EFFICIENT USE OF THE GRAPHICS CARD FOR MATHEMATICAL COMPUTATIONS *

Vítězslav Vít VLČEK

University of West Bohemia in Pilsen
Department of Mathematics
vsoft@kma.zcu.cz

1. Introduction

Recently the new graphics cards were introduced which have the high performance processor. This processor is called graphics processing unit (GPU) and it can process a lot of graphics data at a time. The performance of the GPU can be more than common CPU (central processor unit) in some cases. This is the reason why I would like to use the graphics card for mathematical computation especially as far as the vector or matrix addition.

2. Programming Languages and GPU

At first I would like to introduce the programming languages of the GPU. The programming languages for the GPU are divided into two worlds: 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. The HLSL is a part of Microsoft Direct X 9.0 (D3D) while the Cg can use the OpenGL (OGL) or the D3D.

Of course, there are existed further languages for the GPU like the Brook (a stream language), the ShLib, etc. These types of the languges facilitate the graphics framework but they have more disadvantages and they are too restrictive. We can use the assembly language for the pixel shader but it is too difficult.

The rest of this section concerns types of the graphics cards. We cannot use any graphics card. The graphics card has to contain the programmable pixel shader (PS). The programmable pixel shader is a processor which processes the graphics data. Unfortunately, every graphics card does not have its own PS because of the high price. The development of the graphics card is too fast hence it has been existing a few version of the PS yet.

*This work was supported by grant No. 1354/2004/G6 of the FRVS of the Czech Republic.

The survey shows the chronological order of the PS version.

- PS 1.0 — The hardware for this version was never released.
- PS 1.1 — The company nVidia released the GeForce card, which started the wave of shader programming.
- PS 1.2–1.4 — This versions are the improvement of previous versions. (speed, new instructions, etc.)
- PS 2.0 — The new features of this version are the floating point textures, the branching of the code, etc.
- PS 3.0 — This version will contain the dynamic code loops, etc.

The further table shows the properties of the some graphics cards.

Manufacturer	**Chip**	**PS version**
nVidia	GeForce2	3.0 (sw emulation)
nVidia	GeForce3–Ti	1.3
nVidia	GeForceFX	2.0
ATI	Radeon 8200/9100/9200	1.4
ATI	Radeon 9600/9800	2.0

Table: The properties of the graphics cards

3. Representation of Matrix in GPU

I focus on the graphics card with the PS 2.0 because the PS 2.0 provides the floating point data processing. The previous versions of the PS only facalitated the 8-bit data processing and it is unsuitable for mathematical computing.

3.1. Textures and Matrices

The main use of the D3D is for the game developing (i.e., the modelling of the 3D world, ...). The building stone of the D3D is a triangle. Two particular triagles do a square. The square can be covered with a texture. So the texture is a bit map which is mapped to the square (polygon). The texture data depends on the texture (or pixel) format (TF). The D3D defines some texture formats (see table). There are the fixed point and the IEEE floating point texture formats. The formats D3DFMT_R32F and D3DFMT_A32B32G32R32F are suitable for mathematical computations because of the floating point structure. The GPU is a vector processor which can parallel process the alfa, red, green, blue data hence the D3DFMT_A32B32G32R32F texture format is the best for us purposes.

The texture is actually a matrix in the mathematical sense but there are some specialities. A poin of the texture consists of four entries (alfa, red, green and blue color) which are the floating numbers (see figure). These four numbers are processed in GPU at a time.

D3D TF	Description
D3DFMT_R8G8B8	fixed point - 24-bit RGB pixel format with 8 bits per channel
D3DFMT_A8R8G8B8	fixed point - 32-bit ARGB pixel format with alpha, using 8 bits per channel
D3DFMT_X8R8G8B8	fixed point - 32-bit RGB pixel format, where 8 bits are reserved for each color
D3DFMT_R32F	IEEE floating point - 32-bit float format using 32 bits for the red channel
D3DFMT_A32B32G32R32F	IEEE floating point - 128-bit float format using 32 bits for the each channel (alpha, blue, green, red)

