ACCELERATING SIGNAL PROCESSING ALGORITHMS USING GRAPHICS PROCESSORS

Ashwin Prasad and Pramod Subramanyan
RF and Communications R&D
National Instruments, Bangalore – 560095, India
Email: {asprasad, psubramanyan}@ni.com

Abstract

There is increased interest in the use of graphics processing units (GPUs) for general purpose computation. This is because GPUs are almost two orders of magnitude faster in terms of floating point throughput compared to conventional CPUs. In this paper we investigate the use of graphics processing units for accelerating signal processing algorithms, specifically FIR filters and the FFT. We describe optimized implementation of these that take advantage of the instruction-level-parallelism and the data parallel stream processing architecture of a GPU. We also present benchmarks illustrating that these signal processing algorithms are significantly faster when run on the GPU as compared to implementations that run on conventional CPUs.

1. Introduction

Modern graphics processors are designed to process large streams of graphics primitives like points and polygons. Since there is no dependence between the processing of one polygon/point and the next, GPUs are designed to have massive parallelism and deep pipelines. Additionally, the architecture tends to emphasize execution over control flow and branching constructs.

The net result of all these factors is that modern GPUs have floating point performance that is two orders of magnitude higher than that of conventional CPUs. For instance the NVIDIA Tesla S870 GPU Server has a peak floating point throughput of 2 *teraflops* as compared to the Quad-Core Intel Xeon X5355 processor which has a floating point throughput of 63.5 *gigaflops*. [11] It should not be surprising that there is considerable interest in leveraging this computational power for general-purpose computation. Unfortunately, difficult to use graphics APIs which are required to control and execute code on the GPU and a complicated programming model for code running on the GPU itself together mean that the full computational capacity of GPUs is yet to be exploited.

2. GPU Architecture and Programming

Figure 1 shows a simplified block diagram of the graphics pipeline[1]. Until about six years ago the graphics pipeline was usually implemented by a fixed function ASIC, but advances in VLSI technology have made it possible for many parts of the pipeline to be programmable. Current GPUs have two logical programmable units: the vertex and fragment[2] processors. The fragment processor on the GPU typically has an order of magnitude more processing power than the vertex processor. (Note that some modern graphics processors have only one type of execution unit dynamically assigned to either vertex or fragment program execution, so the distinction between vertex and fragment processors might not actually be present in hardware)

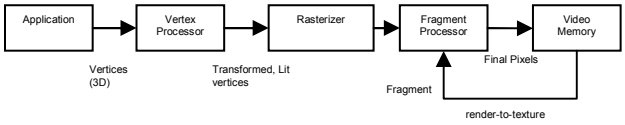

Figure 1 : The graphics pipeline

Traditional GPGPU[3], has mostly focused on leveraging the computational capacity of the fragment processor [3]. The input to the fragment processor is given in textures[4], and instead of writing the rendered output to the screen, the render-to-texture feature of modern graphics systems is used to write the results to another texture. The fragment processor is parallel SIMD (Single Instrument Multiple Data stream) processor, and applies the same operation on all the rendered elements in the input texture. The operation applied is analogous to the inner loop in CPU computation and is referred to as the *computational kernel*.

Multistage algorithms (our implementation of the FFT, for example) are implemented using a technique called ping-pong buffering – two textures t_1 and t_2 are allocated; for the first stage, t_1 is bound as the input texture, and t_2 as the output texture and the fragment program is run. For the next stage of computations, t_2 is made the input texture and t_1 is made the output texture, and the fragment program is executed again. This process of swapping the input and output textures is repeated $N-1$ times for an N stage algorithm. (Continuing on the FFT example, a 1024 point

[1] The graphics pipeline is the sequence of operations involved in converting a stream of three-dimensional vertices, to a two-dimensional image to be rendered on the screen. See [1] for more details on 3D graphics rendering.

[2] A fragment can be roughly defined as a potential pixel.

[3] GPGPU : General Purpose Computation on GPUs

[4] A texture is a bitmap that is projected onto a surface.

FFT which requires 10-stages of processing will swap the input and output textures 9 times).

3. FIR Filter Implementation And Benchmarks

A Finite Impulse Response (FIR) Filter operating on an input signal *x[n]*, to produce the output *y[n]* is defined by the equation:

$$y[n] = \sum_{k=0}^{N-1} x[n-k]h[k] \qquad (1)$$

3.1 FIR Filter Implementation Details

The SIMD processors on modern GPUs have the ability to natively process the *float4* data type[5]. As GPUs can perform addition, multiplication, and other floating point operations on each of these 4 components in parallel, we attempt to restructure the FIR filter equation to take advantage of these operations.

