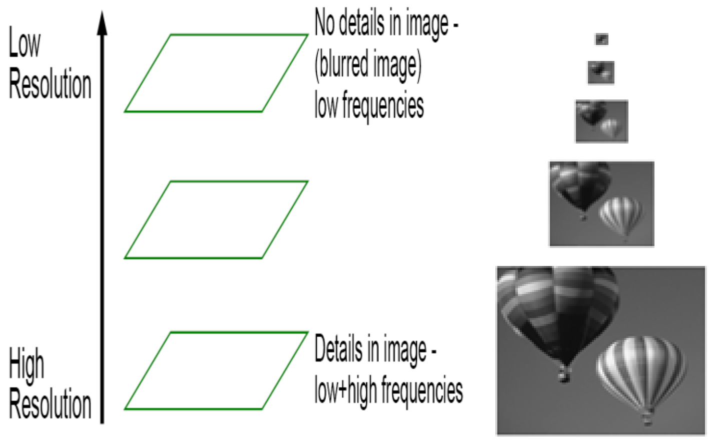

Low Resolution
No details in image - (blurred image) low frequencies
High Resolution
Details in image - low+high frequencies

Figure 1.5 – Image pyramid

CHAPTER 2

ACCELERATING ALGORITHMS USING NVIDIA CUDA

2.1 Optimization Techniques for Template Matching

Template Matching is a popular technique in computer vision for classifying objects. Usually implemented as feature-based or template-based, template matching can be a computationally intensive procedure especially in the spatial domain. Traditionally, to speed up the processes, a mixture of spatial and a feature based approach or the use of an image pyramid can produce significantly better results depending on the application. Since the basic implementation of a template matching algorithm is an iterative process, it can be greatly optimized and sped up by the use of the CUDA GPU architecture for parallel computing. There have been a number of implementations with different approaches in template matching using the CUDA architecture which have mostly produced significant results depending on the size of templates and images.

2.2 Data Parallelism

Data parallelism is the driving force behind GPU computing in the NVIDIA architecture. A software application which requires large amounts of data and has a property whereby many arithmetic operations can be safely performed on the data structures in a simultaneous manner has a high probability to be structured in parallel. For example, matrix multiplications of large dimensions can have a very large amount of data parallelism [5]. Although in a real world application, it is not always as simple as a matrix multiplication; a CUDA device can significantly accelerate the execution over the CPU version.

Multiplying random square matrices of various sizes on the CPU and GPU using CuBLAS demonstrates the overwhelming run time speed up of CUDA parallelism. Using a GPU with 240 cores, speed up for matrices of size N = 4,000 to N = 10,000 increased close to 1,000 than its CPU implementation [6]. Different configurations of threads per block and number of blocks affect the run time of the computation; usually results are experimented on by different configurations to test which is the best to use. Other algorithms such as breadth-first search are also enhanced by parallelism, producing results close to 2,000x faster for a case of 1 million vertices [6]. These results provide simple examples which were greatly benefited by the GPU computing architecture; although not all data parallel structures will result in these overwhelming speeds, it provides a base for all applications and libraries.

2.3 OpenCV and GpuCV

GpuCV is a general purpose computer vision library being developed similar to OpenCV. It is an open source framework for image processing and computer vision meant to help the developer port their existing applications from OpenCV on the CPU to the GPU [7]. These set of libraries provide efficient implementations of simple operators such as adding, multiplying, finding the minimum, and threshold accelerated to speeds 10-20x. Other advanced operators being developed such as erode, sobel filter, DFT-float are accelerated to 44-200x compared to its traditional OpenCV implementation [7]. There is much work to be done with this basic library in which the community can contribute. Adding more accelerated operators and integration into OpenCV applications and image processing libraries are some of the future work going into this project.

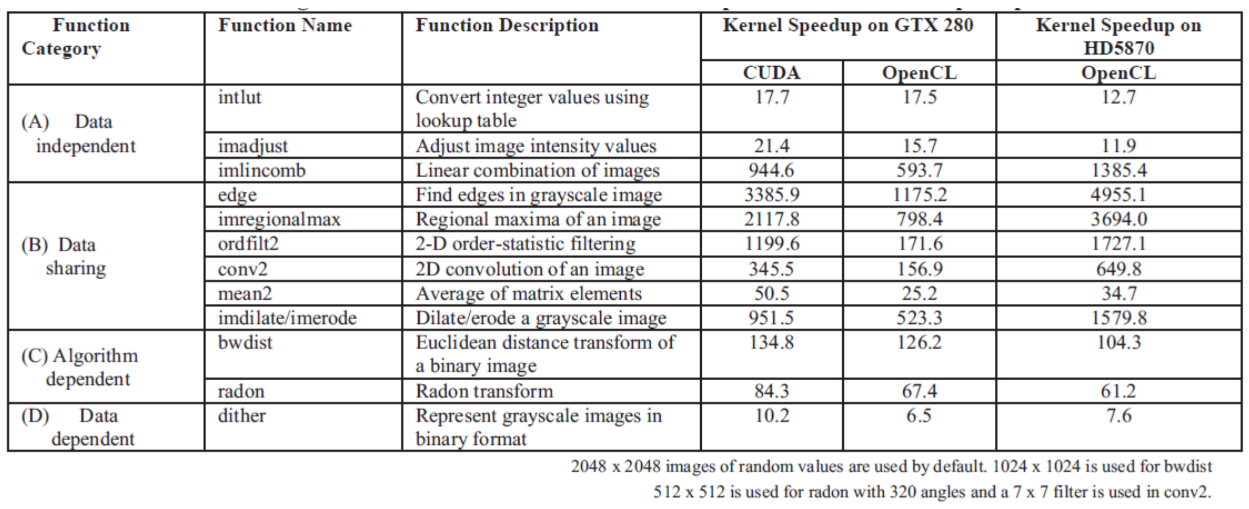
Routines already developed in this library can be used for developing template matching algorithms and new ones could surface from the works.

2.4 MATLAB Integration

The integration of general purpose GPU computing is fairly streamlined with the MATLAB MEX file interface. This provides the user the option of writing their particular application as a CUDA kernel and executing it from MATLAB to seamlessly integrate it into their MATLAB code [3]. Other recent works such as the Jacket commercial product to accelerate MATLAB functions are effective and enables standard MATLAB functions to run on the GPU [8], but the recent GPU Computing Toolbox from MATLAB combined with the MEX file interface provides for a relatively open source alternative in which you only need MATLAB software and the Parallel Computing Toolbox/Image Processing Toolbox to develop parallel computer vision applications in CUDA.

2.5 Accelerated MATLAB Image Processing Functions

MATLAB functions were selected from the Image Processing Toolbox in [8] which are used frequently to perform media processing operations. These functions were accelerated using the MATLAB MEX file interface and achieved significant speed ups using an NVIDIA GTX 280 graphics card.



Function Category	Function Name	Function Description	Kernel Speedup on GTX 280		Kernel Speedup on HD5870
			CUDA	OpenCL	OpenCL
(A) Data independent	intlut	Convert integer values using lookup table	17.7	17.5	12.7
	imadjust	Adjust image intensity values	21.4	15.7	11.9
	imlincomb	Linear combination of images	944.6	593.7	1385.4
(B) Data sharing	edge	Find edges in grayscale image	3385.9	1175.2	4955.1
	imregionalmax	Regional maxima of an image	2117.8	798.4	3694.0
	ordfilt2	2-D order-statistic filtering	1199.6	171.6	1727.1
	conv2	2D convolution of an image	345.5	156.9	649.8
	mean2	Average of matrix elements	50.5	25.2	34.7
	imdilate/imerode	Dilate/erode a grayscale image	951.5	523.3	1579.8
(C) Algorithm dependent	bwdist	Euclidean distance transform of a binary image	134.8	126.2	104.3
	radon	Radon transform	84.3	67.4	61.2
(D) Data dependent	dither	Represent grayscale images in binary format	10.2	6.5	7.6

2048 x 2048 images of random values are used by default. 1024 x 1024 is used for bwdist
512 x 512 is used for radon with 320 angles and a 7 x 7 filter is used in conv2.