Table: The Direct X 9.0 texture formats

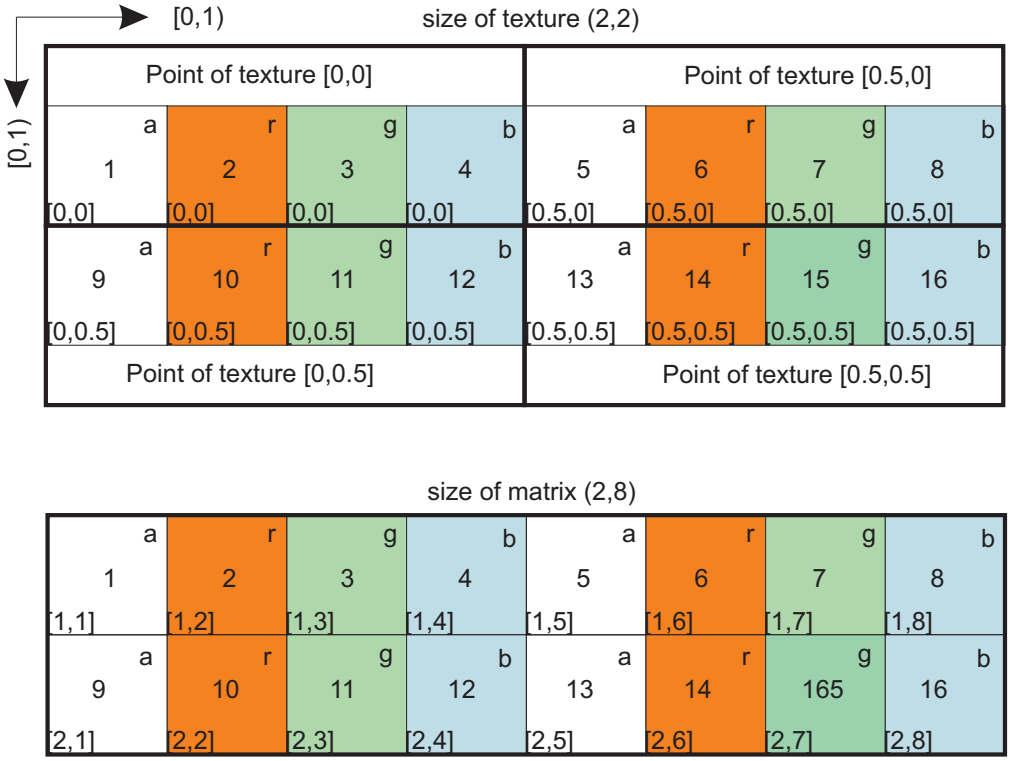

Figure: The mapping of the texture points to the matrix entries

3.2. Pixel Shader Program

I would like to focus on the pixel shader in this section. The PS is a processor to which the program and the data (textures) are incomming. The output of the PS is the texture (the render target in D3D) or the textures, it depends on the features of the graphics card, which contains the computed values. The computed values are in the same format as the input textures. The PS program can only read the bounded count of the texture points (aprox. 12 it depends on the card) and only has to write the one texture point (both in one step). There is further restriction for writing of

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 restriction. The PS can only do the static loops which the compiler unrolls. The PS program cannot access to the output data. For an example of the pixel shader program for an adding of two matrices see appendix.

3.3. Graphics Framework for Mathematics Computation

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. This program (the graphics framework) uses the interface of Microsoft Direct X in case of the HLSL.

The following list shows the points which we have had to do before the PS starts the PS program.

1. Initialize Direct 3D.
2. Create a square for the texture mapping.
3. Create textures and fill in data.
4. Create render target texture.
5. Load the PS program into graphics card.
6. Load the data into graphics card.
7. Start the rendering of the scene. It means that the PS runs the PS program on each pixel of the render target.
8. Read the render target texture for the computed data.

As we can see it is quite difficult.

4. Computation Experiments

I would like to show the test task where I would like to illustrate that the GPU computation is more fast than the CPU computation in this section. I chose the following test task $\mathbf{C} = \mathbf{A} + \mathbf{B}$ (the adding of two big matrices). I created the purely CPU program and the program for GPU. The speed results shows following table.

N	GPU-time	CPU-time	Speedup
1	62/246	19/44	0.2
2	63/243	39/64	0.3
10	86/263	195/221	0.8
100	342/523	1872/1898	3.6
1000	2888/3070	18583/18608	6.1 (8.2)*
10000	28336/28519	185440/185466	6.5

Table: Comparision of the GPU-CPU computing

The size of matrices was 1024×4096. The column **N** means the number of repeating of adding. The **GPU-time** and **CPU-time** contain two values. The first of them is the time of adding in milliseconds and the second value is total running time of the program in milliseconds. The column **Speedup** is the speedup ration=CPU total time/GPU total time. The value with * mean speedup ration of the matrix entry multiplication – if we replace + with * ($\mathbf{C} = \mathbf{A}. * \mathbf{B}$ in the Matlab sense).

I used CPU Athlon XP 1800+ and Radeon 9200.

5. Conclusion

We can see on the comparision table that the GPU computing is not always efficient especially if we have the low count of the data operations or the small data. But if we process a lot of data then we can expect certain speedup. The further problem is significantly limit of the program complication and necessity of the graphics framework.

The reasons why to use the graphics card for mathematical computation are the price of graphics card which is significantly lower than the cost of equivalent CPU, the further advantage is the extra memory on graphics card.

I would like to implement the matrix-matrix, matrix-vector multiplication as the further improvement I expect some speedup here, too.

References

[1] Microsoft: *The MSDN Library.* Microsoft, 2001.

[2] Luna, F.: *Introduction to 3D Game Programming with DirectX 9.0.* Wordware Publishing, ISBN 1556229135, 2003.

Appendix — Pixel Shader Program

The pixel shader program for adding two matrices.

```
texture textA;
texture textB;
float   gMul;
sampler SamplerA = sampler_state
{
    Texture   = <textA>;
};
sampler SamplerB = sampler_state
{
    Texture   = <textB>;
};
struct Vs_Input
{
```

```
   float3 vertexPos  : POSITION;
   float2 texture0   : TEXCOORD0;
   float2 texture1   : TEXCOORD1;
};
struct Vs_Output
{
   float4 vertexPos  : POSITION;
   float2 texture0   : TEXCOORD0;
   float2 texture1   : TEXCOORD1;
};
struct Ps_Output
{
float4 color      : COLOR;
};
Vs_Output VS(Vs_Input IN)
{
   Vs_Output vs_out;

   vs_out.vertexPos = float4(IN.vertexPos, 1);
   vs_out.texture0 = IN.texture0;
   vs_out.texture1 = IN.texture1;

   return vs_out;
}
Ps_Output PS(Vs_Output IN)
{
   Ps_Output ps_out;

   ps_out.color=tex2D(SamplerA, IN.texture0)+
    gMul*tex2D(SamplerB, IN.texture1);
   return ps_out;
}
technique Soucet
{
   pass Pass0
   {
      Lighting = FALSE;

      VertexShader = compile vs_1_1 VS();
      PixelShader  = compile ps_2_0 PS();
   }
}
```