Graphics Hardware (2003)
M. Doggett, W. Heidrich, W. Mark, A. Schilling (Editors)

The FFT on a GPU

Kenneth Moreland[1] and Edward Angel[2]

[1] Sandia National Laboratories, Albuquerque, NM, USA
[2] Department of Computer Science, University of New Mexico, Albuquerque, NM, USA

Abstract

The Fourier transform is a well known and widely used tool in many scientific and engineering fields. The Fourier transform is essential for many image processing techniques, including filtering, manipulation, correction, and compression. As such, the computer graphics community could benefit greatly from such a tool if it were part of the graphics pipeline. As of late, computer graphics hardware has become amazingly cheap, powerful, and flexible. This paper describes how to utilize the current generation of cards to perform the fast Fourier transform (FFT) directly on the cards. We demonstrate a system that can synthesize an image by conventional means, perform the FFT, filter the image, and finally apply the inverse FFT in well under 1 second for a 512 by 512 image. This work paves the way for performing complicated, real-time image processing as part of the rendering pipeline.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Bitmap and framebuffer operations I.4.3 [Image Processing and Computer Vision]: Filtering

1. Introduction

Scientists and engineers have been using digital signal processing to manipulate images since the mid 1960's[1]. Over the past 40 years a wide variety of techniques have been developed to help enhance images for better human visual perception and autonomous machine perception. However, very little of this technology has been applied to real time computer synthesized graphics. The high computational intensity of such operations has been prohibitive. The graphics hardware that generated these images was, until very recently, not flexible enough to perform any but the simplest image filtering. Also, moving the image data to and from another processing unit, such as the CPU, is in itself a serious bottleneck. Fortunately, the latest releases of graphics hardware now have the power and flexibility to perform even the most complicated image filtering.

Digital image processing algorithms should be a good fit for modern GPU hardware. Any digital image processing technique entails a repetitive operation on the pixels of an image. Graphics processors are designed to perform a block of operations on groups of vertices or pixels, and they do this very efficiently. Furthermore, the performance growth of graphics hardware has been exceeding Moore's law, with an annual increase in performance rate of over 2.4. In addition, the functionality of these cards has also increased dramatically. Nearly every component on the GPU can now be reprogrammed. A GPU is no longer a fixed pipeline but is now better described as a SIMD parallel processor[2] or a streaming processor[3]. Furthermore, these processors can now perform computations on full 32 bit floating point values as opposed to 8 bit fixed values of one generation ago.

Fourier domain processing is not currently done for real-time graphics synthesis because performing transforms on a CPU requires data to be moved to and from the graphics card, a serious bottleneck. Furthermore, commodity graphics hardware was not powerful enough to perform the FFT necessary for complicated image processing. However, the current generation of graphics cards have the power, programmability, and floating point precision required to perform the FFT efficiently.

This paper describes how we used a commodity graphics card to perform the FFT and filter images. We start by providing an overview of the FFT al-

gorithm to facilitate understanding of our implementation. We will then discuss some properties of real Fourier transforms that allow us to more efficiently implement the FFT on a graphics card. Finally, we will discuss the details of our implementation.

2. The Fast Fourier Transform

A primary component to signal processing is the impulse response. The impulse response of a filter is its response to a single impulse, or, in the case of digital image filtering, its response to a single active pixel. Any linear and time-invariant filter can be completely characterized by its impulse response[4].

A digital image filter can be applied by performing a *convolution* of the input image and the filter's impulse response. Unfortunately, the convolution operation can be very computationally intensive. Performing a straightforward convolution requires every pixel in the input image to be multiplied by every pixel in the impulse response. Some modern graphics cards have hardware to perform a convolution[5], but they are limited to impulse responses of only three or four pixels wide.

For wider impulse responses, there are faster ways to perform a convolution. A common technique is to use Fourier transforms. A Fourier transform converts an image from the *spatial* domain to the *frequency* domain. An image in the spatial domain, as it is customarily represented, is defined by color values at spatial locations in the image. An image in the frequency domain is defined by amplitudes and phase shifts of the various sinusoids that make up the image. An important feature of the Fourier transform is that a convolution in the spatial domain can be done with a simple piecewise multiplication in the frequency domain. Thus, to filter an M by N image with an M by N frequency response takes $O\left((MN)^2\right)$ time in the spatial domain but only $O(MN)$ time in the frequency domain once the Fourier transform is performed.

