Computation of Inverse Radon Transform on Graphics Cards

Vítězslav V. Vlček

Abstract**—**New graphics cards have a high performance that we can use for mathematical computations. I explain how we can use the graphics performance for mathematical computation in this brief report. I focused on the implementation of the inverse Radon transform by method of the filtered backprojection. The result of this article is the following: The GPU implementation of the filtered backprojection can be 0.5–29 times faster than the standard CPU version.

Keywords**—**Filtered backprojection, graphics processing unit, inverse Radon transform, parallel computation.

I. Introduction

RECENTLY new graphics cards were introduced, which have 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. This is the reason why I try to use the graphics card for mathematical computation, especially for inverse Radon transform. I decided to create the GPU implementation of the filtered backprojection because the GPU implementation of adding of two matrices can be 6 times faster than the CPU implementation [4].

We have two basic 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.

I chose the Filtered Backprojection (FBP) for the GPU realization, because the FBP is used in most scanners today. I would like to compare the computation time of the GPU and CPU program.

II. GPU Programming

A. Programming Languages for GPU

The programming languages for the GPU are divided into two platforms: Microsoft Windows and Linux. The world of MS Windows consists of the high-level shader language (HLSL) in particular, and a system for programming graphics hardware in a C-like language (Cg) is well known in the Linux world. It is necessary to say that the HLSL and the Cg are semantically 99% compatible. The main difference between HLSL and Cg is in the GPU interface. 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.

B. Data and GPU

The main use of the graphics card with the GPU was for game playing. Microsoft has been developing Direct3D (D3D) for game developers to facilitate their game development. The mathematics data, e.g. matrices, are represented by textures. So the texture is a bit-map which is mapped to a square (polygon). A point of the texture consists of four entries (alfa, red, green and blue color) which are the floating point numbers. See Figure 1 for the matrix-texture mapping. The GPU is a vector processor which can process the alfa, red, green and blue entry of the pixel in parallel. The GPU uses the 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 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.

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

Manuscript received November 9, 2004. This work was supported in part by the grant No. 1354/2004/G6 of the FRVS of the Czech Republic.

V. V. Vlček is with the University of West Bohemia, Pilsen, Univerzitní 22, the Czech Republic (corresponding author to provide phone: 420-377-632660; fax: 420-377-632660; e-mail: vsoft@kma.zcu.cz).

usually uses the D3D in Microsoft Windows, or OpenGL in Linux.

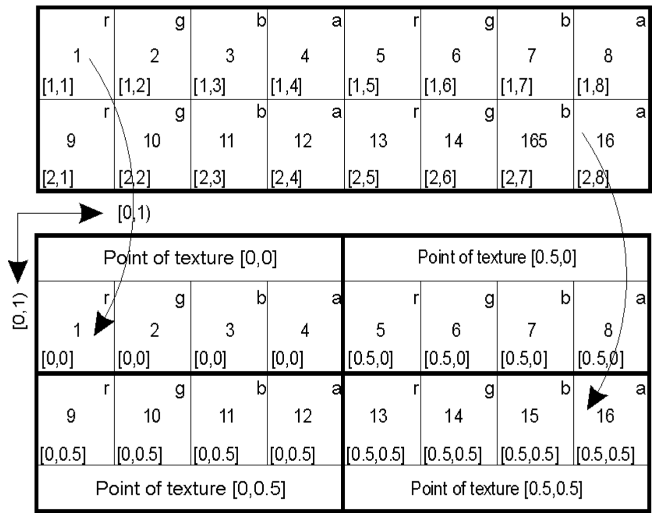

FIGURE 1: Mapping of Matrix Entries (upper) to Texture Points (down).

