LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

UCRL-TR-217422

Evaluating Mobile Graphics Processing Units (GPUs) for Real-Time Resource Constrained Applications

J. Meredith, J. Conger, Y. Liu, J. Johnson

November 30, 2005

This work was performed under the auspices of the U.S. Department of Energy by University of California, Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

Evaluating Mobile Graphics Processing Units (GPUs) for Real-Time Resource Constrained Applications

Jeremy Meredith (PI), Jim Conger, Yang Liu
John Johnson (Programmatic Supervisor)

This work was performed under the auspices of the U.S. Department of Energy by University of California, Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

Executive Summary

Modern graphics processing units (GPUs) can provide tremendous performance boosts for some applications beyond what a single CPU can accomplish, and their performance is growing at a rate faster than CPUs as well. Mobile GPUs available for laptops have the small form factor and low power requirements suitable for use in embedded processing. We evaluated several desktop and mobile GPUs and CPUs on traditional and non-traditional graphics tasks, as well as on the most time consuming pieces of a full hyperspectral imaging application. Accuracy remained high despite small differences in arithmetic operations like rounding. Performance improvements are summarized here relative to a desktop Pentium 4 CPU:

		NVIDIA 6800GT	*NVIDIA Go6800Ultra*	*ATI x300 Mobility*
Artificial Benchmarks	Geo-registration (static)	404.3×	367.5×	69.3×
	Geo-registration (video)	61.5×	50.3×	30.4×
	Text Indexing (peak)	4.6×	5.3×	–
	Convolution (procedural, small kernel)	2.4×	2.0×	0.9×
	Convolution (procedural, large kernel)	29.1×	25.6×	–
	Convolution (2-texture, small kernel)	2.7×	1.7×	0.9×
	Convolution (2-texture, large kernel)	12.4×	11.8×	–
HSI Covariance Matrix and Matched Filter		–	**26.5×**	–

1 Introduction

Revolutionary advances in computer graphics technologies, driven by the needs of 3D gaming, have resulted in specialized SIMD floating point rendering engines known as Graphics Processing Units, or GPUs. These GPUs are programmed via graphics libraries such as OpenGL, but have very general programming architectures. These cards are handily exceeding Moore's law performance predictions and are expected to continue to do so for some time. The size and cost-competitive nature of the gaming industry

combine to make these systems extremely affordable. Today, GPUs with over 100GF compute power can be bought for under $300, and they are expected to increase to around 1000GF for about that same cost in the next two years. Within the past year, mobile GPU technology has reached nearly the same capability as workstation GPUs. The small form factor and lower power requirements of the recently released NVIDIA GeForce Go 6800 signals the potential for GPUs to provide a low cost solution for embedded processing.

This project consisted of two phases. The first phase benchmarked the performance of mobile GPUs on a variety of algorithms. These algorithms covered a wide scale, from those that match more traditional graphics problems for which GPUs were designed, to those that fall in the realm of general purpose programming well outside the normal intent of GPUs. The mobile GPUs we evaluated included both high-performance and low-power varieties; the high performance GPU evaluated was the NVIDIA GeForce Go 6800 Ultra, and the low-power GPU was the ATI Radeon x300 Mobility.

The second phase ported a full-scale hyperspectral imaging (HSI) analysis code to the GPU and evaluated its performance on the NVIDIA card. This port is a hybrid GPU/CPU implementation, where only the algorithms that were determined to be the most time consuming and the best candidates for re-implementation were ported to the GPU. LLNL's current HSI application was used for comparison when benchmarking the results.

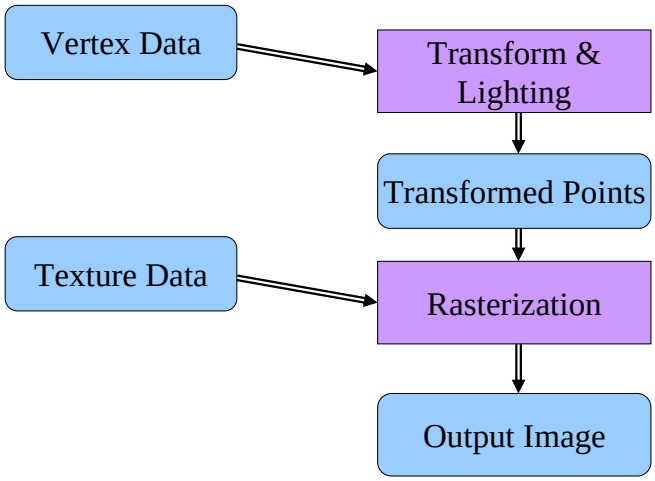
The rest of this report is organized as follows. Section 2 covers background material on GPUs – their origins in computer graphics, their architectures, and the nature of using them for general-purpose programming. Section 3 details the first phase of the project evaluating mobile GPUs as a whole. Section 4 covers second phase of the project porting the hyperspectral imaging application, including background on HSI, the approach taken for porting to GPUs, and the results. Section 5 contains reference shader source code.

2 Graphics Processing Units

2.1 Computer Graphics

The traditional purpose of real-time computer graphics has been to render realistic imagery, and the traditional approach to doing so has been to take three-dimensional models of the world, transform them into image space as seen by a camera, and apply lighting and texturing to these models to make them seem truly solid and three-dimensional.

The graphics processing pipeline reflects this. The models are represented as collections of polygons with two or three dimensional vertices. Along with the models, the light sources and camera are inputs to the first processing stage (Transform and Lighting). The result of this stage is combined with texture data, or images, to input to the second stage (Rasterization), where the final pixels are rendered to the screen. This pipeline is shown below.



2.2 GPU Architectures

The first generation of 3D graphics cards accelerated merely the rasterization stage, rendering the final image to the frame buffer drawn to the screen. However, later generations of graphics cards started to also accelerate the transform and lighting stage. At this point, though programmability was limited at best, the term Graphics Processing Unit, or GPU, was first applied. A few generations later, graphics cards started to show true programmability at both processing stages; inputs were no longer constrained to static light sources and transformation matrices, but could now be real programs. Finally, the most recent generations have started to allow real programming constructs like loops and branch instructions, and they have begun to incorporate true 32-bit floating point data storage and arithmetic.

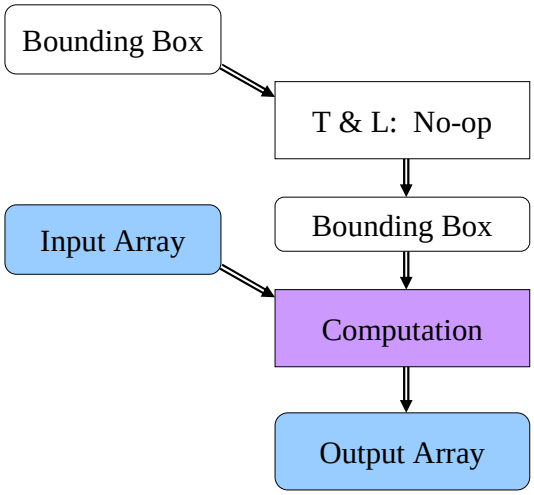
It is these last generations of graphics cards that have shown real promise as an alternative processor for general-purpose computing. For one reason, processing speeds of GPUs have grown at a rate much faster than CPUs; current generations achieve 200 GFLOPS in a single chip, a factor of three faster than those of the previous year. They are able to accomplish this partly because there is a huge mass market for graphics cards driving competition. However, there is a fundamental aspect to computer graphics that helps enable this as well: most graphics operations are highly parallelizable, and processing speed increases can be often be achieved by adding more parallel pipelines.

