Implementation of a visual difference metric using commodity graphics hardware

Jered E. Windsheimer, Gary W. Meyer

Dept. of Computer Science and Engineering, University of Minnesota, 200 Union Street SE, Minneapolis, MN, USA 55455

ABSTRACT

Recent improvements in Graphics Processing Units (GPUs) make it possible to execute complex image-processing tasks on commodity video cards. The vertex and pixel pipelines of modern GPUs are reprogrammable using high-level programming languages to accomplish almost any task a CPU can perform. Additionally, GPUs are designed to execute vector and matrix operations at high speeds with high parallelism. GPUs now support full floating-point precision in each color channel, allowing techniques that require such precision to be more easily supported than in the past. This paper reviews the development of a complete implementation of the Sarnoff Visual Discrimination Model (VDM) that executes almost exclusively upon the GPU. This implementation takes advantage of several properties of modern GPUs to improve the running time by an order of magnitude compared to the CPU implementations. An interactive version of the VDM allows the user to explore, in near real time, the significance of various pictorial artifacts.

Keywords: GPU computing, Graphics Hardware, Visual Difference Metrics

1. INTRODUCTION

The capabilities of Graphics Processing Units (GPUs) have significantly improved over the last few generations. These new capabilities allow the GPU to be treated more like a CPU. It is now possible to develop software which delegates some computationally expensive but highly parallelizable tasks to the GPU.

Several papers have been published recently that exploit GPUs. Some of the papers discuss general algorithms for linear and nonlinear computations[1–3] that are useful for a large variety of situations. Others have described specific software goals such as ray-tracing[4,5] and cloud simulation.[6] The new functionality of the GPU allows an ever-widening set of applications and algorithms to be shifted from the CPU.

The Sarnoff VDM described by Lubin[7] is an excellent example of the sort of software that can benefit from the newest GPU features. This paper reviews the GPU features that make implementing complex software possible, describes some issues regarding developing software for the GPU, and further analyzes the specifics of the Sarnoff VDM implementation.

2. IMAGE PROCESSING ON THE GRAPHICS PROCESSING UNIT

2.1. GPU Capabilities

Early GPUs were much simpler than the current generation. The functionality of the early GPUs was fixed, and fairly limited. The hardware could perform basic geometry rasterization and texture mapping from textures stored in local memory. Over time, the hardware improved in both speed (partially from multiple processing pipelines) and functionality, adding capabilities such as the application of multiple textures in a single pass, evaluation of 3D transformations and lighting equations, and special hardware for a small amount of pipeline programmability (register combiners). These improvements, and a few others that are very recent, allow much more to be done with the GPU than ever before.

Further author information:
J.E.W.: E-mail: windshei@cs.umn.edu
G.W.M.: E-mail: meyer@cs.umn.edu, Telephone: 1 555 123 1234

The programmability of the GPU is undoubtedly the most significant advancement in GPU technology. The GPU no longer applies a fixed-function pipeline to each vertex and pixel it processes. Register combiners previously allowed a small amount of programmability, but the major portions of the pipeline were still fixed. Instead, certain stages of the rendering pipeline have been replaced with stages that apply small user-defined functions to the data. The programs, known as vertex programs and pixel (or fragment) programs, can be developed in a high-level language (such as NVIDIA's Cg,[8] Microsoft's HLSL or OpenGL's glslang) and compiled to the native instruction set of the GPU. With this programmability, one can perform complex processing tasks on the vertices and fragments that come down the pipeline.

A major factor in the high speed of modern GPUs is parallelism. Multiple pipelines exist for pixel processing, allowing different pixels of the output image to be processed at the same time. The GPU handles this transparently, allowing the vertices and fragments being processed to easily share resources such as textures.

GPUs now allow the rendering output to be sent directly to on-card texture memory instead of the frame buffer. This feature allows the use of special rendering targets that are in formats the frame buffer doesn't normally support.

The GPU is fully capable of performing computations using floating-point values with high precision, but when all output is sent to the frame buffer, much of the accuracy is discarded by the 8-bit integer color channels. GPUs now allow textures in texture memory to use high-precision formats (such as 32-bit floating-point representation for each color channel), so the precision can be retained when rendering to texture memory. Using high-precision data types in the pipeline and permitting the results to be output to high-precision texture memory allows the GPU to perform high-precision calculations that are necessary for many complex tasks.

One of the other critical new features is dependent texture lookup. In earlier GPU generations each fragment was only allowed to access textures at the coordinates specified by the pipeline, generally the texture coordinates interpolated from the vertices. Now a pixel program is allowed to access any pixel of its associated textures, at any time the program wishes. This feature lets one treat texture maps much like two-dimensional arrays in normal software. Textures can be used as lookup tables containing approximations to complex functions, or a pixel program can read the texels around its usual texture coordinate, allowing arbitrary filtering to take place. Further discussion of filtering can be found in Sec. 2.3.

It should be noted that there are many GPUs on the market. NVIDIA and ATI are the primary processor designers. While it would have been preferable to test products from both companies, at the time of this research it was not feasible to use ATI. While the ATI hardware has some significant advantages (notably the ability to render to multiple buffers in a single pixel program), it has a relatively low limit on the number of instructions in a pixel program. The ATI cards do implement a hardware F-buffer,[9] but the API extensions required for using this hardware are not available at this time. This limitation prevents ATI hardware from performing some stages in a single pass. Creating a multipass implementation for ATI hardware is an option, but it was not explored in this research. When the F-buffer is available for use an implementation should be straightforward, but the current implementation is limited to NVIDIA hardware.

