Dense Matrix Algebra on the GPU

Ádám Moravánszky

NovodeX AG

adam.moravanszky@novodex.com

1. Introduction

Perhaps the most important innovation of the latest generation of programmable graphics processors (GPUs) is their capability to work with floating point color data. Previous generations of GPUs have worked with up to a byte of integer data per color channel. Developers working on graphics engines with advanced lighting effects often complained about banding artifacts, even in truecolor video modes, because multiplicative effects quickly made the round off error caused by the limited precision noticeable. The advent of GPUs that represent each color channel with a 32-bit floating-point value has thus been widely celebrated in the real time graphics community.

More importantly, while eight-bit color channel precision is often adequate, the dynamic range is quite limited. Floating-point color buffers make it possible to work with brightness values well beyond the maximum value that can be represented in the final image. Though the dynamic range of output device stays the same, intermediate values during a computation are no longer clamped to this range. This way a much more realistic simulation of lighting is possible, resulting in vibrant images.

The topic of this article is made possible by the emergence of floating point color support as well, but we will not be dealing with either of the often-cited advantages of floating point buffers described above. In fact, we will not be rendering images in the conventional sense at all. Instead, we look at the GPU as a powerful vector coprocessor to the CPU. We use it to solve two common problems in scientific computing: Solving systems of linear equations, and linear complementarity problems. Both of these problems come up in dynamics simulation, which is a field drawing increasing interest from the game developer community.

By implementing these algorithms on the GPU, we hope to achieve a performance gain, or at least free up CPU resources, which can then be better spent running algorithms that are not vectorizable. Because the GPU usually has its hands full rendering graphics in a computer game, and because GPUs with floating point color support are anything but widespread, the results of this article are initially primarily of theoretical interest for the game community. However, if we can show convincing performance figures that make such application of GPUs desirable, we may soon find these applications becoming practical and widespread. And if GPU performance continues to grow at its present rate, we may eventually see researchers and engineers abandoning expensive supercomputers for clusters of GPU equipped PCs.

1.1 Previous Work

The fundamental concept of linear algebra is the matrix. Matrices are used in simulation in order to describe a linear relationship in a concise way. A significant amount of research has gone into working with large dense matrices. BLAS (Basic Linear Algebra Subprograms) [2,7] has emerged as the standard interface to linear algebra libraries. Freely available implementations of BLAS include ATLAS (Automatically Tuned Linear Algebra Software) [9], a linear algebra library that includes Intel SSE2 and AMD 3DNow optimized matrix multiply kernels. These fast kernels, combined with ATLAS' cache friendly memory access pattern achieved by special ordering of the input data make it one of the fastest dense matrix libraries available on the PC platform. In [6], Larsen and McAllister first investigated using GPUs for linear algebra. At the time of this publication, floating point pixel processing was not yet available, so their results were not practical for real world problems. The papers [1,5], made available after this paper was initially submitted, tackle the representation of sparse matrices on the GPU.

While ATLAS provides a selection of higher-level linear algebra operations, such as solving linear systems, the code of ATLAS is a high performance matrix multiply kernel, which is then leveraged by the high level operations. We follow the same principle in our GPU matrix library: We implement a few basic matrix operations using shaders, including matrix multiply, and then use these as building blocks to solve the higher level problems. While we have not written a write a full GPU BLAS implementation due to time constraints, we show how to implement all the basic components necessary for this goal.

1.2 Implementation

Our implementation consists of a matrix class that carries out all the core arithmetic operations. It interfaces with the GPU using the DirectX 9 Graphics SDK. The user interface is a script interpreter which parses matrix operation instructions out of a text stream, manages matrix variable names, reads and writes matrix variable data to file, and passes operations for execution to the matrix class. We only discuss the matrix class below, as well as two examples of its use.

2. Matrix Textures

If the GPU is to perform large matrix multiplication for us, the first thing we need to do is represent the matrix data in a format that is accessible by the GPU. GPUs can in principle work on two basic types of data: geometry and texture maps. Textured geometry is preferable because of the more compact representation when compared with highly tessellated geometry with vertex colors. Also, unlike geometry, textures can also be output by the GPU in the form of render target surfaces. If we store a matrix as a texture, and then perform a matrix operation such as matrix addition by rendering two textures with additive blending into a third render target surface, the storage format of the resulting matrix can be identical to the input format. This is a desirable property because this way we can immediately reuse the resulting texture as an input to another operation without having to perform format conversion.