Figure 2.1 – Implemented image processing functions and speed ups [8]

Significant performance increase in comparison to a CPU implementation is established in this experiment. Functions were broken into categories based on communication between its elements; whether it needed information from a neighborhood, have inherent parallelism or if it is very difficult to achieve parallelism. Functions which are data-dependent and data-sharing have the greatest performance boost and provide an insight into which kinds of functions can be parallelized in the future.

2.6 Parallelized Template Matching Algorithms

Recently, a template matching algorithm for iris matching was implemented on GPUs which achieves rates of 44 million iris template matching comparisons per seconds and 4.2 million rotations –invariant templates [9]. This GPU implementation benchmarks in 10-20x speed up comparable to its CPU counterparts. The basic idea behind parallelizing this template matching lies in dividing the labor; each instance of kernel code calculates a single pair wise fractional Hamming distance between a probe template and a gallery template determined by a unique thread ID. The resulting distance is stored in a matrix in global device memory; a listing contains an inline function used to compute a fractional Hamming distance; then a CUDA intrinsic function implements a population count routine. A series of device memory optimizations contribute to speeding up the algorithm, such as using shared memory and texture memory. This experiment provides an insight into other possible results in the context of matching algorithms.

Variable template size for efficient template matching on the CUDA architecture is proposed in [10]. Past applications were smaller templates were applied were not accepted as subjects for acceleration in applications. This paper applies techniques to a correlation function; which has several nested routines in it. This function is broken into steps to improve adaptability without sacrificing performance by combining runtime compilation of kernels and separate kernel launches to introduce adaptability. A main tile size is selected and mapped across the template size associated with the given data set. Then as needed, separate kernels are compiled to handle the remaining tiles [10]. This approach provides a wide range of input parameters not proposed before in GPU implementations with performance speed ups of around 10-20x over a previous GPU implementation.

Experimenting with template matching algorithms with the CUDA architecture offers an expected minimum amount of 10-20x speed up for a regular benchmark dataset without using much of the optimization options available in the CUDA architecture such as streaming, thread block configuration, different memory managements, or using several GPU devices at a time. Most time consuming and iterative algorithms are perfect examples of parallelization using GPUs.

CHAPTER 3

RESEARCH METHODOLOGY

3.1 Template Matching Using GPUs for Still Images

Implementing a basic matching technique will establish a benchmark for which to compare the runtimes of the algorithm and decide which one case produces the best results. The MATLAB timer function will serve as a measure of GPU running time together with the regular CPU function for an overall execution time. A basic implementation of a template match, such as the use of an accumulator array, can be accelerated by the inherent nature of the iterative process. Since the search is applied throughout an entire image in sections, these sections can be assigned to be parallelized to execute concurrently, then the results are concatenated to compute the location of the highest match. Parallelizing for-loops for example will be essential as well as the opportunity to use a full resolution image rather than a scaled down or down sampled version.

Similarly to the basic implementation, pyramid matching algorithms using the CUDA architecture will benefit from parallelizing any iterative processes, except that the number of levels in the pyramid is introduced as a new variable. The process to parallelize pyramid matching algorithms is similar to the basic accumulator array implementation in which certain bottleneck parts of the code can be optimized.

The hybrid approach of using the Fourier Transform in conjunction with the spatial implementation is already an accurate and very efficient method of computing matches, but as the other methods, the computational time increases as the size of the image becomes larger. Though converting the current area being correlated to the frequency domain, it is still a spatial time algorithm in which it still has to look through the image iteratively.

As the implementation of the pyramid and hybrid approach are similar to the basic accumulator array implementation for optimizing certain critical areas of code, the basic implementation will serve as a base if the pyramid and hybrid approaches are to be implemented in future works.

3.2 Factors and Parameters for GPU Implementation

Several underlying factors will affect the performance of a template matching algorithm. For example, the size of the image and the template being searched will affect the efficiency measured in terms of execution time and storage requirements. Templates of sizes 32x32 to 256x256 will be tested to benchmark runtimes. The number of GPU threads will depend on the size of the image and complexity of the template matching method. Typically, the more threads used in a GPU application, the faster the performance will be.

A simple known method of optimizing C/C++ code is the use of one-dimensional array accesses instead of typical two-dimensional array of handling of images. By flattening the array, we can benefit from an improvement of speed rather than twice the memory accesses of two-dimensional arrays to execute computations. The NVIDIA CUDA architecture is optimized to execute pieces of code in this manner.

Color-space parameters can be investigated in the future as they affect performance. Using an RGB image will be 3x more time consuming given that each layer of color will be computed. Converting to a gray level resolution will decrease computational time. Other color-spaces such as HSV and HSL can be tested to measure performance.

The scalability of the NVIDIA GPU architecture should not be a problem if more than one GPU device is used to assign threads to a different memory device. Current Tesla C1060 graphics cards should be enough to support up to 4GB of memory transferred from the host to the device.

Implementing multiple matches within an image is an interesting research problem for future implementations. The current hardware of the Tesla C1060 does not support concurrent kernels being executed, for example a threaded template matching algorithm cannot be executed at the same time as another, and therefore the user will have to execute the algorithm after the first match is finished. The new Fermi architecture simplifies these types of complications that may affect the implementation of the algorithm.

3.3 Algorithmic Detail

The following pseudo-code describes the process by which a basic template matching routine was implemented in the CPU and GPU.

Inputs: full image *F*, template image *T*

Outputs: Minimum error *accum*, row match *r*, column match *c*, speedup *s*

```
1.  Read image $F$, $T$
2.  Convert $F$, $T$ from RGB to gray scale
3.  Find row and column size of $F$, $T$.
4.  Calculate center of image by $cx = floor(rows/2)$,  $cy = floor(columns/2)$
5. Initialize with zeros $accum$
6. Begin template matching
```

```
For $i=cx$ to $columns-cx$, $j=cy$ to $rows-y$, $x=1-cx$ to $cx-1$, $y=1-cy$ to $cy-1$
    $error = ( F( j+y, i+x ) - T( y+cy, x+cx ) )^2$
        $accum( j, i ) = accum( j , i ) + error$
End of template matching.
```

7. Find minimum error values from *accum* to locate *r*, *c*
8. Calculate *s*

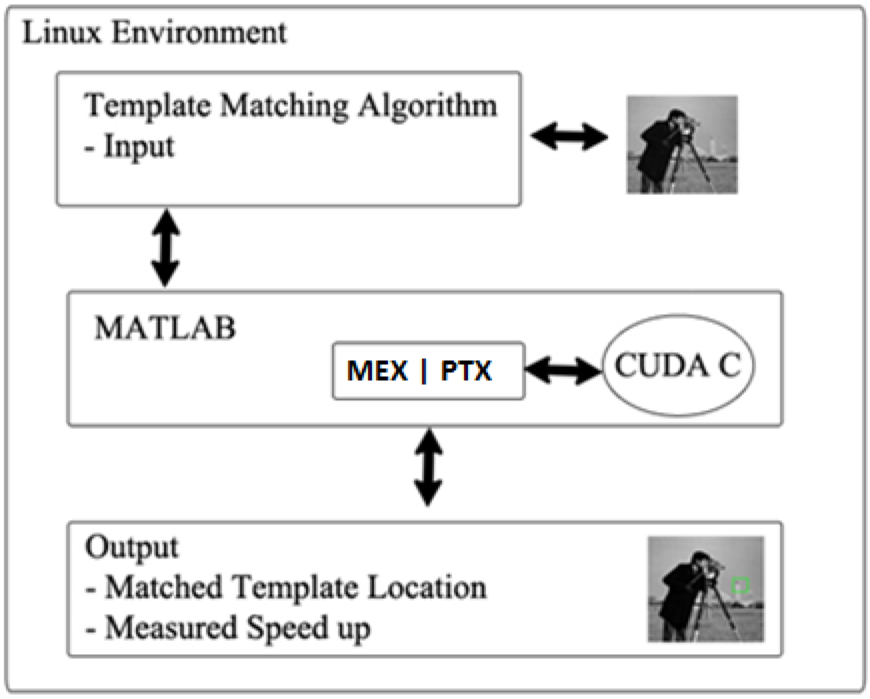
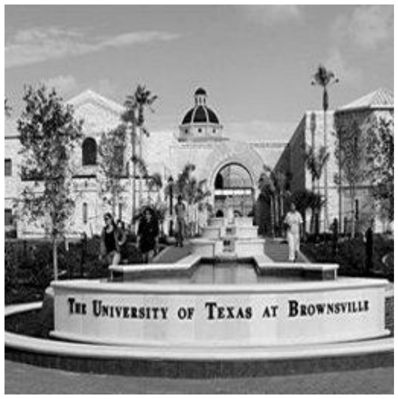


Figure 3.1 – Implementation

CHAPTER 4

EXPERIMENTAL DESIGN

A set of templates of sizes 32x32, 64x64, and 128x128 were created from sample image of size 256x256 and another sample image of 512x512. These images were converted to gray level resolution which provides a traditional setup for a dataset in Image Processing and Computer Vision for which to test algorithm performance. Using common MATLAB Image Processing Toolbox functions to handle images we can focus on implementing the iterative template matching algorithm in both a CPU version written in MATLAB and a GPU version of the algorithm written in CUDA. For this experimental case, using the CUDA PTX alternative to MEX files was the best option for a quick and intuitive implementation.