Finally, the GPU code is written in the Cg language. This code is compiled and executed on an NVIDIA GPU to perform all the rendering. As GPUs approach the CPU in capabilities, they also become more complex. Unfortunately, exact specifications on the hardware are not publicly available. The software developer that desires high performance must either rely upon the Cg compiler to produce optimized code, or manually create machine-level code without a thorough understanding of the hardware. At many points during this implementation it became apparent that much of the code generated by the compiler was not well optimized. Very little manual optimization was performed on the current implementation, and while this does slow the implementation down, it leaves open the possibility that future compilers or manual optimizations could provide a significant speed improvement on current hardware.

2.2. Rendering to Buffers and Pyramids

All of the processing of the Sarnoff VDM takes place on single-channel images or pyramids of images. In the GPU implementation images are represented as four-channel (RGBA) textures, with each channel stored in 32-bit floating-point format (for a total of 128 bits per pixel). Faster but lower-precision formats (such as 16-bit

and 24-bit, depending on the card) are available, but it was not deemed a worthwhile tradeoff for this project. Such a switch may be useful in limited places, or for other projects.

Pyramids of images were represented by simply wrapping multiple independent images into a single class in software. One possibility for improvement would be to represent pyramids as separate regions of a single image. Unfortunately, some stages of the Sarnoff VDM (see Sec. 3) require sampling a different level of the same pyramid. Current GPUs do not allow reading and writing to the same texture at the same time, which forces the use of separate images for separate levels, or large amounts of texture copying.

The rendering process is actually fairly simple. The texture we wish to render to is bound as the active rendering target. Then a simple orthographic projection is set up, with a viewport that is the same size as the texture we are rendering to. A single quadrilateral that covers the entire viewport is drawn using a trivial vertex program, and the appropriate pixel program performs the desired computations on the texture.

2.3. Filtering

With the advent of dependent texture reads, general filtering has become a relatively painless process to implement on a GPU. Pixel programs are allowed to access any pixel of its associated textures at any point in the program. This allows a fragment program to use an arbitrarily shaped and sized kernel for convolution.

The simplest general filtering process requires, for each pixel, three instructions for each component of the filter's kernel. A kernel with a width of S and height of T requires $3ST$ instructions to be executed for each pixel of the output image.

These three instructions per filter kernel component include an add, a texture lookup, and a multiply-and-add. The add is used to determine the texture coordinate of the pixel in the original image that is to be multiplied by the current filter component. The texture lookup retrieves the actual value for that pixel. The multiply-and-add multiplies the pixel by the filter value and adds it to the running total for the output pixel. Careful optimization can reduce the number of adds required, but the texture reads appear to dominate the running time, so this optimization does not significantly improve the speed.

One place where the functionality of the GPU can be exploited is during the multiplication. The GPU can perform a scalar multiplication of a four-element vector just as quickly as a multiplication of two scalars. One can place four different monochrome signals into the four color channels of a single image, and filter them all at the same time. Similarly, one could place four different filters into the four available color channels and apply all four filters at the same time to one channel of an image. This feature can be used in several stages of the Sarnoff VDM, as will be discussed in Sec. 3.

The filter kernel, if precomputed, can be accessed by the fragment program in two different ways. If the filter kernel is small enough it can be placed in the temporary registers of the card. Placing the filter there incurs no penalty during the execution of the fragment program, though transferring the data from main memory to the registers may cause a performance hit. The other method involves placing the filter kernel in a texture. In this case, the filter kernel can be any size that the GPU can support as a texture, but retrieving the values of the filter requires an additional texture lookup, which can cause a significant performance penalty. As GPUs improve, using texture memory for large filters may become less necessary.

One other significant issue remains with regards to filtering. When the regions near the edges of the image are being filtered, the kernel often extends over the edges of the image. Figure 1 shows how this occurs near the edges of images. If the GPU attempts to retrieve pixels at texture coordinates that are beyond the edges of the image, the returned values may not be what the filter expects. As GPUs and their drivers change, these values may change as well. Filtering methods also sometimes desire specific types of values beyond the edges of the input image. There are two ways to deal with this complication. The pixel program can check texture coordinates prior to their use in a texture lookup, and if the coordinates fall outside the edges of the image, special steps are taken to get the desired result. This process can be performed inside the pixel program that does the actual filtering, by applying this process to each texture coordinate used for retrieval of input pixel values. This method can be very expensive, as this special processing must be applied to each texture coordinate, and the process could be several instructions. If the instruction count for this process is c, the instruction count for this additional processing on an M by N image would be $cSTMN$.

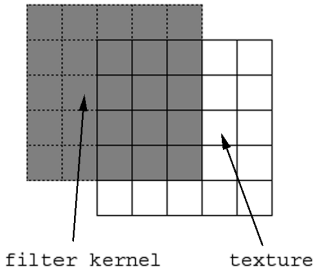

Figure 1. In this example a 5x5 filter kernel is centered near the edge of an image. The texture coordinates used to retrieve the pixels from the texture will be out of the texture's normal range. If these texture coordinates are used, it is likely that the results will not be appropriate.

