An Implementation of a FIR Filter on a GPU

Alexey Smirnov and Tzi-cker Chiueh

***Abstract*— In this paper we describe an implementation of the Finite Impulse Response (FIR) filter on a modern graphics processing unit (GPU). The FIR filter has a number of applications in audio processing. Modern GPUs are known to have higher GFLOPs rates compared to the CPUs. The new PCI-X Express bus enables fast data transfers between the video card and the main memory. We compare the GPU implementation of the FIR filter with the SSE-optimized CPU implementation. We used Geforce 6600 video card and the Pentium 4-HT 3.2 GHz processor-based PC in our experiments. We varied several parameters of the GPU implementation to achieve highest performance. The results indicate that the GPU implementation is faster than the SSE-optimized implementation when the FIR filter has a large number of taps and therefore requires a large number of data. We also evaluated the performance of radio receiving application. The results indicated that using GPU-optimized building blocks improves the performance of the application.**

I. Introduction

Often, numerical algorithms perform a similar computation on each data element they process. Therefore, parallel architectures are likely to improve the efficiency of these algorithms. Over the last decade, the traditional instruction set of Intel 32-bit architecture has been augmented with several extensions such as MMX, SSE, SSE2, and SSE3. Special registers added to the processor can hold as many as 4 double-precision floating-point numbers. The instructions of the SSE extensions allow programmers to manipulate data in these registers and perform SIMD computations on the data contained in them.

Modern video cards can process data in the single-precision floating-point format. Instead of rendering data to the screen, the video card can save the output in a so-called *pbuffer* which is allocated in the video card's memory. The CPU can copy the data from the video memory to the main memory after that. A video card has a number of rendering units called fragment processing units (FPUs). FPUs perform the same sequence of user-specified computations for each pixel using textures as inputs. The number of FPUs on a video card varies from 1 (Geforce FX 5700) to 6 (Geforce 7800). Each fragment unit has a certain amount of cache memory. When a fragment unit requests data that is not in the cache, the data is fetched from the video memory and stored in the unit's cache. The caching algorithm considers the 2D shape of the textures. Whenever a texture element is fetched, elements adjacent to it horizontally and vertically are pre-fetched.

Because of their parallel nature, modern video cards have high computational power. Fatahalian et al. [1] measured the peak FLOPS rates and peak memory bandwidth for several video cards. The results presented in Table I imply that video cards are more powerful than the CPU. However, the data transfers between the main memory and the video memory can degrade the efficiency of GPU computations.

	GFLOPS	CacheBW
P4 CPU	12	44.7
NV 6800	43.9	18.3
ATI X800	63.7	28.4

TABLE I
PEAK COMPUTATION AND BANDWIDTH (GB/SEC) RATES. [1]

In this paper we design and implement a GPU algorithm for the finite impulse response (FIR) filter [2]. A FIR filter has numerous applications in audio processing. We evaluate our implementation using Gnu Radio open-source software [3]. This software includes a library of building blocks from which a programmer can build an audio processing application. We modified the following building blocks of Gnu Radio: FIR filter, frequency translating FIR filter, and Hilbert transformation. We built a simple radio transmitting/receiving application from these blocks and a few others and evaluated the performance of the GPU implementation of these blocks using this application.

The rest of this paper is organized as follows. In Section 2 we review the related work in the area of using GPU to implement numeric algorithms. Section 3 presents an overview of the GPU computation model. The audio processing functions that we have implemented on the GPU are described in Section 4. The implementation is evaluated in Section 5. Finally, Section 6 concludes this paper with a summary of research contributions and directions for future work.

II. Related Work

Implementation of numerical algorithms on a GPU has been an active research area over the last five years. Several researchers considered the problem of matrix multiplication [4], [1], [5], [6]. The results of this research suggest that a P4 3GHz processor can perform matrix multiplication faster than video cards such as Geforce 6800. However, newer video cards Geforce 7800 and ATI X800 XT prove the opposite. The results of Govindaraju et al. [7] and Galoppo et al. [8] confirmed that these video cards can sort an array of elements and solve a dense linear system respectively faster than an optimized CPU implementation. Other researchers [9], [10] described an implementation of the fast Fourier transform (FFT) algorithms on GPU. Krueger et al. [11] described an implementation of linear algebra operators on a GPU. Hillesland et al. [12] developed a non-linear optimization framework for the GPU. The goal of ClawHMMer project [13] was to design and evaluate a GPU implementation of Viterbi algorithm.

GPU computing has been applied to many other areas including accelerated rendering, computational geometry, and database systems.

$a_{1,1}a_{1,2}a_{1,3}a_{1,4}$	$a_{1,5}a_{1,6}a_{1,7}a_{1,8}$	...
$a_{2,1}a_{2,2}a_{2,3}a_{2,4}$	$\ddots$	
...		

TABLE II
DATA REPRESENTED AS A GPU TEXTURE. FOUR MATRIX ELEMENTS ARE STORED IN ONE PIXEL.

