FAST COMPUTATION OF GENERAL FOURIER TRANSFORMS ON GPUS

D. Brandon Lloyd   *Chas Boyd*   *Naga Govindaraju*

Microsoft Corporation

ABSTRACT

We present an implementation of general FFTs for graphics processing units (GPUs). Unlike most existing GPU FFT implementations, we handle both complex and real data of any size that can fit in a texture. The basic building block for our algorithms is a radix-2 Stockham formulation of the FFT for power-of-two data sizes that avoids expensive bit reversals and exploits the high GPU memory bandwidth efficiently. We implemented our algorithms using the DirectX 9 API, which enables our routines to be used on many of the existing GPUs today. We have performed comparisons against optimized CPU-based and GPU-based FFT libraries (Intel Math Kernel Library and NVIDIA CUFFT, respectively). Our results on an NVIDIA GeForce 8800 GTX GPU indicate a significant performance improvement over the existing libraries for many input cases.

Index Terms— graphics hardware, FFT, GPGPU

1. INTRODUCTION

Though commodity graphics processors (GPUs) have been traditionally used for real-time 3D rendering in visualization applications and video games, their raw computational power and relatively low cost has made them increasingly attractive for more general purpose, data-parallel computations. The performance of GPUs comes from their large number of cores and high memory bandwidth. For example, the GeForce 8800 GTX GPU has 128 scalar processors and 86 GB/s peak memory bandwidth. GPUs are well-suited for a number of multimedia applications including signal processing for audio, images, and video. An important component of these applications is the Fast Fourier Transform (FFT). In this paper we discuss how the GPU can be used for high performance computation of general FFTs.

A number of FFT implementations for the GPU already exist, but these are either limited to specific hardware or they are limited in functionality. Probably the most general FFT implementation for GPUs available today is the CUFFT library [1]. CUFFT handles FFTs of varying sizes on both real and complex data. However, CUFFT is written in CUDA [2], a programming interface that is specific to only the most recent NVIDIA GPUs. A recent survey of over 800,000 gaming enthusiasts revealed that only about 15% of those surveyed had a GPU capable of running CUDA [3]. Those numbers are probably much lower for a more general user base. To support multiple generations of GPUs from different vendors, some FFT libraries are written in the high-level shading languages found in standard graphics APIs such as OpenGL or DirectX [4, 5, 6, 7, 8, 9]. However, all of these implementations share the same limitation – they are restricted to sizes that are a power of two.

In this paper we describe our FFT library which is both general in terms of the GPUs it supports and its functionality. Our current FFT library is written using DirectX 9. We handle 1D and 2D FFTs for power-of-two and non-power-of-two sizes on both real and complex data, using a simple API that chooses the appropriate algorithm for a given input. Our focus has been on general algorithms that ensure the availability of functionality over a wide range of GPUs more than on attaining maximum performance for any specific GPU. Nevertheless, even our general implementation on newer GPUs typically outperforms the same computation on the CPU, while achieving comparable performance to vendor-specific implementations such as CUFFT.

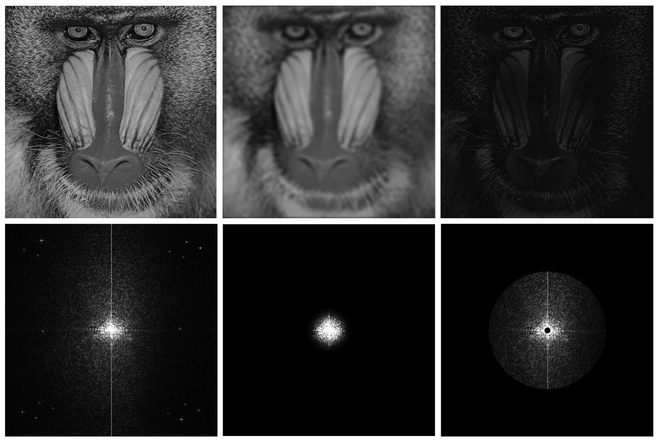

Fig. 1. **Simple filtering application.** A user interactively adjusts a simple bandpass filter. The resulting image is shown on the top row and the power spectrum of its Fourier Transform is shown in the bottom row. Using our FFT implementation the application runs at 66 Hz for a 1024×1024 image.

The rest of this paper is organized as follows. In Section 2 we present some background information on the GPU programming model and the Fourier Transform. We briefly discuss previous work in Section 3. In Section 4 we present the details of our FFT routines. We show some results using our library in Section 5 and conclude with some ideas for future work.