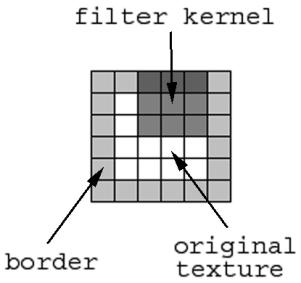

Figure 2. An example of creating a temporary image containing the original image and a border. By only applying the filter to the center region containing the original image, the filter kernel never extends beyond the edges of the temporary image. This ensures that texture lookups that fall outside the bounds of the original image will return appropriate values.

The other option is to create a temporary enlarged version of the input image. The original is placed in the center of the new image, pixel for pixel, and the border region is filled with the appropriate values for the filter to retrieve. This new image is then given to the filtering pixel program, and the texture coordinates are offset to allow the program to stay within the overall bounds of the image. Figure 2 demonstrates how this process prevents bad texture lookups. This method has the advantage of potentially requiring much less work per pixel, but it comes with the overhead of performing an additional pass and it requires additional texture memory. The instruction cost of this additional pass is roughly $c(M+S)(N+T)$ with an additional $(M+S)(N+T)$ texture reads, which are more time consuming than normal instructions.

As an example, consider a filter that is intended to reflect the image around the border. The proper texture coordinate can be calculated by using two sets of texture coordinates, tc_0 and tc_1. In one dimension, if b is the size of the border, and s is the size of the original texture, the new texture has a size of $(2b)+s$, the texture coordinates represented by tc_0 are in the range $[-b...s+b]$, and the coordinates of tc_1 are in the range $[2s+b...s-b]$. By finding the maximum of tc_0 and $-tc_0$, any texture coordinates which are below zero are properly reflected. By taking the minimum of that value and tc_1, texture coordinates above s are also properly reflected. A visual demonstration of this process can be seen in Fig. 3. This process can be performed in a single pass, which increases the instruction count of filtering to five instructions per filter component per pixel, for a total cost of $5STMN$. If the border is precalculated and then the resultant image processed without the reflection checking code, the overall number of number of pixel instructions is approximately $3(M+S)(N+T)+3STMN$.

3. STAGE DESCRIPTION

The stages of the GPU implementation of the Sarnoff VDM correspond fairly well to the stages described by Lubin.[7] The input images are monochrome, and so during the first few stages of the implementation they can

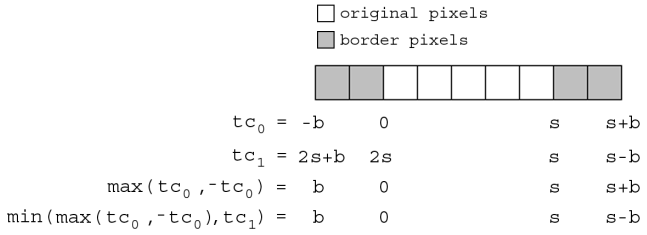

Figure 3. Demonstration of the process to reflect texture coordinates that fall outside the area of the original image. Texture coordinates that fall inside the original image ($[0..s]$) are unmodified, while coordinates that fall below 0 are reflected around 0, and coordinates that fall above s are reflected around s. This demonstration is in one dimension, but the GPU can perform this task in two dimensions at no extra cost.

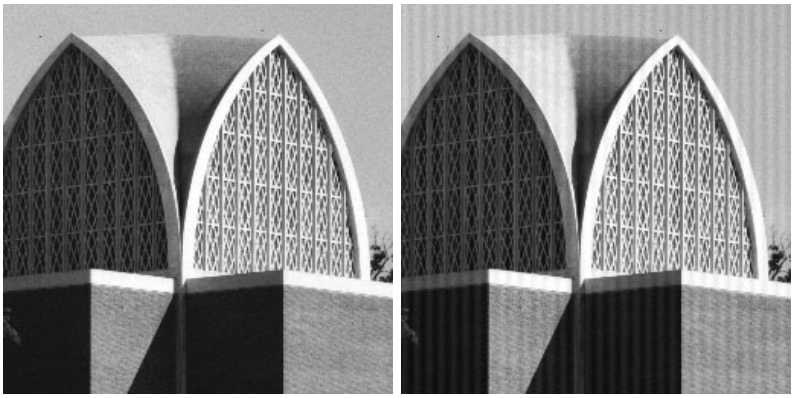

Figure 4. The two images used as input for the example.

each be placed in a different channel of a single image. This single image containing up to four input signals is then the only image that needs to pass through the first few stages. Figure 4 shows an example of input images.

3.1. Optics

Given the default viewing parameters of the implementation, the Optical stage consists of convolving the input images with a 3x3 filter. At this size, the filter fits easily in the registers of the GPU for fast access. Additionally, the filter is single-channel, which means that it can be applied to all four channels of an image at the same time. Therefore, this stage can be run once to process four independent images. Given the small size of the filter, creating a temporary border version of the input images for correct border reflection is not really necessary, and may instead slow down the program due to increased API overhead.

3.2. Sampling

Currently the implementation assumes that to find the retinal image the viewing parameters are chosen so that no resampling needs to take place. If the viewing parameters changed, one would need to perform a simple gaussian filter at this point, similar to the process necessary for the next stage.

