Proceedings of the 5th WSEAS Int. Conf. on SIGNAL, SPEECH and IMAGE PROCESSING, Corfu, Greece, August 17-19, 2005 (pp34-39)

Computation of Filtered Back Projection on Graphics Cards

VÍTĚZSLAV VÍT VLČEK
Department of Mathematics
University of West Bohemia
Faculty of Applied Sciences, P.O.BOX 314, 306 14 PLZEN 6
CZECH REPUBLIC
http://nynfa2-kma.fav.zcu.cz/~vsoft/iradon

Abstract: - Common graphics cards have a programmable processor that we can use for some mathematical computations. I will explain how we can use the performance of the graphics processor in this brief report. I focused on the implementation of the inverse Radon transform by method of the filtered backprojection. The GPU implementation of the filtered backprojection can be 0.5–4 times faster than the optimized CPU version. It depends on used hardware.

Key-Words: - Filtered backprojection, graphics processing unit, inverse Radon transform, parallel computation, FFT, FFT filtering

1 Introduction

Recently common graphics cards have included a high performance processor. This processor is called a graphics processing unit (GPU), and it can process a lot of graphics data at one time. The performance of the GPU can be better than the performance of a common CPU (central processor unit) in some cases – among others – the GPU of GeForce 6800GT contains 222 million transistors and the CPU of AMD 64 has about 105 million transistors. This is the reason why I try to use the graphics card for mathematical computation.

I decided to extend the GPU implementation of the Filtered BackProjection (FBP) on the basis of previous research [5]. There was only implemented the second part of FBP. The second part of FBP is an integration of filtered sinogram. The extension consists in addition of the fast Fourier filtering.

Of course, there are other possibilities of how to realize the inverse Radon transform. The first way is a direct method: Fourier Slice Theorem, Filtered Backprojection and Filtering after Backprojection. The second way is by reconstruction algorithms based on linear algebra: EM Algorithm, Iterative Reconstruction using ART and the reconstruction based on the Level Set Methods.

I chose FBP for the GPU realization, because the FBP is used in most scanners today.

2 GPU Programming

The GPU has a different instruction set to ordinary CPU's, that's why GPU's cannot carry out the same program as CPU's. We need special GPU's languages.

2.1 Programming Languages for GPU

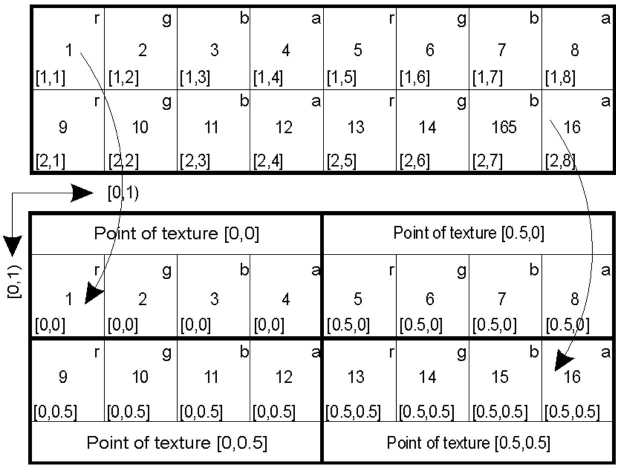
The programming languages for the GPU are divided into two platforms: Microsoft Windows and Linux. Both the high-level shader language (HLSL) and a system for programming graphics hardware in a C-like language (Cg) are used in MS Windows. It is necessary to say that the HLSL and the Cg are semantically 99% compatible. The HLSL is connected with MS DirectX 3D and the Cg is connected with OpenGL, hence we can use the Cg in Linux. Of course, we could use the assembly language for the GPU, but it is too difficult.