2. BACKGROUND

2.1. GPU Programming Model

Standard graphics APIs use a programming model for GPUs that is essentially stream processing. Kernels are run independently on the data elements of input streams to produce an output stream. Two main types of kernels are used on the GPU: vertex programs and fragment programs. Vertex programs transform an input stream of vertex records consisting of vertex positions and other vertex attributes (such as color). The vertices of the transformed output stream are grouped together to form 2D primitives (e.g. every 4 vertices form a quadrilateral). The primitive is rendered to a 2D buffer (either the framebuffer or a texture map). A *fragment* is generated for each buffer element covered by the primitive. A fragment program is run on the fragment stream. To generate an output value for each fragment, the fragment program uses data from a buffer of constants, the bilinearly interpolated vertex attribute values, and texture maps with random read access. The interpolated attribute values are commonly used to compute an address from which to read the texture. The output value is then stored at a fragment's position in the buffer.

2.2. Discrete Fourier Transform

An N point Discrete Fourier Transform (DFT), $\mathcal{F}_N$, of a sequence $f(n)$ is computed using the following equation:

$$F(k) = \mathcal{F}_N\{k, f\} = \sum_{n=0}^{N-1} f(n) e^{-2\pi i k n/N}, \tag{1}$$

where $n \in [0, N-1]$ and $k \in [0, N-1]$. The sequence $f(n)$ is referred to as the time domain and $F(k)$ as the frequency domain. The Fast Fourier Transform (FFT) is a family of algorithms for efficiently computing the DFT. The "Decimation in Time" (DIT) algorithm recursively splits the time domain into a DFT of even and odd elements:

$$\begin{gathered}
\mathcal{F}_N\{k, f\} = \mathcal{F}_{N/2}\{k', f_e\} + T_N(k)\mathcal{F}_{N/2}\{k', f_o\} \\
k' = k \bmod N/2 \\
f_e(n) = f(2n) \qquad f_o(n) = f(2n+1) \\
T_N(k) = e^{-2\pi i k/N}.
\end{gathered}$$

The T_N values are referred to as *twiddle factors*. This algorithm is often called the Cooley-Tukey algorithm after the researchers who published it [10]. The "Decimation in Frequency" (DIF) algorithm is similar to DIT, except that it recursively splits the frequency domain into DFTs for even and odd k.

$$\begin{aligned}
\mathcal{F}_N\{k, f\} &= \begin{cases} \mathcal{F}_{N/2}\{k/2, f_e\} & \text{for } k \text{ even} \\ \mathcal{F}_{N/2}\{(k+1)/2, f_o\} & \text{for } k \text{ odd} \end{cases} \\
f_e(n') &= f(n') + f(n' + N/2) \\
f_o(n') &= T_N(n')\big(f(n') - f(n' + N/2)\big) \\
T_N(n') &= e^{-2\pi i n'/N}
\end{aligned}$$

where $n' \in [0, N/2 - 1]$. Both of these algorithms require that N be a power of two. The algorithms can be performed in-place by simply concatenating the subtransforms in a single array. However, this requires a reordering of the elements. For the DIT, the input must be in *bit-reversed* order, that is, the value corresponding to index n is actually found at the index formed by reversing the bits of n. The DIF starts with the input in natural order, but produces bit-reversed output. The inverse DFT can be computed by simply conjugating the twiddle factors and dividing the final values by N.

3. PREVIOUS WORK

Spitzer [5] and Mitchell et al. [6] perform a DIT FFT that retrieves the input indices and twiddle factors of each output element from a precomputed texture. This makes for a relatively simple fragment program. Moreland and Angel [4] modify the DIT indexing scheme to avoid the bit-reversal step. They also encode FFTs of real data into a single channel texture, which requires several fragment programs to be applied to different parts of the texture. Jansen et al. [7] move the bit-reversal step of a DIF to the input and reorder the dataflow to obtain a simpler indexing scheme more suitable for GPUs. Sumanaweera and Liu [8] render each subtransform of the FFT as a separate quadrilateral. This approach uses interpolators for index and twiddle factor calculations, but is slow for the first iterations of the FFT where there are many subtransforms. For these iterations they use an approach like that of Spitzer and Mitchell et al. Govindaraju et al. [9] use the Stockham formulation of the FFT described later in this paper to avoid the bit-reversal step. They also block the computations to maximize cache performance. All of these FFT implementations are for power-of-two sizes.

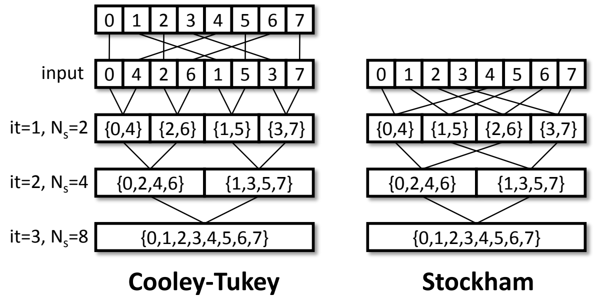

Fig. 2. **Dataflow for two DIT algorithms.** Both algorithms proceed iteratively, merging pairs of smaller FFTs into larger ones. Each box represents the FFT of the listed sequence elements. Unlike the Cooley-Tukey algorithm, the Stockham algorithm does not require an initial bit-reversal step.