3.3. Gaussian Pyramid

The Gaussian (Local Mean) Pyramid is calculated using the result of the Optics stage. The filter kernel is 5x5 and single-channel, which means that it fits easily in the registers of the GPU and can be applied to all four channels of the input simultaneously. The results of this stage can be seen in Fig. 5. With a filter kernel of this size it becomes more plausible to use the two-pass border reflection method. Unfortunately, each level of this

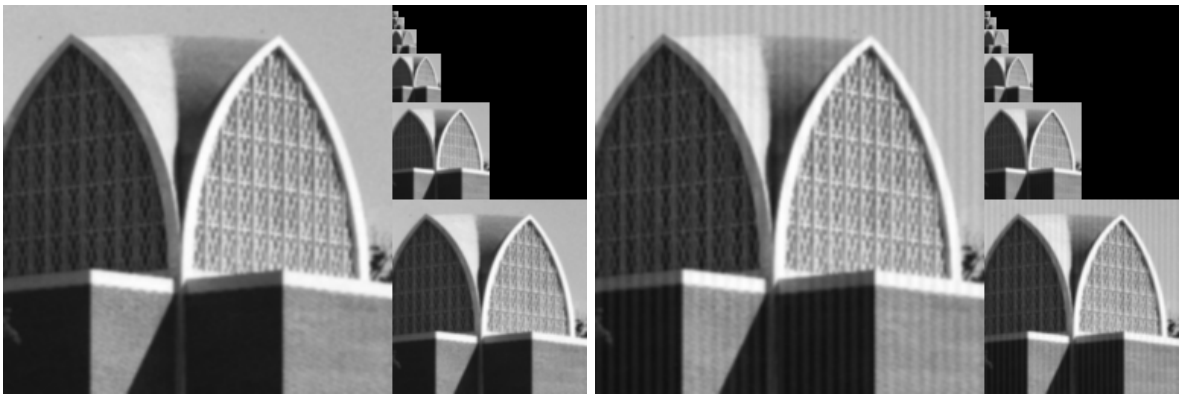

Figure 5. The results of the Gaussian Pyramid stage.

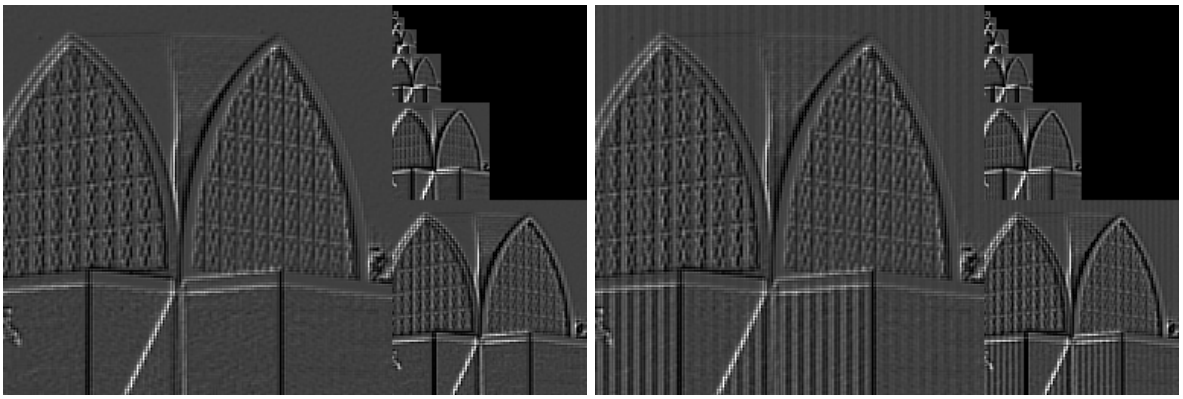

Figure 6. The results of the Contrast Pyramid stage. The base intensity of the images has been increased in this image to highlight that it is possible to properly store negative values in textures.

pyramid is created by downsampling and filtering the previous level, with the first level being a filter of the input image. Thus, to properly perform the two-pass border reflection, one would have to repeatedly switch between performing the border expansion and performing the filtering, and testing seems to indicate that this causes a significant performance hit. Therefore, this stage uses the single-pass border reflection.

3.4. Contrast Pyramid

The Contrast (Laplacian) Pyramid is calculated as the local difference divided by the local mean. The pixel program for level k of this pyramid uses the levels k, $k+1$ and $k+2$ of the Gaussian Pyramid. At a particular pixel, the corresponding pixel from each of the Gaussian levels is retrieved, and the calculation is performed. Overall this is a simple stage, and the GPU calculates it very rapidly. Additionally, there is no interaction between different channels in this stage, so independent images can be processed in each of the channels. Figure 6 shows the results after processing by this stage.

3.5. Energy Pyramid

In terms of implementation, the Energy Pyramid is easily the most complex stage. The primary problem is that its size creates a host of implementation issues. This stage takes the Contrast Pyramid as input and applies eight different 11x11 filter kernels (four orientations, each with two components). All other issues aside, the size of the filter kernel makes this stage a must for the use of two-pass border reflection. Additionally, current GPUs cannot fit all eight orientations into the registers at full precision.

