Assessment of Graphic Processing Units (GPUs) for Department of Defense (DoD) Digital Signal Processing (DSP) Applications

John D. Owens, Shubhabrata Sengupta, and Daniel Horn[†]
University of California, Davis
[†] **Stanford University**

Abstract

In this report we analyze the performance of the fast Fourier transform (FFT) on graphics hardware (the GPU), comparing it to the best-of-class CPU implementation FFTW. We describe the FFT, the architecture of the GPU, and how general-purpose computation is structured on the GPU. We then identify the factors that influence FFT performance and describe several experiments that compare these factors between the CPU and the GPU. We conclude that the overhead of transferring data and initiating GPU computation are substantially higher than on the CPU, and thus for latency-critical applications, the CPU is a superior choice. We show that the CPU implementation is limited by computation and the GPU implementation by GPU memory bandwidth and its lack of a writable cache. The GPU is comparatively better suited for larger FFTs with many FFTs computed in parallel in applications where FFT throughput is most important; on these applications GPU and CPU performance is roughly on par. We also demonstrate that adding additional computation to an application that includes the FFT, particularly computation that is GPU-friendly, puts the GPU at an advantage compared to the CPU.

The future of desktop processing is *parallel*. The last few years have seen an explosion of single-chip commodity parallel architectures that promises to be the centerpieces of future computing platforms. Parallel hardware on the desktop—new multicore microprocessors, graphics processors, and stream processors—has the potential to greatly increase the computational power available to today's computer users, with a resulting impact in computation domains such as multimedia, entertainment, signal and image processing, and scientific computation.

Several vendors have recently addressed this need for parallel computing: Intel and AMD are producing multicore CPUs; the IBM Cell processor delivers impressive performance with its 9 parallel cores; and several stream processor startups, including Stream Processors Inc. and Clearspeed, are producing commercial parallel stream processors. None of these chips, however, have achieved market penetration to the degree of the graphics processor (GPU). Today's GPU features massive arithmetic capability and memory bandwidth with superior performance and price-performance when compared to the CPU. For instance, the NVIDIA

[0]John D. Owens, Shubhabrata Sengupta, and Daniel Horn. "Assessment of Graphic Processing Units (GPUs) for Department of Defense (DoD) Digital Signal Processing (DSP) Applications". Technical Report ECE-CE-2005-3, Computer Engineering Research Laboratory, University of California, Davis, 2005. http://www.ece.ucdavis.edu/cerl/techreports/2005-3/

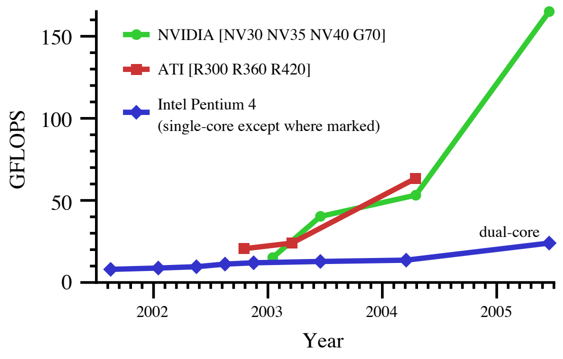

Figure 1: *The programmable floating-point performance of GPUs (measured on the multiply-add instruction as 2 floating-point operations per MAD) has increased dramatically over the last four years when compared to CPUs. Figure from Owens et al. [OLG$^+$05].*

GeForce 7800 GTX features over 165 GFLOPS of programmable floating-point computation, 38.4 GB/s of peak main memory bandwidth, and about 20 GB/s of measured sequentially accessed main memory bandwidth. These numbers are substantially greater than the peak values for upcoming Intel multicore GPUs (24 GFLOPS and 8.2 GB/s)[1], and are growing more quickly as well (Figure 1). And the GPU is currently shipping in large volumes, with a rate of hundreds of millions of units per year. The market penetration of the GPU, its economies of scale, and its established programming libraries make the compelling case that it may be the parallel processor of choice in future systems.

Parallel processing, however, is not without its costs. Programming efficient parallel code is a difficult task, and the limited and restrictive programming models of these emerging parallel processors make this task more difficult. Traditional scalar programs do not efficiently map to parallel hardware, and thus new approaches to programming these systems are necessary. From a research point of view, we believe that the major obstacle to the success of these new architectures will be the difficulty in programming desktop parallel hardware.

This report is arranged as follows. In Section 1, we first describe the architecture of the GPU and how general-purpose applications map to it. We then introduce the fast Fourier transform (FFT) in Section 2 and outline our implementation of it. Section 3 gives high-level metrics for evaluating the FFT, and Section 4 analyzes the performance of the FFT on the GPU and CPU. We offer thoughts about the future in Section 5.

1 GPU Architecture

We begin by describing the architecture of today's GPU and how general-purpose programs map to it. The recent *GPU Gems 2* book has longer articles on these topics: Kilgariff and Fer-

[1]GFLOPS numbers courtesy of Ian Buck, Stanford University; GPU GFLOPS were measured via GPUBench [BFH04a]; GPU peak memory bandwidth is from NVIDIA marketing literature [NVI05], and measured GPU sequential bandwidth from Buck [Buc05a].

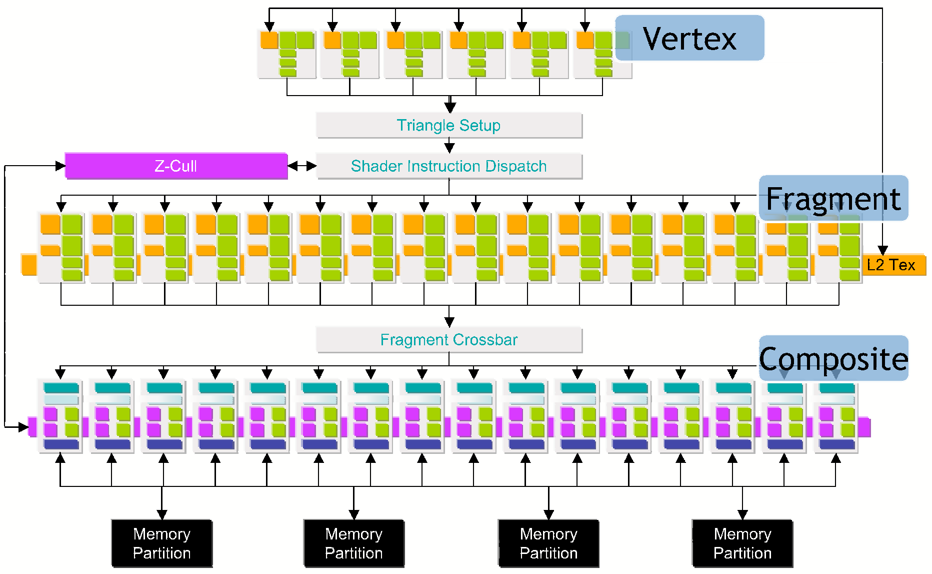

Figure 2: *Block diagram of the NVIDIA GeForce 6800. The vertex and fragment stages are programmable, with most GPGPU work concentrated in the fragment processors. Note there are 16 parallel fragment processors, each of which operates on 4-wide vectors of 32-bit floating point values. (The GeForce 7800 GTX has 24 fragment processors.) Figure courtesy Nick Triantos, NVIDIA.*

nando summarize the recent GeForce 6 series of GPUs from NVIDIA [KF05], Mark Harris discusses mapping computational concepts to GPUs [Har05], and Ian Buck offers advice on writing fast and efficient GPU applications [Buc05b].

The most important recent change in the GPU has been a move from a fixed-function pipeline, in which the graphics hardware could only implement the hardwired shading and lighting functionality built into the graphics hardware, to a programmable pipeline, where the shading and lighting calculations are programmed by the user on programmable functional units. Figure 2 shows the block diagram of the NVIDIA GeForce 6800. To keep up with real-time demands for high-performance, visually compelling graphics, these programmable units collectively deliver enormous arithmetic capability, which can instead be used for computationally demanding general-purpose tasks.

To describe how a general purpose program maps onto graphics hardware, we compare side-by-side how a graphics application is structured on the GPU with how a general-purpose application is structured on the GPU.