4. OUR FFT IMPLEMENTATION

4.1. Stockham FFT

Fig. 2 shows the dataflow for the Cooley-Tukey algorithm. The initial bit-reversal permutation can be expensive because the memory accesses are incoherent. For our library we use the radix-2 Stockham FFT algorithm [11], which reorders the dataflow in order to eliminate the need for the bit-reversal. This algorithm requires twice as much memory because it does not perform the FFT in-place, but because textures cannot have simultaneous read and write access, the FFT must be performed out-of-place anyway.

Our implementation of the Stockham FFT uses 32-bit floating-point textures with 2 channels to store the real and imaginary components of complex data. We store a 1D array in each row of the texture and perform multiple 1D FFTs of the same length simultaneously. We render a single quadrilateral into an output texture that is the same size as the input texture, using the fragment program shown in Fig. 3. We then swap the input and output textures and repeat for all $\log_2(N)$ iterations.

4.2. 2D FFTs

The 2D FFT can be computed simply by computing 1D FFTs along the rows followed by 1D FFTs along the columns. Because traversing columns of a row-major 2D array stored linearly in memory has poor spatial locality, the FFT along columns is usually implemented by transposing the array, performing the transform on the rows, and transposing back. On a GPU, however, textures are swizzled in memory so as to preserve 2D locality. Thus, no transposes are necessary.

```
FFT_Rows(x, y, N, Ns, input)
{
    base   = floor(x / Ns)*(Ns/2);
    offset = x mod (Ns/2);
    x0 = base + offset;
    x1 = x0 + N/2;
    (Re0, Im0) = input[x0][y];
    (Re1, Im1) = input[x1][y];
    angle = -2*M_PI*(x/Ns);
    (ReT, ImT) =  (cos(angle), sin(angle));
    return (Re0 + ReT * Re1 - ImT * Im1,
            Im0 + ImT * Re1 + ReT * Im1 );
}
```

Fig. 3. **Pseudocode for FFT fragment program**. x and y are the positions of the fragment in the buffer. N is the size of the entire FFT and Ns is the size of the subtransform for the current iteration. input is an input texture.

4.3. Non-power of two sizes

Several algorithms exist for handling DFTs of lengths that are not a power of two. Mixed radix algorithms recursively split a DFT of length $N = N_x N_y$ into smaller DFTs of lengths N_x and N_y. This works best for lengths that are highly composite numbers. Other algorithms exist for lengths that are prime numbers. Rather than implement a large number of special cases for different lengths, we currently use what is commonly called the Bluestein z-chirp algorithm [11], which handles all non-power-of-two sizes. The algorithm is derived by substituting $kn = (k^2 + n^2 - (k-n)^2)/2$ into the complex exponential of Eq. 1 and rearranging the terms to get:

$$\begin{aligned} F(k) &= [e^{-\pi i k^2/N}] \sum_{n=0}^{N-1} [f(n) e^{-\pi i n^2/N}][e^{+\pi i (k-n)^2/N}] \\ &= b^*(k) \sum_{n=0}^{N-1} a(n) b(k-n) \\ a(n) &= f(n) b^*(n) \\ b(n) &= e^{-\pi i n^2/N}, \end{aligned}$$

where b^* is the conjugate of b. The summation is a linear convolution $c = a * b$. When a and b are large, the convolution can be performed more efficiently as a component-wise product in the frequency domain due to the following property:

$$\mathcal{F}\{a * b\} = \mathcal{F}\{a\} \otimes \mathcal{F}\{b\}.$$

Before performing the FFT, a and b should be zero padded to a size N' that is at least the size of the convolved signal in order to avoid aliasing. The lengths of a and b are N and $2N-1$, respectively, so the length of $a * b$ is $N + (2N-1) - 1 = 3N - 2$. However, we are only interested in values for $k \in [0, N-1]$. Aliasing beyond this range does not affect the solution, which means that it is sufficient that $N' \geq 2N - 1$.

The range of valid indices for the b is $[-(N-1), N-1]$. After zero padding, the indices lie in $[-(N-1), N' - (N-1)]$. Negative indices are inconvenient, so relying on the fact that the FFT convolution assumes periodic signals, we move the values in the negative range to the other end of the array so that the indices run from 0 to $N' - 1$. The entire convolution can then computed as follows:

$$c(k') = \mathcal{F}_{N'}^{-1}\left\{k', \mathcal{F}_{N'}\{k', a\} \otimes \mathcal{F}_{N'}\{k', b\}\right\},$$

where $k' \in [0, N'-1]$. The advantage of this algorithm is that N' can be chosen to be a power of two, which we already know how to handle efficiently.

