Image Processing on the GPU: a Canonical Example

Cynthia Bruyns and Bryan Feldman

Department of Computer Science, University of California Berkeley, Berkeley CA, USA

Abstract

Recent computer vision work at Berkeley has focused on tracking both human and animal motion in a sequence of images [1]. The most costly stage of the tracking algorithm is searching through a given frame for recognizable limbs at different orientations. In this project, we propose using the GPU as a means of parallelizing the search problem. The use of the GPU for this canonical vision problem is an extension of recent work demonstrated by Moreland and Angel [2]. By transforming the tracking algorithm into various stages of a modified rendering pipeline, we achieve a performance increase ten times that of an algorithm solely implemented on a high-end CPU.

Keywords: Image Processing and Computer Vision, Graphics Processors, Filtering

1. Introduction

The aim of this paper is to explore the use of the GPU for computer vision applications. This includes both demonstrating how a common search algorithm is modified for GPU implementation and comparing the running times of the GPU and a high-end CPU for a canonical search problem

In general, tracking algorithms seek to return some information about an image or sequence of images. Usually, the number of computations per algorithmic pass over an image can be rather significant. That is because it is often assumed that information about the position and orientation of the camera relative to the objects being captured is not known, so one must often look at a variety of orientations and scales for the objects of interest.

Equation 1 demonstrates the computational requirements for a common template matching algorithm:

$$[(1024 \times 768) \times 3colors] \times 12orientations \times 5scales$$

Equation 1: *Search space for a typical 1024 x768 image*

Template matching algorithms normally used for tracking, compute intensity at each pixel using values from a surrounding neighborhood. Often times, the pixels in this neighborhood have no data dependencies for a simultaneous computation. Instead, each pixel computes its intensity based on results executed in a previous computation. As such, much of this computation can be done in parallel, as there is a large degree of independence within the streams of data.

2. A Canonical Example, Limb Finding

In order better illustrate the parallelism often inherent to computer vision algorithms, we now describe the steps involved in the portion of the tracking algorithm implemented for this project – finding candidate limb segments in an image. We assume that rectangular shapes of a specified aspect ratio are possible limbs, and construct templates to represent these limb candidates. As such, the search looks for either dark rectangles on a light background or light rectangles on a dark background. To find and localize these rectangular shapes there are three basic steps. These steps are described below and are repeated for a number of orientations and scales in the image.

The first step is to find the long edges of the rectangles. This is accomplished by convolving with a filter that has a large response to changes in pixel intensity across a long edge that is orientated at the specified angle. The filter is generated using a differencing mask the size of the limb to be searched for. The filter is then rotated in order to match the desired limb orientation. Changes in intensity signify

this long edge border. Figure 1 demonstrates the use of -1 and +1 to signify dark and light intensity and the use of zero padding elsewhere to inhibit response to those pixels.

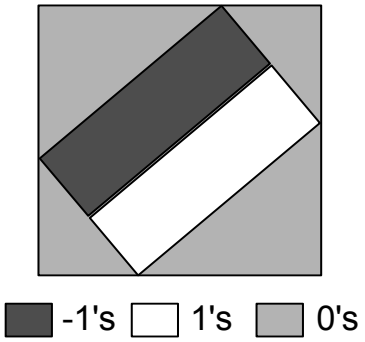

Figure 1: *A rotated filter for long edge*

In order to perform this convolution we have found that it is most efficiently implemented in the frequency domain. First both the image (I) and the filter (F) are transformed into the frequency domain via the FFT. Since the filter is typically smaller than the image it must be padded with zeros before the transformation. Once in the Fourier domain, each element of the transformed image is multiplied by the corresponding element of the transformed filter. Then the result of this element-by-element multiplication must be transformed by the inverse FFT (IFFT) back into the spatial domain. The result of this step is a matrix whose elements are scored according to its likeness to a long edge of a rectangle at the specified orientation (i.e. its impulse response to the limb template). We denote $R_{conv}(i,j)$, as the value of this response where (i,j) refers to the pixel location.