2.3 General-Purpose Programming

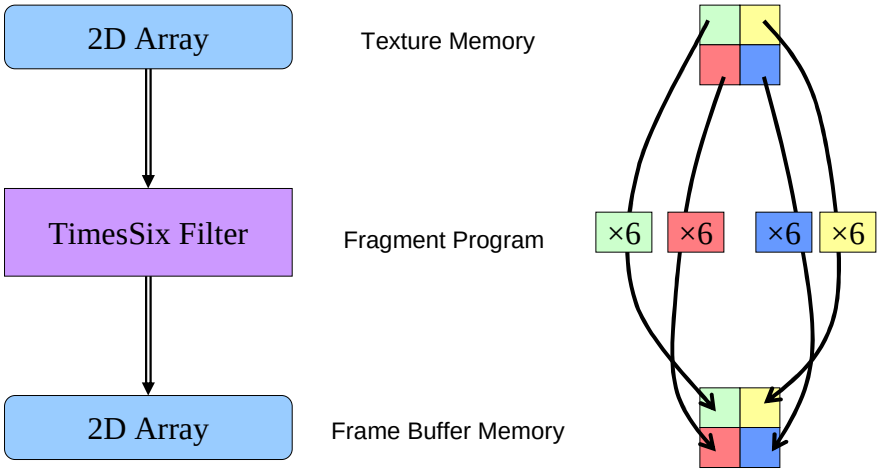
Suppose you wanted to take an array of values and multiply each one by the value six. On a standard CPU program, this is trivial: read each value from memory, multiply it, and write it back to the same location. On a GPU, this is not quite as simple, because in general it is not possible to read from and write to the exact same locations in graphics memory; this situation would require separate input and output arrays.

How would this new “multiply-by-six” operation fit into the standard graphics pipeline? The simplest way of doing it involves using texture memory as the input array, the output image (“frame buffer”) as the output array, and the rasterization stage of the pipeline as the computational kernel. In this case, the transform and lighting stage is trivial and

merely sets up the rasterization stage so that every pixel in the output array is visited exactly once, and the computational kernel is executed exactly once for each output pixel. This can be seen below.



Thus, for our example operation, the computational kernel for this operation becomes: (1) read the input value corresponding to my output location, (2) multiply it by six, and (3) return this new value to the rasterizer, which will write it to the correct output location. Below we see the pipeline with the irrelevant portions omitted, and with 2×2 input and output arrays causing an execution of our computational kernel exactly four times.



It is quite possible to read from more than one input location, but the amount of data actually read and written in each pass affects its performance, and each fragment can only write data to its own location in the output image(s).

A GPU is not self-sufficient; these graphics chips plug in as add-on cards to an existing computer and still require a CPU to direct their actions. Because of this, for a GPU to act

as a co-processor, it must rely on the CPU to send it data to process and retrieve the result when the computation is finished, and performance can thus be limited by speed of the bus. For chained computations, the output array must be either be copied to an input array or re-targeted as one.

2.4 GPUs as Co-processors

Interacting with a GPU is done at a high level with a graphics library like OpenGL or DirectX. These APIs set up the input and output buffers and provide the high level control needed to make use of the graphics cards.

These APIs, however, were not originally designed to provide the programmability needed to execute anything but the most basic kinds of computation on the GPUs. There are several languages in existence that are designed to compile to GPU machine code, such as Cg and GLSL. These languages are often called "shader languages", and the programs are thus called "vertex shaders" (or "vertex programs") for the transform and lighting stage, and "pixel shaders" for the rasterization stage. Note that a "fragment" is what becomes a pixel after successful rasterization, so pixels shaders are commonly called "fragment shaders" or "fragment programs". In our above example where we multiplied the elements of an array by six, our computational kernel was a fragment shader.

There are two primary kinds of buses over which the CPU and GPU communicate – AGP and PCI Express (PCI-E). AGP is the older of the two, and not only has lower bandwidth, but has asymmetric performance, and reading data back from the GPU is much slower than sending it. Luckily, PCI-E is quickly becoming the prevalent standard and seems to alleviate most of the problems with AGP.

For the case where data is generated as the output of one computation step within the GPU and needs to be re-used as the input to another step, again there are multiple options. The older standard for rendering offscreen images was called "pbuffers" (short for pixel buffers), and data had to be copied from the pbuffer to a new input texture. The newer standard, only widely available very recently, is called "FBOs" (short for Frame Buffer Objects), and these allow a rendering target to be re-used as a texture directly without suffering a penalty by waiting for a memory copy.

More details of the approach we used to port the HSI application to GPUs are found in the section on hyperspectral imaging.

2.5 Related GPGPU Work

General purpose programming of GPUs (GPGPU) has become a more common research area recently with the dawn of programmable GPUs. The GPGPU web site [17] highlights many of the recent developments using GPUs for general purpose computation and is often a resource of information on the field. The book *GPU Gems* [7], ostensibly devoted to graphics programming, had a section devoted to general purpose computation, and *GPU Gems II* [14] has almost half of its content devoted to the topic.

Recent research has been investigating the feasibility of GPUs for high performance computing in a variety of areas. Among the applications are 3D fluid simulation [4],

database operations [11,1], computational geometry [5,10], computer vision [16], signal processing [13], scientific computation [2,9], and knowledge discovery [3,12].

There is even research into using GPUs in computational clusters [6], and the concept of using novel architectures in clusters extends beyond the graphics processor, as seen by the Playstation2 cluster at NCSA [19].

3 Mobile GPUs

3.1 Introduction

One goal of this project was to evaluate mobile GPUs for their performance, and this requires comparing algorithms running on mobile GPUs with pure software implementations. However, as desktop GPUs are most commonly targeted for benchmarking in prevalent GPGPU research, it is also worthwhile to compare mobile GPUs with desktop GPUs. This will allow us to evaluate external GPGPU results for their applicability to the mobile processors as well.

In this section, we first comment on issues related to GPU benchmarking, and we then examine a variety of benchmarks on a variety of both CPUs and GPUs.

The following table lists some of the performance characteristics of the cards tested:

Manufacturer	*Card*	*No. of pipelines*	*Core clock*	*Bus*
NVIDIA	GeForce 6800 GT	16	350 MHz	AGP
NVIDIA	GeForce Go 6800 Ultra	12	425 MHz	PCI-E
ATI	Radeon x300 Mobility	4	300 MHz	PCI-E

3.2 GPU Benchmarking

When comparing GPU performance to CPU performance for a particular algorithm, one important question is what kind of implementation we should be comparing against.

For example, suppose we want to evaluate a GPU for its performance with FFTs. It is assumed no GPU implementation exists, and we must write it from scratch. However, we have a number of options for the CPU variant.