When applying an impulse response in the frequency domain, the majority of the work is spent by applying the Fourier transform and its inverse. An efficient Fourier transform algorithm, the fast Fourier transform (FFT), has been known for at least 40 years[6]. The FFT can perform the Fourier transform or its inverse of an M by N image in $O(MN(\log M + \log N))$ time. The FFT makes tangible the computational intensity of processing even large images with complicated filters.

We now give a very brief overview of the discrete Fourier transform and the fast Fourier transform algorithm. More complete and formal descriptions of Fourier theory and the FFT can be found in a wide variety of articles and textbooks[1, 4, 7, 8].

Fourier analysis is based on the idea that every function is composed of a set of potentially infinite sinusoidal waves. The functions we are interested in are discrete functions of finite length. It can be shown that the frequencies for any such discrete function, $f(x)$, can be fully represented by another discrete function, $F(u)$, with an equal number of samples. The discrete Fourier transform gives the relationship between these two functions as

$$\mathcal{F}\{f(x)\} = F(u) = \frac{1}{N}\sum_{x=0}^{N-1} f(x)W_N^{ux} \tag{1}$$

$$\mathcal{F}^{-1}\{F(u)\} = f(x) = \sum_{u=0}^{N-1} F(u)W_N^{-ux} \tag{2}$$

where N is the number of samples in $f(x)$ and where $W_N = e^{-j2\pi/N}$. By convention, we give functions in time or spatial domains lower case letters and functions in the frequency domain upper case letters. To get these equations, we assume the functions $f(x)$ and $F(u)$ are periodic. That is, $f(x) = f(x + iN)$ and $F(u) = F(u + iN)$ for any integer i.

Equations 1 and 2 define the Fourier transform for one dimensional discrete functions, but we wish to use them on images, which are two dimensional. The two dimensional Fourier transform (or its inverse) can be calculated by applying the transform in one direction and then in the other direction. Likewise for higher dimensions. Therefore, for simplicity, we need only discuss the transforms in one dimension and apply that to higher dimensions.

The FFT algorithm works on the principle of divide and conquer. The input is split into two halves, the FFT is applied on each half, and the two are combined to form a full transform. We split the input into the sequence of values with even indices, $f^{\mathrm{e}}(x) = f(2x)$, and odd indices, $f^{\mathrm{o}}(x) = f(2x+1)$. For ease of explanation, let us also temporarily define $F'(u) = NF(u)$. We can rewrite Equation 1 as follows:

$$\begin{aligned} F'(u) &= \sum_{x=0}^{N-1} f(x)W_N^{xu} \\ &= \sum_{x=0}^{N/2-1} f^{\mathrm{e}}(x)W_N^{2xu} + \sum_{x=0}^{N/2-1} f^{\mathrm{o}}(x)W_N^{(2x+1)u} \\ &= \sum_{x=0}^{N/2-1} f^{\mathrm{e}}(x)W_N^{2xu} + W_N^u \sum_{x=0}^{N/2-1} f^{\mathrm{o}}(x)W_N^{2xu} \end{aligned} \tag{3}$$

Using the definition of W_N, we get

$$W_N^2 = e^{-j4\pi/N} = e^{-j2\pi/(N/2)} = W_{N/2}$$

Plugging back into Equation 3, we get

$$F'(u) = \sum_{x=0}^{N/2-1} f^{\mathrm{e}}(x) W_{N/2}^{xu} + W_N^u \sum_{x=0}^{N/2-1} f^{\mathrm{o}}(x) W_{N/2}^{xu}$$
$$= F'^{\mathrm{e}}(u) + W_N^u F'^{\mathrm{o}}(u) \quad (4)$$

Thus, Equation 4 shows us how to combine two half transforms into a full transform. The same analysis yields similar results for the inverse Fourier transform.

3. Index Magic

Note that in Equation 4, the sub-sequences are shuffled together in the original sequence. The straightforward approach to the recursion would be to copy samples from the input array into two separate sub arrays. However, this would require an excessive amount of data copying in what is usually a time critical application.

Instead, one uses a dynamic programming approach that performs the FFT iteratively. A common tactic is to reverse the bits of the input indices[8, 9]. This allows the program to treat the sub-sequences as contiguous chunks of the array, and adjacent chunks are combined in much the way the merge-sort algorithm works. However, to avoid this initial reordering of the array, we instead kept the original order of the sequence in the array. We will now discuss how to index into the array to perform the FFT without any reordering.