Graphics Application	General-Purpose Application
First, the graphics application specifies the scene geometry. This geometry is transformed into screen space, resulting in a set of geometric primitives (triangles or quads) that cover regions of the screen.	In general-purpose applications, the programmer instead typically specifies a large single piece of geometry that covers the screen. For the purposes of this example, assume that geometry is 1000×1000 pixels square.
The next step is called "rasterization". Each screen-space primitive covers a set of pixel locations on the screen. The rasterizer generates a "fragment" for each covered pixel location.	The rasterizer will generate one fragment for each pixel location in the 1000×1000 square, resulting in one million fragments.
Now the programmable hardware takes center stage. Each fragment is evaluated by a fragment program that in graphics applications calculates the color of the fragment. The GPU features multiple fragment engines that can calculate many fragments in parallel at the same time. These fragments are processed in SIMD fashion, meaning they are all evaluated by the same program running in lockstep over all fragments.	The fragment processors are the primary computational engines within the GPU. The previous steps were only necessary to generate a large number of fragments that are then processed by the fragment processors. In the most recent NVIDIA processor, 24 fragment engines process fragments at the same time, with all operations on 4-wide 32-bit floating point vectors.
Calculating the color of fragments typically involves reading from global memory organized as "textures" that are mapped onto the geometric primitives. (Think of a picture of a brick wall decaled onto a large rectangle.) Fragment processors are efficient at fetching texture elements (texels) from memory.	General-purpose applications use the texture access mechanism to read values from global memory; in vector processing terminology, this is called a "gather". Though fragment programs can *read* from random locations in memory, they cannot *write* to random locations in memory ("scatter"). This limitation restricts the generality of general-purpose programs (though scatter can be synthesized using other GPU mechanisms [Buc05b]).
Generated fragments are then assembled into a final image. This image can be used as texture for another pass of the graphics pipeline if desired, and many graphics applications take many passes through the pipeline to generate a final image.	After the fragment computation is completed, the resulting 1000×1000 array of computed values can be stored into global memory for use on another pass. In practice, almost all interesting GPGPU applications use many passes, often dozens of passes.

The designers of the GPU have also made architectural tradeoffs that are different than those of the CPU. Perhaps the most important is an architectural philosophy on the GPU that emphasizes *throughput* over *latency*. CPUs generally optimize for latency; their memory systems are designed to return values as quickly as possible to keep their computation units busy. The GPU, historically, has different goals, because it targets the human visual system, which operates on millisecond time scales. With computation occurring on nanosecond time scales, six orders of magnitude faster than the visual system, the designers of the GPU learned that a tradeoff that increased throughput at the expense of latency did not matter to the users of the GPU and improved overall performance. Consequently pipelines for the GPU are thousands of cycles long (compared to tens of cycles on a CPU); individual operations take much longer than their CPU counterparts, but these deep pipelines permit more concurrency and more throughput.

GPU manufacturers have found that many of their applications of interest are limited not by the GPU but by the CPU, specifically its ability to marshal and send the data to the GPU. A significant effort in GPU architecture and API design is thus toward alleviating this bottleneck. The GPU manufacturers also often find that their applications that are limited by the GPU are limited less by GPU computation and more by its memory system. As we mentioned in the introduction, the NVIDIA GeForce 7800 features 38.4 GB/s of peak main memory bandwidth, and about 20 GB/s of measured sequentially accessed main memory bandwidth. However, the bandwidth on random reads is significantly lower, only about 2 GB/s [Buc05a].

2 The FFT on the GPU

We now describe the high-level structure of the FFT and its mapping to the GPU, discuss the previous work in this field, and then detail the implementation we chose for this study.

The Fourier transform links signal representations between the physical domain (usually time) and the spectral domain (frequency). While the computation of the Fourier transform for n points has complexity of $O(n^2)$, the "fast Fourier transform" (FFT) described by Cooley and Tukey [CT65] reduced this cost to $O(n \log n)$. The FFT and its variants are the most common methods of computing the Fourier transform in today's signal processing applications.

The most common formulation of the FFT is described as "radix-2, decimation-in-time (DIT)" and we use it to describe the general structure of the FFT computation. The FFT is at its core a divide-and-conquer algorithm, and radix-2 means that at each step, the problem is divided into 2 parts. Other radixes are possible, but here, we can conclude that a radix-2 n-point FFT will take $\log_2 n$ stages. Decimation-in-time means that the physical domain signals (usually measurements as a function of time) are split into two halves in an interleaved fashion; even measurements into the first half, odd measurements into the second half. The alternative, decimation-in-frequency, splits the signal in the spectral domain rather than the physical domain.

The recursive nature of the FFT means that the problem, for power-of-2 values of n, eventually decomposes to a series of 2-point transforms, known as "butterflies". These butterflies produce two outputs from two inputs: $o_0 = i_0 + W_n i_1$; $o_1 = i_0 - W_n i_1$. The W_n coefficients are called "twiddle factors" and are complex roots of 1. The computation of the butterflies and (possibly) the twiddle factors comprise the computation needs of the algorithm.

The FFT, then, has the following characteristics that are interesting from the point of view of efficient implementations:

- The FFT is a multi-stage algorithm; for a radix-2 formulation, a n-point FFT requires $\log_2 n$ stages.
- The FFT is typically performed on complex numbers.
- The recursive structure of the FFT, coupled with the interleaving of DIT, means that the inputs to the first stage of butterflies are not in input order. Instead, they are in "bit-reversed" order. Memory systems with bit-reversed load primitives are common in digital signal processors.

  Each successive stage also requires a structured but non-trivial and non-sequential communication pattern; this pattern is implementation-dependent.
- If organized properly, by evaluating blocks of the FFT as a coherent whole, the memory traffic of the FFT is highly cache efficient.

2.1 Previous GPU Implementations of the FFT

In our recent GPGPU survey we compiled a recent list of FFT algorithms [OLG+05], which we copy below:

> Motivated by the high arithmetic capabilities of modern GPUs, several projects have recently developed GPU implementations of the fast Fourier transform (FFT) [BFH+04b, JvHK04, MA03, SL05]. (The *GPU Gems 2* chapter by Sumanaweera and Liu, in particular, gives a detailed description of the FFT and their GPU implementation [SL05].) In general, these implementations operate on 1D or 2D input data, use a radix-2 decimation-in-time approach, and require one fragment-program pass per FFT stage. The real and imaginary components of the FFT can be computed in two components of the 4-vectors in each fragment processor, so two FFTs can easily be processed in parallel. These implementations are primarily limited by memory bandwidth and the lack of effective caching in today's GPUs, and only by processing two FFTs simultaneously can match the performance of a highly tuned CPU implementation [FJ98]. Daniel Horn maintains an open-source optimized FFT library based on the Brook distribution [Hor05b].

Since the publication of this survey (August 2005), we have not come across any new implementations described in the literature.

2.2 Mapping the FFT to the GPU

The most straightforward way to map the FFT to the GPU is to perform one pass through the GPU for each of the $\log n$ stages of the FFT. The previous algorithms (Section 2.1) all use this structure. Each stage requires the computation of many butterflies (a n-point FFT requires $n/2$ butterflies), which can all be computed in parallel and with the same SIMD instruction stream.

The structure of each stage is similar: draw geometry that covers n fragments for a n-point FFT (depending on the packing and the structure of the computation, $n/2$ or $n/4$ fragments might be appropriate); in the fragment program, load the inputs to the butterflies as a texture fetch; and store the outputs of the butterfly as a texture for use in the next stage. Two major challenges to do this efficiently are ensuring that the 4-wide fragment processors are fully utilized and attempting to run the same code on each stage, meaning the communication pattern must be identical in each stage (this is not true in a traditional CPU-based FFT computation).

2.3 Our Implementation

After evaluating the existing implementations and their advantages and disadvantages, for this study we adapted Daniel Horn's "libgpufft" library [Hor05a, Hor05b]. This library was developed at Stanford in conjunction with the Brook stream programming system [BFH$^+$04b] and has several factors that made it the most attractive basis for this study.

- libgpufft has two major technical advantages over other GPU FFT implementations:
  - libgpufft has an identical communication pattern for each stage, allowing the same fragment program to be used across all stages. Consequently libgpufft does not incur the cost of switching the fragment program between stages. This pattern, while structured and consistent across passes, is not sequential. Thus it will not realize the maximum memory bandwidth from the GPU memory system. (We discuss this point further in Section 4.3.) This difficulty is common to all GPU implementations of the FFT.
  - libgpufft fully utilizes the 4-wide floating-point arithmetic units in the GPU's fragment program units. Horn's implementation packs two complex floating-point numbers into each fragment, allowing the 4-wide units to be used at maximum efficiency. Other FFT implementations require two FFTs to run simultaneously to fully utilize the math capabilities of these units.
- libgpufft is written in Brook, a high-level language for GPUs, making its code simpler to understand and adapt than other implementations. Buck indicated that the FFT's loss of efficiency when written in Brook compared to written in a low-level language is small [Buc05a].

- Its source code is publicly available.
- It is efficient on both NVIDIA and ATI hardware.
- Horn and his colleagues have compared their Brook implementation to other FFT implementations, including GPU implementations (Moreland's original FFT implementation [MA03] and ATI's reference implementation) as well as the best-of-class CPU implementation, FFTW [FJ98]. Brook has comparable or better performance than any of the GPU implementations.

Buck indicated in August 2005 that the (2D) libgpufft has twice the performance (throughput) as FFTW [Buc05a][2]. libgpufft, however, only addresses 2D FFTs, thus for this study we adapted its techniques to apply to arrays of 1D FFTs. We plan to incorporate our changes and additions into the libgpufft release.

3 Analyzing FFT Performance

To aid in understanding the results below (Section 4), we describe some of the factors that contribute to the runtime and some of the considerations we should make in analyzing the results.

3.1 Keys to FFT High Performance

Efficient FFT computation depends on the following factors:

Computation FFTs demand high floating-point computation rates. For a 1024-point FFT, the butterflies require on the order of 50,000 operations.