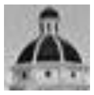


Figure 4.1 – Sample image and template used

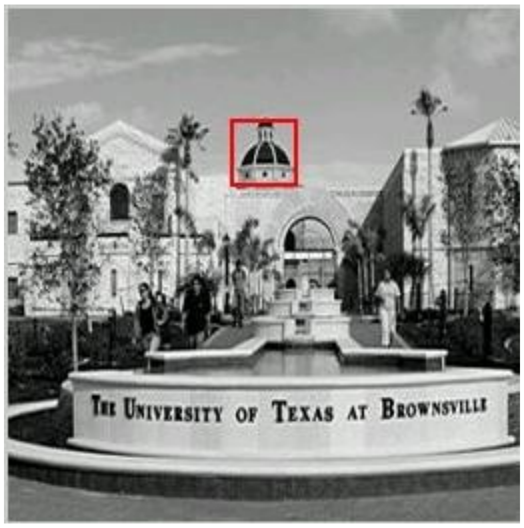


Figure 4.2 – Result of matching

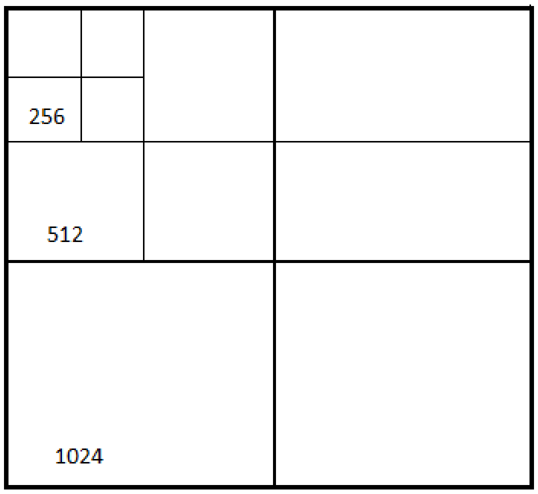


Figure 4.3 – Illustrated image size comparison

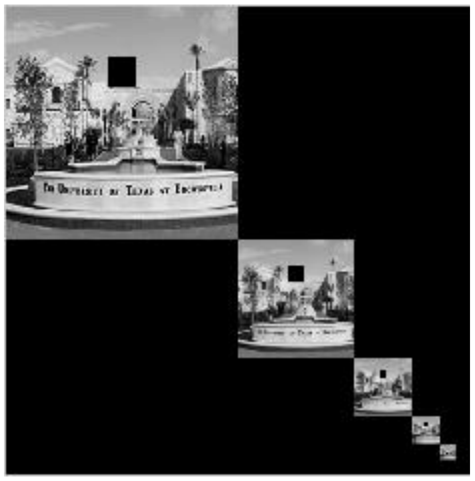


Figure 4.4 – Image size comparison in down sampling

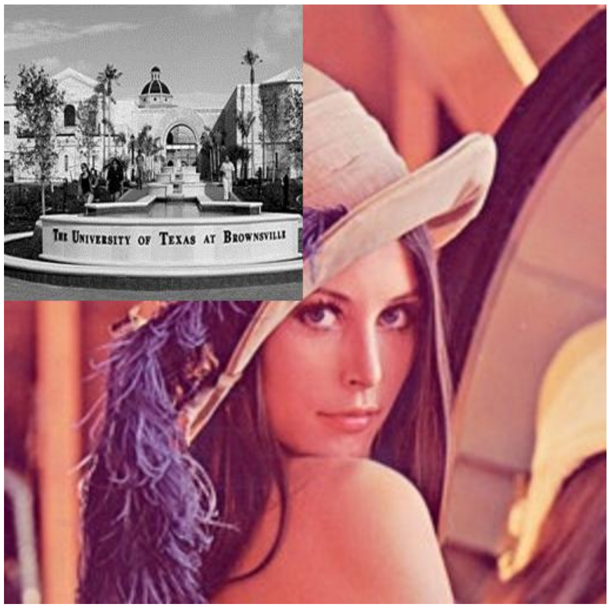


Figure 4.5 – Difference in size of images used

For the CPU version, after reading the input image and a template image, we use the size function to find the size of the image we are working on. This image size variable is used to find the center to begin iterating over the rows and columns of the input image accumulating variables in error. We subtract the input image by the template image at the current location we are iterating over and square the result to disregard any negative numbers. The resulting minimum error throughout the accumulator array gives us the location of the best match.

For the GPU version, we follow the same algorithm with some notable changes and small performance boosters. After reading our image, we then load it directly into the GPU device using the parallel.gpu.GPUArray function. This shortcut skips the step of loading the image variables into MATLAB and then to the GPU device during execution. After initializing our CUDA kernel variable using the parallel.gpu.CUDAKernel function, we can then adjust the parameters for number of CUDA threads and blocks. There is a limit however of how many threads in a block for dimensions X Y Z, which is 512 threads. We can modify our grid configuration which constitutes of a number of blocks to overcome this limitation. A configuration of 128x128 blocks and 16x16x1 threads per block was a comfortable window to be able to compute all the elements in a 256x256 image with variable template sizes being tested. This GPU version of the iterative template matching algorithm benefits from a CUDA implementation in which the image and template are handled in a one-dimensional array format. This lowers the amount of memory accesses the GPU device has to do in global memory optimizing the implementation compared to two-dimensional arrays.

Since we are working with one-dimensional indexes, a common CUDA formula for this configuration is index = threadId.x + blockId.x * blockDim.x. ThreadIdx.x being the distinct thread index, blockIdx.x the block index, and blockDim.x the block dimension, all initialized by the device architecture at runtime. Block dimension in this case is the size of the image we are working on. This configuration is available on a three-dimensional level, x y z, and can be used to expand the indexing after having flattened a two-dimensional array to be able to compute every element. Finally, we can measure the speed up dividing the CPU time by GPU time execution.

4.1 Experimental Results

Our results show a drastic increase in measured speed-up using the CUDA architecture to evaluate our basic pixel by pixel template matching algorithm. In all cases the accuracy of the GPU template matched the CPU implementation 100%. Table 1 and Table 2 shows the algorithm run times in seconds for the tested images and the measured speed-up for comparison. Table 1 used a University of Texas image of size 256x256; Table 2 used a traditional picture used in Image Processing of Lena for a size of 256x256. It is noticeable form Table 3 that an upscale image of Lena to size 512x512 was used to experiment other image sizes. An extreme case of an image size of 2048x2048 was tested to stress the algorithm performance.