Keep in mind that even though we pack all values into a single array of size N, at any particular iteration of the FFT we actually consider the array to be partitioned into a number of valid frequency spectra (remember that Equation 4 shows us how to merge two valid frequency spectra into a single larger spectrum). At iteration $i = 0$, before the FFT starts, we have N partitions of size 1; at $i = 1$, we have $N/2$ partitions of size 2; at $i = 2$, we have $N/4$ partitions of size 4, and so on. In general, at iteration i we have $N/2^i$ partitions of size 2^i. Let $P_{i,p}[u]$ be the uth entry of the pth partition at iteration i. The sequence $P_{i,p}$ contains the frequency spectrum of some subsequence of the input data.

Consider array $A[n]$ containing the data with which we are performing the FFT (and therefore consisting of partitions $P_{i,p}$ for all applicable p at any given iteration i). We pack our partitions in the array such that $P_{i,p}[u] \equiv A[n]$ if $p = n \mod N/2^i$ and $u = n \operatorname{div} N/2^i$ (remember that there are $N/2^i$ partitions at iteration i). Conversely, $A[n] \equiv P_{i,p}[u]$ if $n = (p + uN/2^i) \mod N$. Note that at the final iteration, $i = \log N$, when only one partition remains, $A_{\log N}[n] \equiv P_{\log N,0}[n]$. Using this convention will result in the array containing the final Fourier transform without further reordering.

We now will show that this indexing results in the appropriate Fourier transform. Per Equation 4, we must calculate $P_{i,p}[u]$ as

$$P_{i,p}[u] = P^{\mathrm{e}}_{i_p}[u] + W^u_{2^i} P^{\mathrm{o}}_{i_p}[u] \quad (5)$$

By using previous definitions, we determine that

$$\begin{aligned} P^{\mathrm{e}}_{i,p}[u] &\equiv P_{i,p}[2u] \\ &\equiv A[p + 2uN/2^i] = A[p + uN/2^{i-1}] \\ &\equiv P_{i-1,p}[u] \end{aligned}$$

Likewise

$$\begin{aligned} P^{\mathrm{o}}_{i,p}[u] &\equiv P_{i,p}[2u+1] \\ &\equiv A[p + (2u+1)N/2^i] \\ &= A[(p + N/2^i) + uN/2^{i-1}] \\ &\equiv P_{i-1,p+N/2^i}[u] \end{aligned}$$

From this we can see that each $P^{\mathrm{e}}_{i,p}$ and $P^{\mathrm{o}}_{i,p}$ maps correctly to a unique partition from the previous iteration.

We can now plug the indexing scheme into Equation 5 and get the operation performed on the array at each step of the FFT.

$$A[n] \leftarrow A[n + uN/2^i] + W^u_{2^i} A[n + uN/2^i + N/2^i] \quad (6)$$

where $u = n \operatorname{div} N/2^i$. Note that the indices into A need to be placed in the range $[0..N-1]$ via modulo N. Also note that in order to be correct, Equation 6 must be applied simultaneously to all values of A.

4. Frequency Compression

In general, the values generated by the Fourier transform are complex, even when the original samples are real, as is typical in a graphical image. However, if the original samples are real, the Fourier transform has much symmetry. It can be easily shown from Equation 1 that if f is a real sequence, then $\forall u, F(u) = F^*(N-u)$. That is, nearly all the values in F occur as pairs of complex conjugates. The only two exceptions are for $u = 0$ and $u = N/2$. Mathematically, the values at $F(0)$ and $F(N/2)$ are conjugates with themselves, and therefore must be real.

Computing both parts of these conjugates is inefficient. Press[9] gives an efficient method for computing the FFT of real functions. First, consider two real functions $f(x)$ and $g(x)$. These functions can be any pair of rows in our image. We define a new, complex function, $h(x) = f(x) + jg(x)$. $h(x)$ can be easily defined without any data movement by pointing to one row of our image as the real values and another row as the imaginary values. We can calculate $H(u)$ in half the time it would take to calculate $F(u)$ and $G(u)$ individually.