$$R_{conv} = I * F = FFT^{-1}\{FFT[I] \times FFT[F]\}$$

Equation 2: *Convolution semantics*

In the next step, we compute the likeness to two long parallel edges by looking an appropriate vector offset from each pixel in the positive and negative direction interpolating the values from the previous step. Since we are looking for dark-light-dark (d-l-d) and light-dark-light (l-d-l) transitions, we can efficiently re-use our previous computations by negating one of the values from $R_{conv.}$ Equation 3 gives the expression for this step and Figure 2 illustrates the comparison for a dark-light-dark transition.

$$R_{d-l-d}(i,j) = \min[\ R_{conv}(i+v_i, j+v_j),\ -R_{conv}(i-v_i, j-v_j)]$$

Equation 3: *Cylinder ranking*

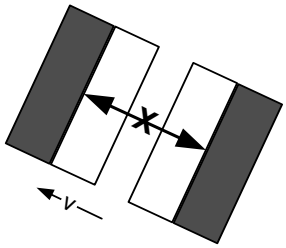

Figure 2: *R_{limb} at the location marked "x" is obtained by looking for response at +/- v away*

The score for a pixel being a limb, $R_{limb}(i,j)$ at a given pixel location is then the maximum of the response for the l-d-l and d-l-d comparison. Equation 4 demonstrates this cylinder ranking computation.

$$R_{Limb}(i,j) = \max[\ R_{l-d-l}(i_i, j), R_{d-l-d}(i_i, j)]$$

Equation 4: *Cylinder ranking*

As you will note the all steps in the calculation of R_{limb} for every pixel location, (i,j) are independent of those for all other elements of R_{limb} and only dependant on R_{conv} or intermediate pixel specific quantities.

The final step is to locate the limbs by finding local maximums in the cylinder ranked image. A local neighborhood surrounding the element is searched for the maximum response. This value is then assigned to the element in the center of the neighborhood according to Equation5, where d indicates local neighborhood size. In order to find the maximum efficiently, this operation is done first in the x then the y directions.

$$L_{\max}(i,j) = \max[R_l(i+k, j+k)] \quad for \quad k = -d..0..d$$

Equation 5: *Max finding*

Finally, we can find the locations of the local maximums by noting that L_{max} contains the values of the local maximums of R_{limb}. So if $R_{limb}(i,j) = L_{max}(i,j)$ then we know that (i,j) is the location of a local maximum. Figure 3 illustrates how a true limb center is determined.

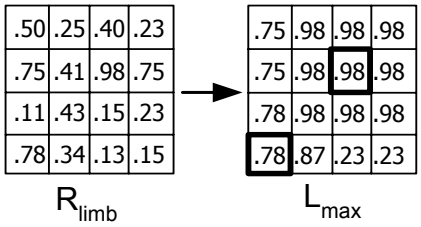

Figure 3: *To obtain L_{max}, each entry in R_{limb} is replaced with its local maximum. In this example the local neighborhood is 1, and the local maximums are highlighted on the right.*

3. The GPU for Image Processing

Given the long running times for a canonical search problem through an image, it is clear why parallelizing this search either through software or hardware is an attractive option. We chose to investigate the use of vector processors for three reasons.

Firstly vector machines are becoming prevalent as GPUs in many desktop machines. Most off-the-shelf desktop systems and portable computers include these cards due to the popularity of graphics intensive applications. GPUs have been used to portion off the rendering computations from the CPU to an adjunct graphics processor so that the CPU is free to perform other non-graphic computations. Initially most GPUs were connected to the CPU via a shared Peripheral Component Interconnect (PCI) bus. However, with the prevalence of 3D animation and streaming video, computer architects developed the Accelerated Graphics Port (AGP), a modification of the PCI bus designed specifically to facilitate the use of streaming video and high-performance graphics.

These stream co-processors have become programmable in ways that are extremely powerful and flexible to the developer. In fact, companies highly invested in this technology have partnered up to develop a language that is approaching the sophistication of C [3, 4, 5].

With development resources being allocated to fast communication between the CPU and the GPU, and given the increasing sophistication of off-the-shelf GPUs it is clear that computer architects are more aggressively structuring their machines to meet the high demands of real-time graphics engines. Our challenge is to find ways to exploit the capabilities of these advanced processors for computer vision problems.