For example, we could write an implementation that is literally identical to the GPU one. This could be used to compare CPUs and GPUs at a very low level, but for algorithms like FFT where the GPU implementation looks nothing like what one would reasonably use for a CPU, it makes very little sense.

As another example, we could simply hand-code a CPU implementation as we normally would if we were to write it from scratch. This often makes sense because we have put about the same effort into trying to optimize both the CPU and GPU variants and appears to make the most fair comparison. (A related question is how much compiler optimization to allow for the CPU variant – since CPU compiler optimization technology is often mature, an order of magnitude improvement for the CPU is often seen merely by adding a single command-line flag. Note, however, that there is less chance to heavily

optimize GPU code as the instruction sets are rather severely restricted, so one might argue that disallowing compiler optimization for the CPU would put it at an artificially large disadvantage.)

A third example would be to use an industry-standard CPU implementation, such as LAPACK or Matlab. This also makes sense, but some implementations, though common, may either simply be slow or might make substantial accuracy tradeoffs for improved speed.

And finally, there exist some algorithms that will profile and tune themselves to a particular CPU and cache and can achieve orders of magnitude speedups relative to a naïve hand-coded implementation. FFTW is one example of this.

For many benchmarks, a case could be made for picking any one of these flavors of CPU implementation to compare GPUs with. As a result, the best practice is simply to disclose what kind of CPU implementation you have used.

One other note to mention here – in the event that the GPU is performing at only a similar level to the CPU, and it appears that little benefit might exist from porting that particular algorithm to the GPU, it can be worth remembering that the CPU is often free to perform other tasks while the GPU is performing the computation. This means that even with no appreciable improvement on the GPU, a worst case scenario might actually be a doubling of performance, and this might be a viable option in situations dual-core or dual-CPU systems are not.

3.3 Geo-Registration

3.3.1 Introduction

Geo-registration refers to removing the distortions from geographic imagery. In this context, we are mapping aerial imagery taken from a known viewpoint to a zenith viewpoint. For some applications, this involves static imagery, and for others, motion video is continually mapped from differing viewpoints.

3.3.2 Approach

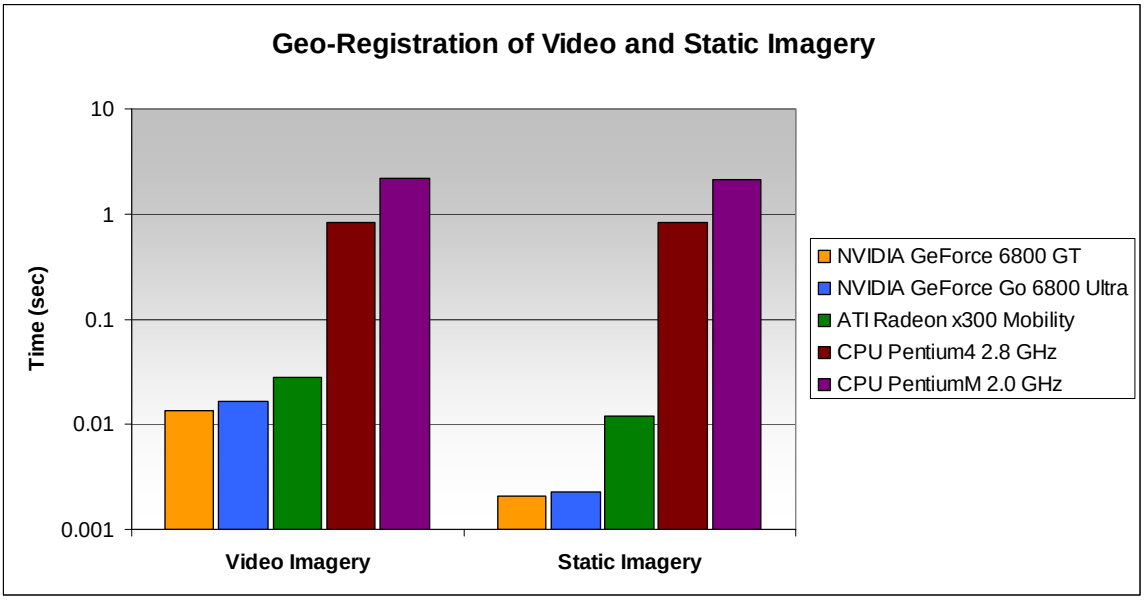
For the CPU and GPU implementations, this corresponds roughly to drawing a texture-mapped polygon to an offscreen buffer repeatedly from a variety of angles. To achieve motion video geo-registration (instead of merely static imagery), the texture is updated for every rendered image.

As usual, for the GPU approaches, we used the OpenGL library. No advanced fragment or vertex shaders were required.

For the CPU implementation, we used the Mesa 3D rendering library. This contains significant optimizations for the Pentium 4 architecture, including hand-coded assembler, and would far surpass in performance any other software rendering algorithm we would have written from scratch. As it shares an API with OpenGL, Mesa is probably the most common 3D software rendering library used in practice.

3.3.3 Benchmarks

In both cases, a 1000×1000 image was registered for every rendering. Registration angle was varied continuously across 180° and rendered to a 6 million pixel output buffer. For the video imagery test, a new texture was generated and used for every output frame.



Note that the figure above shows the comparisons on a logarithmic scale.

3.3.4 Results

One observation to make is a comparison between the GPU implementations and the CPU implementations. For video imagery, the GPU variants were faster than the CPU by a range of 30x to 158x. For static imagery, the GPU variants were faster than the CPU by a range of 70x to 1040x. Note that the Mesa library was not optimized well for the Pentium M; if we restrict the comparison to the only against the Pentium 4, the GPU variants are still 30x to 60x faster for video imagery, and still 70x to 400x faster for static imagery.

Also of interest are comparisons among only the GPUs. The fastest is the NVIDIA GeForce 6800 GT, which is the desktop video card. However, the laptop GeForce Go 6800 Ultra is only 10% to 20% slower in this application, despite having only 12 pipelines versus the desktop's 16. The higher clock rate of the mobile unit probably help the static test almost as much as the extra pipelines, and the PCI-E bus probably helps the video test quite dramatically as well.

The extremely low power ATI Radeon x300 Mobility still performs quite admirably. With only 4 pipelines and a much lower clock, it is still only a factor of two slower than the desktop 6800 GT for video geo-registration. However, in the static test it was almost 6x slower than the 6800 GT. The close performance on the video test illustrates the importance of the bus speed when streaming data onto a GPU.