Let us assume that the number of coefficients of the FIR filter is a multiple of 4. (This causes no loss of generality because the filter coefficients can always be padded with zeros to reach a multiple of four). Using this, (1) may be rewritten as:

$$y[n] = \sum_{k=0}^{N/4-1} \begin{pmatrix} x[n-4k]h[4k] + \\ x[n-4k-1]h[4k+1] + \\ x[n-4k-2]h[4k+2] + \\ x[n-4k-3]h[4k+3] \end{pmatrix} \qquad (2)$$

Notice that the term between the parentheses can be viewed as the component wise multiplication of two *float4* tuples – which means that this interpretation of the FIR filter can be used for an efficient implementation on the GPU. With this objective in mind, we stored the FIR filter input in a single RGBA texture. This leads to an organization similar to the one shown in Figure 2. However, just reordering the computation performed by the filter alone is not sufficient, because the GPU cannot load 4-tuples that span across texture-elements. In the example shown in Figure 2, while the GPU can load *x[0], x[1], x[2]* and *x[3]* into a *float4* variable in a single load operation as it is stored at texture-coordinate (0, 0); it is not possible to load *x[1], x[2], x[3]* and *x[4]*, in a single operation. The elements *x[1], x[2] and x[3]* are stored at coordinates (0, 0) while the element *x[4]* is stored at coordinate (0, 1). This means that to calculate the filter output at *x[4]*, the filter will have to make two load operations, and then combine the two values, before the component-wise multiplication of the *float4* tuples can be performed. In general, for the current data-structure, calculating filter outputs at all locations that are not of the form *x[4n-1]* (*n* being a positive integer) without additional load/combination operations is not possible.

[5] A *float4* is a 4-tuple of single precision floating point numbers. In graphics applications, these are typically used to store either *(x, y, z, w)* coordinates or *(r, g, b, a)* color values.

texture row	*texture column 0*				*texture column 1*			
0	*x[0]*	*x[1]*	*x[2]*	*x[3]*	*x[4]*	*x[5]*	*x[6]*	*x[7]*
1	*x[8]*	*x[9]*	*x[10]*	*x[11]*	*x[12]*	*x[13]*	*x[14]*	*x[15]*
2	*x[16]*	*x[17]*	*x[18]*	*x[19]*	*x[20]*	*x[21]*	*x[22]*	*x[23]*
3	*x[24]*	*x[25]*	*x[26]*	*x[27]*	*x[28]*	*x[29]*	*x[30]*	*x[31]*

Figure 2: Example showing the storage of FIR filter input in an RGBA texture

We avoided the additional load operators by storing four shifted copies of the filter coefficients. An example is shown in Figure 3 for a filter with 4 taps. Notice that by selecting the filter texture row as `(arrayIndex mod 4)`; we can calculate the filter output for all input samples.

3.2 FIR Filter on GPU : Pseudocode

texture row	*texture column 0*				*texture column 1*			
0	*h[3]*	*h[2]*	*h[1]*	*h[0]*	*0*	*0*	*0*	*0*
1	*h[2]*	*h[1]*	*h[0]*	*0*	*0*	*0*	*0*	*h[3]*
2	*h[1]*	*h[0]*	*0*	*0*	*0*	*0*	*h[3]*	*h[2]*
3	*h[0]*	*0*	*0*	*0*	*0*	*h[3]*	*h[2]*	*h[1]*

Figure 3: Example showing the storage of FIR Filter coefficients in an RGBA Texture

Pseudocode equivalent to our fragment program kernel for the GPU is shown below:

```
for each input texture element x[n] do:
  float4 sum1 = {0, 0, 0, 0}, sum2 = {0, 0, 0, 0};
  float4 sum3 = {0, 0, 0, 0}, sum3 = {0, 0, 0, 0};
  float4 filteredOutput;

  for i ← 0 to (filter length / 4) do:
    sum1 ← sum1 + x[n-i]*h[0, i]
    sum2 ← sum2 + x[n-i]*h[1, i]
    sum3 ← sum3 + x[n-i]*h[2, i]
    sum4 ← sum4 + x[n-i]*h[3, i]

  filteredOutput.x = sum1.x+sum1.y+sum1.z+sum1.w;
  filteredOutput.y = sum2.x+sum2.y+sum2.z+sum2.w;
  filteredOutput.z = sum3.x+sum3.y+sum3.z+sum3.w;
  filteredOutput.w = sum4.x+sum4.y+sum4.z+sum4.w;

  store filteredOutput in output texture
```