Other work on the GPU

Due to the lucrative nature of PC gaming and the relative abundance of transistors, companies such as ATI and NVIDIA are engaged in a battle for dominance in the GPU market. As a result, consumers are seeing an explosion in the capabilities of these cards.

Figure 4: *NVIDIA generated character demonstrating the capabilities of modern graphics cards*

Last year, the use of GPUs in the scientific community took a giant leap forward with the release of the **cg** [3,4,5] shading language and the example applications that had been ported to these systems [2,6,7].

One of the most notable applications computed the Fast Fourier Transform (FFT) in a way that changed an otherwise cumbersome operation into simply another stage in a modified rendering pipeline [2]. By using texture buffers, Moreland and Angel were able to perform FFT and convolution in under a second for a 512 x 512 pixel image.

Kruger and Westermans [7] and Botz et. al. [6] developed strategies for implementing linear algebra operators on the GPU, including efficient representations and operations of vectors and sparse matrices. They then used to the GPU to do fast solutions to large sparse linear systems using both multigrid and conjugate gradient methods. One of the ultimate goals of their work was to develop a hardware implementation of the basic operators that would be found in linear algebra software libraries.

Work by Colantoni et. al. explored using the GPU for image processing [8]. They discovered that most

cases yielded faster results using the GPU and up to a ten times speed up for one application. They also mention work on hybrid processing using both the CPU and GPU for use in video stream processing.

4. Review of the GPU architecture

Although there are many different graphics cards available, with each possessing slightly varying capabilities, graphics cards are basically data-flow machines as illustrated in Figure 5.

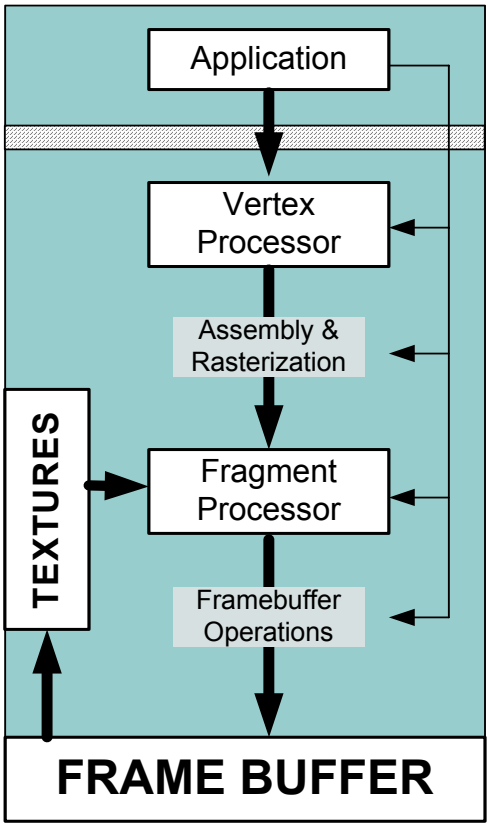

Figure 5: *GPU data flow*

Data is instantiated in the main application that runs on the CPU. Then this data is passed to the GPU either through binding variables or through passing data to graphics registers in groups of values stored in textures.

Once the data is on the graphics card, it follows through the rendering loop being modified at each step.

In a common rendering pipeline, data starts out as models in a virtual scene. These models can be either 3D or 2D objects. In either case, models are composed of vertices and facets, and can be reconstructed by the data structure that describes its connectivity.

The first modification happens in the vertex processor, where routines called vertex programs operate on each vertex of the model. In traditional graphics applications, this is where the shading and lighting computations are performed, as well as information regarding vertex perturbation due to skinning or bump mapping. For image processing applications, the step though the vertex program is used to determine texture coordinates and modifications to the color of the object's surface due to shading.

Next the scene is assembled and rasterized. Rasterization is a process of scan converting a mathematical description of an object's surface (continuous or piecewise-continuous) into a discrete color values stored in a pixel frame buffer [12]. Once broken into pixels, data is sent to the fragment processor, where routines called fragment programs further modify the pixel color data.

The final stage in the GPU rendering pipeline is filling the frame buffer with the final color values for each pixel.

Given its data flow architecture, programming the GPU amounts to transforming an algorithm to fit within the framework of the vertex and fragment programs [13]. These programs act as microcode that is loaded directly onto the GPU and hence no instruction fetching from main memory is needed.