Image Size	Template Size	CPU Time (seconds)	GPU Time (seconds)	Speed-up
256x256	32x32	2.0399	0.0142	144x
256x256	64x64	6.0994	0.0539	113x
256x256	128x128	10.9227	0.1441	76x
2048x2048	256x256	19080.6937	54.6263	349x

Table 1 – Algorithm run times for the image of The University of Texas at Brownsville of size 256x256, and stress test

Image Size	Template Size	CPU Time (seconds)	GPU Time (seconds)	Speed-up
256x256	32x32	2.0544	0.0175	117x
256x256	64x64	7.0281	0.0547	128x
256x256	128x128	11.1977	0.1449	77x

Table 2 – Algorithm run times for the image of Lena of size 256x256

Image Size	Template Size	CPU Time (seconds)	GPU Time (seconds)	Speed-up
512x512	32x32	9.2520	0.0532	174x
512x512	64x64	32.3670	0.2005	161x
512x512	128x128	96.6256	0.7511	129x

Table 3 – Algorithm run times for image Lena of size 512x512

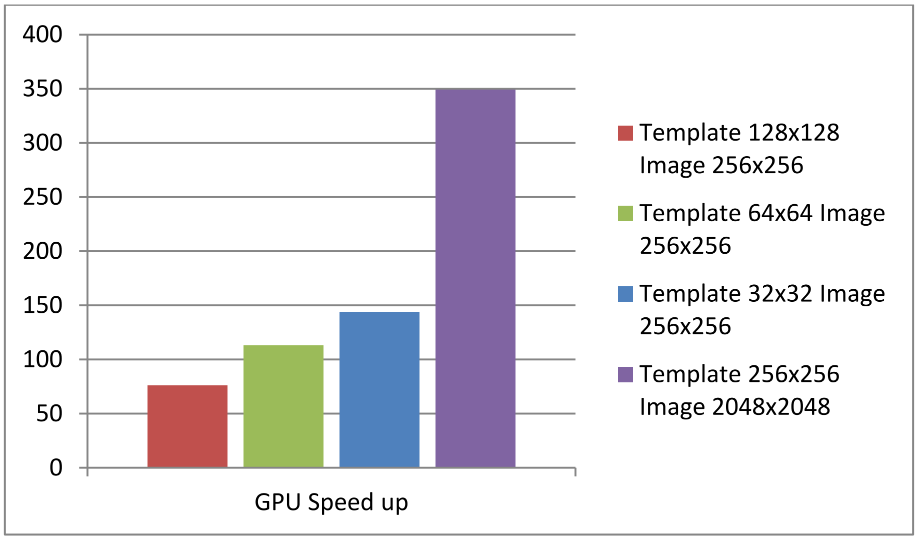


Figure 4.6 – Measured speed-up for image of University of Texas at Brownsville for size 256x256 and stress test

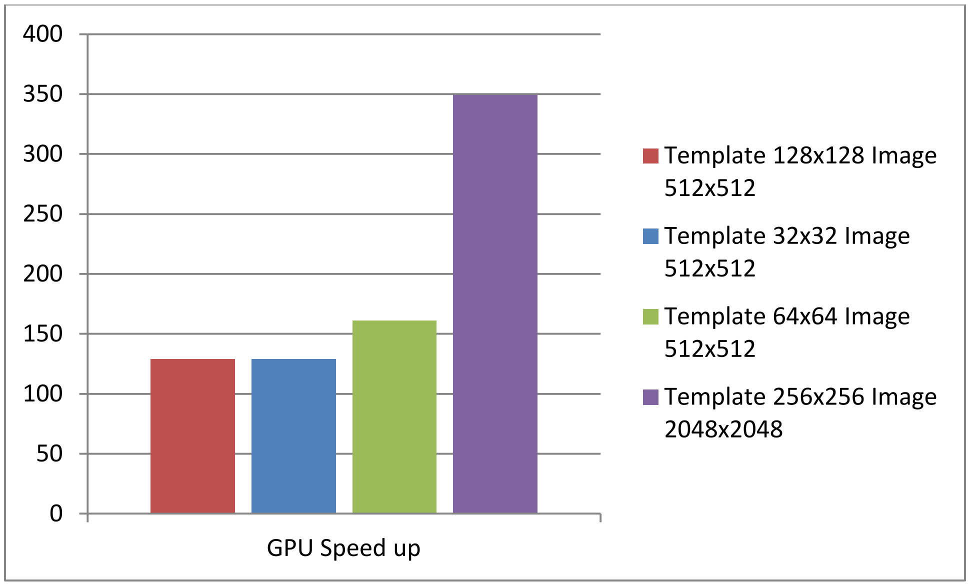


Figure 4.7 – Measured speed-up for image of Lena of size 512x512, and stress test for comparison

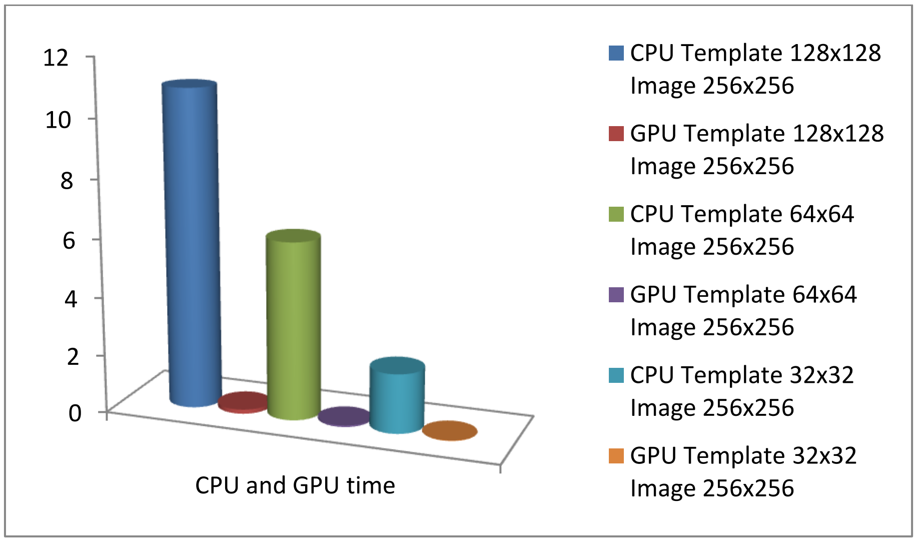


Figure 4.8 – CPU and GPU runtime in seconds for image of the University of Texas at Brownsville of size 256x256

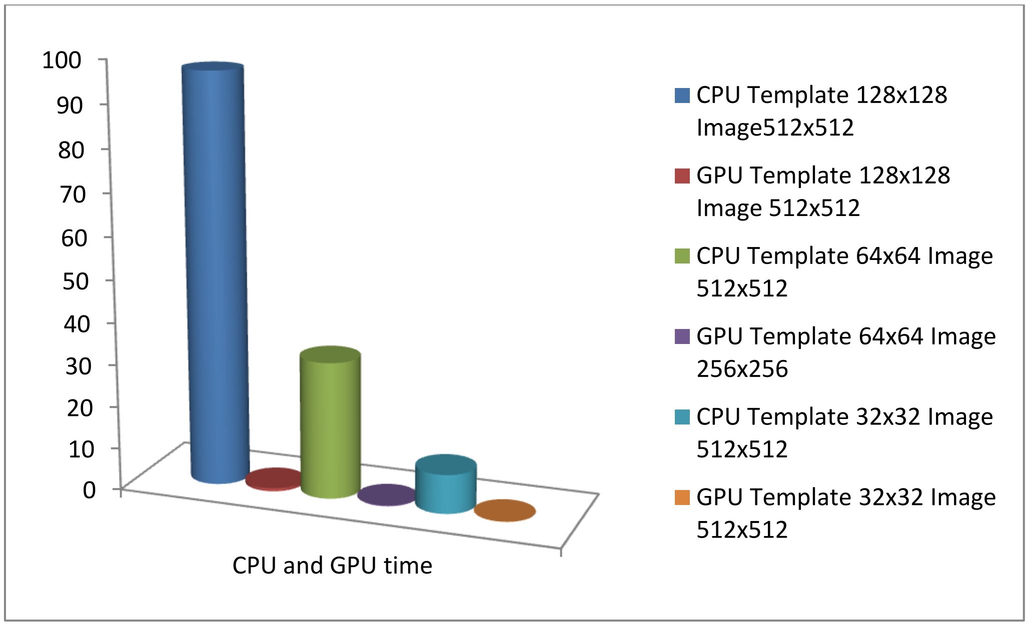


Figure 4.9 – CPU and GPU run times in seconds for the image of Lena of size 512x512