Algorithm 3.1: GPU Filtering Kernel

Our implementation used direct memory access (DMA) to transfer the filter input samples from CPU memory to a texture on the GPU VRAM; Computation was done by drawing a screen-aligned polygon. We implemented the fragment kernel in Cg [5], and used the OpenGL API to interface with the GPU.

3.3 FIR Filter Benchmarks

We compared our FIR filter routine running on the GPU to a CPU and benchmarked the performance increase. The GPU we used was an NVIDIA GeForce 8600 GT, and our CPU was an Intel Pentium 4 "Prescott" clocked at 3.2 GHz.

Figure 4 shows the variation of the filter execution time with the number of filter taps for FIR Filters implemented on the GPU and on the CPU when both filters are processing 2M samples. The GPU kernel is consistently about 8 times faster.

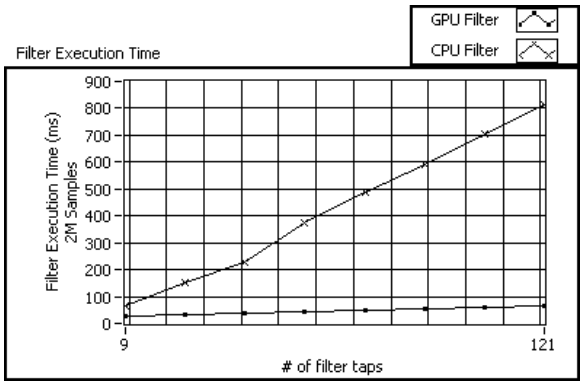

Figure 4

Figure 5 shows the variation of the filter execution time with number of filter taps constant at 57, as the input sample count varies from 100 thousand samples to 4.1 million samples. Note that the GPU filter shows more of a performance increase when processing larger block sizes or when the filter has a higher number of taps.

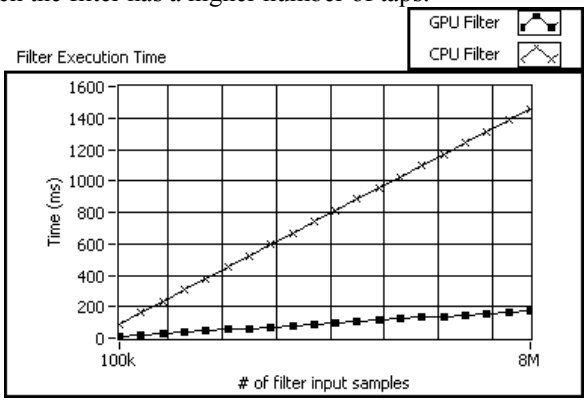

Figure 5

4. FFT Implementation and Benchmarks

We describe an implementation of the radix-2 decimation in time Cooley-Tukey FFT Algorithm for complex inputs below.

4.1 FFT Implementation

Current graphics processors impose a limit on the maximum size along each dimension of a texture (this varies across GPUs, but is usually of the order of 2K). As a consequence, for FFT sizes larger than a few thousand, we cannot store the input samples in a single row of a 2D texture. Secondly, since we are implementing the decimation-in-time version of the Cooley-Tukey algorithm, the input to the FFT will have to be stored in a bit-reversed fashion before the FFT computation can be performed on it. To take advantage of the *float4* data-type on the GPU, we pack two complex numbers into a single *float4* tuple, so each element in the input texture consists of two complex numbers.

4.1.1 Naïve Implementation of FFT on GPU

There are two primary challenges in porting the Cooley-Tukey FFT algorithm to the GPU: (1) it is not possible to write to arbitrary memory locations[6] when executing fragment kernel code, (2) current GPUs do not have efficient support for conditional statements.