Depending on the graphics, and the graphics interface, the capabilities of the fragment and vertex processors vary. For the newer NVIDIA graphics cards, such as the FX 5900, the capabilities of the vertex processor are as follows [14, 15, 16]:

- 4-vector FP32 operations
- True data-dependent control flow
  - Conditional branch instruction
  - Subroutine calls, up to 4 deep
  - Jump table (for switch statements)
- Condition codes
- New arithmetic instructions
- User clip-plane support
- 256 instructions per programs
- 16 temporary 4-vector registers
- 256 "uniform" parameter registers
- 2 address registers (4-vector)

And for the fragment processor:

- 1024 instructions
- 512 constants or uniform parameters
  - Each constant counts as one instruction
- 16 texture units
  - Reuse as many times as desired
- 8 FP32 x 4 perspective-correct inputs
- 128-bit framebuffer "color" output

In addition, the programs for these processors have the following restrictions:

- No branching
- No indexed reads from registers
- No memory writes

Also, GPU's don't support the following data types:

- No integer type
- No support for pointers

However, the ***cg*** Language supports the following useful types:

- float
- bool
- sampler1D, sampler2D, sampler3D, samplerCUBE
- half -- half-precision float
- fixed -- fixed point [-2,2)

Given these capabilities and limitations, the next step in programming a computer vision application for the GPU is to transform the code to fit within these parameters.

5. Implementation

The first step in morphing code from being fully executed on the CPU to being split between the CPU and the GPU is determining where the components of the algorithm should be executed.

Care must be taken in deciding which variables should be shared between them. That is because the process of moving data from one processor to another is currently limited by the PCI or AGP data transfer rates. For the NVIDIA FX 5900, the AGP transfer rate is 2.1 GB/s.

Once you have determined how the algorithm is split and what data is worth the overhead of transporting memory from the CPU to the GPU, the next step is transforming the code to fit within the framework of the GPU rending pipeline. Our implementation choices are described in the following sections.

CPU

First, the image to be searched is loaded into a texture map. Texture maps are used primarily in graphics applications to hold color information about an object's surface that would otherwise need to be computed through procedural means or modeled with great surface detail. In our application, the texture map is simply the image to be searched. This image is applied to a rectangle aligned to an orthographic camera in the virtual environment. Upon invoking the draw routine for the virtual scene, the CPU hands off information about this object and its associated texture to the GPU for processing through the modified rendering pipeline.

GPU

Once in the GPU, data flows in parallel for vertices and fragments. One can think of rendering as a SIMD problem where each instruction is executed on parallel streams of vertices or fragments. These streams only interact in the pipeline, either through the texture buffers or through accessing information in the frame buffer via multiple pipeline passes. In this way, one can think of the texture and frame buffers as pipeline registers. Since each stage is executed in lockstep, RAW hazards are avoided. In addition, by using two buffers, we can avoid RAW hazards between the original image to be processed and intermediate results [17]. In our application we traverse the rendering loop on the GPU for each orientation we are searching.

Vertex Processor

The first stage through the pipeline for our application is determining the coordinates for the texels (texture primitives), and the color at each texel location. In this stage of the pipeline, the texel color can be altered by shading or blending with the underlying polygon. However, in our application, the vertex program execution is a compulsory part of the pipeline, where no actual computations are performed.

Fragment Processor

Instead, fragment programs are used to implement these per-pixel operations. Figure 6 illustrates the use of fragment programs to modify the rendering pipeline.