III. GPU COMPUTATION OVERVIEW

The processing pipeline of a GPU is reasonably complicated. Fragment shaders are used to implement numerical algorithms on a GPU. A fragment shader is a programmable unit that runs a user-specified program. It takes textures as inputs, runs the shader program for each pixel, and returns the 4-tuple color (R, G, B, A) of the destination pixel. OpenGL [14] is a standard for rendering images. It provides several extensions that allow programmer to use advanced features of a GPU. NVidia's web site [15] provides specifications of OpenGL extensions that NVidia supports. For example, extension `ARB_fragment_program` allows programmers to load fragment programs into the fragment processing units. Extension `ARB_pixel_buffer_object` makes off-screen rendering possible. A buffer in the video memory called *pbuffer* is allocated in this case. Floating-point textures are available if extension `NV_float_buffer` is provided. `ARB_render_texture` extension allows an application to render directly into a texture instead of rendering to the screen or to a pbuffer. The data rendered to a texture is ready for being used immediately in another fragment program. One has to copy the data rendered to the screen or an off-screen buffer to a texture before using it in other fragment programs. A typical GPU program is comprised of the following steps:

1) Create floating-point textures containing the input data and load them to the video memory.
2) Load the fragment program and enable multi-texturing.
3) Define vertex and texture coordinates for a figure (such as a rectangle). The interpolation hardware will compute the textures coordinates for each pixel of the destination image and provide them to the fragment program.
4) Draw the figure (the rectangle) to an off-screen buffer or to a texture.
5) If the results were rendered to an off-screen buffer then copy the image to a texture using `glCopyTexSubImage2D()`.
6) Go to step 3 if additional iterations are required.
7) Otherwise, use `glGetTexImage()` to copy data from the video memory to the main memory.

Fragment programs are written in the fragment program assembly language. However, a number of compilers from C-like languages exist. In this project we used NVidia's Cg compiler to implement fragment programs. Geforce 6600 video card that we used in our experiments supports such features as conditional expressions (if operators in C) and loops (for loops). The fragment processor has a limit on the dynamic instruction count of 65,535 instructions.

When floating point data is loaded into the video card it is converted into textures. A texture pixel has four components: R, G, B, and A and each of these components is a single-precision floating-point number. Typically, an input matrix is represented as a texture directly as shown in Table II. However, other representations are possible. Hall et al. [6] propose dividing a matrix into four equally-sized matrices and packing one element from each of these matrices into a pixel. This representation allows one to reduce the number of texture fetches required to compute matrix product and uses fragment processing unit's cache more efficiently.

IV. FIR FILTER ON THE GPU

A. FIR Filter and Its Applications

FIR filter is widely used in audio processing. Parameters $t_1, \ldots, t_l$ called *taps* define the FIR filter. For each input sample i_n the FIR filter computes an output sample o_n using the following equation:

$$FIR(i_n, t) = o_n = \sum_{j=0}^{l} i_{n+j} \cdot t_j \tag{1}$$

Vectors i and t are either complex or real.

A FIR filter is used in many other audio processing tasks. For example, a *Hilbert transformation* of a real input vector i is a vector of complex numbers o computed with the following equation:

$$\begin{cases} real(o_n) = i_{n+\frac{k}{2}} \\ imag(o_n) = FIR(i_n, t) \end{cases}$$

where t are specially defined taps of a Hilbert transformation.

A *frequency translating FIR filter* takes a sequence of real numbers and applies a FIR filter with complex taps. The complex output is then rotated using the following equation:

$$o_n = FIR(i_n, t) \cdot e^{2\pi n f \cdot i}$$

where f is a parameter.

The functions that we described take vectors rather than matrices as their inputs. One can represent vectors using $1 \times n$ textures. However, this limits the vector size to the maximum texture width or length, typically 2048 pixels. Therefore, we decided to represent a vector as a matrix. Vector elements are stored sequentially within a row, adjacent rows store adjacent vector elements. Four real numbers or two complex numbers are stored in each pixel: Re_1, Im_1, Re_2, Im_2. We did a number of experiments to determine the optimal matrix shape.

B. GPU Implementation of a FIR Filter

In this paper we describe the implementation of these three audio processing functions on a GPU. We modified GNU Radio [3] — an open-source software that provides many audio- and video-processing building blocks and allows one to build audio- and video-processing applications. We implemented a simple radio transmitter/receiver application using GNU Radio and evaluated its performance. A C++ class represents each building block. For example,