The location of each fragment on the grid is fixed at the time of fragment creation, and the fragment program kernel cannot alter that location. In other words, fragment programs running on the GPU cannot write to arbitrary locations. In the context of the FFT implementation this poses a problem because, we cannot perform a straightforward translation of the Cooley-Tukey algorithm - iterating over each butterfly, and writing out the two output values (shown in Algorithm 4.1).

```
for i ← 1 to log2N :
  for each butterfly in this stage:
    x1' ← x1 + x2*Wn^k
    x2' ← x1 -  x2*Wn^k
```

Algorithm 4.1: FFT on CPU

Instead the GPU implementation will have to work-backwards based on the coordinate of the current fragment which is being computed. This implementation is shown in Algorithm 4.2.

```
'iterate over FFT stages
for i ← 1 to log2N:
  for each output coordinate x do:
   'points lies on the "upper"
   'end of a butterfly
   if (x mod 2^i) < 2^(i-1):
    output ← input[x] + Wn^k *input[x+2^(i-1)]
   else:
   'points lies on the "lower"
   'end of a butterfly
    output ← input[x] - Wn^k *input[x-2^(i-1)]
   store output value

  'ping-pong texturing
  swap input and output textures
```

Algorithm 4.2: Naïve GPU implementation of FFT

Current graphics processors do not have complete support for conditional statements. Instead, conditional statements are implemented using predicated execution – meaning that the GPU will execute the code for when the if-statement returns true, as well as that for the case when the statement returns false, and finally, only one of the two results will be stored. As a consequence, the naïve implementation of the FFT will actually perform almost 2X the required number of operations, suffering a significant performance hit.

4.1.2 Efficient FFT Implementation on GPU

The conditional statements in the GPU code can be eliminated by using static branch resolution on the CPU [8]. In this technique, the algorithm is split into two specialized fragment programs – one containing the code that will be executed when the condition evaluates to true, and another containing the code to be executed when the condition

[6] The operation of writing to arbitrary memory locations is referred to as "scatter". Restricting memory writes to pre-computed locations significantly simplifies processor architecture. (For instance arbitration circuitry that would otherwise be required to handle multiple-writer conflicts can be avoided)

evaluates to false. The controlling code running on the CPU identifies which of the two programs is to be run on each input texture-element.

For this technique to be efficient, the CPU should not have to evaluate the branching condition for each texel. Instead, sets of texels for which the condition evaluates to the same value need to be identified. Below we outline a method to identify these sets of texture elements when the texture width W is constrained to be a power of 2.

$$\text{Define } W = 2^k \qquad [\text{where } k \text{ is an integer}] \qquad (3)$$

The first stage of the FFT has two special properties that are exploited in our implementation. Firstly, it operates on adjacent elements in the input texture, and because of our packed complex number representation, the elements of each butterfly diagram are going to be stored in the same texture element. Secondly, the *twiddle factors* in the FFT computation for the first stage are all equal to one, and need not be multiplied. This leads to a particularly simple GPU fragment kernel for the first stage of the FFT.

```
for each texture element pt in input texture do:
  val ← float4 ( pt.xy + pt.zw, pt.xy - pt.zw)
  store val in output texture
```

Algorithm 4.3: First stage of GPU FFT Implementation

The condition we use to decide whether the element at array index j is on the upper or lower end of each butterfly structure is:

$$j \bmod 2^{i+1} < 2^i \qquad \text{where } 0 \le i < log_2 N \qquad (4)$$

Consider an arbitrary element in the input texture located at coordinates *(x, y)*. For a row-major mapping from the input array to the texture, the condition (4) can be re-written as:

$$(y \cdot W + x) \bmod 2^{i+1} < 2^i \qquad (5)$$

For stages i such that $0 < i \le k-1$; $(y \cdot W + x) \bmod 2^{i+1}$ is $x \bmod 2^{i+1}$, because W is a multiple of 2^i. This leads to the condition in equation (5) reducing to:

$$x \bmod 2^{i+1} < 2^i \qquad (6)$$

In other words, this means that all the elements which satisfy condition (5) lie in a series of column-aligned rectangles whose diagonals are defined by points $(p \cdot 2^{i+1}, 0)$, $(p \cdot 2^{i+1} + 2^i - 1, H)$; where $0 \le p < 2^{k-i-1}$.