For our implementation of this algorithm, we choose N' to be the next power of two greater than or equal to $2N-1$. We first compute the b vector and its Fourier transform $B(k') = \mathcal{F}_{N'}\{k', b\}$ on the CPU and store them in a texture. These can be reused for multiple transforms of the same size. We then render to a texture of width N' and compute $a(n')$ from the input textures containing b and f, and zero pad elements $n' > N - 1$. We compute $A(k') = \mathcal{F}_{N'}\{k', a\}$ using the power-of-two FFT routine described earlier. ($B(k')$ could also be computed on the GPU in the same way, though we currently do not do this). Another pass computes the component-wise product $A(k') \otimes B(k')$. We compute the inverse FFT of the result to get $c(k')$. The final pass computes $b^*(k) \otimes c(k)$. This algorithm can be performed on both rows and columns to compute 2D FFTs. Because this algorithm requires both a forward and inverse FFT of size N' which is approximately $2N$ in the best case and nearly $4N$ in the worst case, the algorithm is roughly 4 to 8 times slower than the Stockham algorithm would on similar power-of-two sizes.

Note that the reason we compute b on the CPU is that for large values of n, the n^2 term in $b(n)$ cannot be computed directly with enough precision with the single precision available on most GPUs. Only the most recent GPUs support double precision, so we currently compute b with double precision on the CPU.

4.4. Large FFTs

The total number of elements in an FFT in our current implementation is limited to the number of elements in the largest allocatable texture on the GPU. However, the number of rows or columns in a batch of 1D FFTs or a 2D FFT can exceed the maximum texture dimensions, but still have fewer than the maximum number of elements (e.g. a single large 1D FFT). We handle this situation by concatenating the arrays and wrapping them into the rows of the texture. We can compute FFTs on this "wrapped" representation in two ways. The first approach introduces a level of indirection in the indexing, computing a logical index for each fragment from its physical address in the texture. Suppose that (x, y) represents the physical address of the current fragment in the destination texture of width W. We compute the logical address (x_l, y_l) in an $N_x \times N_y$ array as:

$$\begin{aligned} u &= y \cdot W + x \\ x_l &= u \bmod N_x \\ y_l &= \lfloor u / N_x \rfloor. \end{aligned}$$

The logical index can be used in the same fragment programs as described above. When reading from the input texture, the logical input index must be converted back to a physical address:

$$\begin{aligned} u &= y_l \cdot N_x + x_l \\ x &= u \bmod W \\ y &= \lfloor u / W \rfloor. \end{aligned}$$

The advantage of this method is its flexibility. However, it requires extra computation and can destroy 2D locality in its access patterns for some combinations of N_x and W. A second approach can be used for wrapped 1D FFTs whose lengths are a multiple of the texture width. The algorithm, sometimes referred to as the "four-step" framework [12, 13], computes a single, large 1D FFT by computing FFTs along the rows and columns of its wrapped representation.

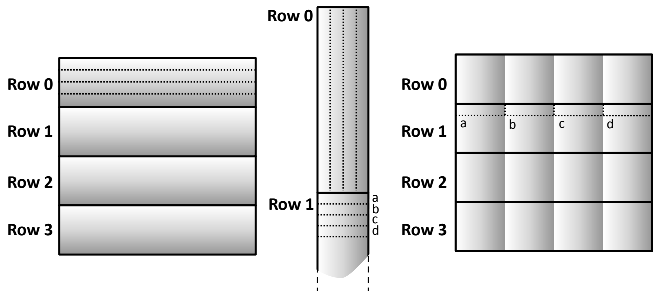

Fig. 4. **Local transposes in the four-step method for large FFTs.** (Left) Four long rows are wrapped at the texture width into multiple rows to form a sub-block of the texture. (Middle) The four-step method transposes each of these sub-blocks independently, and thus exceeds the texture dimensions. (Right) We rewrap the results back into the texture.

Specifically, the algorithm first computes FFTs along the columns. Then each element is multiplied by a twiddle factor:

$$T(x,y) = e^{2\pi ixy/N},$$

where $(x,y) \in [0, W-1] \times [0, H-1]$ are the physical coordinates of the element and $H = N/W$. Finaly, the algorithm computes the FFT along the rows and transpose the result. The advantage of this method is that it is generally faster because it better preserves 2D locality and does not require expensive logical indexing. Currently, we use this method only for large power-of-two FFTs. For large non-power-of-two DFTs, we use logical indexing for all the steps of the z-chirp algorithm except for the two FFTs used to compute the convolution, for which we utilize this four-step method.

With a batch of large 1D FFTs, a single row is wrapped into a sub-block of the texture. The transpose step of the four-step method is performed for each sub-block independently. We refer to these transposes as local transposes (as opposed to a global transpose of the entire 2D array). As shown in Figure 4, physically transposing the sub-blocks can cause the texture to exceed dimension limitations. Instead, we compute the logical equivalent of a local transpose followed by rewrapping to the texture width. Figure 5 explains how this is done. To reduce the number of render passes, we fold in the transposes and twiddle factor multiplications with FFT computation.