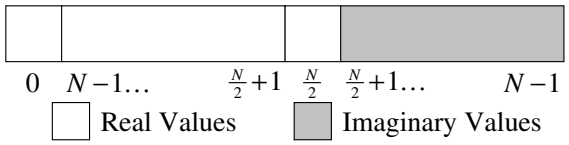


Figure 1: *Packing for 1D Fourier transform.*

Of course, $H(u)$ itself is not useful to us. We need to extract the $F(u)$ and $G(u)$ functions from $H(u)$. Because the Fourier transform is linear, we know

$$H(u) = F(u) + jG(u) \tag{7}$$

We also know that $F(u)$ and $G(u)$ display the conjugate symmetry discussed above because $f(x)$ and $g(x)$ are real. Combining this information, we find that

$$\begin{aligned} F(u)_R &= \tfrac{1}{2}(H(u)_R + H(N-u)_R) \\ F(u)_I &= \tfrac{1}{2}(H(u)_I - H(N-u)_I) \\ G(u)_R &= \tfrac{1}{2}(H(u)_I + H(N-u)_I) \\ G(u)_I &= -\tfrac{1}{2}(H(u)_R - H(N-u)_R) \end{aligned} \tag{8}$$

where the R and I subscripts stand, respectively, for the real and imaginary components of a complex number.

Again, we call out the special cases for when $u = 0$ or $u = N/2$. For these values of u, $H(u) = H(N-u)$. It falls out from Equation 8 that $F(0)_R = H(0)_R$, $F(0)_I = 0$, $G(0)_R = H(0)_I$, $G(0)_I = 0$. Likewise for values of $F(N/2)$ and $G(N/2)$.

Computing the one dimensional FFT by combining pairs of rows as described above takes about half the time as computing the FFT directly. It also saves us from having to allocate separate buffers for real and complex numbers. We can also store the final result, after being "untangled" by Equation 8, in an array the same size as the input. Storing all N complex values would require $2N$ floating points. However, we can throw out $N/2 - 1$ complex values as they are conjugates of values we will be storing. We can also throw away the imaginary part of values at indices 0 and $N/2$ as we know they are always real. This leaves us with only N floating point values we need to store in our array. Figure 1 shows how we efficiently packed the values for the Fourier transform. More formally, the values for transformed function $F(u)$ are stored in a two dimensional array A of size N as follows.

$$\begin{aligned} F(u)_R &= \begin{cases} A[u] & 0 \le u \le \frac{N}{2} \\ A[N-u] & \frac{N}{2} < u < N \end{cases} \\ F(u)_I &= \begin{cases} 0 & u \in \{0, \frac{N}{2}\} \\ -A[N-u] & 0 < u < \frac{N}{2} \\ A[u] & \frac{N}{2} < u < N \end{cases} \end{aligned} \tag{9}$$

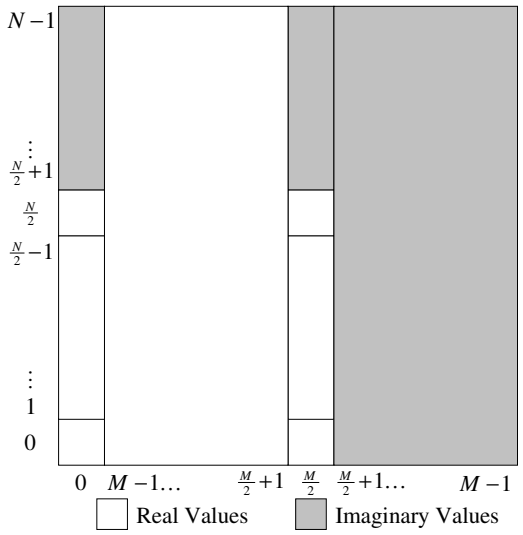


Figure 2: *Packing for 2D Fourier transform.*

After performing the 1D FFT on the rows of our image and packing them as shown in Figure 1, we must perform the FFT on the columns. Two of the columns, those at indices 0 and $N/2$, contain real numbers. We can pair these up and perform the FFT on them together with the method described above. The rest of the columns, when properly paired, form sequences of complex numbers. We perform a traditional complex to complex FFT on these sequences. Figure 2 shows how the final result is packed, which is similar to that given by Pratt[10]. More formally, the values for transformed function $F(u,v)$ are stored in a two dimensional array A of dimensions M by N as follows.