The GPU consists of two vector processors: Vertex Shader (VS) and Pixel Shader (PS). The PS is more suitable for our purposes because it is faster than the VS. The development of the graphics card is too fast, hence there are a few other versions of the PS. The PS of the version 2.0 provides the floating point data processing that is why it is helpful for mathematical computing. The previous versions of the PS only facilitated 8-bit data processing. Nowadays there is PS of version 3.0. It has new features (dynamic branches, in particular).

I decided to use PS of version 2.0 because of compatibility and HLSL because of easier development of the GPU program.

2.2 Data and GPU

Data in the GPU is stored in special structure. Each group of data is called a texture. The texture is similar to a matrix, but there are some differences. A point of the texture is called texel. The texel consists of four entries: red, green, blue and alpha channel. Each of these entries is represented by a floating point number, so one texel is represented by four floating point numbers. See Fig. 1 for the matrix-texture mapping.

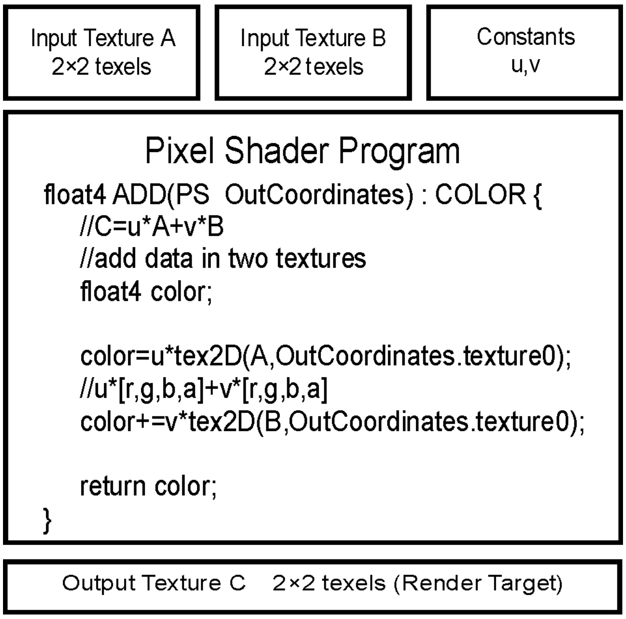


r 1 [1,1]	g 2 [1,2]	b 3 [1,3]	a 4 [1,4]	r 5 [1,5]	g 6 [1,6]	b 7 [1,7]	a 8 [1,8]
r 9 [2,1]	g 10 [2,2]	b 11 [2,3]	a 12 [2,4]	r 13 [2,5]	g 14 [2,6]	b 165 [2,7]	a 16 [2,8]

Point of texture [0,0]				Point of texture [0.5,0]			
r 1 [0,0]	g 2 [0,0]	b 3 [0,0]	a 4 [0,0]	r 5 [0.5,0]	g 6 [0.5,0]	b 7 [0.5,0]	a 8 [0.5,0]
r 9 [0,0.5]	g 10 [0,0.5]	b 11 [0,0.5]	a 12 [0,0.5]	r 13 [0.5,0.5]	g 14 [0.5,0.5]	b 15 [0.5,0.5]	a 16 [0.5,0.5]
Point of texture [0,0.5]				Point of texture [0.5,0.5]			

Fig. 1: Mapping of Matrix Entries (upper) to Texture Points (down).

The reason why the texture consists of the texels comes from computer graphics. The red, green, and blue channels determine a color of the pixel while the alpha channel determines the transparency of the pixel.



```
float4 ADD(PS  OutCoordinates) : COLOR {
   //C=u*A+v*B
   //add data in two textures
   float4 color;

   color=u*tex2D(A,OutCoordinates.texture0);
   //u*[r,g,b,a]+v*[r,g,b,a]
   color+=v*tex2D(B,OutCoordinates.texture0);

   return color;
}
```

Fig. 2: Example of the Pixel Shader Program. This program performs C=u·A+v·B

The GPU is a vector processor which can process the red, green, blue and alpha entries of the texel in parallel. The GPU uses well known technique Single Instruction, Multiple Data (SIMD).