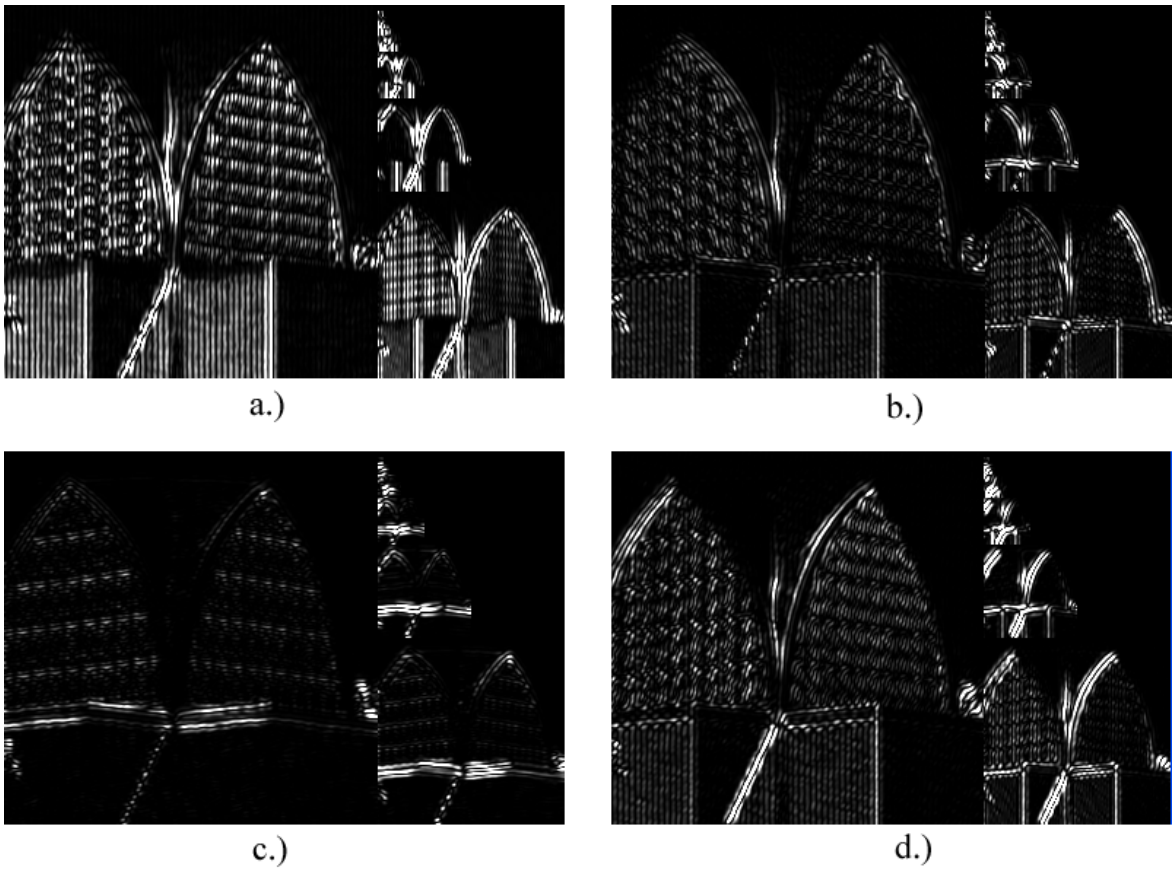

Figure 7. Black and white representations of part of the output from the Energy Pyramid stage. The images show the results of applying the four oriented Gaussian filters to the second Contrast Pyramid in Fig. 6. All four results are output in different color channels of a single output image. a.) The horizonal filter, located in the red channel of the output image. b.) One of the diagonal filters, located in the blue channel of the output image. c.) The vertical filter, located in the green channel of the output image. d.) The other diagonal filter, located in the alpha channel of the output image.

There are several options to deal with the difficulty of handling all eight kernels at once. One option is to encode the filter kernels as two texture maps, each containing four of the kernels. This method certainly works, but the cost of performing two texture lookups is actually very significant. Another option is to use packing to encode the eight kernels into four channels, at half the precision, and store them in the GPU registers. This method does allow all the kernels to fit into the registers, but the approach turns out to be infeasible due to the unpacking process that is currently implemented by the GPU used in this research.

The best option in terms of speed is to break the eight kernels into two sets of four. Four kernels are stored in the GPU registers at full precision, and the stage is run twice, each time using a different set of four kernels and writing to a different texture pyramid. While this does incur the cost of twice as many texture reads from the input images and extra API overhead, the overall speed is much faster than storing the filters in textures.

As a final note for this stage, processing four kernels at once means we can only apply this processing to one channel of input at a time, so at this stage each channel of the input must be treated individually. The output is the convolution of a single channel of the input pyramid with one of the eight kernels, each being placed into a different one of the channels of the two output pyramids. Figure 7 shows the four channels of one output pyramid.