3.4 Document Indexing

3.4.1 Introduction

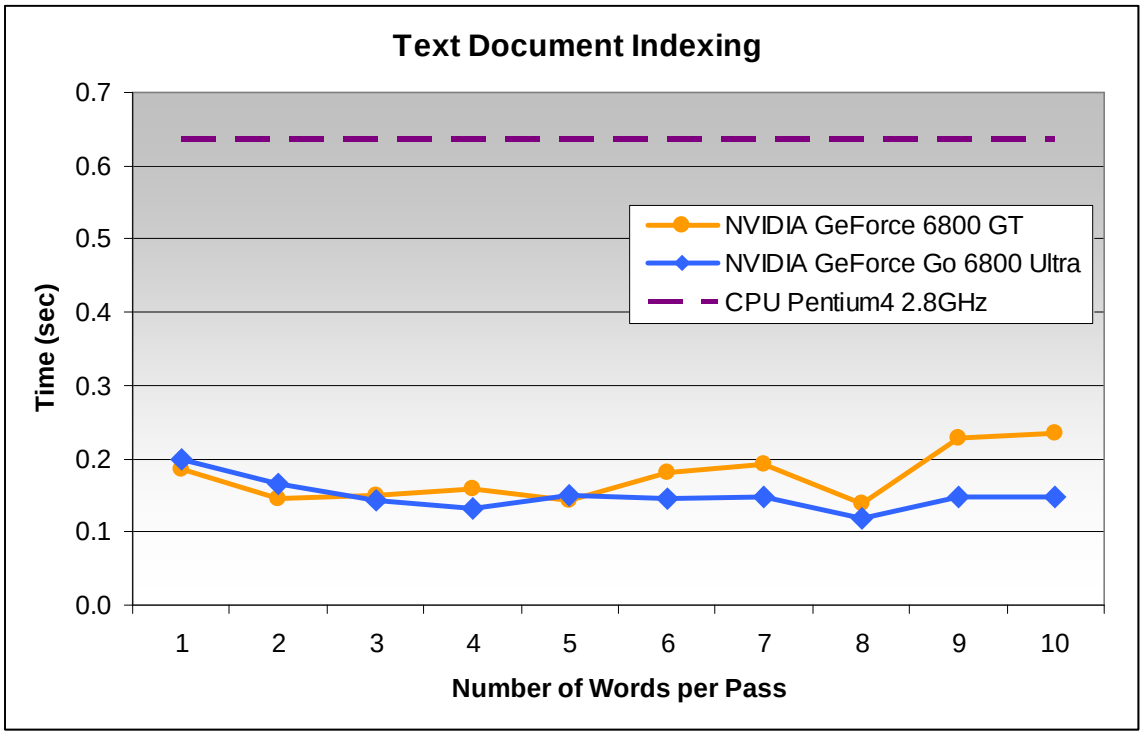
This test is a GPGPU application which converts a document into a fixed-length vector. The input to the application is a plaintext document, and the output is a count of how many times each dictionary word appears in the document. The words in the document and dictionary are stemmed, so that different forms of the same word appear only once. For example, "jump" and "jumped" both map to the same location in the output vector.

3.4.2 Approach

For both the CPU and GPU versions, a hand-coded brute-force lookup was used, where the dictionary words were each compared with the document words. For the GPU variant, the document was broken into small chunks and streamed onto the GPU as texture data, while the dictionary itself was stored in on-card GPU texture memory. Partial results were accumulated in the output buffer using alpha blending. The NVIDIA Cg language and libraries were used for the fragment program.

3.4.3 Benchmarks

For this particular test, the dictionary contained 42,625 stemmed words, with semantically meaningless words such as articles and pronouns removed. The input documents were 4,348 medical abstracts. Stemming was performed before indexing, and so only the indexing times are measured here. The number of words streamed into the GPU were varied from 1 to 10 per rendering pass.



3.4.4 Results

Both GPUs tested are about 5x faster than the CPU on average. Since no matter how many words are evaluated per pass, the total workload is the same, the variation within each card must be due to other factors, such as bandwidth/latency characteristics of the bus, cache performance, and memory layout. For example, both the Go6800Ultra and the 6800GT experienced a reproducible local minimum at 8 words per pass, which would result in a power-of-two size texture internally, and GPU architectures have historically been tuned to achieve the highest performance on textures like this.

Also note that the 6800GT performance drops off with larger chunks of input data and the Go6800Ultra does not. As the two GPUs have different buses, different clock speeds, and different pipelines, it is hard to determine with more specificity the exact cause. However, it is significant that increasing performance by streaming the data in instead of working with data already on the card is not a result unique to this application nor is it relegated to the 6800GT alone – the same behavior is also seen on the Go6800Ultra in the HSI application benchmarks. Thus, this benchmark simply appears likely to exhibit the behavior and the differences between the cards affect only the degree to which it manifests itself.

A related observation is that, in many cases, the mobile Go6800Ultra chip bests the desktop 6800GT, despite having fewer parallel processing pipelines. The simplest explanation is that this application makes poor use of the parallelism on the GPU, and the higher clock speeds of the mobile chip are more advantageous than more parallelism.

3.5 Convolution

3.5.1 Introduction

This test performs a two-dimensional floating-point convolution of one image with a smaller kernel image. While the kernels can be arbitrary, in practice they can often be generated procedurally. Examples of procedural kernels in image processing include a median filter, a Gaussian blur filter, or an edge-detection filter.

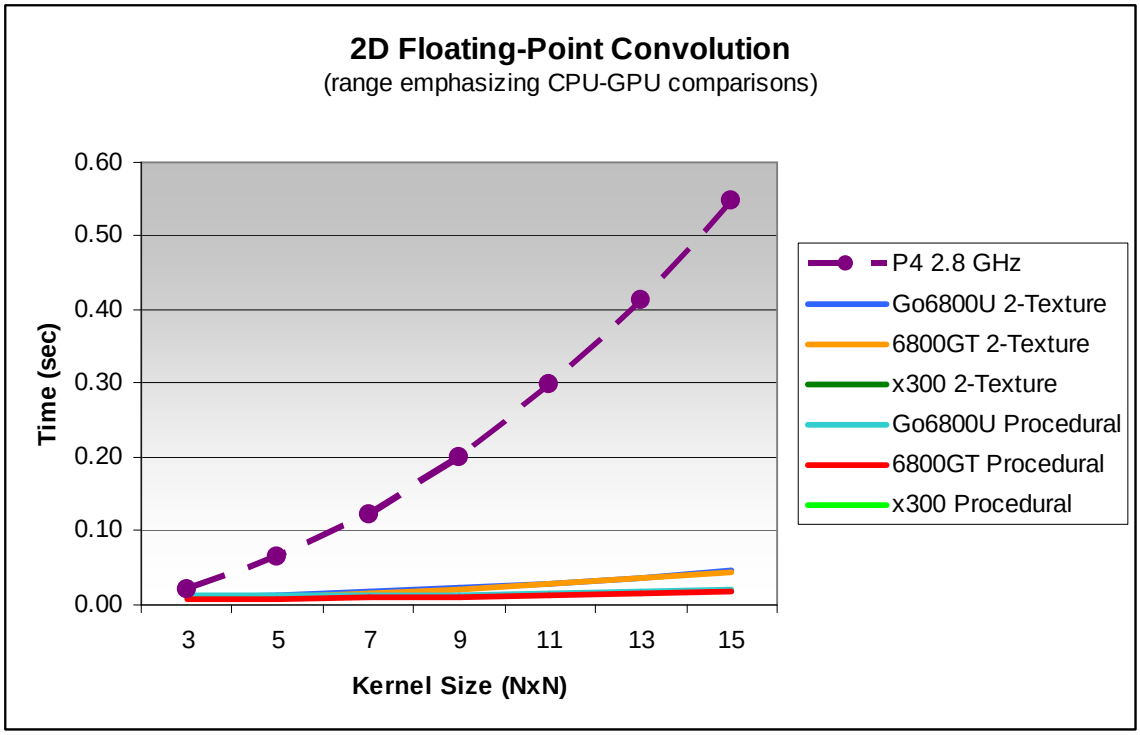
3.5.2 Approach

On the CPU, the implementation is a straightforward convolution of two floating-point input arrays. On the GPU, the input image and the kernel both reside in texture-memory, resulting the “2-texture” algorithm. However, as some kernels are not arbitrary, we also implemented a procedural kernel, in this case a median filter.