Let us now consider the case when $2^{i+1} > W$, specifically, let:

$$W \cdot 2^q = 2^{i+1}. \qquad (7)$$

For a given row (i.e. constant y), x is at most $W-1$. Therefore:

$$max(y \cdot W + x) = W \cdot (y+1) - 1 \qquad (8)$$

Using (8) in equation (5):

$$(W \cdot (y+1) - 1) \bmod 2^{i+1} < 2^i \qquad (9)$$

It is easy to see that this is true when:

$$y + 1 < 2^{q-1} \qquad (10)$$

Therefore the solutions to (5) are of the form:

$$y = m \cdot 2^q + l \qquad (11)$$

where $0 \le m < N/2^{i+1}$, $0 \le l < 2^{p-1}$.

Equation (11) can be interpreted to mean that all the solutions to (5) are a series of row-aligned rectangles, for the stages $k, k+1 \ldots log_2 N - 1$ of the FFT.

Algorithm 4.4 shows the CPU code, that implements the static branch resolution for the GPU FFT.

```
load first stage program on GPU
perform computation on entire texture
swap input and output textures

'iterate over stages from 1 to k-2;
'draw column-aligned rectangles
for i ← 1 to k-1 do:
 load upper-point column butterfly program on GPU
 for j ← 0 to 2^(k-i-1) do:
  perform computation on
        rectangle (j·2^(i+1), 0, j·2^(i+1)+2^i-1, H)
 load lower-point column butterfly program on GPU
 for j ← 0 to 2^(k-i-1) do:
  perform computation on
        rectangle (j·2^(i+1)+2^(i-1), 0, j·2^(i+1)+2·2^(i-1)-1, H)
 'ping pong buffering
 swap input and output texture

'iterate over remaining stages of FFT
'draw row aligned rectangles now
for i ← k to log2N - 1 do:
 load upper-point row butterfly program on GPU
 for j ← 0 to 2^(i-k) do:
  perform computation on
        rectangle (0, j·2^(i-k), W, j·2^(i-k)+2^(i-k-1))
 load lower-point row butterfly program on GPU
 for j ← 0 to 2^(i-k) do:
  perform computation on
        rectangle (0, j·2^(i-k)+2^(i-k-1), W, j·2^(i-k)+2·2^(i-k-1))
 'ping pong buffering
 swap input and output texture
```

Algorithm 4.4: CPU Code

The code avoids the overhead of running conditional statements on the GPU, by creating specialized programs that are controlled by code running on the CPU. Since each of individual fragment programs do not contain any conditional statements, the execution on the GPU is efficient.

Algorithms 4.5 and 4.6 show the programs that are run for the stages *1* to *k-1*.

```
For each texture element pt in input texture do:
  'coords is the coordinate of the current point
  float2 lowerPointCoords ←
        float2(coords.x+2^(i-1), coords.y)
  float4 lowerNum ← inputSamples[lowerPointCoords]
  float4 twiddle ← float4(W_N^k, W_N^(k+1))
  store inputSamples[coords] + lowerNum * twiddle
```

Algorithm 4.5: Upper Point Column Butterfly Program

```
For each texture element pt in input texture do:
  'coords is the coordinate of the current point
  float2 upperPointCoords ←
        float2(coords.x-2^(i-1), coords.y)
  float4 upperNum ← inputSamples[upperPointCoords]
  float4 twiddle ← float4(W_N^k, W_N^(k+1))
  store upperNum - inputSamples[coords] * twiddle
```

Algorithm 4.6: Lower Point Column Butterfly Program

Algorithms 4.7 and 4.8 show the programs that are run for the stages k to $log_2 N - 1$.

```
For each texture element pt in input texture do:
   'coords is the coordinate of the current point
   float2 lowerPointCoords ←
          float2(coords.x, coords.y+2^i/W)
   float4 lowerNum ← inputSamples[lowerPointCoords]
   float4 twiddle ← float4(W_N^k, W_N^(k+1))
   store inputSamples[coords] + lowerNum * twiddle
```

Algorithm 4.7: Upper Point Row Butterfly Program

```
For each texture element pt in input texture do:
   'coords is the coordinate of the current point
   float2 upperPointCoords ←
          float2(coords.x^i, coords.y-2^i/W)
   float4 upperNum ← inputSamples[upperPointCoords]
   float4 twiddle ← float4(W_N^k, W_N^(k+1))
   store upperNum – inputSamples[coords] * twiddle
```

Algorithm 4.8: Lower Point Row Program

4.2 FFT Benchmarks

Figure 6 shows the execution times for the GPU and CPU implementations of the FFT. The GPU is faster than the CPU by a factor of about 2.5X.