We would like our library to work with matrices of real numbers, because this domain is the most generally useful for simulation problems, and dynamics simulation in particular. Integers would be too restrictive while complex numbers are usually not required. Note that the system we present could be extended to handle complex numbers should this be the case. Real numbers are most efficiently approximated on computers using floating-point numbers of various precisions. Unfortunately GPUs still only support single precision floating point, and future support for double or higher precision is unlikely, as this sort of precision is not thought to be needed for graphics applications. Nonetheless, single precision floating point is adequate for many applications.

2.1 Storage Format

There are several ways in which the elements of a matrix can be mapped to the pixels of a texture image. Perhaps the most obvious approach would be to take a luminance (one channel per pixel) image and fill it with the matrix data using a direct mapping of elements to pixels, in either row or column major format. The disadvantage is, of course, that GPUs are optimized to process RGBA pixels, and thus have four way SIMD for executing pixel shaders. A luminance texture would only use a quarter of the available bandwidth.

Instead, we pack four adjacent matrix elements into a single pixel's RGBA channels. The simplest possibilities are to either pack rows or columns of four. While this packing does make square matrices into 4:1 aspect rectangular textures, it makes the writing of the pixel shaders for multiplication quite straightforward. Other schemes, such as packing 2x2 rectangular submatrices into each pixel complicates the pixel shaders for doing matrix multiplication, and offers no clear advantage. It is interesting to note that CPU linear algebra packages like ATLAS primarily get their speed boost by storing the matrices in a convoluted but very cache friendly way. Data locality is an important key to performance. Unfortunately, in contrast to CPU programming, we have only a relatively high level control of the GPU. In particular, the order in which pixels get processed is an undocumented implementation detail. Usually, the GPU automatically stores textures in a swizzled form to improve cache coherence. It may be interesting to investigate if more exotic storage formats can boost performance, but one would have to do quite a bit of experimentation, without necessarily being able to generalize the results to different GPUs.

The final question regarding data storage is whether there is any difference between packing rows or columns of four into a pixel. One important difference comes up when we consider doing vector operations. It is important that pixels be created along the length of a vector, instead of across. In the latter case a vector would only fill one color channel and leave three empty. In this implementation we arbitrarily decided to go with storing CPU matrices in row major format, and working with column vectors. Thus we put 4×1 sub-column vectors into each pixel. The width of a texture that corresponds to an $n\times m$ matrix is thus m, while the height is $\lceil \frac{n}{4} \rceil$.

To create a matrix texture from some source data, we create an appropriately sized render target surface using the `D3DFMT_A32B32G32R32F` floating point pixel format. We don't need any mipmapping, in fact we render with point sampling to prevent texture filtering from falsifying our computations.

Creating a render target texture is technically only necessary if we want the matrix to serve as a destination for matrix operations; in our application we choose not to keep track of this distinction, and treat all matrices equally for the sake of simplicity.

Unfortunately in DirectX9 it is not possible to lock render target surfaces, so we need to create an identically formatted temporary texture in the `SYSTEMMEM` pool. This texture's surface is then locked and the matrix data is read into it. Finally we use the DirectX method `UpdateTexture()` to copy the temporary texture into our render target texture.

Reading back from the matrix texture happens in the same way. This time the method `GetRenderTargetData()` is used to copy from the matrix texture to the temporary texture.

3. Matrix Operations

After reading in the data, we are ready to perform some matrix operations. We start by implementing three basic operations – matrix assignment, addition and multiplication. Later we will add some others as required by our higher level algorithms. Note that some operations are not strictly necessary and could be expressed using others. For example, assignment could be emulated by adding a zero matrix to the source matrix. Still, writing special case code when optimizations are possible is a good idea.