$$F(u,v)_R = \begin{cases} A[u][v] & u \in \{0, \frac{M}{2}\}, v \le \frac{N}{2} \\ A[u][N-v] & u \in \{0, \frac{M}{2}\}, v > \frac{N}{2} \\ A[u][v] & 0 < u < \frac{M}{2}, v \in \{0, \frac{N}{2}\} \\ A[u][N-v] & 0 < u < \frac{M}{2}, 0 < v < \frac{N}{2} \\ A[u][N-v] & 0 < u < \frac{M}{2}, \frac{N}{2} < v < N \\ A[M-u][v] & \frac{M}{2} < u < M \end{cases}$$

$$F(u,v)_I = \begin{cases} 0 & u \in \{0, \frac{M}{2}\}, v \in \{0, \frac{N}{2}\} \\ -A[u][N-v] & u \in \{0, \frac{M}{2}\}, 0 < v < \frac{N}{2} \\ A[u][v] & u \in \{0, \frac{M}{2}\}, \frac{N}{2} < v < N \\ -A[M-u][v] & 0 < u < \frac{M}{2}, v \in \{0, \frac{N}{2}\} \\ -A[M-u][N-v] & 0 < u < \frac{M}{2}, 0 < v < \frac{N}{2} \\ -A[M-u][N-v] & 0 < u < \frac{M}{2}, \frac{N}{2} < v < N \\ A[u][v] & \frac{M}{2} < u < M \end{cases} \tag{10}$$

The inverse FFT can also be performed in a similar manner. First, we "tangle" pairs of rows by evaluating the function in Equation 7. Once the inverse FFT is performed on the tangled equation, the two correct real functions will already be placed in their respective entries in the image.

5. Implementation

We implemented the FFT algorithm as described in sections 2 through 4 on an nVidia GeForce FX 5800 Ultra graphics card. This graphics card features fully programmable vertex and fragment units, 32 bit floating point frame buffers and textures, and full 32 bit floating point enabled throughout the entire pipeline. The vertex and fragment units were programmed with OpenGL and the Cg computer language and runtime libraries[11, 12].

Implementing the FFT on the graphics card is relatively straightforward. The FFT is inherently an image-based algorithm. As such, we perform the FFT by executing a fragment program on every pixel at each step in a SIMD-like fashion. To execute these fragment programs, we draw quadrilaterals that are parallel to the screen, which causes the rasterizer to create a single fragment for every pixel on the screen and thereby invoking the fragment program for every pixel. Because the vertex programs are trivial (and in fact provided only for Cg convenience), we will focus solely on describing the fragment programs necessary for implementing the FFT.

Figure 3 shows a block diagram containing the steps required to perform a full 2D FFT on an image buffer or a full 2D IFFT on an array of frequencies. Each block represents a single operation performed on each pixel of the buffer. Abstractly, we can consider the operations on all the pixels to be performed atomically and in parallel. In reality, each pass of a fragment program reads its inputs from one memory area (a texture) and writes to another memory area (a frame buffer), which eliminates any possibility of read-after-write control hazards. All the calculations can be done with only two frame buffers by using the render-to-texture extensions and swapping the buffer that behaves as a texture.

Although the same basic operation is performed on every pixel at every step in our algorithm, the fragment program will need to exhibit slightly different behaviors based on the position of the pixel. The GeForce FX architecture does not allow branching within its fragment programs, so the developer is left with two choices: calculate all possible cases and multiplex the result or load a different program variant for all possible cases. We chose the latter figuring the number of

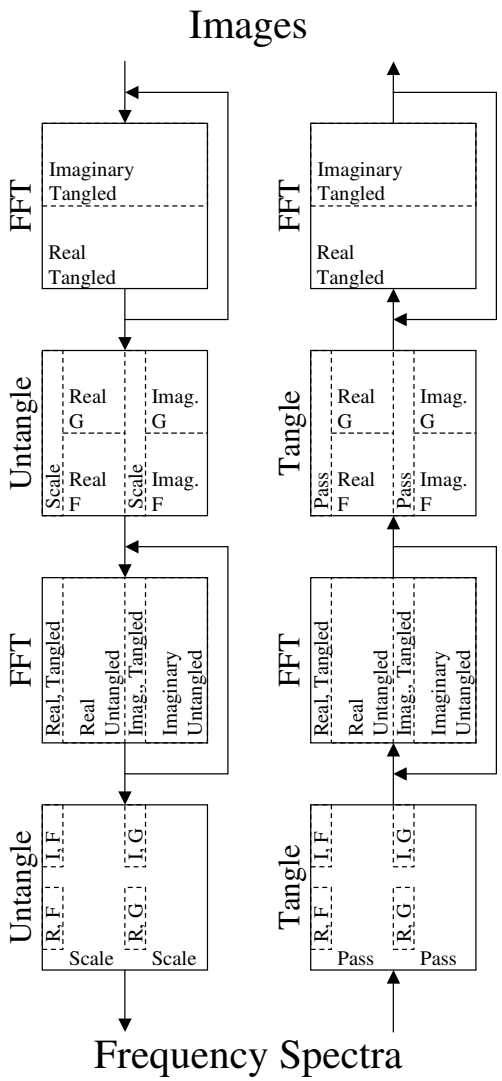


Figure 3: *Program flow for performing the Fourier transform and the inverse Fourier transform.*

instructions saved would outweigh the cost of changing fragment programs. Each block in Figure 3 shows a breakdown of the different cases encountered within an array and the program variants to handle these cases. We shall now describe the function of each fragment program required to perform the Fourier transform on an image. For purposes of this discussion, we will assume an image of dimensions M by N.