The PS is a processor to which the program and the data (textures) are incoming. The output of the PS is the texture (the render target texture in the D3D) or more textures, it depends on the features of the graphics card, which contain the computed values. The computed values are in the same format as the input textures. The PS program can only read a finite number of the texture points (approx. 12, it depends on the card). There is a further restriction for writing the texture point; the PS program obtains the output texture coordinates from the PS. So the PS program cannot write where it wants but the PS program has to write there where the PS wants. This is one of biggest restrictions. The PS can only do the static loops which the compiler unrolls. The PS program cannot read the output data during the pass.

You can see very simple PS program in Fig. 2. The program only performs matrix operation C=u·A+v·B, where u, v are any constants and C, A, B are matrices. The PS program only carries out operation with single texel of the output matrix C that is determined by OutCoordinates. The multiplication `u*tex2D(A,OutCoordinates.texture0)` is scalar–vector operation, because tex2D returns texel. The matrix multiply is not possible easily to implement, because we need another loop for the inner product, which is not allowed for PS version 2.0 because of a reduced instruction set [4].

We need an additional program for both the data and the PS program loading into the graphics card. I call this program the graphics framework.

3 Filtered Backprojection

The FBP is a very famous inverse scheme. I introduce some useful notations for simplification. Let $g(x, y)$ be a source signal, let $g^*(\theta, t) = R_{x,y\to\theta,t}\{g(x, y)\}$ be the Radon transform $R\{\}$ of the function $g(x, y)$. Let $H(\tau)=F_{x\to\tau}\{h(x)\}$ be the direct Fourier transform $F\{\}$ of the function $h(x)$ and the inverse Fourier transform $IF\{\}$ be denoted by $h(x)= IF_{\tau\to x}\{H(\tau)\}$.

The FBP can be expressed by the formulae

$$\begin{aligned} g^{\#}(\theta,\rho) &= IF_{\upsilon\to\rho}\{|\,\upsilon\,|\,FT_{t\to\upsilon}\{R_{x,y\to\theta,t}\{g(x,y)\}\}\}, \\ g(x,y) &= \int_0^{\pi} g^{\#}(\theta, x\cos\theta + y\sin\theta)d\theta. \end{aligned} \qquad (1)$$

The terms (1) consist of two parts: the first is a filtering part and the second is an integration part [1]. We can derive a discrete implementation of the FBP [1].

3.1 Algorithm of Discrete FBP

The following algorithms of the FBP were written in a pseudo-code.

```
//1D FFT filtering part of the FBP
//It performs 1D fast Fourier
//transform on each row of matrix g.
$fg = \mathrm{FFT}(g)$
//Filtering and 1D Inverse FFT
$ifg = \mathrm{IFFT}(fg \cdot |\upsilon|)$

//the integration part of the FBP
//optimized for CPU
for $m = 0$ to $M-1$
   for $n = 0$ to $M-1$
      $sum = 0$
      for $f = 0$ to $F-1$
         $pos = \mathrm{floor}(m \cdot \cos(f \cdot \Delta_f)$
                 $+ n \cdot \sin(f \cdot \Delta_f) - \rho_{\min})/\Delta_\rho$
         $sum = sum + ifg(f, pos)$
      end
      $h(m,n) = sum \cdot \Delta_f$
   end
end
```

The integration part of this algorithm is optimized for the CPU because it uses the memory cache efficiently. Unfortunately, this version is unsuitable for the GPU implementation because the PS carries out the loops m and *n* over implicitly and it cannot perform the loop over *f* that is why I had to shift the loop *f* to the loops *m*, *n* then I got the new GPU optimized version of the integration part of the FBP.

```
//1D FFT filtering part of the FBP
//It performs 1D fast Fourier
//transform on each row of matrix g.
$fg = \mathrm{FFT}(g)$
//Filtering and 1D Inverse FFT
$ifg = \mathrm{IFFT}(fg \cdot |\upsilon|)$

//the integration part of the FBP
//optimized for CPU
$h(m,n)=0$ //clear all output matrix
for $f = 0$ to $F-1$
   for $m = 0$ to $M-1$
      for $n = 0$ to $M-1$
         $pos = \mathrm{floor}(m \cdot \cos(f \cdot \Delta_f)$
                 $+n \cdot \sin(f \cdot \Delta_f) - \rho_{\min})/\Delta_\rho$
         $h(m,n) = h(m,n) + ifg(f, pos) \cdot \Delta_f$
      end
   end
end
```