III. 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

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

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].

```
//1D filtering part of the FBP
$frg$=FFT($rg$) // 1D Fast Fourier Transform
$ifrg$=IFFT($frg \cdot |\upsilon|$) // 1D Inverse FFT
//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
      $pos$=($m$-$M$/2) cos($f \cdot delta\_f$) - ($n$-$M$/2) sin($f \cdot delta\_f$)
      $sum$=$sum$ + $delta\_t \cdot ifrg(f, pos)$
    end
    $g(m,n)$=$sum$
  end
end
```

The algorithm of the FBP was written in a pseudo-code. 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 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 filtering part of the FBP
$g(m,n)$=0 //clear all output matrix
$frg$=FFT($rg$) // 1D Fast Fourier Transform
$ifrg$=IFFT($frg \cdot |\upsilon|$) // 1D Inverse FFT
//the integration part of the FBP – optimized for GPU
for $f$=0 to $F$-1
  for $m$=0 to $M$-1
    for $n$=0 to $M$-1
      $pos$=($m$-$M$/2) cos($f \cdot delta\_f$) - ($n$-$M$/2) sin($f \cdot delta\_f$)
      $g(m, n)$=$g(m, n)$ + $delta\_t \cdot ifrg(f, pos)$
    end
  end
end
```

The previous algorithm is written in a pseudo-code. The variables rg, frg, $ifrg$ and g are matrices (the textures in the D3D). The constants M, $delta\_f$, $delta\_t$ depend on the discrete Radon transform of the source signal $g(m, n)$.

IV. GPU Implementation

The FBP can be divided into two parts. The first part is the data filtering by the 1D FFT and 1D IFFT. I used the CPU version of the FFT [3]. The second part of the FBP is the integration part. I transformed the CPU version of the integration part into the GPU version.

V. Computational Experiment

The Pentium 4 at 2.8 GHz and ATI Radeon 9800 were used. The instruction set of Pentium 4 contains the SSE2 instructions for the SIMD optimalization.

I compared the 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 rg.

TABLE I
COMPUTATION TIME OF CPU AND GPU FILTERED BACKPROJECTION

Size of the matrix rg	GPU time (ms)	CPU time (ms)	SSE2 time (ms)	*CPU time (ms)	*SSE2 time (ms)
128	344	179	125	195	133
256	296	3125	2750	1469	1038
512	1265	29110	26406	12125	8515
1024	8531	249828	230171	103453	72094
Size of the matrix rg	CPU/ GPU	*CPU/ GPU	*SSE2/ GPU	CPU/ SSE2	*CPU/ *SSE2
128	0.5	0.6	0.4	1.4	1.5
256	10.6	5.0	3.5	1.1	1.4
512	23.0	9.6	6.7	1.1	1.4
1024	29.3	12.1	8.5	1.1	1.4

The column GPU contains the time of the GPU implementation. The column CPU, SSE2 contains the time of the CPU implementation. The columns *CPU and *SSE2 contain the time of CPU optimized FBP algorithm.

The first part of Table I contains the computation time of the CPU and GPU implementations. The second part of the Table I contains the speedups of the individual implementations for a variety of sizes of the matrix *rg* and the implementations.

VI. Conclusion

We can see in the Table I that the GPU computing is not always efficient, especially if we have a low size of the matrix *rg*. But if we process large data then we can expect a certain speedup. Further problems are the limit of the program length, the impossibility of dynamic loops, the size of the video memory, and necessity of the graphics framework.

The reasons why to use the graphics card for mathematical computation are the high performance of the graphics card, the price of the graphics card (which is significantly lower than the cost of an equivalent CPU cluster), and the further advantage is the extra memory on graphics cards.

A further improvement of this method is the transformation of the first part of the FBP – the FFT data filtering.

Acknowledgment

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

References

[1] P. Toft, "The Radon transform, theory and implementation (Thesis or Dissertation style)," Ph.D. dissertation, Dept. Math. Modelling, Technical Univ. of Denmark, Kongens Lyngby, Denmark, 1996. [Online]. pp. 95–113. Available: http://pto.linux.dk/PhD

[2] *Microsoft Developer Network*, Microsoft, [Online]. ch. DirectX graphics. Available: 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 (Conference Paper Style—Accepted for publication)," in 3rd international mathematic workshop, Brno, 2004.