Main memory bandwidth For a n-point FFT, each stage of computation requires reading and writing n points from and to memory. FFT-specific hardware also often features a bit-reversal primitive in the memory system, but its effect on the overall performance here would likely be minimal.

Cache efficiency Because main memory bandwidth is often the limiting factor in FFT implementations, any gains in memory bandwidth from caching may directly improve FFT performance.

Using these factors as a basis, we can divide the runtime of our computation into four components and compare them between the CPU and the GPU. Table 1 summarizes these

[2]Buck compared CPU (Pentium 4 3.0 GHz) vs. GPU (NVIDIA GeForce 7800) performance for several benchmarks. On the microbenchmark SAXPY, for instance, the GPU's advantage was over 8 to 1. The discrepancy between the 2:1 GPU performance advantage on 2D FFTs and what we report here for 1D FFTs is because the 2D FFT requires a cache-unfriendly transposition step on the CPU, whereas the GPU has a native 2D memory system that has equal efficiency for horizontal and vertical strides.

components. In summary, the raw computation and main memory bandwidth rates of the GPU are superior to the CPU, but the superior cache performance of the CPU, and its substantially smaller transfer and setup times, give it an advantage over the GPU in many cases.

3.2 Evaluating the FFT

The user of the FFT may be interested in two aspects of FFT performance. First, the *latency* of the FFT may be most important, and implementations should strive for minimizing the amount of time to compute a single FFT. Second, FFT *throughput* may be most important, so implementations may instead attempt to maximize the number of FFTs performed in a given amount of time.

These two aspects are often contradictory, particularly as systems move from scalar to parallel. The additional parallel hardware can either be used to speed up a single FFT (improving latency) or perform multiple FFTs in parallel (improving throughput). In general, the parallelism exploited by the GPU is better suited for improving throughput rather than latency.

4 Results and Analysis

4.1 Experimental Setup

We evaluated the FFT on two platforms, a laptop running Windows XP[3] and a desktop also running Windows XP[4]. All results shown are for the (faster) desktop.

For the CPU FFT, we used the FFTW version 3.01 available at http://www.fftw.org/ compiled with Microsoft Visual C++; for the GPU FFT, we used the CVS version of Brook with the OpenGL[5] backend (also compiled with Microsoft Visual C++).

All tests were run 10 times and the results averaged for our final measurements.

4.2 Single Issue FFTs

We begin by analyzing the amount of time to evaluate a single FFT. Figure 3 illustrates the runtime of a single FFT as a function of the size of the FFT, and Figure 4 shows MFLOPS (millions of floating point operations per second). This test primarily measures FFT *latency*.

The results indicate several interesting points. First, at all sizes, the CPU-based FFTW has a much smaller latency than the GPU-based libgpufft. We also note that, as we expect, FFTs with more points take longer to compute on the CPU using FFTW than FFTs with fewer

[3]CPU: 2128 MHz Pentium M 770; 64 KB L1, 2048 KB L2 cache; main memory: 512 MBx2 133 MHz DDR2-SDRAM; GPU: GeForce GO 6800 Ultra PCI-E (NV41) with 256 MB memory.

[4]CPU: 2412 MHz AMD Athlon 64 FX 53; 128 KB L1, 1024 KB L2 cache; main memory: 1024x1 200 MHz DDR-SDRAM; GPU: GeForce 7800 GTX with 256 MB memory.

[5]OpenGL is a cross-platform API for graphics supported by modern graphics cards; the alternative is the Windows-only "DirectX" API.

Component	CPU	GPU	Comments
Transfer time	Zero	Fixed	For the GPU to begin computation it must first transfer the data from the CPU's memory to the GPU's memory over a system bus. Though the recent introduction of the PCI Express (PCI-E) bus has improved the bandwidth of this transfer, and continued driver improvements have helped the latency, this cost is still an important component for any GPU-based computation, particularly for small amounts of GPU computation.
Setup time	Smaller	Larger	To run a program over a dataset, the GPU must set up the graphics pipeline, load the fragment program, run the input through the entire pipeline, and flush the pipeline at the end. Collectively this is termed a "pass". On our GPU FFT implementation, one pass is necessary for each stage. While the cost of a pass has decreased over the past few years due to better drivers and hardware, processing speed has increased even faster, so ever-larger amounts of work are necessary to mitigate the setup cost. In any case the setup cost is still certainly larger than the cost of setting up and initiating a comparable CPU program.
Memory loads/stores	Slower, but cacheable	Faster	The GPU memory system has 3–4 times as much main memory bandwidth as a CPU memory system, but the CPU caches can cache intermediate results, unlike the GPU. (GPU caches are read-only, so no intermediate results can be written back to them. CPUs, on the other hand, take advantage of the read-write capabilities of their caches to place an entire block of the FFT within their caches and fully compute it without intermediate accesses to main memory.) To take full advantage of GPU memory bandwidth, the dataset should allow sequential memory fetches as much as possible.
Computation	Slower	Faster	If structured in a GPU-friendly manner, GPU computation is substantially more capable than CPU computation. The CPU can make up some ground by effectively using its vector (SSE and SSE-2) units, but the CPU still has many fewer functional units than the GPU. The peak floating-point performance for today's hottest GPU is more than 5 times as high as its CPU counterpart, and this gap is growing. The challenge for GPUs is to organize the computation to take advantage of the full power of their hardware.

Table 1: *FFT runtime components compared between CPU and GPU.*

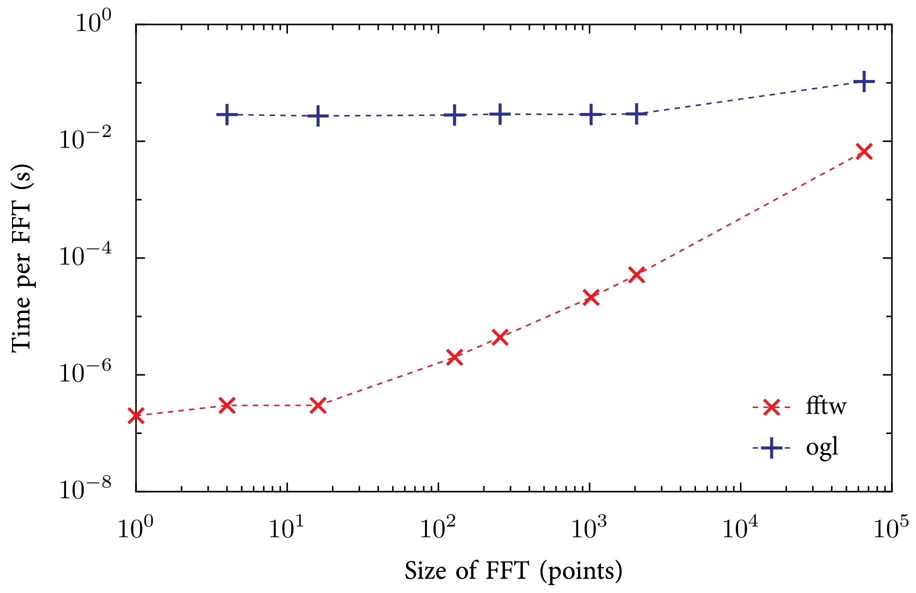

Figure 3: *Time to compute one FFT (latency), as a function of the size of the FFT. Toward the bottom of the graph is faster. "fftw" is the CPU-based FFTW; "ogl" is the GPU-based Brook OpenGL back end to libgpufft.*

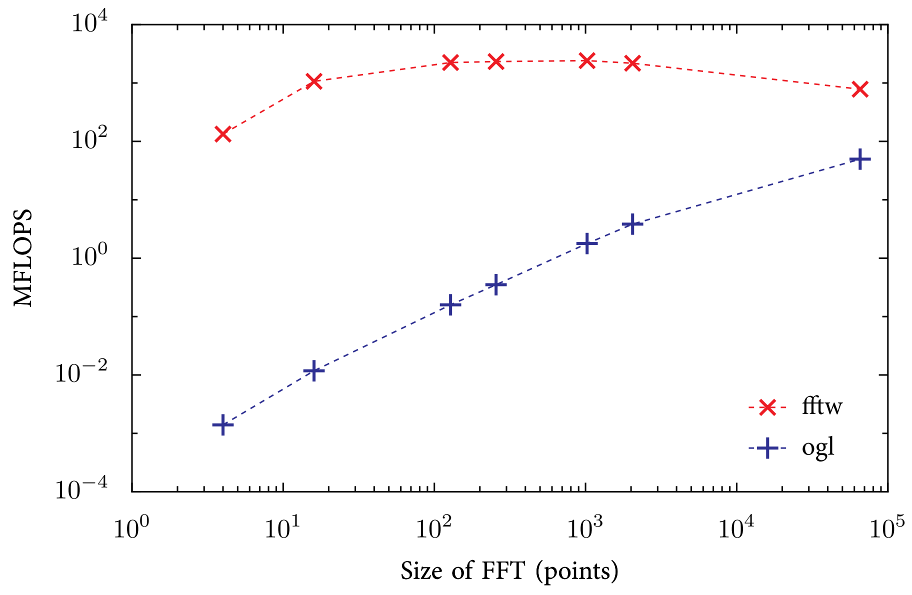

Figure 4: *Measured MFLOPS (millions of floating-point operations per second), as a function of the size of the FFT. Toward the top of the graph is higher MFLOPS. "fftw" is the CPU-based FFTW; "ogl" is the GPU-based Brook OpenGL back end to libgpufft.*

points. However, to first order, the latency of one single FFT on the GPU does not depend on the size of the FFT except for very large FFTs (65k points). Why is this?