The previous algorithm is written in a pseudo-code. The variables fg, ifg, g and h are matrices (the textures in the D3D). The constants M, F, $\rho_{\min}$, Δ_f, Δ_ρ depend on the discrete Radon transform of the source signal $g(m,n)$.

3.2 GPU Implementation

The FBP can be divided into two parts. The first part is the data filtering by the Fourier filtering. The Fourier filtering consists of the 1D FFT, then the spectrum filtering and at last the 1D IFFT. The second part of the FBP is the integration part, so we have to write two PS programs.

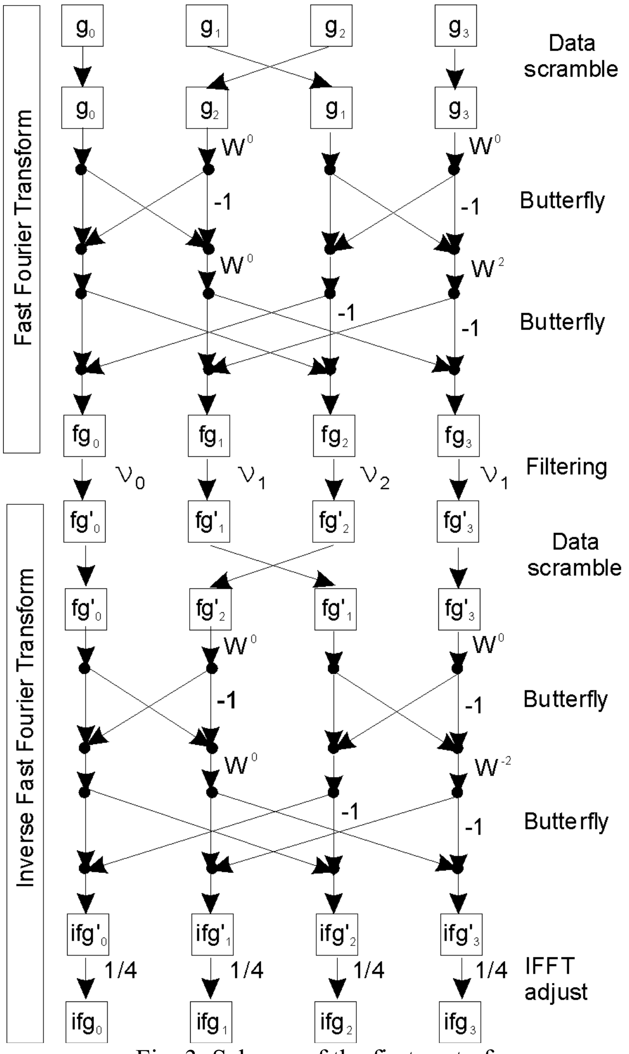


Fig. 3: Scheme of the first part of the Filtered BackProjection

3.2.1 GPU Implementation of Fourier Filtering

The Fast Fourier Transform is divided into two parts. The first part is data scramble and the second part is application of the butterfly operations.

The data scramble has special form so-called bit-reverse. The binary number 1110_2 is 14_{10} in decadic number system and its bit-reverse is $0111_2= 7_{10}$. The data scramble means we have to exchange the locations of data in bit-reverse sense.

Once the data has been scrambled we perform series of butterfly operations. The butterfly operation carries out both complex multiply and complex addition of source data. The inability of the PS to write to random positions in memory causes, we have to perform many additional operations than standard implementation of the FFT.