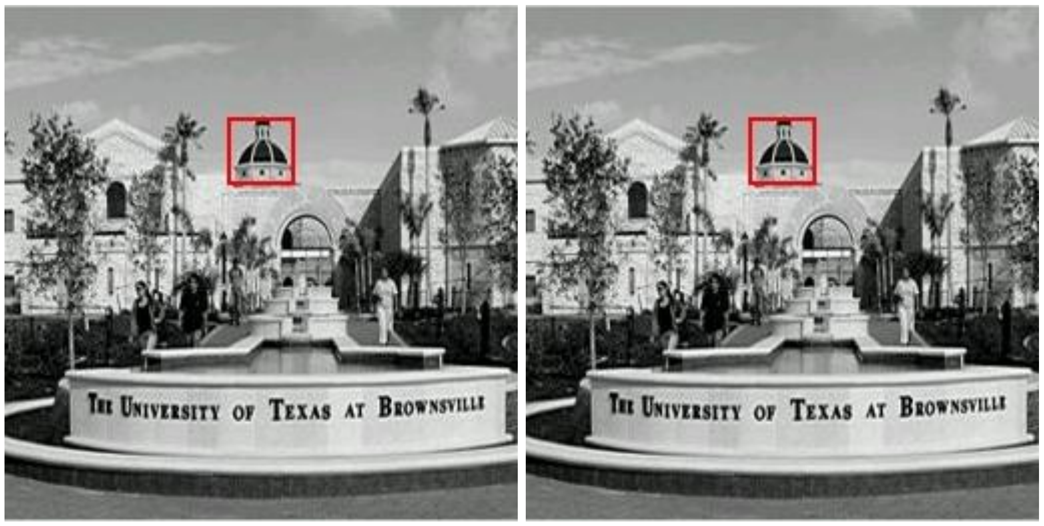

The University of Texas at Brownsville
The University of Texas at Brownsville

Figure 4.10 – Matched Images for CPU and GPU, 256x256 image size, template size 32x32

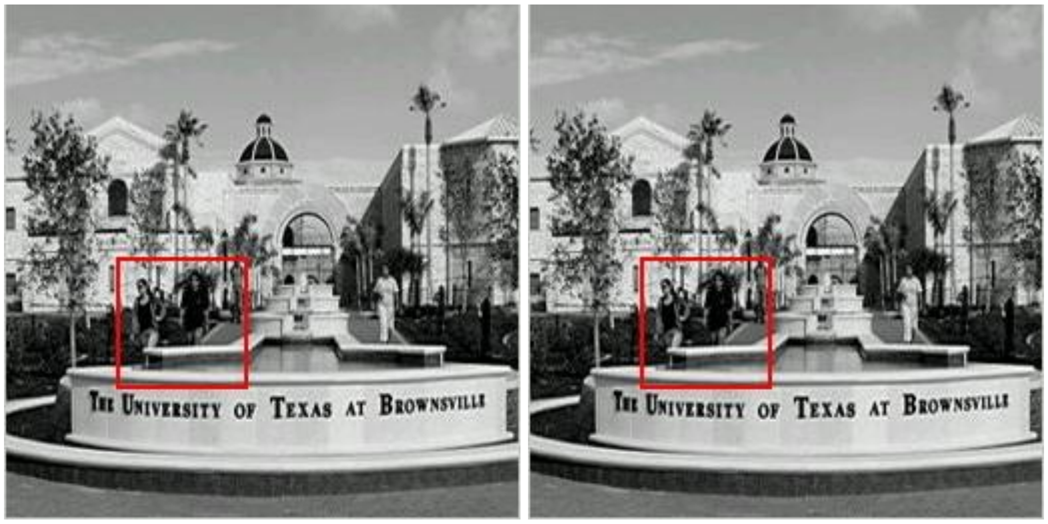


Figure 4.11 – Matched Images for CPU and GPU, image size 256x256, template size 64x64

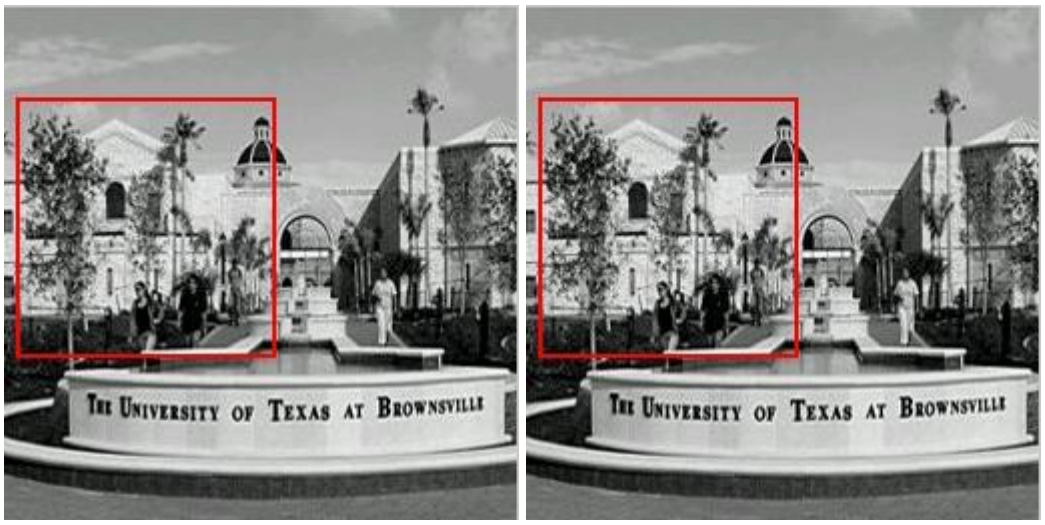


Figure 4.12 – Matched Images for CPU and GPU, image size 256x256, template size 128x128

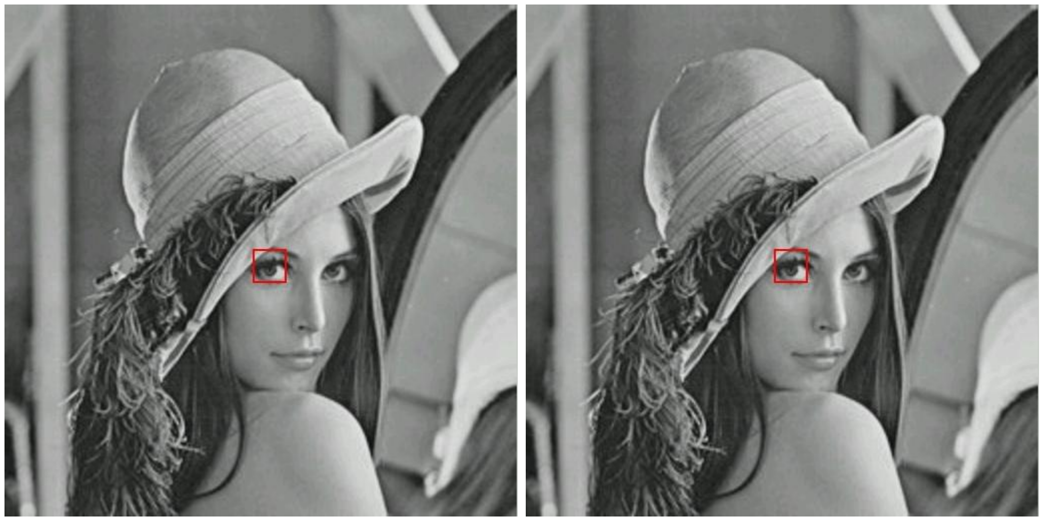


Figure 4.13 - Matched Images for CPU and GPU, image size 512x512, template size 32x32