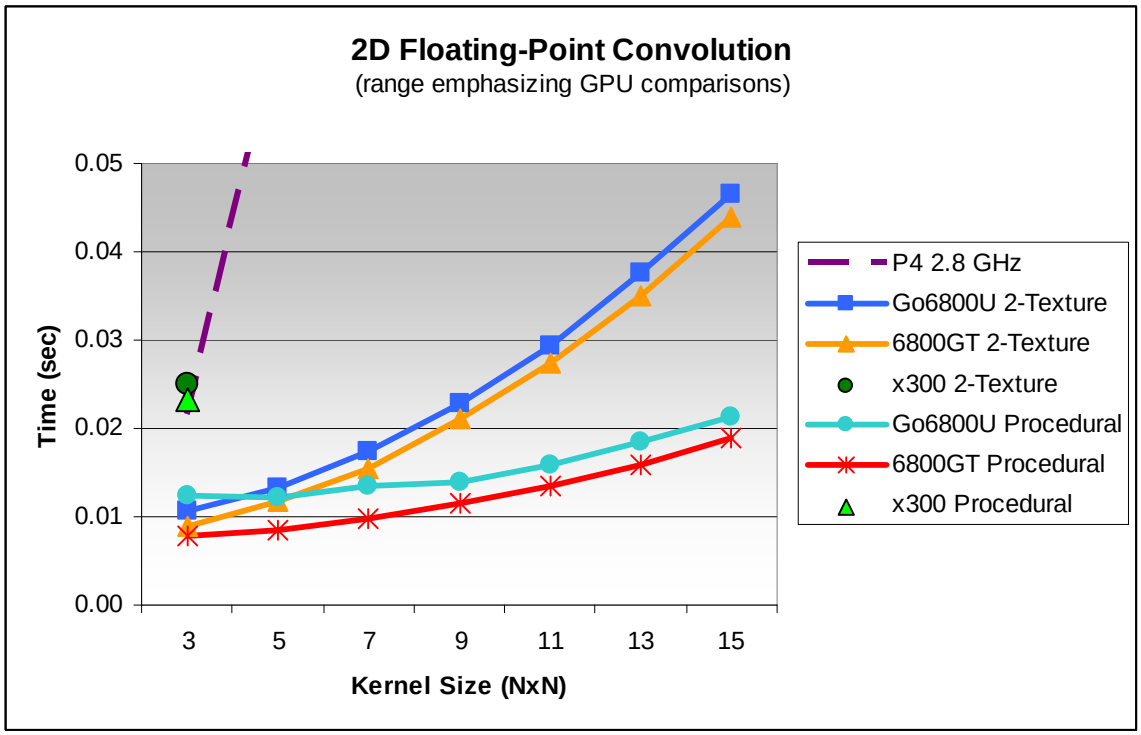
3.5.3 Benchmarks

All benchmarks were run on a 512×512 input image with full floating-point precision while the kernel size was varied from 3×3 to 15×15. The two charts below show the same data at different scales.

The first chart has an expanded range to emphasize comparisons between the GPU implementations and the CPU implementation.



The second chart, below, has a reduced range to show the GPU variants in comparison with one another.



3.5.4 Results

For this application, the higher performing GPUs clearly hold an advantage over CPUs. While not a traditional operation for a graphics card, convolution nevertheless makes use of similar image processing operations for which the GPUs are designed. On the NVIDIA GPUs, the two-texture implementation ran up to 12x faster than the CPU, and the procedural implementation up to 30x faster than the CPU.

Unfortunately, because we were using the Cg language from NVIDIA, we were limited in the features we could use for the ATI GPUs, and thus the ATI Radeon x300 was not able to scale past the 3x3 kernel size. However, it is important to note that even having a much lower clock speed and many fewer pipelines, this card nevertheless performed nearly as well on this GPGPU task as the optimized CPU. Furthermore, if one were to extrapolate the NVIDIA GPU results to the ATI card for higher kernel sizes, it is likely the x300 would have handily beaten the GPU had we been able to rewrite the benchmark using another shader language.

Comparing between only the desktop 6800GT GPU and the mobile Go6800Ultra GPU, we see that the mobile GPU runs about 5% to 18% slower for the two-texture version, and about 12% to 36% slower for the procedural version. Here, again, we see the tradeoff for fewer pipelines but a higher clock speed.

Slightly more interesting is the fact that for most kernel sizes, the procedural implementation is much faster than the two-texture implementation. What this implies is that too much reading from texture memory can become very expensive, and in many cases one could improve performance by replacing texture lookups with more computation.

3.6 Analysis

With the NVIDIA GeForce Go 6800 Ultra, it is clear that very few sacrifices need to be made for mobile GPUs compared with even high end GPUs; a high-end mobile GPU performs almost as well, and sometimes better, than a high-end desktop GPU, depending on the test.

One consideration for embedded applications is power usage. While exact power usage statistics are difficult to obtain, we can provide reasonable estimates for our cards. First, note that while our benchmarks used the 6800 GT, a 6800 Ultra desktop GPU *does* exist; however, it has such high power requirements (maximum draw of 110W) that for practical reasons no available test systems had it installed. In real-life tests, the 6800 GT we used draws about 80% the power of the desktop 6800 Ultra (and has about 80% the clock speed as well), putting its maximum draw around 88W. By comparison, the Go 6800 Ultra GPU has a clock speed comparable to the desktop 6800 Ultra, and its maximum power draw is relatively low at 66W.

For lower power usage embedded applications, it is worth pointing out that a mobile Go 6800 (*not* an Ultra) is also available, which draws at most 27W. Finally, as the above benchmarks show, the ATI Radeon x300 can perform admirably in some applications, with power usage far below any of the other options we explored in this report.

3.7 Future Technologies

A new generation of graphics cards appears regularly, and it comes as no surprise that as of this writing, the next generation of GPUs is already here. From NVIDIA comes the GeForce 7800 series, with the high-end desktop version containing 24 pixel pipelines – a 50% increase over the desktop 6800 series. ATI has its new generation as well, with updated versions of the x300 and x800 called the x1300 and x1800, the latter of which has a 650 MHz core clock.

Concerning mobile GPUs, only the NVIDIA is here so far, with a high-end GeForce Go 7800 GTX already available. Specifications are similar to the desktop version, with 24 pixel pipelines, and the same power budget as the older Go 6800 Ultra.

4 Hyperspectral Imaging

4.1 Background

Hyperspectral instruments (HSI) typically collect between 128 and 256 spectral channels simultaneously for each pixel in the image, with two-byte precision. A single image might contain 256 × 256 × 256 × 2 = 32 MB of data, and is often referred to as a "data cube". Instruments can be tasked to collect a large number of data cubes in a short time, which results in substantial data volumes to process.