Once the FFT has been done we apply the filter on the spectrum and then we perform the inverse Fourier transform. See Fig. 3 for whole Fourier filtering process that is applied on each row of matrix g in parallel.

The computation of the bit-reverse cannot be done in the PS program, because the PS does not have suitable instructions for bit operations. Since, we have to create a temporary vector of bit-reverse vaules. See Table 1.

Position	$0_{10}=00_2$	$1_{10}=01_2$	$2_{10}=10_2$	$2_{10}=10_2$
Bit reverse	$0_{10}=00_2$	$2_{10}=10_2$	$1_{10}=01_2$	$2_{10}=10_2$

Table 1: Vector of bit-reverse

The butterfly operations are implemented in the following way. I created the special temporary FFT map [6]. The FFT map is texture, that has $\rho_{\mathtt{max}}=\upsilon_{\mathtt{max}}$ columns and $2\cdot\log_2(\rho_{\mathtt{max}})+1$ rows. $\rho_{\mathtt{max}}$ is a count of columns of the matrix g. The red and green channel contains the location of the first and second operand of butterfly operation. The blue and alpha channel contains real and imaginary part of weight W, so we have necessary data for the performing $\rho_{\mathtt{max}}$ butterfly operations on each row. The FFT performs $\log_2(\rho_{\mathtt{max}})$ passes, hence we need $\log_2(\rho_{\mathtt{max}})$ rows in the FFT Map. We do not forget to perform the data scramble, so we modify all position entries in the first row of the FFT Map.

The next step is filtering, for which we need another row in the FFTMap, the so-called filter row. Because I want to use same the PS program for filtering, I add the filter row at the end of the FFT part. Then I can perform the last step of the IFFT (adjusting) during the filtering.

The filter row has the following structure: both red and green channel contain same position and the blue channel contains $\upsilon/\rho_{\mathtt{max}}$-1 and the alpha channel is filled by zero.

At last we add rows for IFFT at the end of the FFT Map. We do not forget to change locations in the first row of the IFFT (the data scramble). The count of rows in the FFT Map is given by FFT+filtering+IFFT, i.e. $\log_2(\rho_{\mathtt{max}})+1+\log_2(\rho_{\mathtt{max}})$.

Unfortunately, the PS program cannot perform dynamics loops hence we have to carry out “texture pingpong”. The texture pingpong is a special technique for smart use of data from prior render steps. In case of the FFT we create two textures. The first texture A is used as the render target and the second texture B is filled with matrix g. We set the PS constant `FFTPass` to 0 and then we perform the first FFT pass. Once the FFT pass has been done we exchange texture B with A, we set `FFTPass` to the next line in the FFT map. We are ready to perform the next FFT pass. See Table 2 for the FFT Map, the first part of the Fig. 4, GPU Fourier filtering algorithm and Pixel Shader Programs.

GPU Fourier filtering algorithm

1. create texture A, B
2. fill texture B with matrix g
3. set `FFTPass`
4. run the PS program for fast Fourier filtering
5. switch A with B
6. go to 3 until `FFTPass`<count of FFTMap rows

The implementation of the BackProjection part can be done in the same way. We have to use the pingpong technique again for sum over the loop f. See Fig 4.

r	g	b	a	r	g	b	a	r	g	b	a	r	g	b	a
0	2	1	0	0	2	-1	0	1	3	1	0	1	3	-1	0
0	2	1	0	1	3	0	1	0	2	-1	0	1	3	0	-1
0	0	-1	0	1	1	U	0	2	2	V	0	3	3	U	0
0	2	1	0	0	2	-1	0	1	3	1	0	1	3	-1	0
0	2	1	0	1	3	0	-1	0	2	-1	0	1	3	0	1

Table 2: FFT Map
Rows 1, 2 are used for FFT, row 3 is used for filtering and rows 4, 5 are used for IFFT.
$\mathrm{U}=\upsilon_1/4$ -1, $\mathrm{V}=\upsilon_2/4$ -1
The data in all columns r and g are multiplied by 4 for better reading.