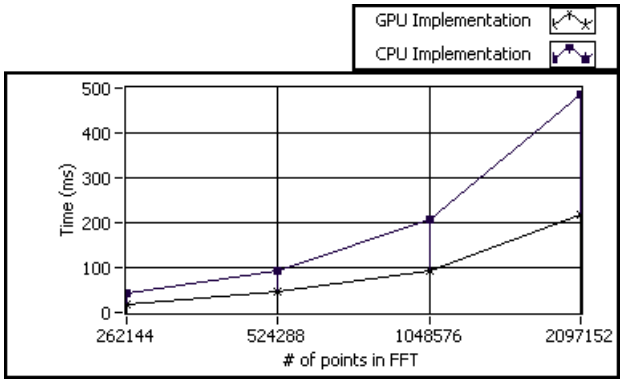

Figure 6: Plot of FFT Size vs. Execution Times

Figure 7 shows the speedup of the GPU FFT over the CPU FFT across FFT sizes.

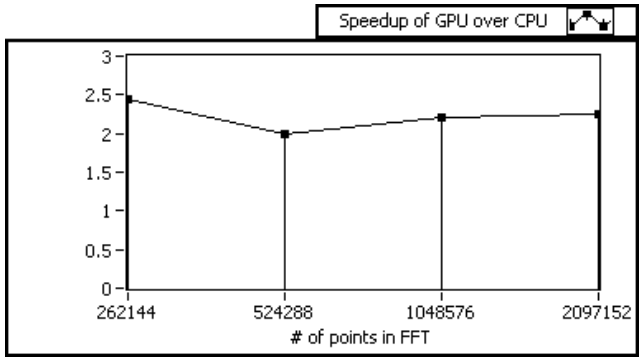

Figure 7: Plot of Speedup of GPU FFT over CPU.

5. Conclusions and Future Work

We presented innovative algorithms for implementation of FIR filters and FFT on graphics processors that take advantage of the data-parallel streaming architecture of these devices; we also provided benchmarks comparing our implementations to implementations running on the CPU, showing that a significant performance gain was achieved by implementing these algorithms on the GPU. We showed speedups of about 8X for the FIR filter, and about 2X for the FFT. The graphics processor we used was the NVIDIA GeForce 8600 GT, which retails for just $150. Our results show that even with inexpensive graphics processors many signal processing algorithms can be accelerated significantly using GPUs.

We are in the process of investigating GPU implementations of routines for performing linear algebraic operations on large datasets. In future work, we also intend to explore efficient implementations of other signal processing algorithms which are expensive for CPU implementations. Specifically resampling is an application that we are planning to target.

Sengupta et al [9] have recently outlined a set of primitives for GPU computation that can be used for the efficient implementation of many algorithms that are not seemingly data-parallel. An analysis of these primitives with the aim of implementing signal processing algorithms that require feedback: IIR filters for instance, could also prove to be a fruitful direction for future research.

References

[1] M. Woo, J. Neider, T. Davis, and D. Shreiner, "The OpenGL Programming Guide", Reading. Massachusetts. Addison Wesley, 3rd ed., (1999).

[2] 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", Computer Graphics Forum, volume 26, number 1, 2007, pp. 80-113.

[3] Mark Harris: "Mapping computational concepts to GPUs", Proceedings of SIGGRAPH 2005.

[4] Emmett Kilgariff and Randima Fernando, "The GeForce 6 GPU Architecture", GPU Gems 2. NVIDIA Corp. pp. 471-491.

[5] W. R. Mark, R. S. Glanville, K. Akeley and M. J. Kilgard, "Cg: A system for programming graphics hardware in a C-like language", In Proceedings of SIGGRAPH 2003.

[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] N. Govindaraju, S. Larsen, J. Gray, and D. Manocha, "A memory model for scientific algorithms on graphics processors", Proc. Supercomputing, p. 6, 2006.

[8] K. Moreland and E. Angel, "The FFT on a GPU", Graphics Hardware, 2003.

[9] Mark Harris and Ian Buck, "GPU Flow Control Idioms", GPU Gems 2. NVIDIA Corp. pp. 547-555.

[10] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John. D. Owens, "Scan primitives for GPU computing", Graphics Hardware 2007.

[11] "Intel Xeon Processor: HPC Benchmarks – Dense Floating point". http://www.intel.com/performance/server/xeon/hpcapp.htm [2007]

[12] "FFT Benchmark Results", http://www.fftw.org/speed, [2007]