The HSI processing logic strives to extract regions that are spectrally anomalous compared with the average background. Detecting anomalies generally requires the following processing steps on every HSI cube:

1. Calibration of the data, and bad pixel repair. The raw image cube is calibrated by comparing each pixel's value in the image to the value of the same pixel when looking at one or more calibration sources with known temperatures. Typically two calibration sources are used to bound the temperature of the scene between known values for each pixel.

2. Calculation of a background model for the image. A covariance matrix is used as a concise model of the background, and captures the average relationship between each spectral band in the image.

3. Inversion of the covariance matrix. The inverse covariance matrix provides a transformation matrix that can be used to suppress background features from the image.

4. Matched filtering using reference spectra. The basic matched filter formula is as follows:

$$\mathrm{Mf} = s \times \mathrm{K}^{-1} \times (x - x_{\mathrm{avg}}) \qquad (1)$$

where s is the spectrum
K^{-1} is the inverse covariance matrix
$(x - x_{\mathrm{avg}})$ is the mean-subtracted image cube

5. A threshold is applied to the matched filter image to highlight pixels which are anomalous. Typically a threshold of two to three standard deviations is appropriate.

6. Clusters of anomalous pixels which connect are aggregated to create 'super pixels". Super-pixels represent detections of spectral anomalies, but are not considered definitive.

7. The spectrum of each super pixel (after whitening) is compared to that of library spectra. The quality of the least-squares fit of the super-pixel spectrum compared to that of the library spectra is a strong indication of the existence of that spectra in the image.

With typical desktop computers the HSI processing chain requires about 10 minutes of computation per data cube. Step 1 (calibration) is rapid for sensors which operate at very low temperature (e.g. 5° K), but can be 50% of the computational time when working with sensors that have 1% or more non-functional pixels. Steps 2 and 4 are computationally expensive, and take 40-80% of the processing time. Steps 3 and 5-7 are generally fast.

4.2 Approach

4.2.1 Introduction

The match filtering step detects the presence of known signatures from a library $\mathbf{s}_n$ in the sampled hyper-spectral data cube $\mathbf{x}$ using their spectral signatures, and is given by the following equation:

$$\frac{\mathbf{s}_n^T \Sigma^{-1} (\mathbf{x} - \bar{\mathbf{x}})}{\sqrt{\mathbf{s}_n^T \Sigma^{-1} \mathbf{s}_n}} \qquad (2)$$

The covariance matrix Σ is necessary to pre-filter the data and remove background noise. The inputs to our system are $\mathbf{x}$ and $\mathbf{s}_n$, where $\mathbf{x}$ is a 3D calibrated hyper-spectral data cube (band × sample × frame) laid out in memory in band-major order, and $\mathbf{s}_n$ is a 2D matrix (spectra × bands) of spectra and their frequency response. Throughout the computation, $\mathbf{x}$ is interpreted as a 2D matrix, and $\bar{\mathbf{x}}$ is a vector of means computed from its columns. The computation of the covariance matrix Σ and its inverse from $\mathbf{x}$ is followed by a few matrix products. The columns of the resulting 2D matrix (signatures × sample × frame) are each interpreted as a 2D image, with the respective spectral features highlighted. Equation 2 is very expensive to evaluate for our large data sets, but fortunately there are many opportunities to exploit instruction/data parallelism in each of the three principal components of this computation: covariance calculation, matrix inversion, and matrix multiplication. Each component can be implemented efficiently on a multi-processor system.

We selected the NVIDIA GeForce Go 6800 Ultra (Go6800Ultra) mobile graphics card as a hardware platform to implement these operations. The Go6800Ultra contains 12 processor cores with 256MB of 400MHz GDDR3 texture memory and is capable of 234 GFLOPS and 38.4 GB/s of memory bandwidth. In contrast, a dual-core Intel Pentium D 840 processor running at 3.2GHz achieves roughly 25.6 GFLOPS using the SSE3 extensions. We program the Go6800Ultra card using the OpenGL API and the GLSL shading language. As usual, the OpenGL function calls are responsible for ancillary tasks such as initializing data transfers, setting the state of the graphics card, and invoking

execution passes. The actual work of performing the computational task is mostly specified by GLSL shaders, which are compiled at run-time into microcode by the graphics card driver. GLSL is a high level programming language based on ANSI C, and is extended with vector and matrix types — and SIMD instructions that operate on these types — to concisely describe many of the typical operations involved in 3D computer graphics [18]. Due to the nature of the graphics hardware architecture and its associated resource limitations, the compiler is highly restrictive in the types of instructions and control flow that it allows. For example, recursion is not allowed, all loops must terminate, and all parameters to functions are call-by-value.

In general, GLSL shaders function exactly as described earlier, with only subtle restrictions. For example, we use vertex shaders to specify the area of computation for a task and to assign a 2D texture coordinate to each constituent fragment within this area, and we use fragment shaders to specify how each fragment should be processed. However, the complexity of a GLSL shader cannot ever exceed 32 temporary registers since they cannot be paged during an execution pass. During each execution pass, a fragment processor can use a fragment's texture coordinates as a virtual memory address to read data from up to 16 input textures and write data to 4 output textures. Computational tasks that cannot fit within the limitations of a single execution pass must obviously span across multiple fragment shaders and execution passes. Since large memory transactions are very expensive across the PCI-E bus (PCI-E has only 4 GB/s of bandwidth), we attempt to keep intermediate data resident in texture memory between passes and copy out only the final output data.

4.2.2 Covariance Matrix

The covariance matrix Σ is a symmetric matrix given by:

$$\sigma_{ij} = \mathrm{cov}\left(\mathbf{x}_i, \mathbf{x}_j\right) \equiv \left\langle \left(\mathbf{x}_i - \mu_i\right)\left(\mathbf{x}_j - \mu_j\right) \right\rangle \tag{3}$$

where $\mu_i = \left\langle \mathbf{x}_i \right\rangle$ is the mean of variate $\mathbf{x}_i$. Calculating the covariance using Equation 3 requires a first summation pass over the data samples to compute μ, followed by a second summation pass to compute the mean dot product of the mean-subtracted variates. Fortunately, Equation 3 can also be rewritten to compute the covariance in a single summation pass:

$$\mathrm{cov}\left(\mathbf{x}_i, \mathbf{x}_j\right) = \left\langle \mathbf{x}_i \mathbf{x}_j \right\rangle - \mu_i \mu_j \tag{4}$$

When written in this form it is obvious that the covariance computation involves three dot products:

$$\mathrm{cov}\left(\mathbf{x}_i, \mathbf{x}_j\right) = \frac{1}{n}\left(\left(\mathbf{x}_i \cdot \mathbf{x}_j\right) - \left(\mathbf{x}_i \cdot \mathbf{1}\right)\left(\mathbf{x}_j \cdot \mathbf{1}\right)\right) \tag{5}$$

Each σ_{ij} is computed independently in exactly the same way; this lack of data dependency allows us to efficiently exploit SIMD parallelism in carrying out the computation. Moreover, since Σ is symmetric, only half of its entries (along with the main diagonal) need to be computed. The dot product summations occur over very large sets of numbers and require several execution passes to complete. Each pass operates on a segment of the data set and is initiated by using OpenGL calls to draw a lower triangle

that covers the computation area. The fragment processor loads the data segment and the accumulated tally (originally initialized to zero), computes dot products using the native four component dot product instruction, and then writes the updated tally into the output texture. After computing the dot products, the final execution pass combines the results together by Equation 5. The results are copied out of texture memory and then $\sum$ is formed by completing its upper triangle using data values from the lower triangle.