3.2.2 Pixel Shader Program

```
texture tSinogram;
texture tPingpong;
texture tFFTMap;
```

```
sampler Sinogram = sampler_state {
    Texture   = <tSinogram>; };
sampler FFTMap = sampler_state {
    Texture   = <tFFTMap>; };
sampler Pingpong = sampler_state {
    Texture   = < tPingpong>; };

struct Vs_Input {
   float3 vertexPos  : POSITION;
   float2 texture0   : TEXCOORD0; };
struct Vs_Output {
   float4 vertexPos  : POSITION;
   float2 texture0   : TEXCOORD0; };

//constants for Back Projection
float4 theta;
//x_delta, x_min, x_max,0
float4 x;
//y_delta, y_min, y_max,0
float4 y;
//rho_delta, rho_minimum, rho_count, 0
float4 rho;

// The BackProjection pixel shader...
float4 PS_BP(Vs_Output In) : COLOR {
 float2 position;
 float4 color;
 float  tmp;

 tmp= dot(float3(
          //(x*x_delta+1*x_min)+
          dot(float2(In.texture0.y*x.b,1),x.rg),
          //(y*y_delta+1*y_min)+
          dot(float2(In.texture0.x*y.b,1),y.rg),
          -1),float3(theta.gb,rho.g));
 tmp=floor(tmp/rho.r)/rho.b;
 position=float2(tmp,theta.r);
 color=tex2D(Pingpong,In.texture0)
      +theta.a*tex2D(Sinogram,position);
 return color;
}

//constant to tell which pass is being used
float FFTPass;
// FF filterring pixel shader
float4 PS_FFT( Vs_Output In ) : COLOR {
 float2 sampleCoord;
 float4 butterflyVal;
 float2 a;
 float2 b;
 float2 w;
 float temp;

 sampleCoord.x = In.texture0.x;
 sampleCoord.y = FFTPass;
 butterflyVal= tex2D( FFTMap, sampleCoord);
 w = butterflyVal.ba;

 //sample location A
 sampleCoord.x = butterflyVal.r;
 sampleCoord.y = In.texture0.y;
 a = tex2D( Pingpong, sampleCoord).ra;

 //sample location B
 sampleCoord.x = butterflyVal.g;
 sampleCoord.y = In.texture0.y;
 b = tex2D( Pingpong, sampleCoord).ra;

 //multiply w*b (complex numbers)
 temp = w.x*b.x - w.y*b.y;
 b.y = w.y*b.x + w.x*b.y;
 b.x = temp;

 //perform a + w*b
 a = a + b;
 return a.xyxy;
}
```

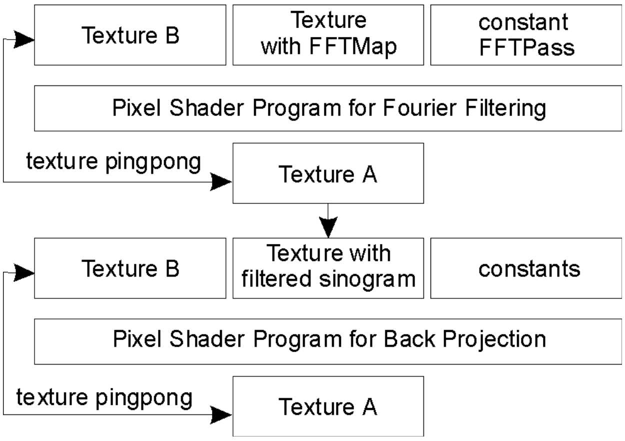

Texture B
Texture with FFTMap
constant FFTPass
Pixel Shader Program for Fourier Filtering
texture pingpong
Texture A
Texture B
Texture with filtered sinogram
constants
Pixel Shader Program for Back Projection
texture pingpong
Texture A