We currently do not directly support large 2D FFTs that can not fit within a single texture. However, our library can be used to handle large 2D FFTs by performing the FFTs on several rows at a time — as many as can fit into one texture. After all the rows have been transformed, the same thing is done for the columns. We support large FFTs along columns by first performing a global transpose on the wrapped representation, performing a batch of FFTs along the rows, and performing a global transpose at the end.

4.5. Real-valued FFTs

In many applications, the input of the Fourier Transform has no imaginary component. The DFT of real-valued sequence has special symmetry that can be used to compute the FFT more efficiently:

$$F(k) = F(N-k)^* \tag{2}$$

F is periodic so $F(N) = F(0)$. We pack two real-valued sequences x and y into a complex sequence z, thus reducing the amount of data

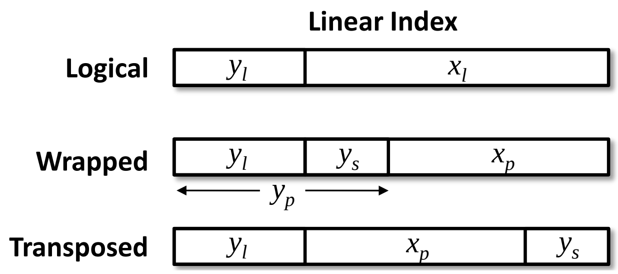

Fig. 5. **Computing indices for the local transposes in the four-step FFT.** (Top) The linear index of an element in a batch of large 1D FFTs is composed by concatenating the bits of the logical address (x_l, y_l), where the length of the FFTs is a power of two. (Middle) The address in the physical index in the wrapped representation (x_p, y_p) in a texture with power-of-two dimensions is computed by simply reinterpreting the appropriate bits of the linear index. y_s is the y coordinate within a sub-block corresponding to one logical row. (Bottom) A local transpose exchanges y_s and x_p. The final physical coordinates are computed by reinterpreting this last linear index. The DX9 API does not support scatters, so we actually gather to the physical destination by inverting this process to compute the corresponding logical index.

to transform by a factor of two:

$$z(n) = x(n) + iy(n).$$

The process can be inverted:

$$\begin{aligned} x(n) &= (z(n) + z^*(n))/2 \\ y(n) &= -i\,(z(n) - z^*(n))/2. \end{aligned}$$

Because the DFT is a linear operator we can recover $\mathcal{F}\{x\}$ and $\mathcal{F}\{y\}$ from $\mathcal{F}\{z\}$:

$$\begin{aligned} X(k) &= (Z(k) + Z^*(k))/2 \\ Y(k) &= -i\,(Z(k) - Z^*(k))/2. \end{aligned}$$

By symmetry, $Z^*(k)$ can be computed as:

$$Z^*(k) = (Z(N-k))^*.$$

For a 1D FFT with N even, we pack together even and odd elements, $x = f_e$ and $y = f_o$. After performing the FFT on the resulting data, we unpack $X = F_e$ and $Y = F_o$ and perform the final iteration of the FFT in the same pass. If N is odd, we cannot use this packing scheme. For batches of odd length FFTs, however, we can pack adjacent rows together. If there are an odd number of FFTs in the batch, an extra row of zeros can be added.

For 2D FFTs, the symmetry relationship is similar to that in 1D:

$$F(k_0, k_1) = F(N_0 - k_0, N_1 - k_1)^*. \tag{3}$$

If the length is even along one of the dimensions d, we pack together even and odd elements, f_e and f_o along d, perform the 2D FFT, unpack F_e and F_o, and compute the last FFT iteration along d. When the lengths are odd in both dimensions, we can treat each dimension as a batch of real 1D FFTs. This requires an unpack after transforming the rows, followed by another pack before transforming the columns. It is possible, however, to eliminate one of these extra intermediate passes as illustrated in Figure 6.