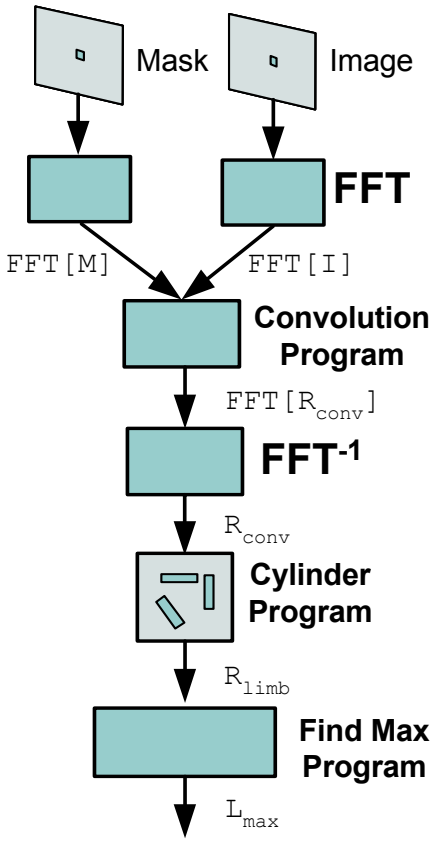

Figure 6: *Program loop*

We use a four color channel texture map which means that for each arithmetic computation, all color channels are operated on concurrently. As a result, no additional loops are needed for multiple color channel operations. In addition, we can store the results of our search in another texture buffer, encoding the orientation and intensity information in a four-vector. In this way, we can report the necessary information and avoid limiting our reporting to fragment program data types.

The first operation that is performed is to transform the image from the spatial domain into the frequency domain.

FFT of the Image
The FFT was computed on the GPU using the code supplied by Moreland and Angel [2]. At the beginning of the application each image to be searched is transformed and stored in the frequency domain as a texture map. Then the search for each orientation becomes a process of convolving this texture with each template mask. The following operations are performed per orientation:

- Generation of masks
- FFT of masks
- Convolution of image and mask
- Inverse FFT
- Taking highest contract color channel
- Search left and right
- Search for local max in X direction
- Search for local max in Y direction

Generation of Masks
Next, masks are generated to represent candidate limb orientations. These masks are embedded in an image by first generating the mask aligned with the screen, then rotating the mask incrementally until a full 180 degrees is swept. We chose to increment by 15 degrees to include enough variation to search a natural image for limb orientations.

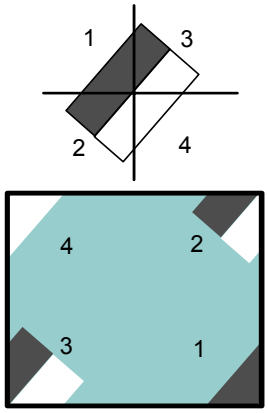

Figure 7: *Mask embedding. Numbers correspond to quadrants of mask and show where they map into an image of larger size.*

FFT of Masks
Once the mask image has been generated, it is passed through the pipeline for transformation via the FFT fragment program.

Convolution of Image and Mask
The convolution between the image and mask in the frequency domain is another fragment program, which essentially performs a piecewise multiplication between components. Each fragment looks into a stored buffer to retrieve the mask counterpart and their intensities are multiplied.

Inverse FFT

The result of the convolution is another image which contains the impulse response of each pixel to the given template. In this way we can think of the results at this stage, to be an image indicating how much each pixel looks like the dark to light transition at a given orientation. In order to further process the image, the result is transformed back to the spatial domain via an Inverse FFT.

Search Left and Right

The next stage in the pipeline is to score each pixel based on its likeness to being part of a cylinder. In this case, pixels to the left and right[1] are queried, and the pattern of light-dark-light or visa versa is checked. If the current pixel does not match that template, then its intensity (amount of color) is set to zero and subsequent operations will have very little effect on this pixel. If however it does appear to be part of a cylinder, then its intensity is maintained and further processed in the next stage of the pipeline.

```
float2 leftSide;
leftSide.x =
input.wposition.x + local.x;
leftSide.y =
input.wposition.y + local.y;

float2 rightSide;
rightSide.x =
input.wposition.x - local.x;
rightSide.y =
input.wposition.y - local.y;

float4 resp1 =
f4texRECT(source, leftSide);
float4 resp2 =
f4texRECT(source, rightSide);
float4 currResp =
f4texRECT(source, input.wposition.xy);

output.value =
max(max(max(-resp1, resp2),
max( resp1,-resp2)), currResp);
```

Listing 1: *Pseudocode for look left and right*

Search for Max in X Direction

At this stage the image contains pixels that appear to be at the center of cylinders at a given orientation, but what is most important is that only the true center is reported. In order to find these true centers, we search the image for local maxima. As with the previous search, the notion of locality is subject to the size of the limb template.