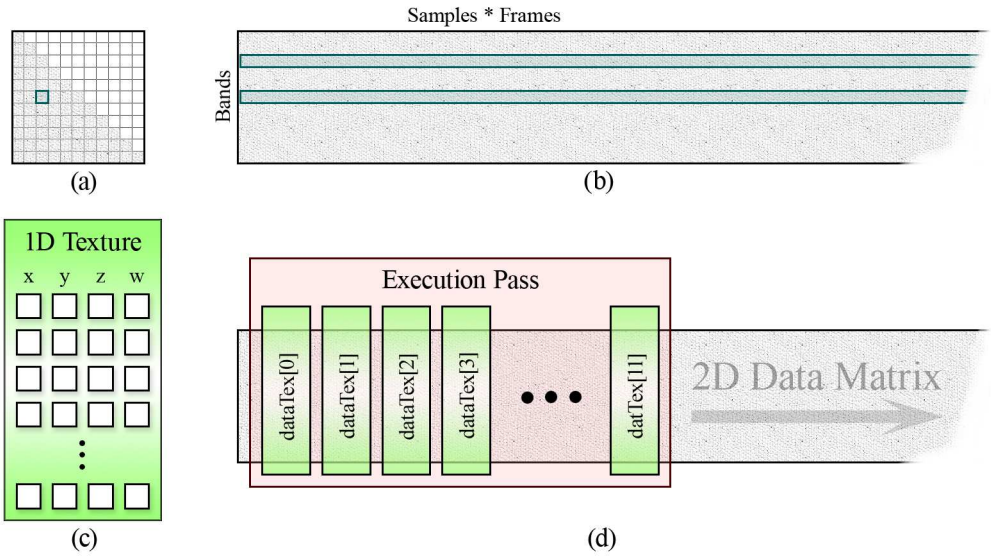


Figure 1: Data layout for covariance matrix computation. Each execution pass processes a segment of the data cube using 12 four-component 1D textures.

The GLSL shader code for dot product portion of the computation is listed in **sum_of_terms.vs** and **sum_of_terms.fs**. During each execution pass, the input data segment is stored as 12 four-component 1D textures as shown in Figure 1. The vertex shader assigns a position and a pair of texture coordinates to each fragment. The fragment shader uses its position to retrieve the current accumulated sum (line 33) and the texture coordinates to index the corresponding data elements from the textures (lines 34-45). The dot products are computed (lines 47-58), and finally summed together (lines 60-62). In each execution pass the fragment processor sums together 48 data elements for each dot product. The floating point arithmetic performed by fragment processors is not fully IEEE compliant, and may introduce rounding off errors that are magnified and become significant when many numbers together are summed together. To improve the precision, we simulate double floating point arithmetic by representing operands using two floating-point numbers and defining special arithmetic functions (lines 6-27) that operate over them. Since computing both $\mathbf{x}_i \cdot \mathbf{1}$ and $\mathbf{x}_j \cdot \mathbf{1}$ is redundant for each fragment, we only compute $\mathbf{x}_i \cdot \mathbf{1}$ (this is also redundant, but we observe that it is cheaper in practice to perform this computation while evaluating $\mathbf{x}_i \cdot \mathbf{x}_j$ than to execute its own set of summation passes). The output from the fragment processor are the intermediate results for $\mathbf{x}_i \cdot \mathbf{x}_j$ and $\mathbf{x}_i \cdot \mathbf{1}$, which both occupy two components (line 66). After the dot products

terms are computed, they are combined using the fragment shader **covariance.fs** (and the default vertex shader **vertex.vs**). The computation here is straightforward: the terms are loaded and then combined together to output the covariance value σ_{ij} and the mean μ_i (to be used later for the mean subtraction step).

4.2.3 Matrix Inverse

There are several ways to compute the matrix inverse. Gauss-Jordan elimination (with full pivoting) is the most straightforward, but not the most efficient. Furthermore, Σ is typically ill-conditioned, and its inverse matrix computed by Gauss-Jordan Elimination will likely be very sensitive to numerical error. A better approach would be to compute the singular value decomposition (Takagi factorization) of Σ, and zero out the small eigenvalues to improve the conditioning of the matrix and compute a pseudo-inverse. However, this step is even more computationally expensive. The preferred strategy would be to calculate the LU factorization of Σ and use back-substitution to obtain its inverse. However, we can use back-substitution to instead compute $\mathbf{s}^T\Sigma^{-1}$ without performing either an explicit inverse or a matrix multiplication. This is more numerically stable and also more efficient than the latter two approaches. In our system, these three routines are implemented in C (adapted from Numerical Recipes in C):

```
void gauss_jordan_inv(int n, float *data);
void svdcmp(float **a, int m, int n, float w[], float **v);
void ludcmp(float **a, int n, int *indx);
void lubksb(float **a, int n, int *indx, float b[], float x[]);
```

These routines can all be efficiently implemented on graphics cards [8], but the CPU implementations actually perform better for our small problem size. Nevertheless, the matrix inverse computation requires very little processing time relative to the covariance computation and matrix multiplication, so our focus for this step is on improving the precision of the results, rather than the performance of the computation. To this end, we can use double precision whenever appropriate to preserve the numerical accuracy. We currently use the LU factorization and back-substitution C routines in our system to compute the term $\mathbf{s}^T\Sigma^{-1}$.

4.2.4 Matrix Multiplication

We implement matrix multiplication routines to evaluate $(\mathbf{s}^T\Sigma^{-1})\mathbf{s}$ and $\mathbf{s}^T\Sigma^{-1}(\mathbf{x}-\bar{\mathbf{x}})$. In the first term, we are only interested in the diagonal of the matrix, and full matrix multiplication is required for the latter term so these computations are implemented separately using different shaders. However, in both cases, we need to compute the dot products between the column vectors from one matrix and the row vectors from another. The two implementations are very similar to the covariance computation, but we batch the data segments differently to maximize the available resources; we use 4 four-component 1D textures to store data segments from each operand matrix and during an execution pass, each fragment processor computes 4 dot products. The vectors we process here are much shorter than the ones in the covariance computation and the

numerical rounding off errors have little impact on our results, so we do not implement double-precision arithmetic for this summation.

The first term $(\mathbf{s}^T\Sigma^{-1})\mathbf{s}$ is evaluated using the shaders **norm_sqrd.vs** and **norm_sqrd.fs**, and its square root is computed by executing **sqrt_norm.fs** (using **default.vs**) and the second term $\mathbf{s}^T\Sigma^{-1}(\mathbf{x}-\bar{\mathbf{x}})$ is computed using **multiply_data.vs** and **multiply_data.fs;** the vector of sample means $\bar{\mathbf{x}}$ is subtracted from $\mathbf{x}$ before it is copied into texture memory. The matrix multiplication is performed via dot products, and the results are then normalized using the computed norms. Since $\mathbf{x}$ is typically very large, the matrix multiplication is computed in blocks and requires many execution passes.