As we saw in Section 3.1, the runtime of a FFT depends on several factors. In the case of the CPU, the most significant component is the time to compute the FFT. For the GPU, however, the time to compute one single GPU is insignificant compared to other factors. It is difficult to separate the various components that account for the GPU's runtime, but they would include the time to load the dataset from CPU memory, the time to set up and perform the transfer to the GPU, the time to initially set up the graphics pipeline and load the fragment program, the time to flush the pipeline when the computation is complete, the time to set up and perform the transfer from the GPU back to the CPU, and the time to store the result back in the CPU's main memory.

A second factor, for GPUs, is their inefficiency with small numbers of fragments. Modern GPUs appear to have a threshold of a minimum number of fragments to fill the pipeline for optimal efficiency. On the latest GPUs, Horn et al. report this threshold is roughly 4000 fragments for their HMMer application (described as "not unlike libgpufft") [HHH05]. What this means in practice is that processing any number of fragments less than 4000 will take the same amount of time as processing 4000 fragments. Horn et al. also report that a larger threshold is necessary to get peak performance, and again on the latest GPUs, this threshold is roughly 10,000 fragments. Running a batch of fewer than 10,000 fragments will not approach peak performance on these GPUs, and these fragment thresholds certainly will not shrink but are instead likely to grow with new GPU generations.

The lesson to be learned here is that the CPU has substantially smaller overhead associated with initiating a FFT than the GPU, and in these tests, the GPU FFT is limited by this overhead.. Though many of the factors that contribute to GPU overhead have improved over the past few GPU generations, it is probably safe to say that CPU overhead will continue to be substantially lower than GPU overhead in the near to intermediate future. As we discussed in Section 1, latency is not the goal of GPU designers; thus **if your task involves running a single FFT, or latency is your primary concern, the CPU's FFT is a better choice.**

4.3 Varying FFT Issue

Evaluating only a single FFT at a time is a task poorly suited for graphics hardware. The GPU is much better at emphasizing throughput and evaluating multiple FFTs at the same time. For a CPU the cost of additional FFTs is linear: evaluating m FFTs takes m times as long as evaluating 1 FFT. The GPU has a very different tradeoff. For example, in our tests, evaluating 1024 1024-point FFTs takes only 3.2 times as long as evaluating 1 1024-point FFT. This is because the cost of evaluating a single FFT is dominated by overhead (as we saw in Section 4.2), but this overhead changes very little when we calculate multiple FFTs at the same time. Also, running a larger number of fragments through the fragment programs (in this case, one million fragments, well above the thresholds we discussed in the previous section) allows the fragment program to reach maximum efficiency.

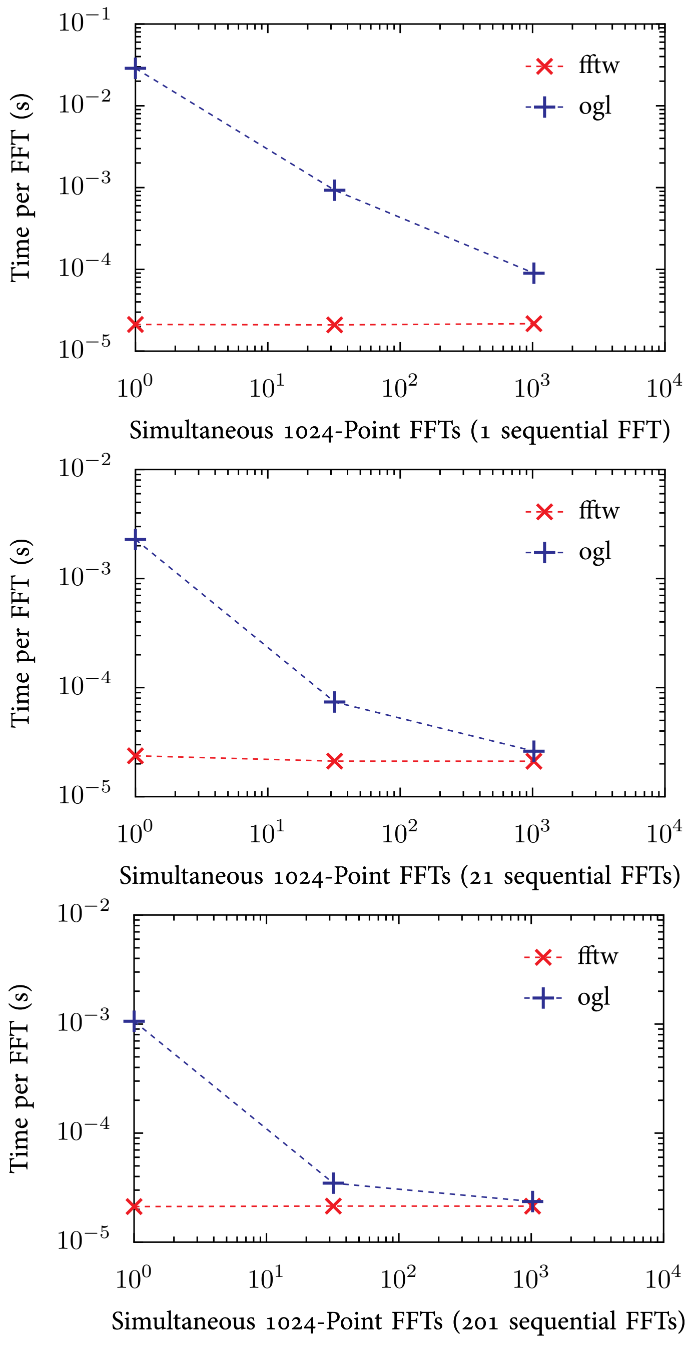

Figure 5: *Time per 1024-point FFT, varying the number of simultaneous (parallel) FFTs (moving toward the right on the graphs above) and varying the number of sequential FFTs (moving down between the graphs above). "fftw" is the CPU-based FFTW; "ogl" is the GPU-based Brook OpenGL back end to libgpufft.*

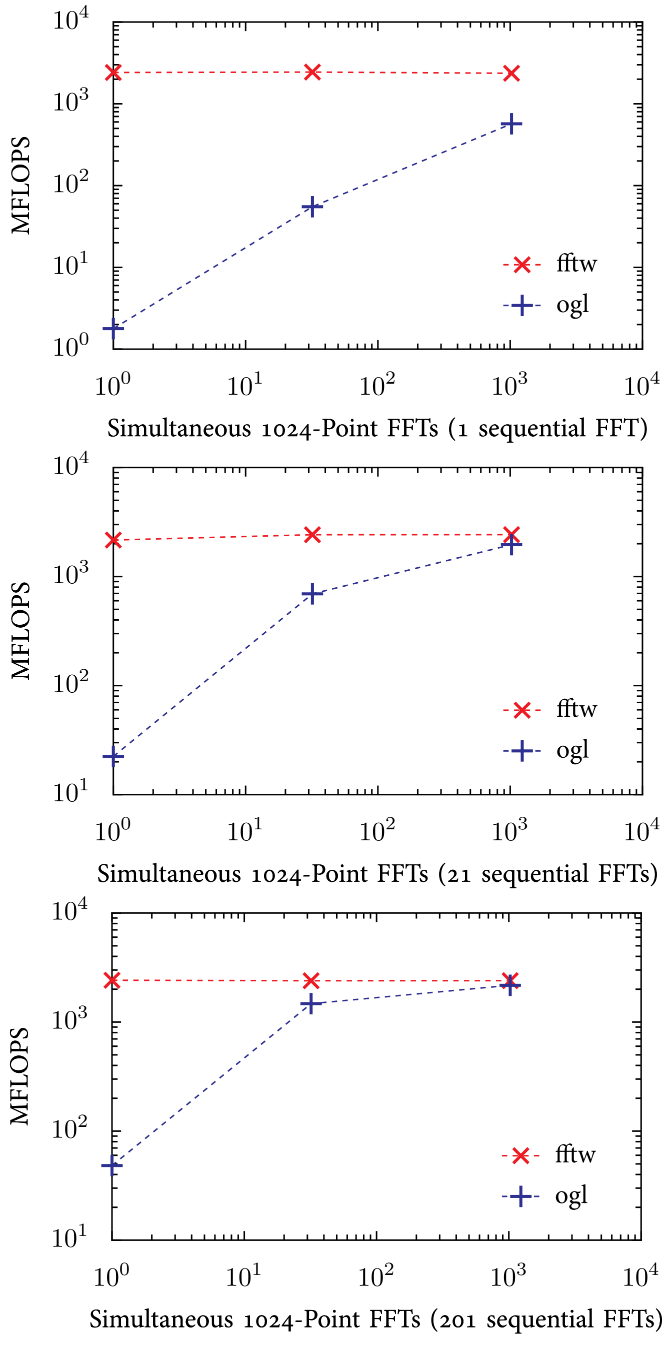

Figure 6: *Measured MFLOPS (millions of floating-point operations per second) for the 1024-point FFT, varying the number of simultaneous (parallel) FFTs (moving toward the right on the graphs above) and varying the number of sequential FFTs (moving down between the graphs above). "fftw" is the CPU-based FFTW; "ogl" is the GPU-based Brook OpenGL back end to libgpufft.*