First, we perform an FFT on each row of the image. To save computation and avoid doubling our memory for handling complex numbers, we use the tangling

technique discussed in section 4. The lower $N/2$ rows are the real values and the upper $N/2$ rows are the imaginary values. The real values in row n are coupled with the imaginary values in row $n + N/2$. The fragment programs used for real values and imaginary values are similar, but differ in their indexing and the value computed. We therefore load a separate program for the real and imaginary values. By grouping all the real values together and likewise all the imaginary values rather than interlace them, we can run the appropriate fragment program on all the entries by "rendering" only two quadrilateral primitives. Each fragment program performs one iteration of the FFT algorithm, so this computation must be repeated $\log M$ times.

Second, we untangle the values using Equation 8, scale by $1/M$, and lay out the data in each row as shown in Figure 1. Note that the lower half of the values are indexed differently than the upper half of the values. There are also slightly different equations for computing the real and imaginary values. We therefore load four fragment programs to handle these four cases. Also recall that the untangling processes has special cases for the columns 0 and $M/2$. The FFT operation of the previous step lays down the data in such a way that these two columns need only be scaled.

Next, we must perform the Fourier transform for each column. At this point, there are only two columns that contain real values. We perform the FFT on these two columns in the same way that we performed the FFT on all the rows earlier, by using the tangling technique. The rest of the entries in the array contain one part of a true complex number. The tangling technique therefore no longer applies, and we can perform the FFT directly on these values without increasing memory space or calculations. In all, we have four programs: real and imaginary forms of the tangled FFT as well as real and imaginary forms of the standard FFT. In truth, the operation of the FFT is the same regardless of whether the values are tangled. The only difference between the tangled and untangled forms of the FFT is one of indexing. As before, we must run the FFT fragment programs $\log N$ times to perform a complete FFT.

Finally, we must perform the untangling and scaling. As stated above, the majority of the columns are not tangled in the first place. The FFT operation of the previous step leaves its values in the correct places per Figure 2, and those values therefore need only to be scaled by $1/N$. The two columns at indices 0 and $M/2$ are untangled in the same manner as we untangled the rows before.

The inverse FFT is performed by reversing the order of operations of the forward FFT. The only difference besides the order of operations is that tangling per Equation 7 is performed in lieu of untangling and that scaling is not performed.

Our ultimate goal is to perform image filtering by multiplying the frequency spectra of images and impulse responses. We therefore need one final fragment program to perform this operation. Building a single pass fragment program to do this is trivial. Simply look up the appropriate real and imaginary values, which can be located with equation 10, perform the complex multiplication, and write the values in the same packing as shown in Figure 2.

Program	Instructions		Invocations
	Arithmetic	Texture	
FFT	27	3	$2MN(\log N + \log M)$
Untangle	4	2	$MN - 4$
Scale	1	1	$MN + 2N$
Tangle	1	2	$MN - 4$
Pass	0	1	$MN + 2N$
Multiply	66	4	MN

Table 1: *Instructions executed per fragment. All program variants (e.g. real vs. imaginary) are grouped together because their instruction counts are equal.*

Table 1 shows how many instructions are executed per fragment for each fragment program involved in computing the Fourier transform. The final column, Invocations, specifies the number of fragments each program processes while filtering an M by N image using the FFT method. That is, we apply the FFT to an image, piecewise multiply the frequencies by a static impulse response, and then apply the inverse FFT. Each texture lookup instruction involves the reading of four 32 bit floating point numbers (an RGBA tuple) from texture memory. Also, each iteration of any one of the fragment programs results in the writing of four 32 bit floating point numbers (again a single RGBA tuple) into the frame buffer. Due to the nature of the hardware, each arithmetic instruction can perform an operation on up to four different sets of values held in four-tuple registers. As such, we compute the FFT for all four color channels of an image in one shot.