4.3 Benchmarks

We compare the performance of our GPU implementation against our software implementation using a data cube consisting of 240 spectral bands, 240 spatial samples, and 400 frames with a library of 16 spectra. The results of the computations are stored as 16 intensity images that are each normalized to [0,255], quantized to 8-bit, and saved in the portable gray map (PGM) image format.

4.4 Results

4.4.1 Performance

Our GPU implementation generated the images in 4.25 seconds while the software implementation required 112.47 seconds, representing a speedup factor of **26.46**.

It is worth noting that this speedup factor is independent of the input data, and should remain largely the same with different data set sizes. Specifically, as the size of the data used for this benchmark is near expected levels, this same speedup factor is expected in real-life application as well.

4.4.2 Accuracy

We measure the peak signal-to-noise ratio (PSNR) of our results with the software generated images to be **33.36 dB**. As a reference, PSNR values in image compression are typically between 30 and 40 dB.

Later steps in the process find anomalous pixels in terms of a threshold at some distance from the mean. We can calculate the similarity in a very strict manner by calculating the overlap of anomalous pixels between the CPU and GPU results. Specifically, this is the number of pixels in the intersection between the set of anomalous pixels in the CPU and GPU divided by the number of pixels in the union of these sets. The average similarity across the test spectra for 2σ is **80.1%** and for 3σ is **80.9%**. (Note, that this similarity metric is strict in that it counts both extraneous *and* missing pixels as errors. If one were to simply ask, for example, how many of the CPU anomalies were also found by the GPU, the accuracy would be above **90**% for both 2σ and 3σ.)

As pointed out earlier, however, many of the pixels outside the given threshold are due to noise and magnified by small numerical errors. To estimate the true error with more accuracy, we can help eliminate this noise by comparing only the anomalous pixels that are surrounded by other anomalous pixels – i.e. by using a simple implementation of the

super-pixel concept described above as step 6 of the HSI application. For this case, the average similarity for 2σ is **86.8%** and for 3σ is **88.5%.** (Again, using a weaker definition of similarity, this means that the GPU found about **94**% of the anomalous pixels that the CPU found for both 2σ and 3σ.)

By combining and automating both step 5 *and* step 6, we can perform an even more accurate comparison of how the CPU and GPU results affect the final analysis in real-world usage. Specifically, we implemented a comparison that would count the number of *clusters* of anomalous pixels (i.e. super-pixels) larger than a certain size, using the largest threshold that still results in at least one cluster. These most closely estimate real features in the data. Specifically, after the first noise reduction step, we counted the number of anomalous clusters with more than 10 connected pixels at 3σ. If at 3σ a certain spectrum contained no clusters of this size for either the CPU or GPU, we lowered the threshold to 2σ, and finally to 1σ if necessary. For this case the GPU and CPU found **100%** of the same clusters for every spectrum, implying that in general there will most likely be no differences in the actual features found in the HSI application. The actual number of clusters found using this technique, and the threshold used to find them, are listed here:

Spectrum Number	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*
Max Threshold	2σ	2σ	2σ	3σ	2σ	3σ	2σ	3σ	3σ	3σ	3σ	3σ	3σ	1σ	2σ	3σ
GPU Super-pixels	**1**	**2**	**1**	**1**	**3**	**1**	**1**	**2**	**2**	**1**	**4**	**3**	**4**	**0**	**1**	**3**
CPU Super-pixels	**1**	**2**	**1**	**1**	**3**	**1**	**1**	**2**	**2**	**1**	**4**	**3**	**4**	**0**	**1**	**3**

References

[1] Nagender Bandi, Chengyu Sun, Divyakant Agrawal, Amr El Abbadi. "Hardware acceleration in commercial databases: A case study of spatial operations." Technical report, Computer Science Department, University of California, Santa Barbara, 2004.

[2] Jeff Bolz, Ian Farmer, Eitan Grinspun and Peter Schröder. "Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid." ACM Transactions on Graphics 22, 2003.

[3] David Bremer, John Johnson, Holger Jones, Yang Liu, and Jeremy Meredith, "Graphics Processing Units (GPUs) for General Purpose High Performance Computing", To appear in the ACM/IEEE SC|05 Conference, 2005.

[4] Youquan Liu, Xuehui Liu, Enhua Wu. "Real-Time 3D Fluid Simulation on GPU with Complex Obstacles," 12th Pacific Conference on Computer Graphics and Applications (PG|04), 2004.

[5] Quanfu Fan, Alon Efrat, Vladlen Koltun, Shankar Krishnan, and Suresh Venkatasubramanian. "Hardware Assisted Natural Neighbour Interpolation". Proceedings of the 7th Workshop on Algorithms Engineering and Experimentation, 2004.

[6] Zhe Fan, Feng Qiu, Arie Kaufman, Suzanne Yoakum-Stover. "GPU Cluster for High Performance Computing". Proceedings of the 2004 ACM/IEEE SuperComputing (SC|04), 2004.

[7] Randima Fernando, *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*. Addison-Wesley Professional, 2004.

[8] Galoppo N, GovinDaraju NK, Henson M, and Manocha D. "LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware." In Proceedings of the ACM/IEEE SC|05 Conference, 2005.

[9] Nolan Goodnight, Cliff Woolley, Gregory Lewin, David Luebke, and Greg Humphreys. "A Multigrid Solver for Boundary Value Problems Using Programmable Graphics Hardware." Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware, 2003.

[10] Naga K. Govindaraju, Stephane Redon, Ming C. Lin and Dinesh Manocha. "CULLIDE: Interactive Collision Detection Between Complex Models in Large Environments Using Graphics Hardware". Proceedings of the Eurographics/SIGGRAPH Graphics Hardware Workshop, 2003.

[11] Naga K. Govindaraju, Brandon Lloyd, Wei Wang, Ming Lin, Dinesh Manocha. "Fast computation of database operations using graphics processors". Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, 2004.

[12] John Johnson and Jeremy Meredith, "The Evaluation of GPU-Based Programming Environments for Knowledge Discovery", High Performance Embedded Computing Workshop, 2004.

[13] Kenneth Moreland and Edward Angel. The FFT on a GPU. Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware, 2003.

[14] Matt Pharr and Randima Fernando, *GPU Gems 2 : Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley Professional, 2005.

[15] William H Press, et al. *Numerical Recipes in C*, Cambridge, University Press, 1992.

[16] Robert Strzodka, Marc Droske and Martin Rumpf. “Image Registration by a Regularized Gradient Flow - A Streaming Implementation in DX9 Graphics Hardware”. Computing, 73(4), 373-389, Springer, 2004.

[17] GPGPU Web Site. http://www.gpgpu.org

[18] OpenGL Shading Language. http://www.opengl.org/documentation/oglsl.html

[19] Scientific Computing on the Sony Playstation 2. http://arrakis.ncsa.uiuc.edu/ps2/index.php