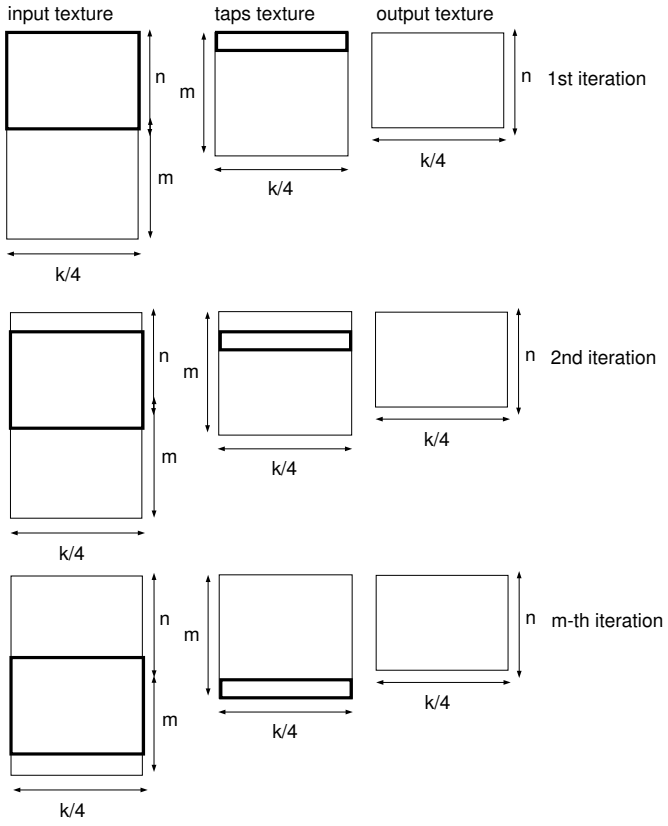

Fig. 1. Texture coordinates computation for the FIR filter algorithm.

FIR filters are represented using classes `GrFIRfilterFFF`, `GrFIRfilterFCF` for different types of input, types, and output. The Hilbert transformation is represented using class `GrHilbert`. The frequency translating FIR filter is implemented in class `GrFreqXlatingFIRfilter`.

Suppose our goal is to compute FIR filter transformation for $n \cdot k$ elements using $m \cdot k$ taps where k is the parameter of our choice. We represent input data and taps using textures of size $n+m$ rows $\times$ $k/4$ columns and $m \times k/4$ respectively. The output texture size is $n \times k/4$. It takes $(m+n) \cdot k$ input elements to compute $n \cdot k$ outputs because computing the $n \cdot k$-th output element requires $m \cdot k$ additional input elements according to Equation 1.

The GPU algorithm proceeds in m iterations. At iteration i the i-th row of the taps texture and rows from i-th to $i+n$ of the input texture are used. Figure 1 illustrates texture coordinate setup of the algorithm. For each element of the output texture the fragment program computes dot product of the adjacent k elements of the input texture and k taps of i-th row.

Figure 2 presents the source code of the fragment program. The program is applied m times. We describe one iteration of the algorithm. Because the matrix elements are stored in four-tuples in textures, different components of the input pixel are multiplied with different components of the taps pixel. The program has a loop that processes all taps. For a given output pixel (x, y) the texture pixel is initialized as follows:

$$\begin{aligned} o_r^0 &= i_r t_r^0 + i_g t_g^0 + i_b t_b^0 + i_a t_a^0 \\ o_g^0 &= i_g t_r^0 + i_b t_g^0 + i_a t_b^0 \\ o_b^0 &= i_b t_r^0 + i_a t_g^0 \\ o_a^0 &= i_a t_r^0 \end{aligned} .$$

Then the fragment program's loop starts. The j-th iteration uses the $j-1$-th and j-th pixels of the taps texture. It updates the output pixel as follows:

$$\begin{aligned} o_r^{j+1} &= o_r^j + i_r t_r^j + i_g t_g^j + i_b t_b^j + i_a t_a^j \\ o_g^{j+1} &= o_g^j + i_r t_a^{j-1} + i_g t_r^j + i_b t_g^j + i_a t_b^j \\ o_b^{j+1} &= o_b^j + i_r t_b^{j-1} + i_g t_a^{j-1} + i_b t_r^j + i_a t_g^j \\ o_a^{j+1} &= o_a^j + i_r t_g^{j-1} + i_g t_b^{j-1} + i_b t_a^{j-1} + i_a t_r^j \end{aligned} .$$

It is convenient to represent this computation as matrix-vector product $O^{j+1} = O^j + M^j I^j$ where

$$M^j = \begin{pmatrix} t_r^j & t_g^j & t_b^j & t_a^j \\ t_a^{j-1} & t_r^j & t_g^j & t_b^j \\ t_b^{j-1} & t_a^{j-1} & t_r^j & t_g^j \\ t_g^{j-1} & t_b^{j-1} & t_a^{j-1} & t_r^j \end{pmatrix}$$

and

$$I^j = \begin{pmatrix} (i_{x+j,y})_r \\ (i_{x+j,y})_g \\ (i_{x+j,y})_b \\ (i_{x+j,y})_a \end{pmatrix}.$$

Matrix M is updated at each iteration of the fragment program loop. The final value of the output pixel is computed as

$$O = O^k + \begin{pmatrix} 0 & 0 & 0 & 0 \\ t_a^{k-1} & 0 & 0 & 0 \\ t_b^{k-1} & t_a^{k-1} & 0 & 0 \\ t_g^{k-1} & t_b^{k-1} & t_a^{k-1} & 0 \end{pmatrix} \cdot I^k$$

The matrix representation of the computation allows to reduce the number of GPU instructions per loop iteration. The naive computation will require 16 multiplications and 16 additions per iteration. The matrix representation requires 8 assignment instructions to update the matrix before and after multiplication and 4 dot-product instructions available in the GPU instruction set (DP4).