Program	Arithmetic Operations	Megabytes Read	Megabytes Written
FFT	1,132,462,080	1,920	640
Untangle	4,194,288	32	16
Scale	1,050,624	16	16
Tangle	1,048,572	32	16
Pass	0	16	16
Multiply	69,206,016	64	16
Total	1,207,961,580	2,080	720

Table 2: *Operations performed per fragment program and total for performing FFT on a* 1024^2 *image.*

Table 2 gives the number of operations required to perform an image filtering with the FFT method. It gives the total amount of arithmetic operations performed and the total amount of data read from and written to memory while performing a filtering of an image of size 1024 by 1024.

6. Results

We built a test application that renders a simple geometry (the Utah teapot) and applies a filter on the resulting image with the FFT method described in this paper. The application filters four color channels (red, green, blue, and alpha). The graphics hardware, which was optimized for calculation on four component color values, performs the calculations of all four color channels simultaneously. Figure 4(a) shows an unfiltered image. Figure 4(b) shows the frequencies of that image. Figures 4(c) through 4(f) show various filters that we applied to the image.

Image Size	Rendering Rate (Hz)	Arithmetic (sec)	Texture Lookup (sec)
1024^2	0.37	1.9	0.60
512^2	1.6	0.44	0.13
256^2	6.7	0.09	0.03
128^2	25	0.01	0.007

Table 3: *FFT filtering performance.*

Table 3 gives the performance of our test application. The rendering rate is measured with respect to the total time needed to produce a frame, including issuing drawing commands to render the teapot. The last two columns were calculated by removing the texture and arithmetic operations from the fragment programs and observing the effect on the frame rate. Combining the information from Tables 2 and 3, we find that our application is performing on average about 2.5 GigaFLOPS (assuming each arithmetic operation performs four floating point calculations) and reading texture memory at a rate of about 3.4 GB/sec. Although our current implementation may not be fast enough to support real-time or interactive applications yet, performance improvements from future generation graphics cards should put this FFT within interactive rates.

We now compare our FFT run on a GPU with an FFT run on a traditional CPU. Using the FFTW library (freely available from www.fftw.org) on our 1.7 GHz Intel Zeon, we performed a Fourier transform, a piecewise complex multiplication, and an inverse Fourier transform on four 1024 by 1024 arrays of 32 bit floating point numbers. We found this operation to take about 0.625 seconds (or up to 2 seconds if scaling is not performed correctly). We feel that our GPU implementation, which took about 2.7 seconds, is comparable to the highly optimized FFTW library. Also realize that if we wish to perform Fourier analysis on images generated by or otherwise already stored in the graphics card, the GPU implementation saves us from having to the transfer these data between CPU and GPU memory.

As we mentioned earlier, the use of Fourier analysis extends well beyond the range of digital image filtering. Another example related the the realm of computer graphics is the use of the Fourier projection-slice theorem to perform renderings of translucent 3D volumes[13, 14]. Another use of Fourier transforms is the generation of random textures[15]. Figures 5(a) and 5(b) show textures generated by building frequencies with known magnitudes and random phase angles.

7. Acknowledgments

This work was partially supported by Sandia National Laboratories. Sandia is a multi-program laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000. The work was also partially supported by the National Science Foundation under grant number CDA-9503064.

We would like to especially thank the Graphics Hardware 2003 papers co-chairs William Mark and Andreas Schilling as well as all those who reviewed this paper for their quick turnaround and insightful comments.

References

1. R. C. Gonzalez and P. Wintz, *Digital Image Processing*. Addison Wesley, (1983). ISBN 0-201-03045-4.
2. S. Krishnan, N. H. Mustafa, S. Muthukrishnan, and S. Venkatasubramanian, "Extended intersection queries on a geometric SIMD machine model", in *14th Annual ACM Symposium on Parallel Algorithms and Architectures (submitted)*, (2002).
3. B. Khailany, W. J. Dally, S. Rixner, U. J. Kapasi, P. Mattson, J. Namkoong, J. D. Owens, B. Towles, and A. Chang, "Imagine: Media processing with streams", *IEEE Micro*, (2001).
4. C. L. Phillips and J. M. Parr, *Signals, Systems, and Transforms*. Englewood Cliffs, New Jersey: Prentice Hall, (1995). ISBN 0-13-795253-8.
5. M. Woo, J. Neider, T. Davis, and D. Shreiner,

OpenGL Programming Guide. Reading Massachusetts: Addison Wesley, 3rd ed., (1999). ISBN 0-201-60458-2.