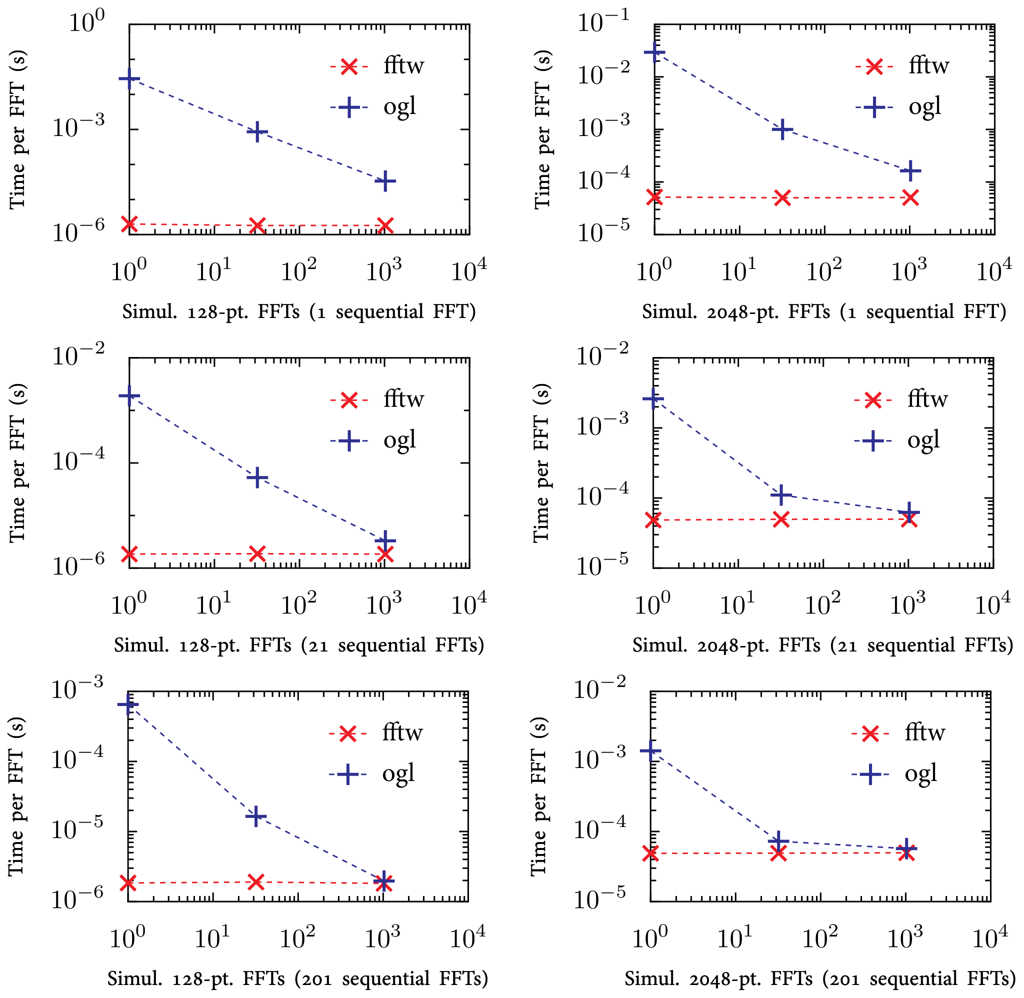

Figure 7: *Time per 128-point (left) and 2048-point (right) FFT, varying the number of simultaneous (parallel) FFTs (moving toward the right on the graphs above) and varying the number of sequential FFTs (moving down between the graphs above). "fftw" is the CPU-based FFTW; "ogl" is the GPU-based Brook OpenGL back end to libgpufft.*

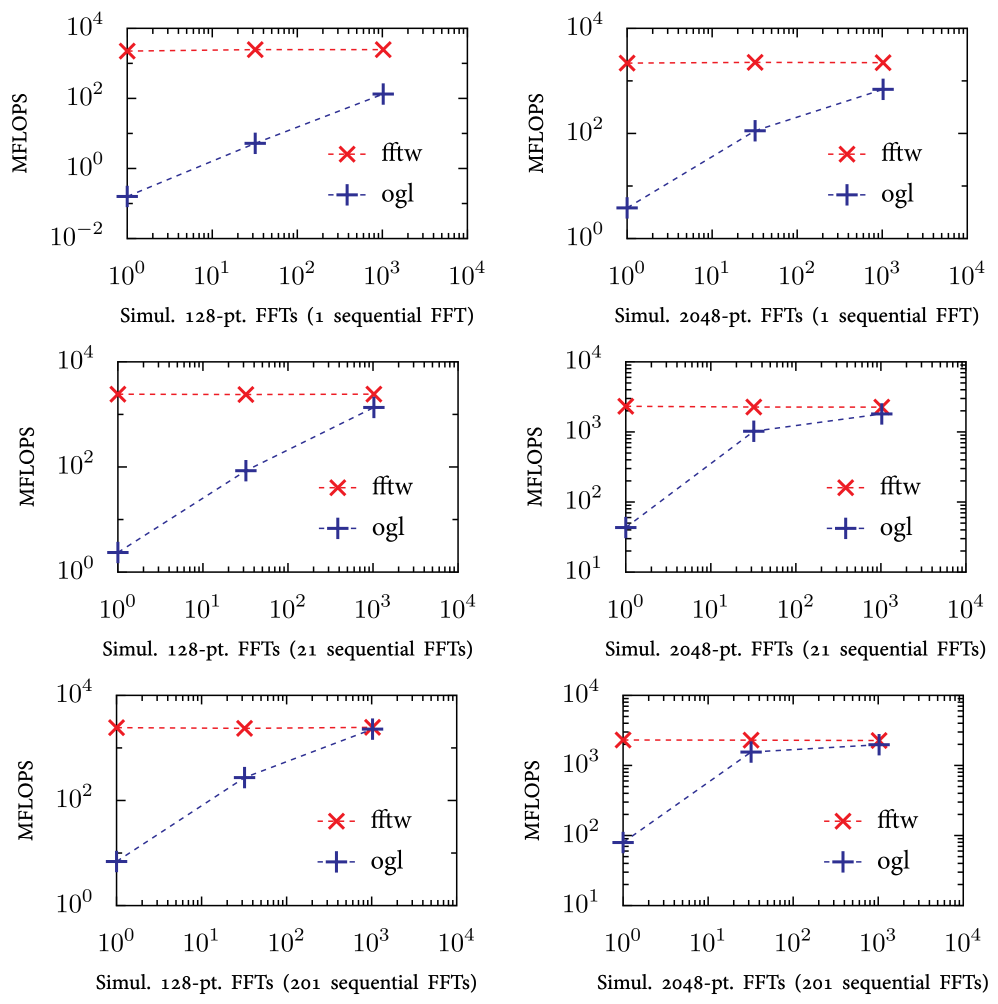

Figure 8: *Measured MFLOPS (millions of floating-point operations per second) for the 128-point (left) and 2048-point (right) FFT, varying the number of simultaneous (parallel) FFTs (moving toward the right on the graphs above) and varying the number of sequential FFTs (moving down between the graphs above). "fftw" is the CPU-based FFTW; "ogl" is the GPU-based Brook OpenGL back end to libgpufft.*