C. GPU Implementation of a Frequency Translating FIR Filter

Frequency translation is applied to the results of the FIR filter with real inputs and complex taps. The results of this FIR filter are complex numbers. Each pixel of the taps texture contains only two complex numbers instead of four real numbers. Therefore, the taps texture size is $m \times k/2$ and the output texture size is $n \times k/2$. The four real numbers stored in one pixel of the input texture are used to compute four complex numbers of the output texture that are saved in two consecutive pixels. The current fragment processing units cannot generate two different pixels of the same destination buffer in one iteration. Therefore, each pixel of the input texture is used twice to generate two output pixels. The

```
float4 color0 = texRECT(tex0, IN.texcoord0);
tmp=float4(0,0,0,0);
mat=float4x4(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0);

for (float i=0; i<Params.x; i+=1) {
  coord1 = IN.texcoord1+float2(i, 0);
  coord2 = IN.texcoord2+float2(i, 0);
  if (coord1.x>=Params.x) {
    coord1.x -= Params.x; coord1.y += 1;
  }

  color1=texRECT(tex1, coord1);
  color2=texRECT(tex2, coord2);
  mat[0]=color2;
  mat[1].yzw=color2.xyz;
  mat[2].zw=color2.xy;
  mat[3].w=color2.x;
  tmp=tmp+mul(mat, color1);
  mat[1].x=color2.w;
  mat[2].xy=color2.zw;
  mat[3].xyz=color2.yzw;
}

coord1 = IN.texcoord1+float2(Params.x, 0);
if (coord1.x>=Params.x) {
  coord1.x -= Params.x; coord1.y += 1;
}

color1=texRECT(tex1, coord1);
mat[0]=float4(0,0,0,0);
mat[1].yzw=float3(0,0,0);
mat[2].zw=float2(0,0);
mat[3].w=0.0f;
tmp=tmp+mul(mat, color1);
OUT.Color0=tmp+color0;
```

Fig. 2. FIR filter implementation on a GPU in Cg language.

$(m+n)\times k/4$ input texture is stretched horizontally to size $(m+n)\times k/2$. Pixel $x\cdot 2, y$ of the output texture stores FIR filter result of input numbers $(i_{x,y})_r$ and $(i_{x,y})_g$ and pixel $x\cdot 2+1, y$ stores the FIR filter result of inputs $(i_{x,y})_b$ and $(i_{x,y})_a$. The fragment program proceeds differently depending on the result of $x \bmod 2$. In the following equations the first tap pixel t_{2j} of j-th iteration of the loop of the fragment program is denoted as $Re_1^j, Im_1^j, Re_2^j, Im_2^j$, the second tap pixel t_{2j+1} is denoted as $Re_3^j, Im_3^j, Re_4^j, Im_4^j$. Pixels t_{2j}, t_{2j+1}, $t_{2(j-1)}$, and $t_{2(j-1)+1}$ are used at each iteration of the fragment program's loop.

x **is even.** The output pixel is initialized using the following equation:

$$\begin{array}{l} o_r^0 = i_r Re_1^0 + i_g Re_2^0 + i_b Re_3^0 + i_a Re_4^0 \\ o_g^0 = i_r Im_1^0 + i_g Im_2^0 + i_b Im_3^0 + i_a Im_4^0 \\ o_b^0 = i_g Re_1^0 + i_b Re_2^0 + i_a Re_3^0 \\ o_a^0 = i_g Im_1^0 + i_b Im_2^0 + i_a Im_3^0 \end{array}.$$

The fragment program's loop increments the output pixel as follows:

$$O^{j+1} = O^j + \begin{pmatrix} Re_1^j & Re_2^j & Re_3^j & Re_4^j \\ Im_1^j & Im_2^j & Im_3^j & Im_4^j \\ Re_4^{j-1} & Re_1^j & Re_2^j & Re_3^j \\ Im_4^{j-1} & Im_1^j & Im_2^j & Im_3^j \end{pmatrix} \cdot I^j.$$

The final value of the output is computed as

$$O = O^k + \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ Re_4^{k-1} & 0 & 0 & 0 \\ Im_4^{k-1} & 0 & 0 & 0 \end{pmatrix} \cdot I^k.$$

x **is odd**. The output pixel is initialized as follows:

$$O^0 = \begin{pmatrix} 0 & 0 & Re_1^0 & Re_2^0 \\ 0 & 0 & Im_1^0 & Im_2^0 \\ 0 & 0 & 0 & Re_1^0 \\ 0 & 0 & 0 & Im_1^0 \end{pmatrix} \cdot I^0.$$

The output is updated at each loop's iteration using the following equation:

$$O^{j+1} = O^j + \begin{pmatrix} Re_3^{j-1} & Re_4^{j-1} & Re_1^j & Re_2^j \\ Im_3^{j-1} & Im_4^{j-1} & Im_1^j & Im_2^j \\ Re_2^{j-1} & Re_3^{j-1} & Re_4^{j-1} & Re_1^j \\ Im_2^{j-1} & Im_3^{j-1} & Im_4^{j-1} & Im_1^j \end{pmatrix} \cdot I^j.$$

The final value of the output pixel is computed as follows:

$$O = O^k + \begin{pmatrix} Re_3^{k-1} & Re_4^{k-1} & 0 & 0 \\ Im_3^{k-1} & Im_4^{k-1} & 0 & 0 \\ Re_2^{k-1} & Re_3^{k-1} & Re_4^{k-1} & 0 \\ Im_2^{k-1} & Im_3^{k-1} & Im_4^{k-1} & 0 \end{pmatrix} \cdot I^k.$$

The source code of the fragment program is presented in Figure 3.