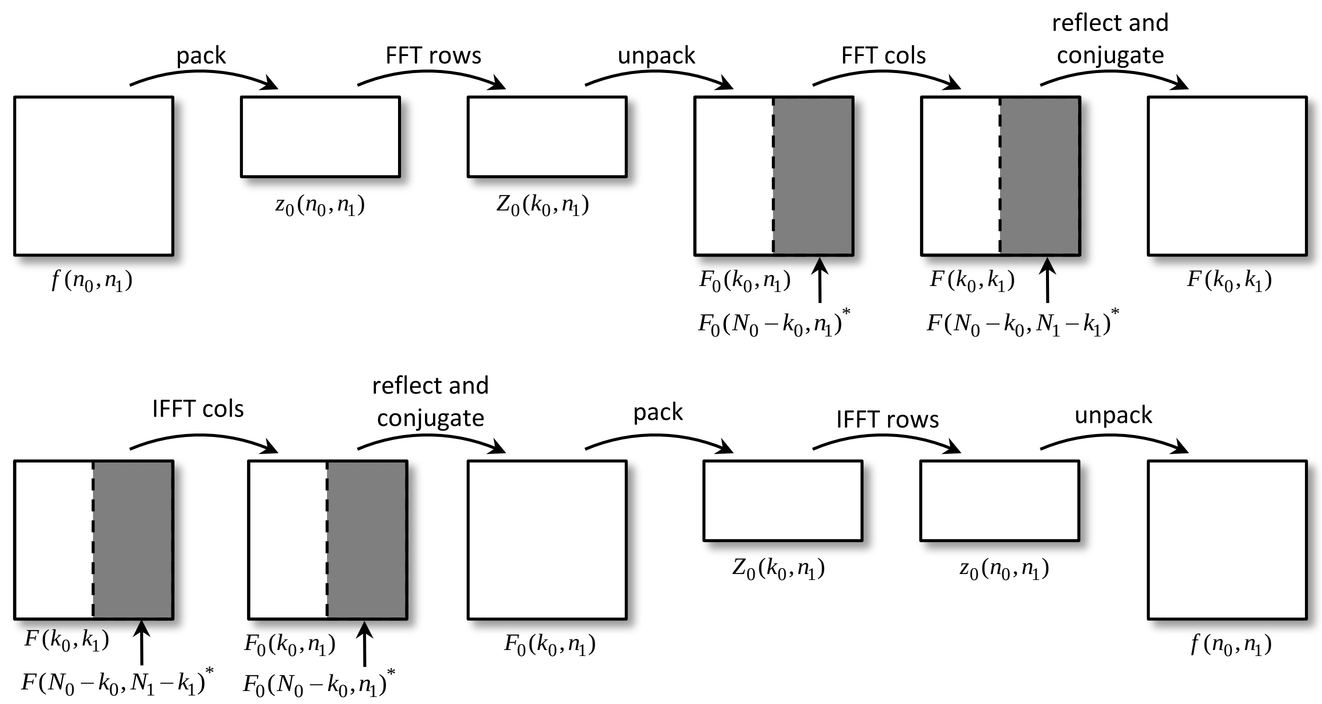

Fig. 6. **Packing real data for 2D FFTs with odd lengths in both dimensions.** (Top) Forward transform. Adjacent pairs of rows (except the last) are packed together to form z. After performing the FFT along rows of z, only half of the data is unpacked. The FFT is performed on the columns of this half. The other half is then obtained by reflecting and conjugating the result. (Bottom) Inverse transform. The inverse FFT is first performed on half of the columns. The rest of the steps are similar to the forward transform.

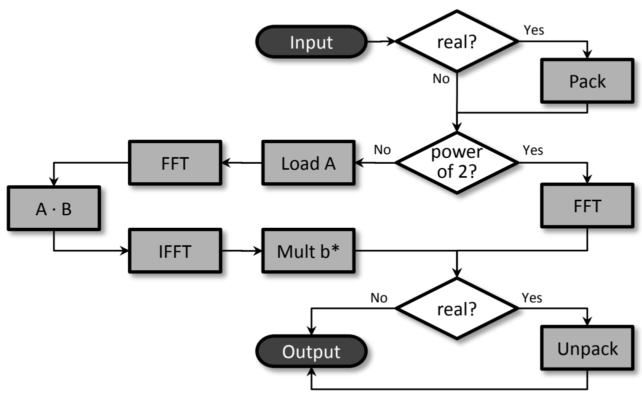

Fig. 7. **Fragment programs used for a 1D FFT computation.**

Another way to implement FFTs of real-valued data is to do the computation on only the first half of each sub-transform in the algorithm. This approach can save the packing step that we currently use. However, it cannot be used with the z-chirp algorithm that we use for evaluating non-power-of-two length FFTs.

4.6. Numerical precision issues

DirectX 9 does not support integer data types. Indices must be computed with floating point, which can lead to precision errors. Precision errors arise from two sources. First, the coordinate interpolation may not be exact. We noticed that on some GPUs the interpolated integer coordinates actually have a fractional component. This is not the case in the reference rasterizer or on other GPUs. For GPUs that support Shader Model 3.0, we use the VPOS semantic, which returns the integer pixel location of a fragment, thereby eliminating interpolation errors. We also tried interpolating linear addresses to save some instructions in the fragment program. However, some GPUs do not interpolate with enough precision for this to work. Second, imprecision can also result from mod operations. This is due to the way that mod is implemented. It is based on the `frac()` instruction, which returns the fractional part of a floating point number:

```
  float mod(x,y) { return frac(x/y)*y; }
```

This code assumes that `x` and `y` are non-negative. Even when they are exact integers, this code can produce non-integer results for values of `y` that are not a power of two. The result of the division, implemented as `x` multiplied by the reciprocal of `y`, cannot always be represented exactly in floating point, leading to a loss of precision.

It is possible to remove imprecision by computing $x \leftarrow \lfloor x + 0.5 \rfloor$ after every operation that can potentially introduce imprecision. However, floor operations are relatively expensive. Therefore, we use floors sparingly. We assume that the input coordinates are imprecise and make judicious use of small offsets so as to prevent a number falling onto the wrong side of an integer boundary. We also use an offset and nearest neighbor sampling when fetching a value from a texture.