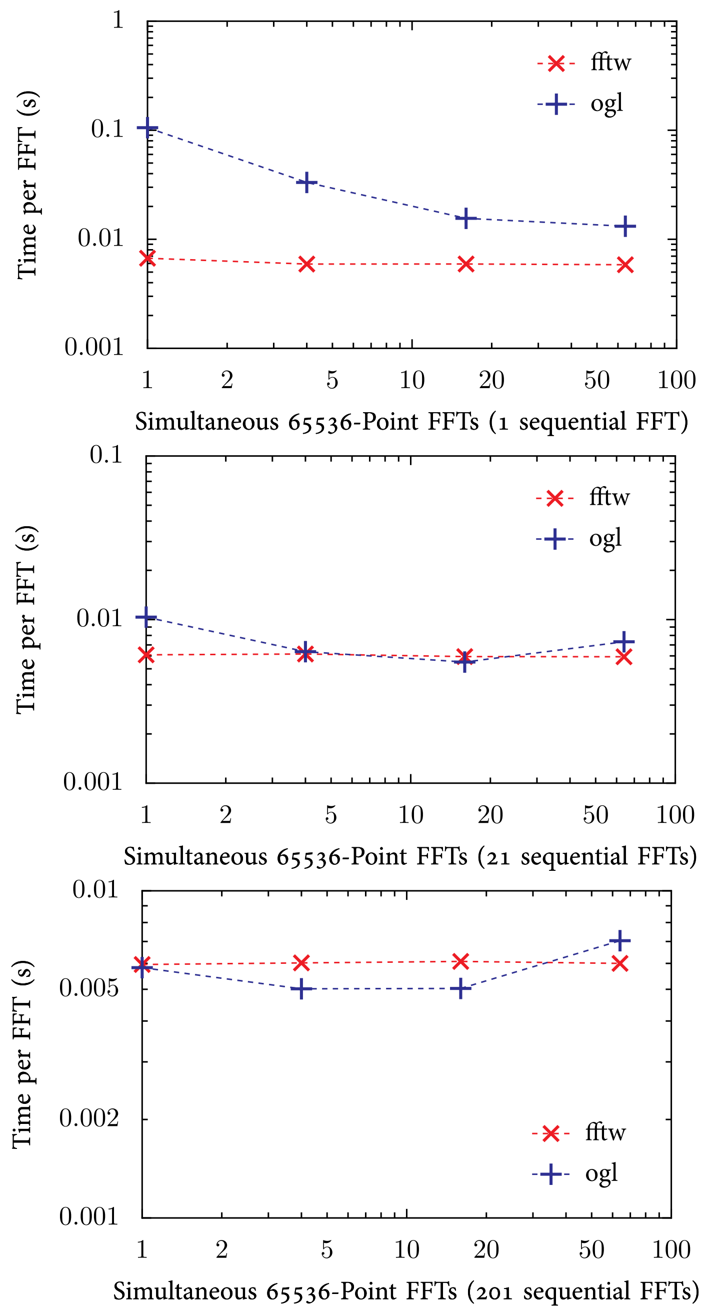

Figure 9: *Time per 65,536-point FFT, varying the number of simultaneous (parallel) FFTs (moving toward the right on the graphs above) and varying the number of sequential FFTs (moving down between the graphs above). "fftw" is the CPU-based FFTW; "ogl" is the GPU-based Brook OpenGL back end to libgpufft.*

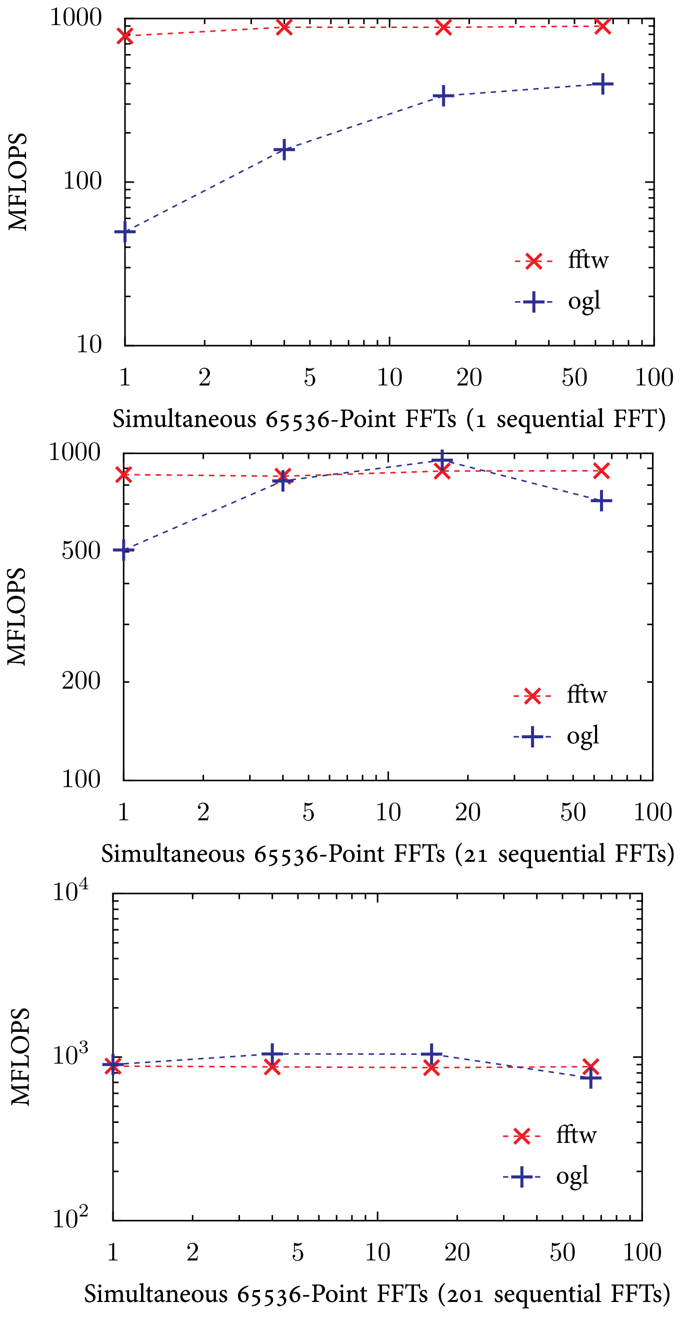

Figure 10: *Measured MFLOPS (millions of floating-point operations per second) for the 65,536-point FFT, varying the number of simultaneous (parallel) FFTs (moving toward the right on the graphs above) and varying the number of sequential FFTs (moving down between the graphs above). "fftw" is the CPU-based FFTW; "ogl" is the GPU-based Brook OpenGL back end to libgpufft.*

Figures 5 and 6 summarize our results from varying FFT issue for 1024-point FFTs. (For completeness, Figures 7 and 8 have results for 128-point and 2048-point FFTs, and Figures 9 and 10 have results for 65,536-point FFTs. The latter is an interesting case: the size of this FFT makes it cache more poorly than smaller FFTs on the CPU, but this degradation in performance is balanced by the GPU's need to introduce address translation arithmetic to handle the large size of the necessary data storage.) We describe the results of two experiments here.

- First, we vary the number of simultaneous (parallel) FFTs calculated. Instead of measuring the amount of time to calculate 1 FFT, we measure the amount of time to measure m FFTs. As the number of simultaneous FFTs grows, the gap between the CPU's throughput and the GPU's throughput shrinks. This is because the marginal cost of issuing more FFTs after the overhead is paid for the first is much smaller for a GPU. In Figures 5 and 6, as we move right along the graphs, the number of simultaneous FFTs increases.

- Second, we replace a single FFT with a series of FFT-IFFT pairs. These are calculated sequentially; Thus instead of running a single FFT, we run FFT-IFFT-FFT-IFFT .... On the GPU, this mitigates the overhead of transferring the data to and from the GPU. In Figure 5 and 6, as we move down between the graphs, the number of sequential FFTs increases.

Both of these experiments mitigate the overhead associated with the GPU. While they have little impact on the throughput of the CPU, **the throughput per FFT on the GPU rises with more simultaneous FFTs and with more sequential FFTs**. The throughput of the CPU and the GPU are roughly equivalent for 1024 parallel, 20-or-more sequential 1024-point FFTs. The GPU performs correspondingly better on larger FFTs than smaller FFTs (to a limit—moving to the huge 65,536-point FFT did not produce an additional performance advantage for the GPU). It is important to note, however, that in the parallel case, the latency does not change on the GPU compared to the serial case.

By running multiple FFTs on the GPU in parallel and emphasizing throughput as our metric, we avoid our performance being limited by the overhead of the transfer and pipeline setup. We can now determine the limitation of the FFT on the GPU proper.

In Figure 5, we can see that the maximum throughput of 1024-point FFTs on the GPU is roughly one every 20 microseconds. The memory requirement for a 1024-point FFT is roughly 250 kB[6], and at 1 FFT every 20 microseconds, the memory bandwidth sustained by the GPU in our tests is a little more than 12 GB/s. This bandwidth falls between the maximum measured memory bandwidth for random accesses (2 GB/s) and for sequential accesses (20 GB/s) discussed in Section 1, which is to be expected since the memory pattern is structured and regular but not sequential.

On the other hand, the 1024-point FFT requires on the order of 50,000 instructions (Section 3.1), and at 20 microseconds per FFT, the required arithmetic capability is 2.5 GFLOPS.

[6] 10 stages $\times$ 6 words per stage (2 read, 2 write, 2 for reading the twiddle factors) $\times$ 4 bytes per word $\times$ 1024 points = 246 kB.