D. GPU Implementation of a Hilbert Transformation

Hilbert transformation uses FIR filter for the imaginary component of the output and assigns an input element to the real component of the output. Fetching the input element for the real component of the output pixel is the main implementation difficulty because input elements are stored in four-tuples.

To fetch the input element i_{n+l} for the output element o_n the C++ program computes the offsets x_{off} and y_{off} of element i_{n+i} in the input texture and passes them to the fragment program as parameters. The fragment program fetches the input texture pixel with the specified offsets and the pixel next to it. Two pixels are required because data is stored in four-tuples in the input texture. For example, if $l = 7$ then the required element for input value stored in the r-component of the current input pixel is stored in the a-component of the pixel next to it, whereas the required element for the g-component of the current pixel is stored in r-component of the pixel two pixels off the current pixel. The fragment program computes the output pixel using the two fetched pixels s^1 and s^2 and masks m^1 and m^2 defined for R, G, B, and A components of the pixel. For example, the following equation is used to compute the r-component of the output pixel:

$$o_r = dot\left(s^1, m_r^1\right) + dot\left(s^2, m_r^2\right).$$

```
tmp=float4(0,0,0,0);
mat=float4x4(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0);
xmod2=fmod(IN.texcoord0.x+0.1, 2)-1.0f;
for (float i=0; i<Params.x; i+=1) {
  tmp=float4(0,0,0,0);
  coord1=IN.texcoord1+float2(i, 0)
  if (coord1.x>=Params.x) {
    coord1.x -= Params.x; coord1.y += 1;
  }
  color1=texRECT(tex1, coord1);
  color2=texRECT(tex2, IN.texcoord2
  +float2(i*2, 0));
  color3=texRECT(tex2, IN.texcoord2
  +float2(i*2+1, 0));
  if (xmod2>0) {
    /* odd */
    mat[0].zw=color2.xz; mat[1].zw=color2.yw;
    mat[2].w=color2.x; mat[3].w=color2.y;
    tmp=tmp+mul(mat, color1);
    mat[0].xy=color3.xz; mat[1].xy=color3.yw;
    mat[2].xyz=float3(color2.z, color3.x, color3.z);
    mat[3].xyz=float3(color2.w, color3.y, color3.w);
  } else {
    /* even */
    mat[0]=float4(color2.x, color2.z,
    color3.x, color3.z);
    mat[1]=float4(color2.y, color2.w,
    color3.y, color3.w);
    mat[2].yzw=float3(color2.x, color2.z, color3.x);
    mat[3].yzw=float3(color2.y, color2.w, color3.y);
    tmp=tmp+mul(mat, color1);
    mat[2].x=color3.z; mat[3].x=color3.w;
  }
}
coord1 = IN.texcoord1+float2(Params.x, 0);
if (coord1.x>=Params.x) {
  coord1.x -= Params.x; coord1.y += 1;
}
color1=texRECT(tex1, coord1);
if (xmod2>0) {
  mat[0].zw=float2(0,0); mat[1].zw=float2(0,0);
  mat[2].w=0.0f; mat[3].w=0.0f;
} else {
  mat[0]=float4(0,0,0,0);
  mat[1]=float4(0,0,0,0);
  mat[2].yzw=float3(0,0,0);
  mat[3].yzw=float3(0,0,0);
}
tmp=tmp+mul(mat, color1);
```

Fig. 3. A GPU implementation of a complex FIR filter.

```
xmod2=fmod(IN.texcoord0.x, 2)-1.0f;
tmp=float4(0,0,0,0);
coord1.x=floor(IN.texcoord1.x)+Params.z;
coord1.y=floor(IN.texcoord1.y)+Params.w;
if (coord1.x>=Params.x) {
  coord1.x -= Params.x; coord1.y += 1;
}
color1=texRECT(tex1, coord1);
coord1.x=floor(IN.texcoord1.x)+Params.z+1;
coord1.y=floor(IN.texcoord1.y)+Params.w;
if (coord1.x>=Params.x) {
  coord1.x -= Params.x; coord1.y += 1;
}
color2=texRECT(tex1, coord1);
if (xmod2>0) {
  OUT.Color0.x = dot(ParamZ1, color1)+
  dot(ParamZ2, color2);
  OUT.Color0.z = dot(ParamW1, color1)+
  dot(ParamW2, color2);
} else {
  OUT.Color0.x = dot(ParamX1, color1)+
  dot(ParamX2, color2);
  OUT.Color0.z = dot(ParamY1, color1)+
  dot(ParamY2, color2);
}
// FIR code follows
```

Fig. 4. Cg program of the Hilbert transformation. `Params.z` equals x_{off}, `Params.w` equals y_{off}.