Floating point precision also limits the size of the Fourier transforms that we can support. The logical indexing that we use when the data exceeds the texture size limits requires the computation of

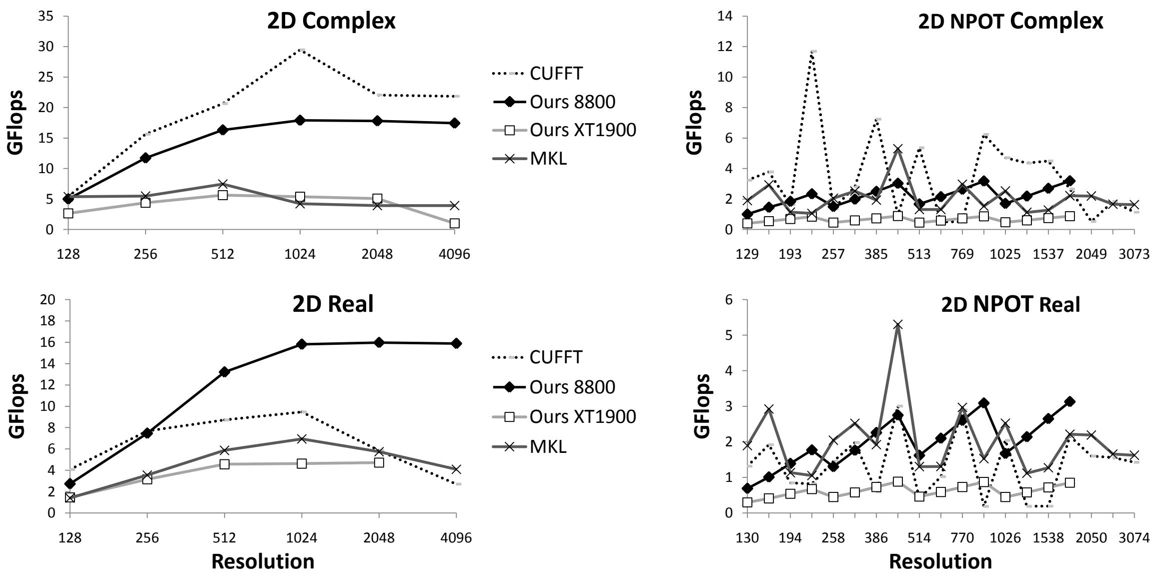

Fig. 8. **Performance comparisons of 2D FFTs.** We compare 2D FFTs for power-of-two and non-power-of-two (NPOT) sizes. CUFFT and MKL are optimized, vendor-provided libraries for the GPU and CPU, respectively. The GFlops numbers are computed as $5N \log_2(N)/t$ where t is execution time. We use $N/2$ instead of N for computing GFlops for the real transforms.

linear addresses. The maximum size of the linear address is limited to the precision of a floating point number, which unfortunately, is less than the maximum number of elements in a texture for some GPUs.

4.7. API

The API for our library is fairly simple. It consists of two main functions. The first function is a configuration function that tells the library what format to expect for the data:

```
Configure( int SpatialFormat,
           int FrequencyFormat,
           int DataSize[2],
           int SpatialRectWidth,
           int FrequencyRectWidth
           int DimMask )
```

We refer here to the domain the FFT as the spatial domain instead of the time domain because it can be two dimensional. The `SpatialFormat` can have the values `REAL` or `COMPLEX`, while `FrequencyFormat` can be `COMPLEX` or `HALF_COMPLEX`. The half-complex format can be used for transformed real data. Only the first $N/2 + 1$ values of an FFT are stored and computed. The other values can be obtained by symmetry. The format is particularly useful for minimizing the size of data transferred on the bus between the GPU and main memory. `DataSize` represents the logical dimensions of the data. For 1D FFTs, one of the dimensions indicates the length of the FFTs and the is the number in the batch. The `*RectWidth` variables are the widths of the data as laid out in the textures. If `DataSize[0]` (or `DataSize[0]`$/2 + 1$ in the case of the half-complex format) is not the same as `*RectWidth`, then the slower logical indexing must be used. `DimMask` is a bit mask with a bit for each dimension to which the FFT should be applied. Thus for a 2D transform, the bits for both rows and columns. Once the library is configured, the user passes input and output textures to the `Compute()` function:

```
Compute( bool Inverse,
         IDirect3DTexture9* pInputTexture,
         IDirect3DTexture9* pOutputTexture )
```

The direction of the transform is determined by the `Inverse` flag. We provide helper functions for loading textures with values from arrays on the CPU. The intended usage model for this API is that after configuring, a user will compute multiple forward and inverse FFTs of the same size and type.

Our library allocates two additional temporary buffers for computing intermediate results. We provide a more advanced API that allows the user to provide the temporary buffers. This is important when computing large FFTs because the input and output themselves can be used as the temporary buffers in order to conserve memory. The advanced API also provides other conveniences such as specifying which channel to read from or write to for real data and specifying an origin for the texture sub-rectangle that contains the data.