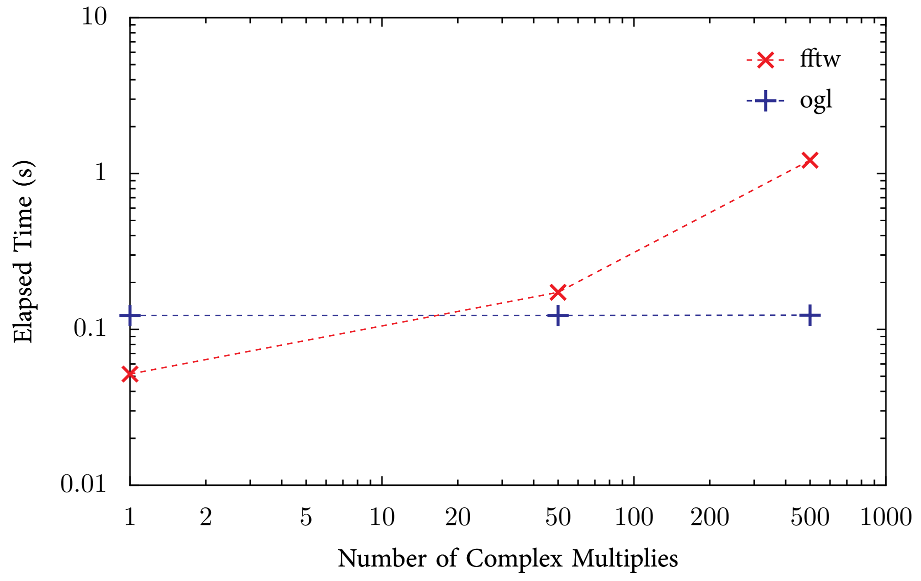

Figure 11: *Time to compute 1024 1024-point FFTs, a series of complex multiplies on each point the result (indicated on the x axis), and 1024 1024-point IFFTs. Toward the bottom of the graph is faster. "fftw" is the CPU-based FFTW; "ogl" is the GPU-based Brook OpenGL back end to libgpufft.*

This is far below the 165 GFLOPS of GPU floating-point capability. We can thus conclude that **the core GPU FFT is limited by memory bandwidth and not by computation**.

Recall that the GPU and CPU throughputs were similar for many parallel, sequential FFTs. Thus in our tests, we conclude that our CPU could sustain roughly 2.5 GFLOPS. FFTW's internal tests on a 2.8 GHz Pentium 4 Xeon show that 1024-point 1D FFTs sustain roughly 5.5 GFLOPS [FFT05b], and on a similar but slower processor to ours, roughly 3.25 GFLOPS [FFT05a].

4.4 Adding Intermediate Computation

Any real application would be unlikely to run just the FFT. Instead, the FFT is more typically part of a chain of processing kernels. For instance, an image processing task may require a FFT, a complex multiply on each frequency component to implement a filter, and then an inverse FFT. Other applications may require much more complex intermediate computation.

Our next experiment measures the impact of placing a task between an FFT and an inverse FFT (Figures 11 and 12). The task is a series of complex multiplies on each frequency component generated by the FFT. For a simple filter, only 1 complex multiply may be necessary. To simulate more complex tasks, we scaled the number of complex multiplies, up to 500 complex multiplies on each frequency component. Recall that a 1024-point FFT requires on

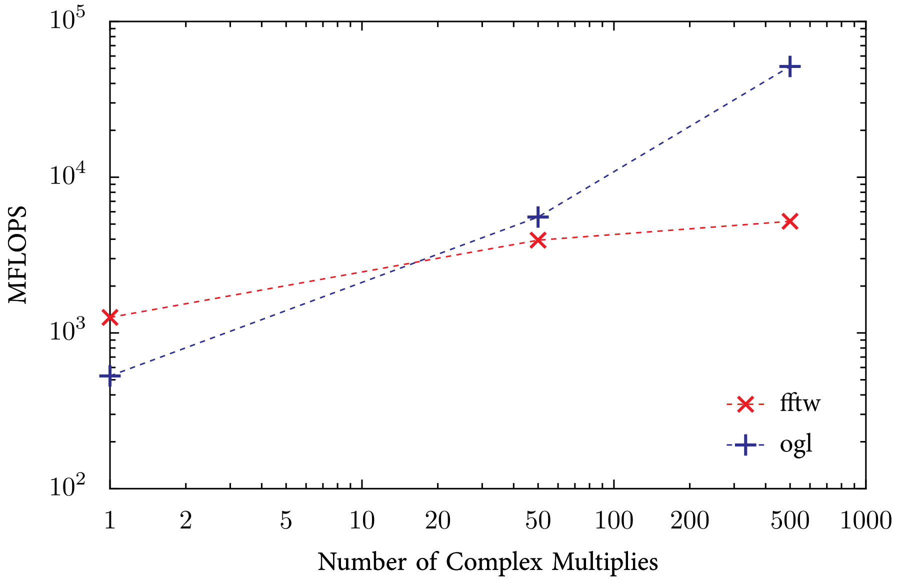

Figure 12: *Measured MFLOPS (millions of floating-point operations per second) while computing 1024 1024-point FFTs, a series of complex multiplies on each point the result (indicated on the x axis), and 1024 1024-point IFFTs. Toward the bottom of the graph is faster. "fftw" is the CPU-based FFTW; "ogl" is the GPU-based Brook OpenGL back end to libgpufft.*

the order of 50,000 floating-point operations. 500 complex multiplies on each of 1024 points is over 6 million operations.

When adding GPU-friendly intermediate computation like 500 complex multiplies, the GPU demonstrates a clear advantage over the CPU. We should qualify this result because the complex multiplies are ideal for the GPU: they require only raw computational horsepower without need for any communication through memory or caches. This particular benchmark is thus a best-case scenario for the GPU. Still, on this benchmark (1024 1024-point FFTs, then 500 complex multiplies on each of the million frequency components, then 1024 1024-point IFFTs), the GPU is sustaining over 60 GFLOPS (Figure 12), a little less than half its theoretical peak and substantially more than the maximum computation rate of any current CPU. Note that while increasing the number of complex multiplies from 1 to 500 changes the performance of the CPU by more than an order of magnitude, it makes almost no impact on the GPU at all. This confirms the conclusion from the previous section that memory bandwidth is the bottleneck for the GPU. Thus on this benchmark, the bottleneck for the CPU is the complex multiplies; on the GPU, the memory bandwidth of the FFT.

4.5 GPU Caveats

GPU hardware, and programming environments for the GPU, are not nearly as mature as their CPU counterparts. Consequently there are several caveats that must be offered when discussing GPU computation. Each of these are active research topics, and none seem insurmountable at this time; however, for any projects that are concerned with making practical use of the GPU today they must be considered.

- GPU computation today is limited to single-precision floating-point arithmetic. No integer capability is currently available, though we expect future hardware to support integers; double precision is not available at all and does not appear likely in the near to intermediate future (though synthesizing it in software is a possibility).

  GPU floating-point arithmetic is still not IEEE-compliant [HL04], with deficiencies in rounding, precision, and denormals. In addition, GPUs do not support any sort of exceptions (such as a trap on divide-by-zero).

- GPU programs have hard limits on their size; the Pixel Shader 3 standard allows 512 unique instructions with loops (typically) permitting 65k instructions. Though software techniques can divide large programs into multiple smaller programs, such techniques are not part of today's production programming environments. We note, however, that the size of the FFT kernel is far below the program sizes of today's GPUs; it would only be much larger programs that would run into this restriction.

- The programming model for the GPU is quite limited compared to the CPU. Programming parallel hardware is difficult, and not all tasks map well to the SIMD-parallel model and the limited feature set of the GPU. Much of the research in the GPGPU field is devoted to finding good mappings between tasks and the GPU hardware. Though continued progress has been made in this area, it is unlikely that we will ever see vanilla C or Fortran code compiled directly and efficiently to a GPU or other parallel architectures.

5 The Future

The future holds both good and bad news for the FFT. The good news is for the future of the GPU. The bad news is for the technology trends that will shape future computation hardware.

Details on next-generation GPUs are scant. The best way to judge what is coming in the future is to study the software trends, in particular the upcoming Microsoft DirectX standards, since GPU vendors must build hardware to conform to these standards. Current hardware is compliant with DirectX 9.0. The next generation standard will be called Windows Graphics Foundation (WGF) (WGF 2.0 will be the name for what is logically DirectX 10) and chips that will comply with it are still under development. Details of these chips have not yet been released, so it is difficult to speculate on what might come next.

Nonetheless, from these chips I believe we can expect the following.

Technology Improvements Certainly the computation rates, and particularly the programmable computation rates, will continue to rise as they have historically. Figure 1 shows the incredible historical growth of programmable floating-point performance. From a technology point of view this may not be sustainable in the long term, but it is reasonable to predict that GPU performance improvements will track clock-speed and transistor-density technology improvements of 70% a year [Sem03] rather than the 40% annual performance increases of recent CPUs [EWN05]. GPUs expect these faster rates because they are more easily able to take advantage of parallelism. (For more on technology trends, please read our recent article in GPU Gems 2 [Owe05].)

Memory bandwidth and latency are largely outside the domain of the GPU manufacturer and will thus improve at industry rates (the 2003 International Roadmap for Semiconductors forecasts a 25% annual improvement for DRAM bandwidth [Sem03], and historically, DRAM latency has improved by 5% annually). As the core GPU FFT is currently limited by GPU memory performance, and future memory performance will improve more slowly than compute performance, it seems most likely that the future of the GPU FFT will continue to be limited by slowly improving memory. Adding writable caches to the GPU would be the best antidote to this trend.

PCI Express (PCI-E) is a scalable standard. The 16x PCI-E buses common in workstations today appear sufficient for graphics applications.

Mitigating GPU Disadvantages We saw in the results that the primary disadvantages for the GPU are the high overhead for initiating computation and the lack of a writable cache.

Newer buses have helped lessen the CPU-to-GPU transfer time for large data, but the initiation cost is still high, and will continue to be a factor. For tasks where latency is paramount, and the compute time is modest, the CPU will likely be the target of choice for the near future. (Note that changes in CPU architectures, however—in particular, the move to multicore—prioritize throughput over latency.)