In our example, $m_r^1 = (0,0,0,1)$, $m_r^2 = (0,0,0,0)$, and $m_g^1 = (0,0,0,0)$, $m_g^2 = (1,0,0,0)$. Masks $m_b^{1,2}$ and $m_a^{1,2}$ are defined similarly. The C++ program passes the masks to the fragment program as parameters.

The source code of the fragment program is presented in Figure 4.

E. Optimization of GPU programs

We have tried to optimize the GPU implementations using a few standard techniques that improve performance of assembly programs on Pentium processors. In particular, we tried to:

- Break the loop of the GPU program into two thus getting rid of the conditional expression in loop's body;
- Unroll the loop body with and without the conditional expression in it;
- Process two rows of input and taps textures inside the loop, that is to reduce the number of iterations of the outer loop in the C++ program;
- Use different texture units with unrolled loops. The idea was that fetching from different texture units would outperform fetching from the same texture unit.

However, none of these changes improved the performance of our original program.

V. Performance Evaluation

We evaluated the performance of our implementation of the FIR filter using a Pentium 4 HT 3.2 GHz with a Geforce

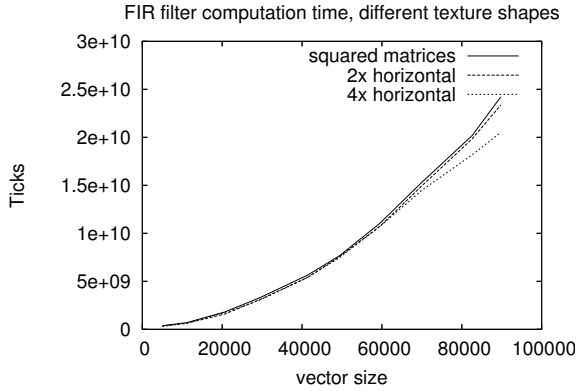

Fig. 6. GPU algorithm performance results for different shapes of the input matrices.

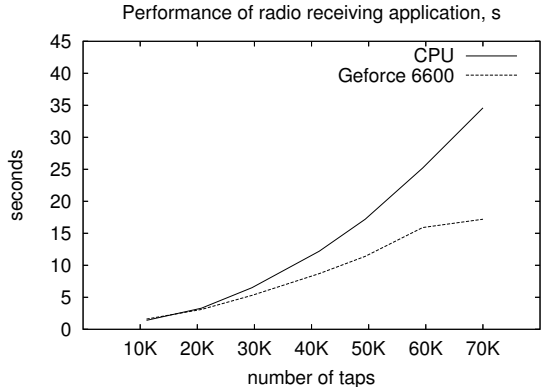

Fig. 9. Performance of the radio receiving application for different number of taps of FIR filters. The measured time is the time to generate 500 output samples.

6600 video card. The GPU implementation was compared with an SSE-optimized CPU implementation from Gnu Radio software. The results presented in Figure 5 imply that the GPU implementation outperforms the CPU implementation for vector sizes greater than 60,000.

We varied several parameters of GPU algorithm to achieve highest performance. Specifically, we varied the matrix shape that the input vectors are converted into. We tried squared matrices, 2x horizontally increased matrices, and 4x horizontally increased matrices. The results are presented in Figure 6. The results prove that matrices wider horizontally better performance-wise. Wider matrices decrease the number of iterations of the GPU algorithm.

Figure 7 presents the running time of the GPU implementation of the frequency translating FIR filter that uses complex taps and generates complex output versus the CPU implementation of the algorithm. The GPU implementation outperforms the CPU implementation because the latter is not SSE-optimized.

The results of comparing the CPU implementation of Hilbert transformation with the GPU implementation are shown in Figure 8. The conclusion is that the GPU implementation is significantly slower than the unoptimized CPU implementation. The reason is the fact that Hilbert transformation uses a small number of taps, 31 in our experiments. We ran a minimal GPU program that only did two texture fetches and compared it with the CPU implementation of Hilbert transformation. It turned out that the minimal GPU program is significantly slower than the CPU implementation of Hilbert transformation as well. These results imply that it is not worthwhile to run Hilbert transformation on a GPU.

We also measured the percentage each step of the GPU algorithm takes: texture upload time (from main memory to video memory), the GPU computation time, and texture download time (from video memory to the main memory). The texture upload time was less than 1% of the total time. The texture download time was 10% on average. Therefore, the computation time was 90% of the total time. The results are presented on the right hand side of Figures 5, 7, and 8. These results show that the FIR filter and the frequency translation FIR filter are computation-bound programs, whereas the Hilbert transformation is a memory bandwidth-bound program. The results also imply that connecting several blocks together and eliminating the unnecessary texture transfers will improve performance.