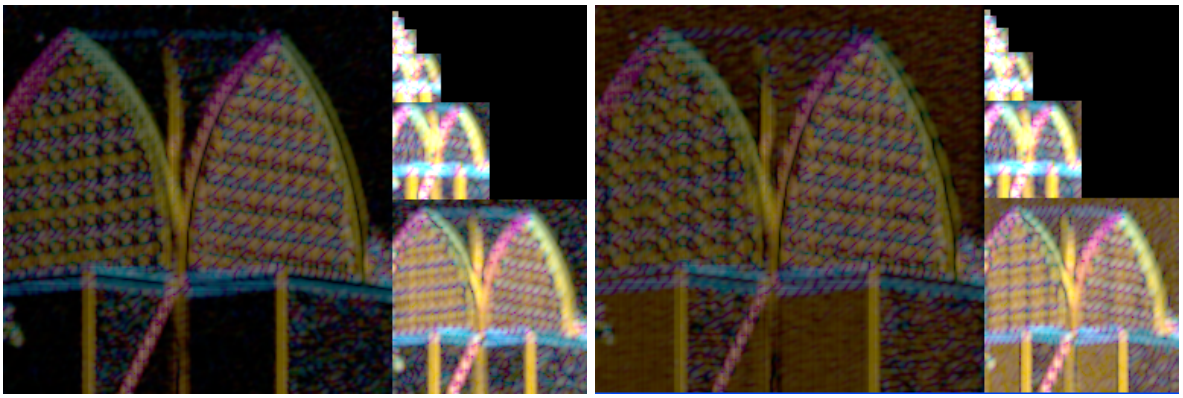

Figure 8. The results of the CSF and Transduer Pyramid stage. The intensity of these images has been scaled down.

3.6. CSF and Transducer Pyramid

Given the implementation of the Energy Pyramid stage, this stage processes two Energy Pyramids and a single channel of the Local Mean Pyramid. First, the two Energy Pyramids are recombined into the proper four-orientation energy representation. Then, for each pixel, the contrast detection threshold function M_t is calculated and applied using both the luminance from the corresponding pixel in the Local Mean pyramid and the peak frequency for the level. Following this operation, the non-linear sigmoid function is applied. Both of these functions are somewhat expensive, and they could probably be precalculated and encoded into a texture. Then instead of performing expensive calculations at runtime, the proper value can be found by performing a dependent texture lookup. This optimization is not currently implemented. Figure 8 shows the output of this stage.

3.7. Pooling Pyramid

The Pooling Pyramid is a simple stage where the value of a pixel is calculated as the average over a 5x5 region surrounding it. As such, it's similar to the Gaussian Pyramid stage, but a bit simpler. It executes reasonably fast using single-pass border reflection, so two-pass border reflection will probably not improve the speed.

3.8. Difference Pyramid

The Difference Pyramid takes two Pooling Pyramids and applies the distance calculation. The Result is a single-channel pyramid that describes the difference found in each pixel of each level. This is an intermediate stage that exists in case the GPU can't support the number of active textures it would require to calculate the Difference Map in one pass. Figure 9 shows the output from this stage.

3.9. Difference Map

The Difference Map stage takes a Difference Pyramid, upsamples all the stages to the same size, and combines them. This requires as many as 7 active textures, which is fine for most modern GPUs. Figure 10 shows the results this stage, a JND Map. This stage would likely be faster if pyramids were implemented as a single texture with multiple regions, rather than multiple textures.

3.10. Statistics

The Difference Map is complete after the last stage has finished. However, if statistical information (such as the mean, minimum and maximum) about the JND values it contains is desired, there are two choices. The entire map can be copied to CPU memory and the appropriate values calculated, but this can be slow. The GPU can be used for this task as well. First one places the minimum, maximum and total sum of the intensity values of each pixel into separate channels (this is already the case if the image is using four channels but is monochrome). Then, by repeatedly downsampling the image, the desired statistical values of two-by-two pixel regions can be found and stored in the proper channels of the smaller image. Care must be taken during the downsampling to

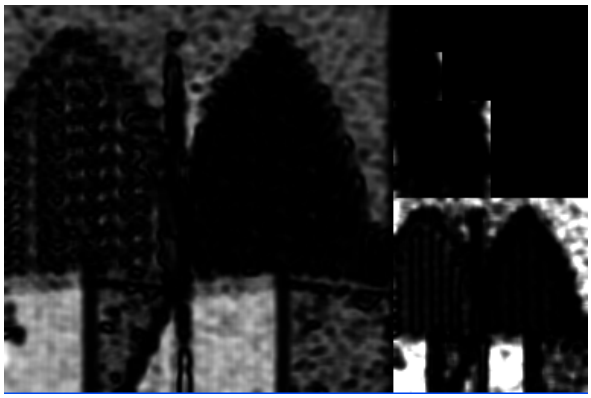

Figure 9. The result of the Difference Pyramid stage. The intensity has been scaled down significantly.

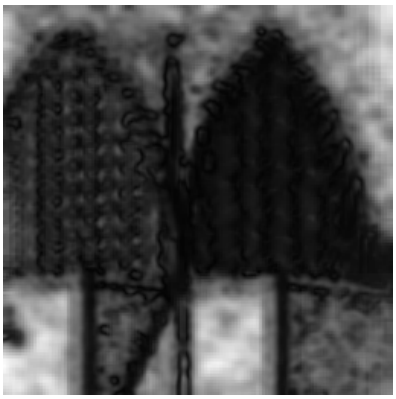

Figure 10. The result of combining the levels of the Difference Pyramid. This is the JND Map depicting likely areas of detectable difference between the two input images. The intensity has been scaled down.

not include pixels at texture coordinates that are outside the actual texture. The process is complete when the result is a single pixel with the minimum, maximum and total intensity in three of its four channels. This single pixel can then be copied to the CPU and the mean intensity calculated, saving bandwidth and CPU processing time.