The cost of setting up a pass has decreased over the past few years as multipass techniques have become more common in graphics applications. This trend should continue. In addition, recent additions to graphics APIs allow more flexible and faster manipulation of data in graphics memory: two years ago Bolz et al. indicated that the cost of switching data buffers from output to input was the bottleneck for their linear algebra application [BFGS03], but graphics vendors have greatly reduced that cost over the last two years. The current "frame buffer object" (FBO) data container is a promising one.

Finally, recent additions to the graphics API that allow more flexible use of buffers, and proposed changes in future WGF-supporting hardware, will allow data to enter and leave the pipeline at locations in the pipeline that are not the beginning or the end. In the newest hardware, the output of a pass can be used as geometry or texture, for instance, and WGF allows an intermediate "stream out" of data from the pipeline into memory before it reaches the rasterizer.

As for cache, that is difficult to predict. From a strictly graphics point of view, a writable cache is not useful, and adding the ability to write to a cache significantly increases the complexity of its design. We have heard no rumors of a writable cache in future graphics hardware at this time; the only real argument for it is its usefulness in general-purpose computation. One possible direction for the GPU manufacturers is to make the shared on-chip level-2 cache writable while maintaining read-only level-1 caches at each fragment processor. **Users who are primarily interested in FFT-like signal processing applications that have potential gains from adding writability to the cache would be advised to make that known to GPU vendors.**

Other WGF Changes WGF-compliant hardware will add a third programmable unit to the pipeline, the "geometry shader". It will fall between the vertex shader and the rasterizer in the pipeline, will work on entire triangles at the same time, and (unlike the other programmable units) will be able to produce output more flexibly than the fragment and vertex shaders' one-in, one-out model (the geometry shader will be able to produce 0–4 outputs for each input).

Microsoft Windows is the most popular platform for GPGPU application development, primarily because Windows' ubiquity has led GPU vendors to make its drivers the most stable and fully featured. One current concern about WGF is Microsoft's push toward layering the OpenGL graphics API on top of it. Graphics researchers in general prefer using OpenGL for several reasons. Perhaps the most important is that OpenGL is cross-platform and works under other platforms such as Linux (which is common for cluster research) and the Apple Macintosh. Though this issue is far from resolved, graphics researchers fear that layering OpenGL atop WGF will reduce its performance in an unacceptable way.

Opinions These are my personal thoughts and should be interpreted in that light. I believe we will see improvements in several areas over the near to intermediate future. Orthogonality is an important goal for the GPU community: the same instruction sets work across the programmable units, each supported feature works properly with each datatype, and tools will become more fully featured and interoperable. Stability is another goal of the GPU vendors (and a goal of WGF): currently new drivers or new tool revisions have significant and undesirable impacts on performance and functionality. This has improved recently and must continue to improve. Finally, I hope that we will see new and varied interfaces, APIs, libraries, and abstractions emerge as graphics and general-purpose computation continue to converge.

However, there are aspects of the GPU that will likely not change in the near to intermediate future. One issue for GPGPU programmers is the constant churn of new features in graphics hardware and its resulting impact on writing the most efficient code. Though this makes programming difficult, GPU vendors will continue to introduce new features for business reasons, as it keeps them ahead of possible competition. The vendors' concentration on entertainment applications will continue as it is by far the largest market for them; the important task for the GPGPU community is to identify a "killer app" that will make an impact on GPU sales. Such a killer app may promote features like 64-bit floating-point that would not otherwise be on the GPU horizon. Finally, the concentration on parallelism for GPU performance, and the difficulty of programming parallel hardware, will not change.

6 About the PI

John Owens is Assistant Professor of Electrical and Computer Engineering at the University of California, Davis, where he leads research groups in graphics hardware (with an emphasis on general-purpose programmability on graphics processors, or GPGPU) and sensor networks. He was awarded the Department of Energy Early Career Principal Investigator Award in 2004 and frequently gives presentations (such as an invited talk at the 2004 High Performance Embedded Computing Conference) and tutorials (such as at IEEE Visualization in 2004 and 2005) on GPGPU. John was also the author of the recent comprehensive survey of GPGPU [OLG$^+$05] published as a Eurographics State of the Art Report in September 2005.

John earned his Ph.D. in electrical engineering in 2003 from Stanford University, where he was an architect of the Imagine Stream Processor and was responsible for major portions of its architecture, programming system, and applications. He earned his B.S. in electrical engineering and computer sciences from the University of California, Berkeley, in 1995. He can be contacted at the Department of Electrical and Computer Engineering, One Shields Avenue, Davis, CA 95616 USA; at jowens@ece.ucdavis.edu; or at +1 530 754-4289.

Acknowledgements

We gratefully acknowledge the financial support of Lockheed-Martin and the Department of Defense that made this work possible.

Bibliography

[BFGS03] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröder. Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. *ACM Transactions on Graphics*, 22(3):917–924, July 2003.

[BFH04a] Ian Buck, Kayvon Fatahalian, and Pat Hanrahan. GPUBench: Evaluating GPU performance for numerical and scientific applications. In *2004 ACM Workshop on General-Purpose Computing on Graphics Processors*, pages C–20, August 2004.

[BFH$^+$04b] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: Stream computing on graphics hardware. *ACM Transactions on Graphics*, 23(3):777–786, August 2004.

[Buc05a] Ian Buck. GPGPU: General-purpose computation on graphics hardware—high level languages for GPUs. *ACM SIGGRAPH Course Notes*, August 2005.

[Buc05b] Ian Buck. Taking the plunge into GPU computing. In Matt Pharr, editor, *GPU Gems 2*, chapter 32, pages 509–519. Addison Wesley, March 2005.

[CT65] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19:297–301, 1965.

[EWN05] Magnus Ekman, Fredrik Warg, and Jim Nilsson. An in-depth look at computer performance growth. *ACM SIGARCH Computer Architecture News*, 33(1):144–147, March 2005.

[FFT05a] FFTW. single-precision complex, 1d transforms [AMD Athlon XP 1.467 GHz]. http://www.fftw.org/speed/athlonXP-1467MHz/amd.1d.scxx.p2.png, 2005.

[FFT05b] FFTW. single-precision complex, 1d transforms [Pentium 4 Xeon 2.8 GHz]. http://www.fftw.org/speed/p4-2.8GHz-new/vce-new.1d.scxx.p2.png, 2005.

[FJ98] Matteo Frigo and Steven G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proceedings of the 1998 International Conference on Acoustics, Speech, and Signal Processing*, volume 3, pages 1381–1384, May 1998.

[Har05] Mark Harris. Mapping computational concepts to GPUs. In Matt Pharr, editor, *GPU Gems 2*, chapter 31, pages 493–508. Addison Wesley, March 2005.

[HHH05] Daniel Reiter Horn, Mike Houston, and Pat Hanrahan. ClawHMMer: A streaming HMMer-search implementation. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, November 2005.

[HL04] Karl E. Hillesland and Anselmo Lastra. GPU floating-point paranoia. In *Proceedings of the ACM Workshop on General Purpose Computing on Graphics Processors*, pages C–8, August 2004.

[Hor05a] Daniel Horn. Fast Fourier transforms. Unpublished, 2005.

[Hor05b] Daniel Horn. libgpufft. http://sourceforge.net/projects/gpufft/, 2005.

[JvHK04] Thomas Jansen, Bartosz von Rymon-Lipinski, Nils Hanssen, and Erwin Keeve. Fourier volume rendering on the GPU using a Split-Stream-FFT. In *Proceedings of Vision, Modeling, and Visualization*, pages 395–403, November 2004.

[KF05] Emmett Kilgariff and Randima Fernando. The GeForce 6 series GPU architecture. In Matt Pharr, editor, *GPU Gems 2*, chapter 30, pages 471–491. Addison Wesley, March 2005.

[MA03] Kenneth Moreland and Edward Angel. The FFT on a GPU. In *Graphics Hardware 2003*, pages 112–119, July 2003. http://www.cs.unm.edu/~kmorel/documents/fftgpu/.

[NVI05] NVIDIA Developer Relations. NVIDIA GeForce 7800 GPUs specifications. http://www.nvidia.com/page/specs_gf7800.html, August 2005.

[OLG+05] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Tim Purcell. A survey of general-purpose computation on graphics hardware. In *Eurographics 2005, State of the Art Reports*, pages 21–51, September 2005.

[Owe05] John Owens. Streaming architectures and technology trends. In Matt Pharr, editor, *GPU Gems 2*, chapter 29, pages 457–470. Addison Wesley, March 2005.

[Sem03] Semiconductor Industry Association. International technology roadmap for semiconductors. http://public.itrs.net/, 2003.

[SL05] Thilaka Sumanaweera and Donald Liu. Medical image reconstruction with the FFT. In Matt Pharr, editor, *GPU Gems 2*, chapter 48, pages 765–784. Addison Wesley, March 2005.

Revision History

- 30 September 2005: Submitted to Lockheed-Martin.
- 8 October 2005: MFLOPS graphs added.
- 16 October 2005: Integrated Daniel Horn comments; filed as technical report.