In our final experiment we evaluated the performance of a radio receiver application. The application uses the following building blocks:

- GrFreqXlatingFIRFilterFCF — frequency translating FIR filter, uses GPU;
- GrReal — converts complex output of the previous block to real numbers, uses CPU;
- GrHilbert — Hilbert transformation, uses GPU;
- VrQuadratureDemod — demodulation, uses CPU;
- GrFIRFilterFFF — FIR filter, uses GPU.

We compared the time it takes to generate 500 output samples when all blocks use CPU and when the GPU implementations described in this paper are used. The results are presented in Figure 9. They indicate that implementing certain blocks on GPU improves application performance for FIR filter with a large number of taps. The GrReal and VrQuadratureDemod blocks are unlikely to improve from implementing them on GPU because they process all input elements only once. The only possible improvement could result from eliminating redundant memory copying from video to the main memory. To evaluate this possible performance gain we replaced all video to main memory copying functions with a function that generated random numbers. It turned out that the performance improvement was insignificant. Therefore, we concluded that implementing GrReal and quadrature demodulation blocks on the GPU will not improve performance of the application.

VI. Conclusion

In this project we developed a GPU implementation of a FIR filter, evaluated its performance, and compared it to that of the SSE-optimized implementation. The results indicate that the GPU implementation is better than the CPU implementation for larger inputs. However, the GPU implementation is worse when inputs are not large. We evaluated the GPU run-time breakdown that includes the GPU computation time and data

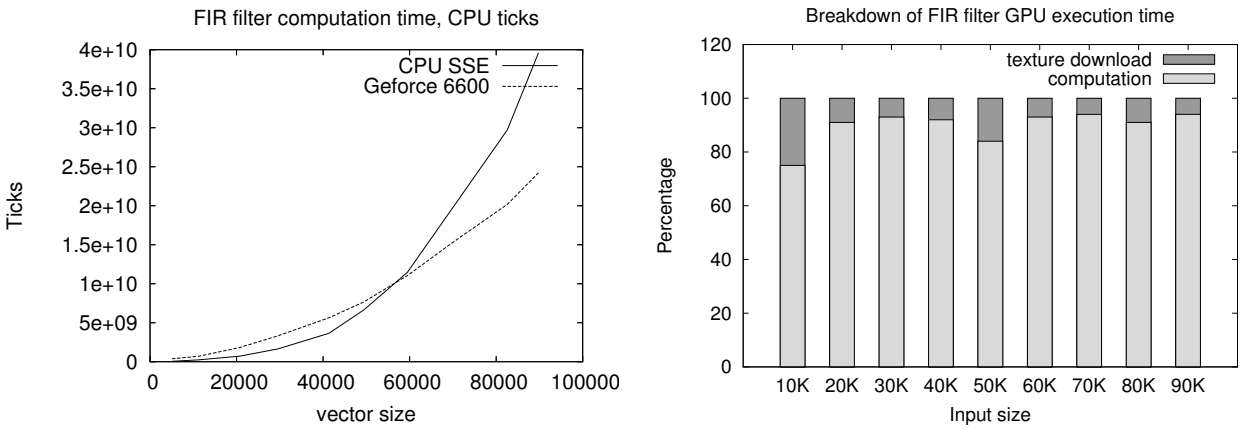

Fig. 5. Performance comparison of CPU and GPU implementations. The number of taps equals the number of output elements.

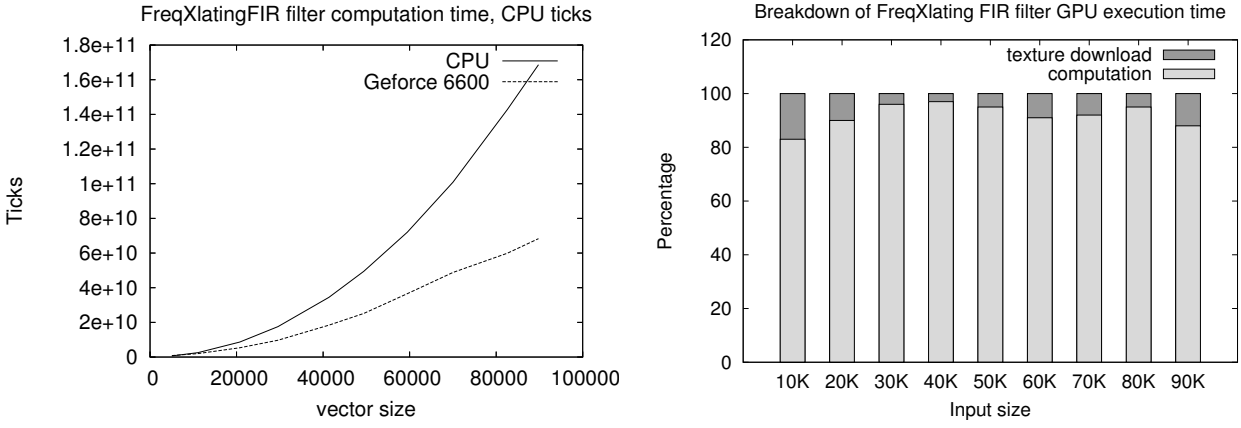

Fig. 7. Performance of the GPU implementation of frequency translating FIR filter versus the non-optimized CPU implementation.