4. COST ANALYSIS

The total cost of running the VDM on the GPU depends on several factors. While some of the GPUs can support a very large number of instructions per pixel, high instruction count programs do take longer to run. Texture operations can also be significantly slower than basic operations because they must fetch data from memory. Another consideration is the ability of the various stages to process multiple inputs. The first several stages can handle up to four monochrome input images at a time, allowing a significant savings if used. Table 1 shows the instruction count of the programs for each stage using the NVIDIA FX architecture, as well as the multi-image processing capabilities.

Given the instructional cost of these stages, one must then consider the time it takes to execute them. The first time a stage is executed it incurs a penalty for compiling and loading the program. Repeated application of the program during the same session does not incur this penalty. Timing was based on repeated execution of the stages, so the initial execution time of a stage was ignored. Accurate per-stage timing information is difficult to obtain, but Fig. 11 shows the relative amount of time spent in each stage for a typical two-image input.

Table 1. The instruction count of each stage's fragment program.

Stage	Basic Ops	Texture Reads	Total Ops	Multi-Image
Optics	35	9	44	Yes
Gaussian	99	25	124	Yes
Contrast	8	3	11	Yes
Energy	245	121	366	No
CSF and Transducer	51	3	54	No
Pooling	99	25	124	No
Difference	12	2	14	No
Difference Map	22	8	30	No
Statistics	14	4	18	No
Border Creation	2	1	3	N/A

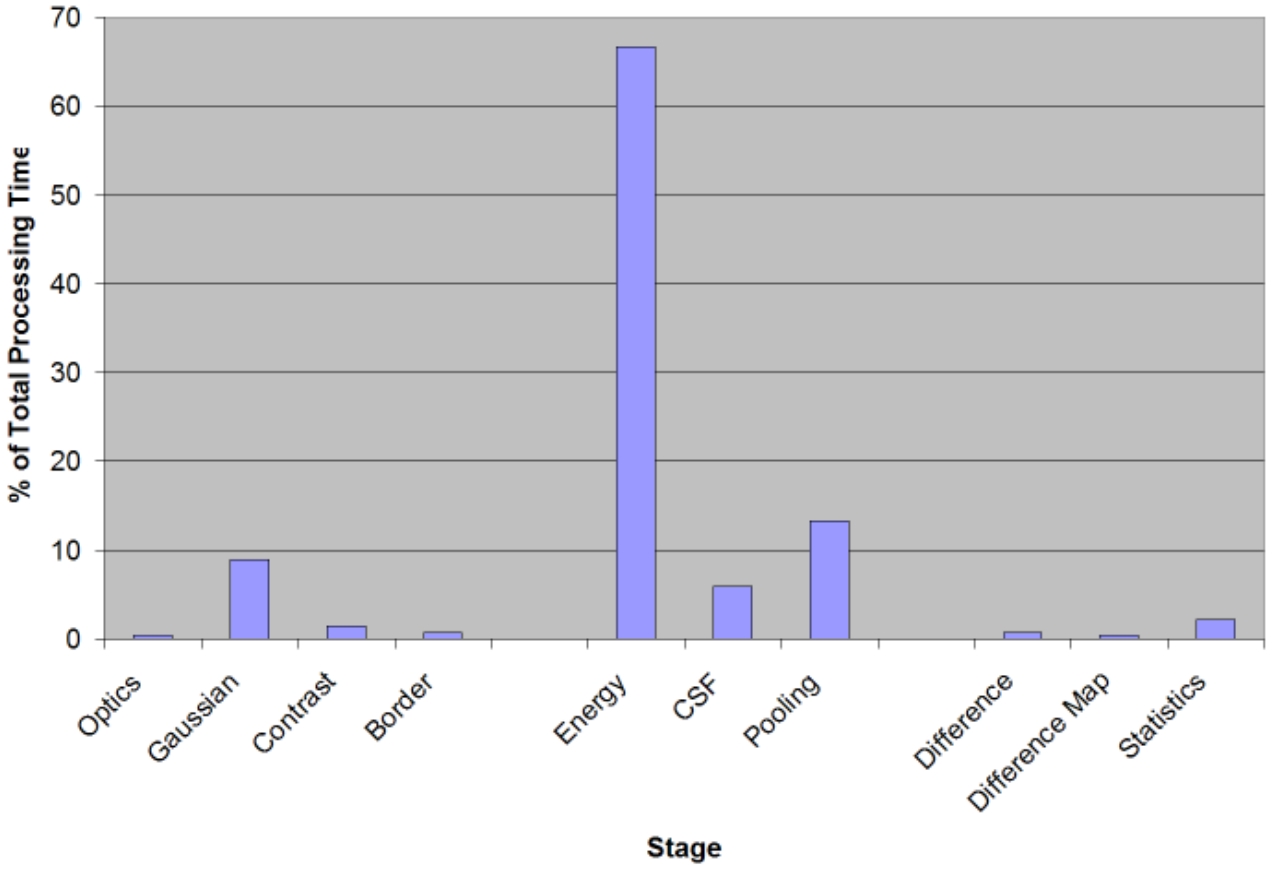

Figure 11. How total processing time is distributed among the stages.