5. RESULTS

We tested our FFT algorithms using both an NVIDIA GeForce 8800 GTX and an ATI XT1900 on a PC with an Intel Core 2 Duo E6600 CPU clocked at 2.66 GHz. We compared the performance of both complex and real 2D FFTs of power-of-two (POT) and non-power-of-two (NPOT) sizes to NVIDIA's CUFFT library running on the 8800 and Intel's Math Kernel Library (MKL) on the CPU. The results are shown in Fig. 8. For complex POT FFTs, CUFFT performs better than the others due to its use of shared memory. Our algorithms perform relatively well on the same GPU even though we do not use shared memory. In fact, for real data our implementation significantly out-performs CUFFT. The performance of our algorithms on the 8800 is better than on the older, less expensive XT1900. We used the latest drivers for both GPUs. The relative performance may vary due to differences and tuning in the drivers. For POT FFTs, the performance of the XT1900 is still competitive with MKL. In general the GPU algorithms do not perform as well on small sizes due to the overhead of the graphics API. For NPOT FFTs, our FFTs show a fairly consistent pattern in performance that correlates with steps in the size of the underlying POT FFT and inverse FFT used to perform the convolution. The graphs for CUFFT and MKL are a bit more erratic, performing both better and worse than ours for some values.

The performance of our FFTs is mostly bandwidth limited. When we eliminate the computations from our fragment programs and only read the input values, the performance improves by less than 10%. We also experimented with vectorizing our code by using 4 channel textures and packing multiple values into one texture element. We see only a small improvement ($< 10\%$) on the XT1900 and no improvement on the 8800.

A simple image processing application that utilizes our FFTs is shown in Fig. 1. The user manipulates sliders to change the cutoff frequency and falloff of a band pass filter. Because the FFT computations take place on the GPU, the relatively expensive memory transfers between GPU and CPU are avoided. The high performance of our library sustains interactive frame rates even for large images. The immediate feedback makes it easier for the user to tweak a filter to achieve a desired result.

6. CONCLUSION

The GPU can be an effective coprocessor for signal processing operations, including FFTs. Our FFT library makes it easier to use a wide variety of GPUs for FFTs of various sizes and types, while still delivering good performance. For future work we would like to add support for FFTs of higher dimensions.

7. ACKNOWLEDGMENTS

We would like to thank Peter-Pike Sloan and Yuri Dotsenko for fruitful discussion on this project, as well as John Bronskill and Steve White for their useful feedback. We would also like to thank John Manferdelli and Craig Mundie for their support of this project.

8. REFERENCES

[1] NVIDIA Corp., *CUDA CUFFT Library*, 2007.

[2] NVIDIA Corp., *NVIDIA CUDA Compute Unified Device Architecture*, 2007.

[3] Valve Corp., "Hardware survey results," `http://steampowered.com/status/survey.html`, 2007.

[4] Kenneth Moreland and Edward Angel, "The FFT on a GPU," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, 2003, pp. 112–119.

[5] John Spitzer, "Implementing a GPU-efficient FFT," *SIGGRAPH Course on Interactive Geometric and Scientific Computations with Graphics Hardware*, 2003.

[6] Jason L. Mitchell, Marwan Y. Ansari, and Evan Hart, "Advanced image processing with DirectX 9 pixel shaders," in *ShaderX^2: Shader Programming Tips and Tricks with DirectX 9.0*, Wolfgang Engel, Ed. Wordware Publishing, Inc., 2003.

[7] Thomas Jansen, Barosz von Rymon-Lipinski, Nils Hanssen, and Erwin Keeve, "Fourier volume rendering on the GPU using a split-stream-FFT," in *Proceedings of the Vision, Modeling, and Visualization Conference 2004*, 2004, pp. 395–403.

[8] Thilaka Sumanaweera and Donald Liu, "Medical image reconstruction with the FFT," in *GPU Gems 2*, Matt Pharr, Ed., pp. 765–784. Addison-Wesley, 2005.

[9] Naga K. Govindaraju, Scott Larsen, Jim Gray, and Dinesh Manocha, "A memory model for scientific algorithms on graphics processors," *Supercomputing 2006*, pp. 6–6, 2006.

[10] James W. Cooley and John W. Tukey, "An algorithm for the machine calculation of complex fourier series," *Math. Comput.*, vol. 19, pp. 297–301, 1965.

[11] Charles Van Loan, *Computational Frameworks for the Fast Fourier Transform*, Society for Industrial Mathematics, 1992.

[12] W. M. Gentleman and G. Sande, "Fast Fourier transforms for fun and profit," *Proceedings of AFIPS*, vol. 29, pp. 563–578, 1966.

[13] D. H. Bailey, "FFTs in external or hierarchical memory," *Supercomputing*, pp. 23–35, 1990.