To begin this search each pixel is visited and its local neighborhood in the x direction is searched for the largest intensity. If there is a pixel with higher intensity than the current pixel, then its intensity is replaced with the maximum.

Since this is a per pixel operation, at the end of this stage in the modified rendering pipeline, the image contains rows of local maxima in groups according to the defined notional of neighborhood size.

```
float4 v1;
float4 currResp =
f4texRECT(source, input.wposition.xy);

for (i=1; i<local; i++) {
    float2 indexL =
        {max(0,input.wposition.x-i) ,
             input.wposition.y};
    float4 respL =
         f4texRECT(source, indexL);

    float2 indexR =
        {max(frameWidth-1,
             input.wposition.x+i),
            input.wposition.y};
    float4 respR =
         f4texRECT(source, indexR);

      v1 = max(v1,max(respL,respR));
}
output.value = max(v1,currResp);
```

Listing 2: *Pseudocode for max X*

Take Highest Contrast Color Channel

Next we select only the highest contrast channel by adding the following step to the end of the previous fragment program.

```
output.value.x =
max(output.value.x, output.value.y);
output.value.x =
max(output.value.x, output.value.z);
output.value.x =
max(output.value.x, output.value.w);

output.value.y = 0;
output.value.z = 0;
output.value.w = 0;
```

Listing 3. *Max color channel selection*

[1] The pixels queried are offset from the current pixel by an amount considered to be comparable to the scale of the width of the limb for which the search is conducted. The direction of the offset is obtained from the potential limb orientation.

Search for Max in Y Direction
The next step is to search the results of the previous stage for local maximums in the Y direction. Since the results from the previous stage contain local maximums grouped according to a local neighborhood, a search for local maximum in the Y direction at this step amounts to finding the local maximums in the neighborhood squared.

For each pixel, if its value matches that of the local max in the Y direction, then it is a local max and a limb is reported as being centered at this pixel in this direction. Listing 4 demonstrates this procedure.

```
float4 v1;

for (i=1; i<local; i++) {
    float2 indexB =
      {input.wposition.x,
       max(0 ,  input.wposition.y-i};
    float4 respB =
       f4texRECT(source, indexB);

    float2 indexT =
       {input.wposition.x,
       max(frameHeight-
       1,input.wposition.y+i};
   float4 respT =
       f4texRECT(source, indexT);

      v1 = max(v1,max(respT,respB));
}
```

Listing 4*: Pseudocode for max Y*

Final Search Through Image
Because the type of data sent to and retrieved from the GPU is limited, we restructured our limb reporting method. The maximums were stored in a texture buffer so that the location of the limb centers and the limb angle could be encoded within the pipeline stage. As such, we determined the maximum number of limbs to be searched for and created an appropriately sized texture map. We then choose to encode the same information as the CPU implementation. That is, at each pixel that is determined to be a local maximum, an entry is placed in the texture map that reports back this pixel's x and y location, intensity and the angle for which this pixel was found to be a maximum (zero angles shifted by 180 degrees). In this way, if a pixel is reported to be a maximum for several angles, only the last one is maintained. The final reporting of limb locations occurs as an addition to the above step as follows:

```
float4 currResp =
f4texRECT(source, input.wposition.xy);

if (v1.x == currResp.x) {
    output.value.x = input.wposition.x;
    output.value.y = input.wposition.y;
    output.value.z = angle+180;
    output.value.w = currResp.x;
}
else {
    output.value = {0,0,0,0};
}
```

Listing 5: *Reporting max in a texture buffer*

6. Results

The following results were obtained using a 2GB 2.53GHz Pentium IV PC running Windows XP and a 512 MB 800 MHz Pentium III PC running Windows XP with a 128MB RAM 400MHz 8xAGP NVIDIA FX 5900 graphics card using the NV30 profile.

First we present the timing for the overall limb finding operation and then break the algorithm down by its components for a closer examination of the performance.

Overall Timing
When comparing the performance of the CPU implementation to the GPU implementation, it is interesting to note the effect on performance as the image size increases and as the mask size increases. This is because the FFT is an O(MN(log M + log N)) operation [2] where M and N are width and length of the image. As a result, a larger M or N results in a longer computation. In addition, convolution between and image and a mask in the spatial domain is $O((MN)^2)$ and O(MN) in the frequency domain. Therefore it is important to examine at what point the overhead of FFT is profitable enough to obtain the lower convolution time.