Table 2. The overall running time of each implementation for a two-image comparison. * Due to an error in the Unix implementation, this run uses a slightly simpler method of CSF calculation than the other implementations.

Implementation	Time (ms)	
	256x256	512x512
Unix Software	6060*	24750
Windows Software	3734	12125
Windows GPU (Go700)	680	2490
Windows GPU (5900 Ultra)	320	1140

Processing time is easier to determine when considering the execution of the entire metric. Two test machines were used to examine running time for the GPU implementation, a laptop and a desktop. The laptop machine is a 1.4 GHz Intel Pentium M with 1 GB of RAM, and its GPU is a NVIDIA Quadro FX Go700 with 128 MB of RAM. The desktop machine is a dual 2.8 GHz Intel Pentium 4 with 1 GB of RAM, and its GPU is a NVIDIA GeForce FX 5900 Ultra with 256 MB of RAM.

To find the speed improvement a comparison was made against two reference software implementations. One implementation was developed under Unix,[10] and the other implementation was a Windows port of the Unix implementation. The Unix machine used was a SunBlade 2000. The Windows machine used is the same machine used for the GPU tests. The parameters for the software implementations were set to match the GPU implementation. As can be seen in Table 2, the GPU-based solutions offer vastly improved speed, with more than an order of magnitude difference between the fastest software implementation and the GPU implementation running on the GeForce Fx 5900 Ultra.

GPU texture memory did not become a factor at any time in these tests. Assuming a good layout of texture memory, a 128-bits-per-pixel pyramid with a base of 512x512 pixels should require approximately 5.33 MB if created with independent images. Given the number of pyramids and images necessary for each stage, the texture memory required for a comparison of two 512x512 images is roughly 76 MB. Thus even the Laptop GPU with 128 MB is sufficient.

5. CONCLUSION

Modern GPUs can clearly function as the processor for the primary tasks of complex software like the Sarnoff VDM. With their specialized hardware, image-based algorithms or other matrix-heavy computations can actually perform much faster on the GPU than their CPU counterparts without fear of lost accuracy.

The implementation of the Sarnoff VDM on a GPU provides a solution fast enough to be used interactively, especially when comparing images synthesized on the GPU itself. Additional improvements to the implementation, in terms of both algorithms and programming techniques, would serve to improve the running time. None of these, however, were necessary for the initial implementation effort. It is also likely that future generations of GPUs will trigger large speed improvements.

Implementing the Sarnoff VDM on a GPU makes it possible to develop a number of interesting applications. An interactive version of the VDM allows the user to explore, in near real time, the significance of various pictorial artifacts. If modified for motion, a VDM on a next generation GPU could potentially be used to process live video. A graphics card with a VDM implementation on board makes it practical to integrate complex perceptual calculations into interactive graphics applications. Novel versions of the VDM itself can be developed and tested much more quickly on a GPU rather than a CPU. Moving beyond the VDM paradigm, other models of human vision can be more easily explored, and the increasingly powerful GPU can become a new platform for vision research.

ACKNOWLEDGMENTS

This research was conducted at the Digital Technology Center at the University of Minnesota. It was funded by *NSF stuff*.

REFERENCES

1. J. Bolz, I. Farmer, E. Grinspun, and P. Schroder, "Sparse matrix solvers on the gpu: Conjugate gradients and multigrid," in *Proceedings of SIGGGRAPH*, pp. 917–924, 2003.
2. J. Kruger and R. Westermann, "Linear algebra operators for gpu implementation of numerical algorithms," in *Proceedings of SIGGGRAPH*, pp. 908–916, 2003.
3. K. Hillesland, S. Molinov, and R. Grzeszczuk, "Nonlinear optimization framework for image-based modeling on programmable graphics hardware," in *Proceedings of SIGGGRAPH*, pp. 925–934, 2003.
4. T. Purcell, I. Buck, W. Mark, and P. Hanrahan, "Ray tracing on programmable graphics hardware," in *Proceedings of SIGGGRAPH*, pp. 703–712, 2002.
5. N. Carr, J. Hall, and J. Hart, "The ray engine," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, 2002.
6. M. Harris, W. Baxter, T. Scheuremann, and A. Lastra, "Simulation and computation: Simulation of cloud dynamics on graphics hardware," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, 2003.
7. J. Lubin, "A visual discrimination model for imaging system design and evaluation," in *Vision Models for Target Detection and Recognition*, E. Peli, ed., pp. 245–283, World Scientific, 1995.
8. W. Mark, R. Glanville, K. Akeley, and M. Kilgard, "Cg: A system for programming graphics hardware in a c-like language," in *Proceedings of SIGGGRAPH*, pp. 896–907, 2003.
9. W. R. Mark and K. Proudfoot, "The f-buffer: A rasterization-order fifo buffer for multi-pass rendering," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pp. 57–64, ACM Press, 2001.
10. B. Li, G. Meyer, and R. Klassen, "A comparison of two image quality models," in *Human Vision and Electronic Imaging III*, B. E. Rogowitz and T. N. Pappas, eds., **3299**, Proc. SPIE, (San Jose, California), 1998.