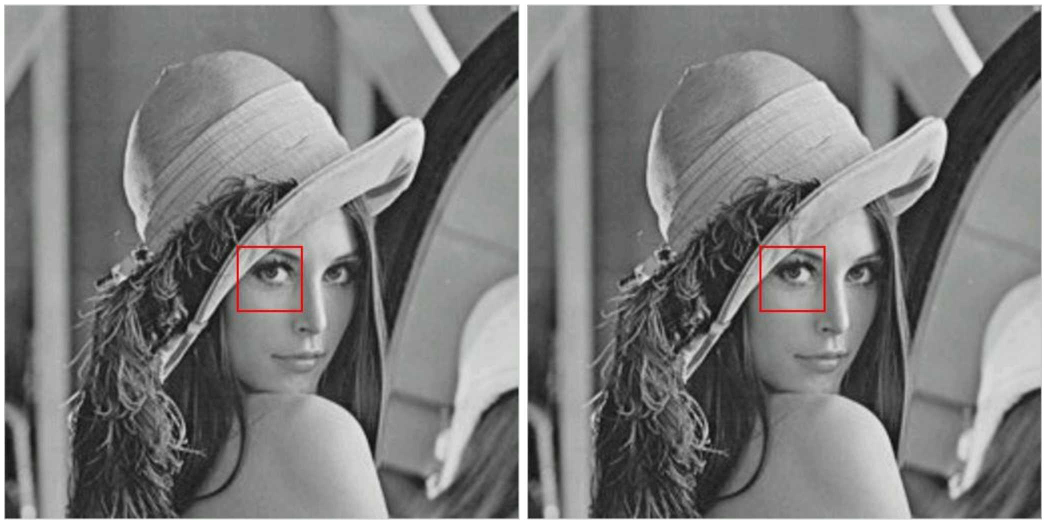


Figure 4.14 - Matched Images for CPU and GPU, image size 512x512, template size 64x64

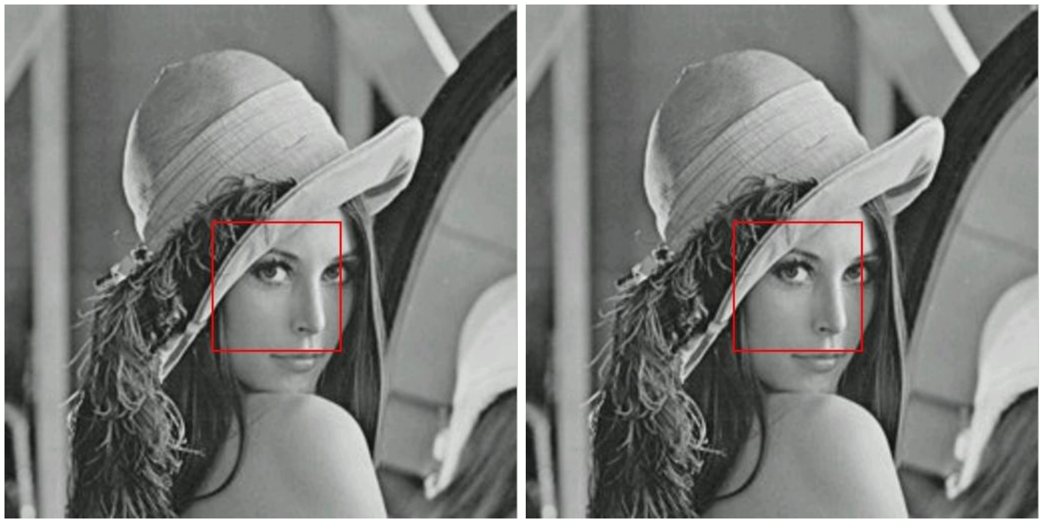


Figure 4.15 - Matched Images for CPU and GPU, image size 512x512, template size 128x128

4.1.1 Motion Tracking Using Template Matching

A motion tracking sample application implemented on the GPU using a sequence of images was developed using the same template matching algorithm to test the algorithm accuracy. Sequences of images used were 512x512 and a template size of 128x128 of gray scale origin. Table 4 shows similar results to a previous test using the Lena image of same size and template size providing an example of the stability of the algorithm achieving an approximate speed-up of 128x for each frame. A template size of 128x128 was chosen because it provides the most resolution to be able to find a correct match. A lower resolution provides less detail of the object and causes a higher rate of error and template mismatch. The vehicle in the sequence was successfully detected in each frame with the exception of the last frame where the vehicle rotated to a point where it did not match the template at a minimal error.