The results below are the times to execute the entire pipeline as in Figure 6, for each orientation searched in a single image. In addition, the algorithm was run several times and the average results were taken. In doing so, we included times for caching. We will discuss later the effect of caching on performance.

The following result is time in seconds for a fixed mask size of 22 x 13 and an image size increasing from 256 x 256 to 1024 x 1024 for a CPU implementation in the spatial and frequency domain and a GPU implementation.

	256^2	512^2	1024^2
CPU	16.28	69.49	581.38
CPU FFT	14.45	59.58	252.81
GPU	3.94	6.30	25.95

Table 1: *Overall running time, mask size 33 x 12 increasing image size*

As seen above, the time to perform the search on the GPU is several orders of magnitude faster than on the CPU and the CPU using a FFT. Even for very large images, the GPU performs roughly ten times better, which is in agreement the work by Colantoni et. al [8].

The following is time in seconds for a 512 x 512 image with mask sizes varying from 11 x 7 to 44 x 25.

	11x7	**22x13**	**44x25**
CPU	47.84	69.49	142.45
CPU FFT	59.53	59.58	59.39
GPU	6.27	6.28	6.30

Table 2: *Overall running time, image size 512 x 512 increasing mask size*

Consequently, using the GPU as a machine for performing this computation is several orders of magnitude faster. Even with small mask sizes, when conventionally one would avoid the computation of the FFT in favor of a spatial convolution, the GPU performs better.

Next, we look closer at each stage in the modified rendering pipeline to determine where improvements can be made if any.

Time to FFT Images

The time to perform the FFT for the original image and the mask images was compared next.

	256^2	512^2	1024^2
CPU FFT	0.1278	0.5201	2.0991
GPU	0.0608	0.0714	0.0937

Table 3: *Time to FFT images*

Time to Convolve

Performing the convolution between the image and the mask in the frequency domain is an interesting operation. Once the mask has been embedded in an image, as long as the mask fits within the boundaries of the original image to be searched, the time to perform the convolution is independent of mask size (See Figure 7 for an example). Because of this independence, we only report increasing image size for the rest of the pipeline. The following Table gives the time to convolve the image and mask.

	256^2	512^2	1024^2
CPU FFT	0.0309	0.1217	0.4889
GPU	0.0010	0.0011	0.6522

Table 4: *Time to convolve*

IFFT Time

Table 5 lists the time to perform the inverse FFT on the results of the convolution.

	256^2	512^2	1024^2
CPU FFT	0.1548	0.6250	2.5455
GPU	0.0127	0.0143	0.0177

Table 5: *Time for IFFT*

Search Times

The time to search the image for cylinders is broken into two sections, time to search left and right, and time to find maximums.

	256^2	512^2	1024^2
CPU FFT	0.0701	0.2626	1.0165
GPU	0.0009	0.0010	0.0011

Table 6: *Time to search left and right*

	256^2	512^2	1024^2
CPU FFT	0.2526	0.1794	7.1003
GPU	0.0021	0.0022	0.0025

Table 7: *Time to search max, including max channel selection*

7. Discussion

The results demonstrate that for all stages of the pipeline, the GPU performs several orders of magnitude faster than the CPU. The only exception was performing the convolution between the filter and mask for a 1024 x 1024 image, where the GPU implementation was 30% slower. In this case, we believe the GPU was unable to fit the data in cache, and was spending time managing memory. This assumption is supported when we performed timing without caching. On cold-start, the time to perform the search operation once is several seconds slower. Given that this is currently a limitation of the hardware, we believe that future generations of GPUs will not have this deficiency. Another method to avoid paging to system memory and increase traffic over the graphics card bus would be to drop precision in the texture map, i.e. by ignoring the alpha channel. For the 1024 x 1024 image, this would at least allow the image to fit within the GPU memory using 32FP precision.