6. J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series", *Mathematics of Computation*, **19**(90), pp. 297–301 (1965).

7. J. W. Cooley, P. A. W. Lewis, and P. D. Welch, "Application of the fast fourier transform to computation of fourier integrals, fourier series, and convolution integrals", *IEEE Transactions on Audio and Electroacoustics*, **AU-15**(2), pp. 79–84 (1967).

8. S. J. Orfandis, *Introduction to Signal Processing*. Upper Saddle River, New Jersey: Prentice Hall, (1996). ISBN 0-13-209172-0.

9. W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C++*. Cambridge University Press, second ed., (2002). ISBN 0-521-75033-4.

10. W. K. Pratt, *Digital Image Processing*. John Wiley & Sons, Inc, (1978). ISBN 0-471-01888-0.

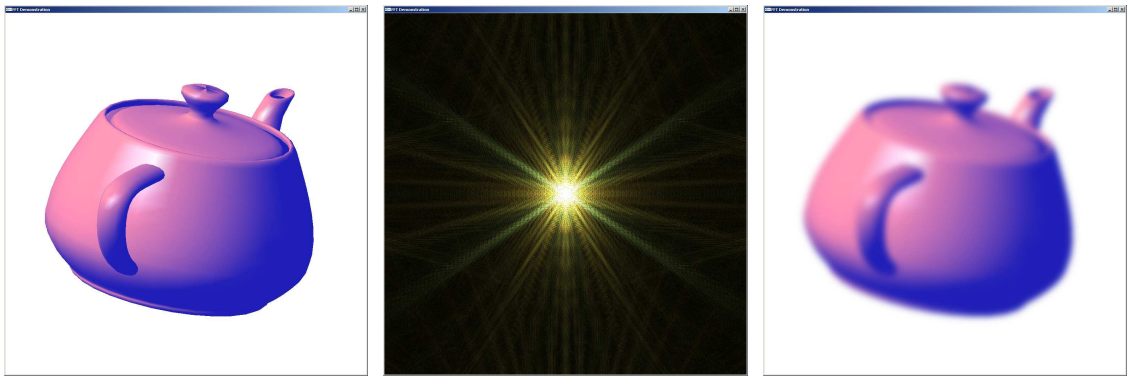
11. R. Fernando and M. J. Kilgard, *The Cg Tutuorial*. Addison Wesley, (March 2003). ISBN 0-321-19496-9.

12. W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard, "Cg: A system for programming grapihcs hardware in a c-like language", in *Proceedings of SIGGRAPH 2003*, (July 2003).

13. T. Malzbender, "Fourier volume rendering", *ACM Transactions on Graphics*, **12**(3), pp. 233–250 (1993).

14. T. Totsuka and M. Levoy, "Frequency domain volume rendering", in *Proceedings of ACM SIGGRAPH 1993*, pp. 271–278, (July 1993).

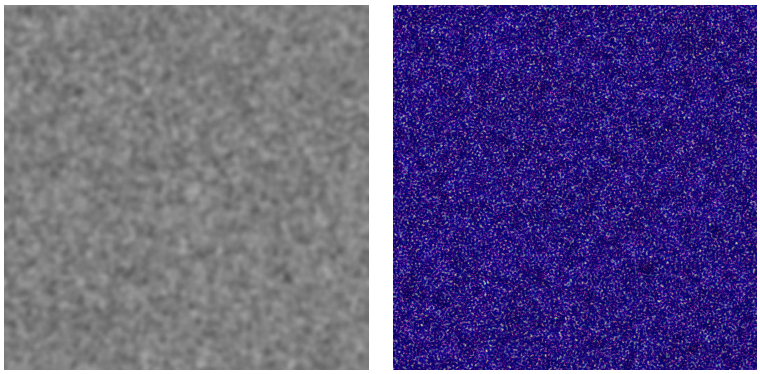
15. D. S. Ebert, F. K. Musgrave, D. Peachey, K. Perlin, and S. Worley, *Texturing & Modeling: A Procedural Approach*. AP Professional, 2nd ed., (July 1998). ISBN 0-12-228730-4.



(a) Original image. (b) Frequency magnitudes. (c) Image with low pass filter.

(d) Image with high pass filter. (e) Image with Laplacian filter. (f) Image with non-photorealistic motion blur.

Figure 4: *Examples of using the FFT for image filtering.*



(a) Low frequencies. (b) Mixed frequencies.

Figure 5: *Textures generated from frequencies with random phase angles.*