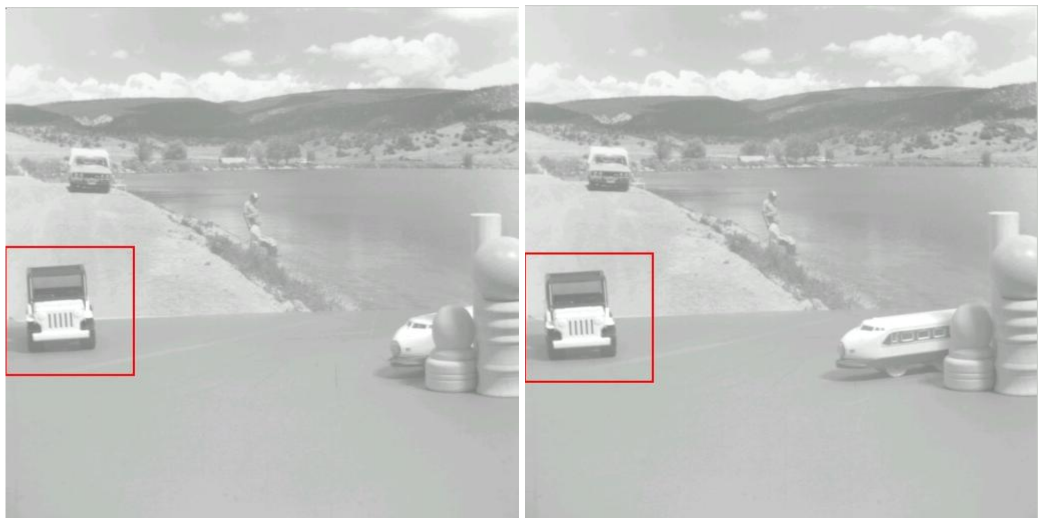


Figure 4.16 – Frame 1 and frame 2 detection

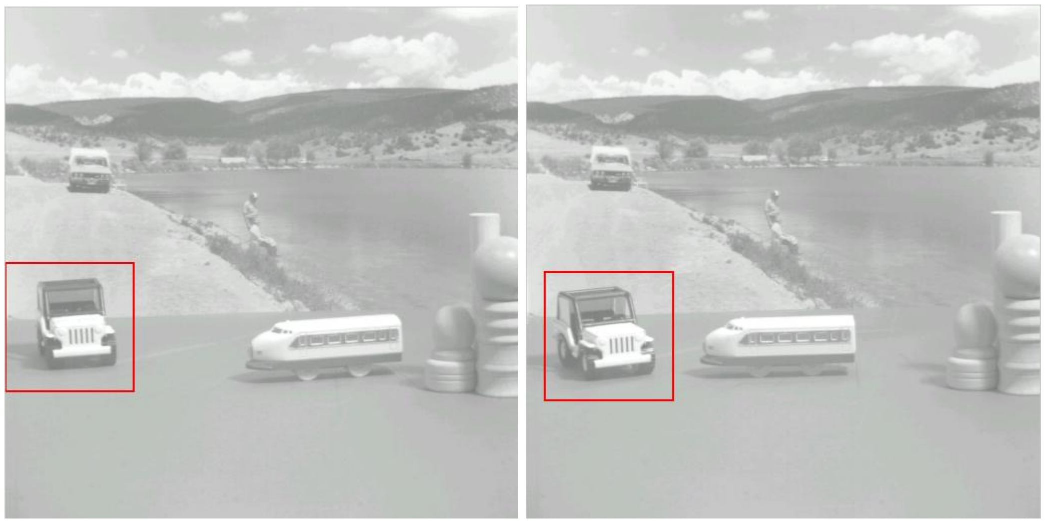


Figure 4.17 – Frame 3 and 4 detection

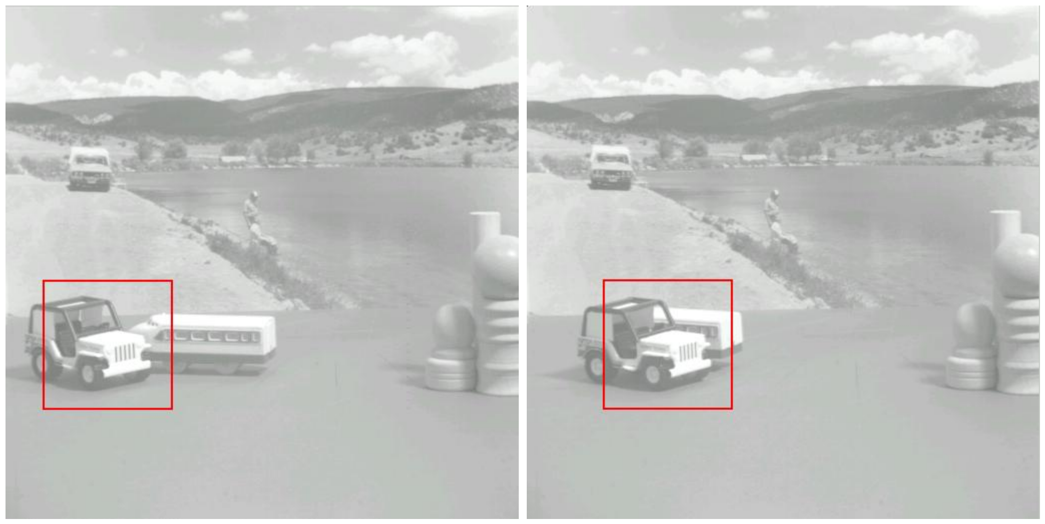


Figure 4.18 – Frame 5 and 6 detection

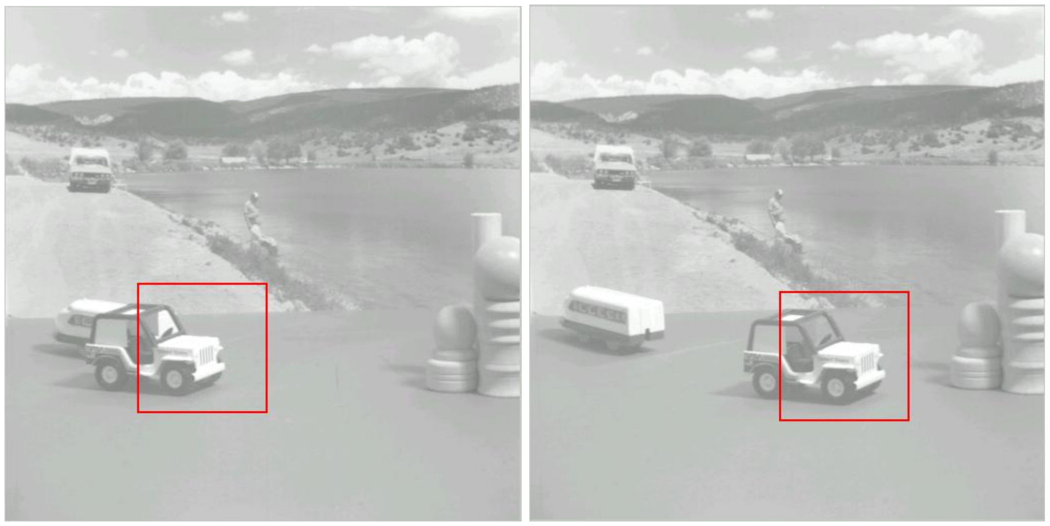


Figure 4.19 – Frame 7 and 8 detection

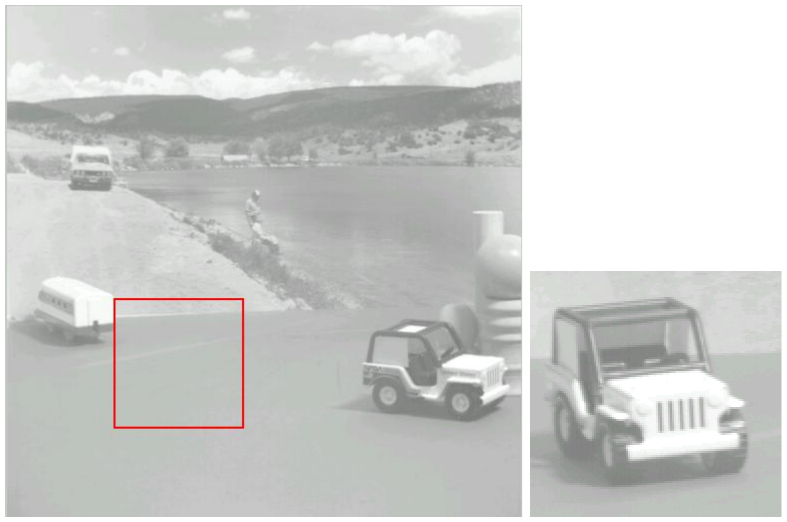


Figure 4.20 – Frame 9 detection and template used

Frame	Match	CPU Time (seconds)	GPU Time (seconds)	Speed-up
01	yes	96.1726	0.7474	128x
02	yes	96.0051	0.7497	128x
03	yes	95.7590	0.7491	128x
04	yes	96.0130	0.7475	128x
05	yes	96.1586	0.7503	128x
06	yes	95.743	0.7475	128x
07	yes	95.6928	0.7490	128x
08	yes	95.8390	0.7503	128x
09	no	95.8686	0.7503	128x

Table 4 – Motion tracking results

4.2 Discussion of Experimental Results

The results obtained from the experiment using a basic non-efficient template matching algorithm exceeded initial expectations of 10-20x GPU speed-up showing improvement of approximately 75-350x speed-ups for the small selection of test datasets used. It is clear from the results observable in figures 4.1.1 through 4.1.10 that although small image and template size images benefit from an optimization, the real performance difference is notable when large images are computed because the GPU processing cost is very low compared to a CPU.

A CPU core is completely overshadowed by a group of 128x128 blocks each executing 16x16 threads. This CUDA kernel configuration executed 256 threads per block, there being 128 blocks, a total of 32,768 threads were executed for each dimension to calculate every pixel by pixel computation with 100% accuracy.

4.2.1 Effect of Image Size and Template Size

Starting with an image NxN, multiple templates and images can be created by factors of NxN/2, NxN/4, NxN/8, NxN/16 and so on and so forth. For example an image 2048x2048 can have smaller templates of sizes 1024x1024, 512x512, 256x256, 128x128, etc. This was the method we chose to create our templates to provide a simple system of standardized images. The higher the resolution of the images in size, the more computationally intensive it is for matching algorithms. Smaller images have fewer elements to compute and therefore have quicker runtimes.

From our experiments, we see that the CPU processing power is not enough to compute large size images and produce real-time results as the GPU can. For the cases of images 512x512, the algorithm runtime was still less than one second while the CPU version finished executing in more than 10 seconds for a template size of 32x32.

4.2.2 Dealing with Color Images

Manipulating color images is very simple in the MATLAB environment. By using the rgb2gray function we can flatten a three channel RGB image to a flattened image of values 0-255. If all RGB channels were to be taken into account in a template matching algorithm, it would take 3x more to execute because of each color channel that needs to be computed. For our experiment, we didn't care for the color detail as the purpose of the experiment was to quantify algorithm speed-up using the CUDA architecture.

4.2.3 Effect of Input/Output (I/O)

The speed of the transfer of information from the host to the device and returning the result to the host will depend on the hardware of the GPU card. For example, the

Tesla C1060 GPU card has a memory bandwidth of 102GB/sec. while a Quadro FX 4600 has a memory bandwidth of 67.2 GB/sec. Reproducing the experiment with different hardware might produce slightly different results as the Tesla C1060 GPU card is built for more high speed general purpose computations than other cards that might be available in the market at the time. Memory size restrictions and number of CUDA cores available in the hardware will influence the processing time and other threads-per-block configurations if it is the case. Given in the table of results from different sized images, we notice the implicit formula for speed is $O(N^2m^2)$. However, variations in the execution time can be attributed to the slow operation of I/O.

4.2.4 Other Factors

The decision to choose what image to use for experimental purposes is important because the detail in an image can affect computational cost. The images of the University of Texas at Brownsville fountain and the image sequences for motion tracking have enough detail for comparison. Features such as vehicles, trees, buildings, and people are important to test because it provides a better environment to test if the algorithm is to be used in a real-time application such as motion tracking for example. The picture of Lena provides subtle details to compare such as eyes, face, and clothing which may appear similar in certain locations of the image.

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

In conclusion, the results have shown that implementing a non-efficient pixel by pixel iterative template matching algorithm in the CUDA GPU architecture can produce good results that may be applied to any Image Processing application. Our thesis highlighted the major issues and factors that a researcher encounters when moving from a traditional CPU based architecture into a GPU one. This work provides a base for implementing other similar template matching techniques such as the hybrid and hierarchical approaches in the future. The framework for developing these algorithms is very similar in which certain critical parts of code can be parallelized and optimized using the same techniques. Applications in Image Processing and Computer Vision can benefit greatly from this increase in performance to provide real-time results to the user and expand to a higher level of resolution in images for processing, otherwise a problem too complex to solve using just CPU processing. Similar results can be obtained for audio and other multimedia formats. Real-time applications such as motion-tracking, facial recognition, robotics, and other areas from engineering to medical can benefit from the performance GPU Computing brings.