By comparing the overall algorithm running time to that of the components of the pipeline, we see that for a 1024 x 1024 image, the time spent in the rendering components is 9.20 seconds, while the overall time is 25.95 seconds. This suggests that the time to assemble the data into textures and the time to transfer the data to the graphics card is a considerable portion of this operation. In fact, for the operation that copies the pixels from the rendering window into a texture buffer, the time is 1.09 seconds on cold start and for a 1024 x 1024 image. This operation is performed for each image to be searched and for each mask. Additional latencies are then due to creation of the mask and storage of the data into texture buffers. Future work on this project will explore ways to avoid these overhead operations. However it should be noted that, as the complexity of the modified rendering pipeline increases, the costs due to memory management become amortized over several operations performed on the GPU.

One optimization that could be made for smaller images would be to tile the images to be searched within a larger image for concurrent processing. Tiling four 256 x 256 images in a 512 x 512 image would only require an additional 2.36 seconds, while performing the entire operation again for a different 256 x 256 image would take an additional 6.94 seconds.

8. Conclusions

The results of this study demonstrate that certain classes of computer vision algorithms can benefit from implementation of the GPU. The process of transforming the problem into a modified rendering pipeline has been established and the results are promising. We expect that developments in hardware will address current limitations in GPU performance.

9. Acknowledgements

The authors would like to acknowledge the assistance of Kenneth Moreland in the implementation of his FFT code as well as useful conversations. Deva Ramanan was instrumental in designing the cylinder finding program. Lastly, the authors extend a special thanks to Okan Arikan.

References

1. Ramanan, D. and Forsyth, D. A. "Finding and Tracking People from the Bottom Up" In Proceedings of Computer Vision and Pattern Recognition (CVPR), Madison, Wisconsin, June 2003.

2. Moreland, K. and Angel, E. “The FFT on a GPU” In SIGGRAPH/Eurographics Workshop on Graphics Hardware 2003 Proceedings, pp. 112–119, July 2003

3. Mark, W., Glanville, R., Akeley, K. , Kilgard, M. “Cg: A System for Programming Graphics Hardware in a C-like Language” in Proceedings of SIGGRAPH 2003

4. Fernando, R. Kilgard, M.J., The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics Addison-Wesley Pub Co 2003

5. Kilgard, M.J., “Cg in Two Pages”, ArXiv Computer Science E-prints, Feb 2003

6. Bolz, J. Farmer, I., Grinspun, E. and Schröde, P “Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid“,Proceedings of SIGGRAPH 2003

7. Krüger, J., Westermann, R., “Linear Algebra Operators for GPU Implementation of Numerical Algorithms”, Proceedings of SIGGRAPH 2003

8. Colantoni, P. , Boukala, N. Rugna, J. “Fast and Accurate Color Image Processing Using 3D Graphics Cards” in 8th Fall Workshop Vision, Modeling, and Visualization 2003,

9. Strzodka, R. Ihrke, I. ,Magnor, M. A “Graphics Hardware Implementation of the Generalized Hough Transform for fast Object Recognition, Scale, and 3D Pose Detection” Proc. IEEE International Conference on Image Analysis and Processing (ICIAP'03), Montova, Italy, pp. 188-193, September 2003

10. Goodnight, N. , Wang, R. Woolley, C., and Humphreys, G. ”Interactive Time-Dependent Tone Mapping Using Programmable Graphics Hardware” in Eurographics Symposium on Rendering, 2003

11. Viola, I. “Applications of Hardware-Accelerated Filtering” Masters Thesis VRVis Research Center. Vienna, Austria

12. Woo, M, Neider, J., Davis, T., OpenGL Programming Guide Addison Wesley, Reading MA, 1997.

13. Chan, E. , Ng, R., Sen, P., Proudfoot, K., Hanrahan, P., “Efficient Partitioning of Fragment Shaders for Multipass Rendering on Programmable Graphics Hardware” in Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware 2002

14. cg Language Specification ftp://download.nvidia.com/developer/cg/Cg_Specification.pdf

15. cg Toolkit User’s Manual ftp://download.nvidia.com/developer/cg/Cg_Users_Manual.pdf

16. NVIDIA OpenGL Specifications http://developer.nvidia.com/object/nvidia_opengl_specs.html

17. Harris, M. Render Texture Computer Software. 2003 http://www.cs.unc.edu/~harrism/misc/rendertexture.html