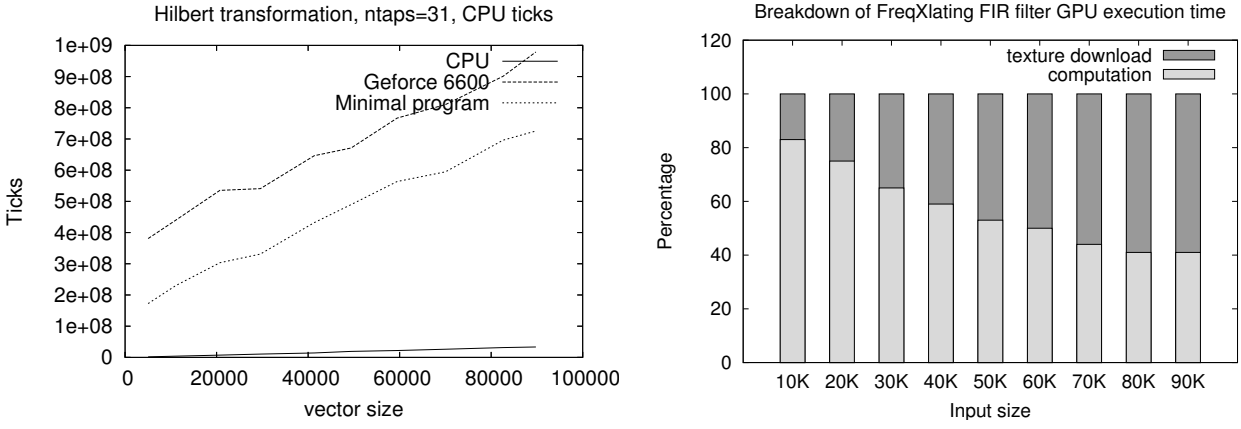

Fig. 8. Performance of the GPU implementation of Hilbert transformation. The number of taps is 31.

transfer time. This experiment allowed us to identify GPU-computation-bound and memory-bound building blocks. A radio receiving application was evaluated using the GPU implementation of the building blocks. The application performance improved when FIR filters have a large number of taps and therefore, require a large number of input data.

The performance evaluation showed that a GPU computation can take several seconds compared to a millisecond time it takes to render a frame in a computer game. Therefore, special OS support is required when several applications that use GPU run at the same time. The goal of the OS support is to ensure that each application gets a fair share of the GPU resources. Another interesting direction of future research is to implement a compiler that can recognize and optimize automatically source code that can improve from running it on GPU.

REFERENCES

[1] K. Fatahalian, J. Sugerman, and P. Hanrahan, "Understanding the efficiency of GPU algorithms for matrix-matrix multiplication," in *Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 2004.

[2] R. G. Lyons, *Understanding Digital Signal Processing, Second Edition*. Prentice Hall PTR, 2004.

[3] GNU Radio — The GNU Software Radio, http://www.gnu.org/software/gnuradio/.

[4] A. Moravanszky, "Dense matrix algebra on the GPU," http://www.shaderx2.com/shaderx.PDF.

[5] E. S. Larsen and D. McAllister, "Fast matrix multiplies using graphics hardware," in *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, 2001.

[6] J. D. Hall, N. A. Carr, and J. C. Hart, "Cache and bandwidth aware matrix multiplication on the GPU," Tech Report UIUCDCS-R-2003-2328, University of Illinois Dept. of Computer Science, Tech. Rep., 2003.

[7] N. K. Govindaraju, N. Raghuvanshi, M. Henson, and D. Manocha, "A cache-efficient sorting algorithm for database and data mining computations using graphics processors," http://gamma.cs.unc.edu/GPUSORT/.

[8] N. Galoppo, N. K. Govindaraju, M. Henson, and D. Manocha, "LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware," in *Proceedings of the ACM/IEEE Supercomputing Conference*, 2005.

[9] K. Moreland and E. Angel, "The FFT on a GPU," in *Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 2003.

[10] T. Jansen, B. v. Rymon-Lipinski, N. Hanssen, and E. Keeve, "Fourier volume rendering on the GPU using split-stream-FFT," in *Proceedings of the Vision, Modeling, and Visualization Conference*, 2004.

[11] J. Krueger and R. Westermann, "Linear algebra operators for gpu implementation of numerical algorithms," *ACM Transactions on Graphics*, vol. 22, no. 3, pp. 908–916, 2003.

[12] K. E. Hillesland, S. Molinov, and R. Grzeszczuk, "Nonlinear optimization framework for image-based modeling on programmable graphics hardware," *ACM Transactions on Graphics*, vol. 22, no. 3, pp. 925–934, 2003.

[13] D. R. Horn, M. Houston, and P. Hanrahan, "ClawHMMer: A streaming HMMer-search implementation," in *Proceedings of the ACM/IEEE Supercomputing Conference*, 2005.

[14] O. the industry standard for high-performance graphics, http://www.opengl.org.

[15] NVIDIA OpenGL Extension Specifications, http://developer.nvidia.com/object/nvidia_opengl_specs.html.