REFERENCES

[1] Sanders, Jason, and Edward Kandrot. CUDA by example: An introduction to general-purpose GPU programming. New York: Addison-Wesley, 2011.

[2] Kirk, David, and Wen Hwu. Programming massively parallel Processors. San Francisco, CA: Elsevier:, 2010.

[3] MATLAB MEX files guide, http://www.mathworks.com/support/tech-notes/1600/1605.html,2009.

[4] Aguado, Alberto S , and Mark S Nixon. Feature extraction and image processing, second edition. 2nd ed. Toronto.: Academic Press/Elsevier Ltd., 2008.

[5] Steger, Carsten, and Markus Ulrich. Machine vision algorithms and applications. Weinheim: Wiley-VCH, 2008.

[6] Erik Wynters. 2011. Parallel processing on NVIDIA graphics processing units using CUDA. J. Comput. Small Coll. 26, 3 (January 2011), 58-66.

[7] Yannick Allusse, Patrick Horain, Ankit Agarwal, and Cindula Saipriyadarshan. 2008. GpuCV: an opensource GPU-accelerated framework for image processing and computer vision. In Proceeding of the 16th ACM international conference on Multimedia (MM '08). ACM, New York, NY, USA, 1089-1092.

[8] Jingfei Kong, Martin Dimitrov, Yi Yang, Janaka Liyanage, Lin Cao, Jacob Staples, Mike Mantor, and Huiyang Zhou. 2010. Accelerating MATLAB Image Processing Toolbox functions on GPUs. In Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU '10). ACM, New York, NY, USA, 75-85.

[9] Vandal, N.A.; Savvides, M.; , "CUDA accelerated iris template matching on Graphics Processing Units (GPUs)," Biometrics: Theory Applications and Systems (BTAS),2010 Fourth IEEE International Conference on , vol., no., pp.1-7, 27-29 Sept. 2010.

[10] Moore, N.; Leeser, M.; Smith King, L.; , "Efficient template matching with variable size templates in CUDA," Application Specific Processors (SASP), 2010 IEEE 8th Symposium, vol., no., pp.77-80, 13-14 June 2010.

[11] MATLAB R2012a Documentation; Parallel Computing Toolbox, http://www.mathworks.com/help/toolbox/distcomp/bslohnr-1.html

[12] Gonzalez, Rafael C., Richard E. Woods, and Steven L. Eddins. Digital Image processing using MATLAB. Toronto: Pearson Prentice Hall, 2004.

[13] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. 2008. Scalable Parallel Programming with CUDA. Queue 6, 2 (March 2008), 40-53.

[14] Yuancheng Luo; Duraiswami, R.; , "Canny edge detection on NVIDIA CUDA," Computer Vision and Pattern Recognition Workshops, 2008. CVPRW '08. IEEE Computer Society Conference on , vol., no., pp.1-8, 23-28 June 2008.

[15] Seung In Park; Ponce, S.P.; Jing Huang; Yong Cao; Quek, F.; , "Low-cost, high-speed computer vision using NVIDIA's CUDA architecture," Applied Imagery Pattern Recognition Workshop, 2008. AIPR '08. 37th IEEE , vol., no., pp.1-7, 15-17 Oct. 2008.

[16] Fung, J.; Mann, S.; , "Using graphics devices in reverse: GPU-based Image Processing and Computer Vision," Multimedia and Expo, 2008 IEEE International Conference on , vol., no., pp.9-12, June 23 2008-April 26 2008.

[17] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. 2008. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming (PPoPP '08). ACM, New York, NY, USA, 73-82.

[18] Huang, Jing; Ponce, Sean P.; Park, Seung In; Yong Cao,; Quek, Francis; , "GPU-accelerated computation for robust motion tracking using the CUDA framework," Visual Information Engineering, 2008. VIE 2008. 5th International Conference on , vol., no., pp.437-442, July 29 2008-Aug. 1 2008.

[19] In Kyu Park; Singhal, N.; Man Hee Lee; Sungdae Cho; Kim, C.W.; , "Design and Evaluation of Image Processing Algorithms on GPUs," Parallel and Distributed Systems, IEEE Transactions on , vol.22, no.1, pp.91-104, Jan. 2011.

[20] Zhiyi Yang; Yating Zhu; Yong Pu; , "Parallel Image Processing Based on CUDA," Computer Science and Software Engineering, 2008 International Conference on, vol.3, no., pp.198-201, 12-14 Dec. 2008.

[21] NVIDIA CUDA Zone, http://developer.nvidia.com/category/zone/cuda-zone, 2011.

[22] NVIDIA CUDA Programming Guide 4.0; http://developer.download.nvidia.com/compute/cuda/4_0/toolkit/docs/CUDA_C_Programming_Guide.pdf, 2011.

[23] General Purpose GPU Programming (GPGPU) Website, http://www.gpgpu.org,2011.

[24] GPU Computing.net Website, http://gpucomputing.net/, 2011.

[25] NVIDIA CUDA C Best Practices Guide, http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Best_Practices_Guide.pdf, 2011.

[26] OpenCL Programming Guide for the CUDA Architecture, http://developer.download.nvidia.com/compute/DevZone/docs/html/OpenCL/doc/OpenCL_Programming_Guide.pdf, 2011.

[27] CUDA Toolkit 4.0 Thrust Quick Start Guide, http://developer.download.nvidia.com/compute/DevZone/docs/html/CUDALibraries/doc/Thrust_Quick_Start_Guide.pdf, 2011.

[28] NVIDIA CUDA C Getting Started Guide for Windows, http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Getting_Started_Windows.pdf, 2011.

[29] CUDA Global Memory Usage & Strategy, http://developer.download.nvidia.com/CUDA/training/cuda_webinars_GlobalMemory.pdf, 2011.

[30] NVIDIA Advanced CUDA Webinar Part 1, Memory Optimizations, http://developer.download.nvidia.com/CUDA/training/NVIDIA_GPU_Computing_Webinars_CUDA_Memory_Optimization.pdf, 2010.

[31] NVIDIA Advanced CUDA Webinar Part 2, Optimizing CUDA, http://developer.download.nvidia.com/CUDA/training/NVIDIA_GPU_Computing_Webinars_Further_CUDA_Optimization.pdf, 2010.

APPENDIXES

Appendix A

The following program shows the basic operation of converting a CPU based function into a GPU based as given by [1].

```
#define N 10
__global__ void add( int *a, int *b, int *c ) {
int tid = threadIdx.x;
if (tid < N)
c[tid] = a[tid] + b[tid];
}

int main( void ) {
int a[N], b[N], c[N];
int *dev_a, *dev_b, *dev_c;
// allocate the memory on the GPU
cudaMalloc( (void**)&dev_a, N * sizeof(int) );
cudaMalloc( (void**)&dev_b, N * sizeof(int) );
cudaMalloc( (void**)&dev_c, N * sizeof(int)  );
// fill the arrays 'a' and 'b' on the CPU
for (int i=0; i<N; i++) {
a[i] = i;
b[i] = i * i;
}
// copy the arrays 'a' and 'b' to the GPU
cudaMemcpy( dev_a,a,N * sizeof(int),cudaMemcpyHostToDevice  );
cudaMemcpy( dev_b,b,N * sizeof(int),cudaMemcpyHostToDevice  );

add<<<1,N>>>( dev_a, dev_b, dev_c );

// copy the array 'c' back from the GPU to the CPU
cudaMemcpy( c,dev_c,N * sizeof(int),cudaMemcpyDeviceToHost );
// display the results
for (int i=0; i<N; i++) {
printf( "%d + %d = %d\n", a[i], b[i], c[i] );
}
// free the memory allocated on the GPU
cudaFree( dev_a );
cudaFree( dev_b );
cudaFree( dev_c );
return 0;
}
```

Sum of two vectors a,b, using the CUDA architecture