Fig. 4: Scheme of GPU implementation of the Filtered BackProjection

3.2.4 Pixel Shader Program

The framework program for the Fourier filtering and BackProjection is the following.

```
//Fourier Filtering
FFTMap=CreateFFTMap(ρ$_{max}$);
PingpongA=CreateTexture();
PingpongB=CreateTexture();
LoadPixelShader("PS_FFT");
SetRenderTaget(PingpongA);
SetTexture("tPingpong",PingpongB);
SetTexture("tFFTMap",FFTMap);
for i=0 to log$_2$(ρ$_{max}$)-1
 SetConstant("FFTPass",i/log$_2$(ρ$_{max}$));
 Render();
 SwapRenderTagetAndPingpong();
end
Release(FFTMap);

//BackProjection
Sinogram=CreateTexture();
LoadPixelShader("PS_BP");
CopyRenderTargetInto(Sinogram);
ClearTexture(PingpongB);
SetTexture("tPingpong",PingpongB);
SetTexture("tSinogram",Sinogram);
SetRenderTaget(PingpongA);
for f=0 to F-1
 SetConstants();
 Render();
 SwapRenderTagetAndPingpong();
end
```

4 Computational Experiment

The AMD 64 at 3.2 GHz and nVidia GeForce 6800GT were used for tests. The instruction set of AMD 64 contains the SSE2 instructions for the SIMD optimalization. The SSE2 optimalization was switched on during the tests.

I compared both the Fourier filtering and integration part of the FBP algorithm in the CPU version with the GPU version, and I measured the computation times of these programs for a variety of sizes of the matrix g.

The column GPU contains the time of the GPU implementation. The column CPU contains the time of the CPU implementation.

The first part of Table 3 contains the computation time of the CPU and GPU implementations. The second part of the Table 3 contains the speedups of the individual implementations.

	CPU			GPU		
Texture	FFT	BP	Total	FFT	BP	Total
128	3	28	31	6	36	100
256	6	209	215	6	111	152
512	120	1984	2106	14	708	808
1024	710	24369	25078	27	5994	6256

	CPU/GPU		
Texture	FFT	BP	Total
128	0,49	0,78	0,31
256	0,97	1,88	1,42
512	8,52	2,80	2,61
1024	26,67	4,07	4,01

Table 3: Computation time
of GPU and CPU Filtered BackProjection

5 Conclusion

We can see in the Table 3 that the GPU computing is not always efficient, especially if we have a low size of the matrix g. But the GPU implementation of the fast Fourier filtering is very successful. On the other hand we need to process large data to be able to expect a certain speedup.

The reasons why to use the graphics card for mathematical computations are the high performance level of the graphics card, as well as the price of the graphics card.

6 Acknowledgment

V. V. Vlček would like to thanks I. Hanák for many helpful discussions concerning the GPU realization of this problem.

References:

[1] P. Toft, *The Radon transform, theory and implementation*, Ph.D. dissertation, Dept. Math. Modelling, Technical Univ. of Denmark, Kongens Lyngby, Denmark, 1996. pp. 95–113. http://pto.linux.dk/PhD

[2] *Microsoft Developer Network*, Microsoft, ch. DirectX graphics. http://msdn.microsoft.com

[3] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, *Numerical Recipes in C.* Cambridge: Cambridge University Press, 1992, ch. 12

[4] V. V. Vlček, Efficient use of the graphics card for mathematical computation, *3rd international mathematic workshop*, Brno, 2004. pp. 109-110

[5] V. V. Vlček, Computation of Inverse Radon Transform on Graphics Cards, *Proceedings of the International Conference on Signal Processing*, Istanbul 2004, pp. 149-151

[6] Jason L. Mitchell, Marwan Y. Ansari, and Evan Hart, Advanced Image Processing with DirectX 9 Pixel Shaders, *ShaderX2: Shader Programming Tips &Tricks with DirectX 9,* Wordware Publishing, Inc. 2004, pp. 457-463