3.1 Assignment

Matrix assignment is the most elementary operation, so we cover it first to introduce some details in our code:

```
void Matrix::copy(Matrix & other)   {
```

Note that while the reference rasterizer works fine with the render target surface being the same as one of the source textures, this case is not officially supported by Direct3D, and should be avoided. In the case of assignment it is obviously a null operation to assign a matrix to itself, so we can early out in this case.

```
if (this == &other)     return;
```

If the destination texture is not the same size as the source texture, it needs to be resized. We resize a texture by releasing it and creating a new one of the correct size.

```
resize(other.getNRows(), other.getNCols());
```

If one of the dimensions of the matrix is zero, there is nothing to do:

```
if (nRows * nCols == 0)     return;
```

Next, we set the destination texture as the render target, begin the scene, assign vertex and pixel shaders, and assign the source texture to the 0th sampler. For this simple operation we do not really need shader support, and could do the same operation with the fixed function pipeline and texture combiners. On the other hand any hardware that supports floating point pixel formats will most likely have shader support as well, so we might as well use them. We omit DirectX error handling in the cited code for clarity.

```
d3dDevice->SetRenderTarget(0,mathSurface);
d3dDevice->BeginScene();
d3dDevice->SetVertexShader( vertexShaders[VS_SINGLE_TEX_QUAD] );
d3dDevice->SetPixelShader( pixelShaders[PS_COPY] );
d3dDevice->SetTexture(0,other.mathTexture);
```

Next, we render a single quadrilateral polygon that exactly covers the destination texture, by using a triangle fan with four vertices. This is what our vertex buffer contains:

```
MathVertex quad[4]= {
    //  x        y
    { -1.0f, -1.0f},
    { +1.0f, -1.0f},
    { +1.0f, +1.0f},
    { -1.0f, +1.0f}};
```

We have 2D clip space coordinates for each vertex. Because we won't be rendering 3D shapes, and because texture coordinates can be trivially generated in the vertex shader from this basic data, it is all we need. We place this data into a managed pool vertex buffer and do not worry about it anymore. It is used for all the matrix operations except multiplication.

The actual rendering code looks like this:

```
d3dDevice->SetStreamSource( 0, quadVertexBuffer, 0, sizeof(MathVertex));
float TexcoordBiasW = (1.0f/cols2TextureWidth(nCols))  * 0.5f;
float TexcoordBiasH = (1.0f/rows2TextureHeight(nRows)) * 0.5f;
float consts[4 * 2] = {
    0.5, -0.5, 0.5, 1,
    0.5+ TexcoordBiasW,  0.5 + TexcoordBiasH, 0  , 0  };
d3dDevice->SetVertexShaderConstantF(0, consts, 2);
d3dDevice->DrawPrimitive( D3DPT_TRIANGLEFAN, 0, 2 );
d3dDevice->EndScene();
}
```

The function of the texture coordinate bias values that get passed to the vertex shader is to line up the destination pixels with the source texel centers by shifting the texture coordinates by ½ texel. If we were to omit this, at each pixel the texture would be sampled half way between texels, making it effectively random which of the four neighboring texels the point sampling would pick.

`cols2TextureWidth()` and `rows2TextureHeight()` simply map matrix dimensions to texture dimensions using the formula mentioned previously:

```
inline unsigned roundUpDivide(unsigned a, unsigned b) { return (a + b-1) / b; }
inline unsigned rows2TextureHeight(unsigned rows) { return roundUpDivide(rows,4); }
inline unsigned cols2TextureWidth (unsigned cols) { return cols; }
```

The vertex shader we use, “SINGLE_TEX_QUAD”, is shown below:

```
// c0 = [   0.5, -0.5, 0.5, 1]
// c1 = [   0.5+ TexcoordBiasW,  0.5 + TexcoordBiasH, 0  , 0]
vs_1_1
dcl_position v0
mov oPos, v0
mov oPos.zw, c0.zw
mov r0, c1
mad oT0.xy, v0.xy, c0.xy, r0.xy
```

We basically emit the vertices that we put in the vertex buffer in clip space after assigning some constant values to the z and w coordinates. The texture coordinates are computed from the vertex position in a single instruction, which involves the flipping of the vertical axis and the application of the bias constants described above.

Finally, the pixel shader is shown below. It serves to simply copy the input texture to the destination surface:

```
//PS_COPY   out = tex0
ps_2_0
dcl_2d s0
dcl t0
texld r0, t0, s0
mov oC0, r0
```

We have tried using HLSL to produce these shaders, and several of them were prototyped that way, but the DirectX shader compiler failed to produce efficient code for the more involved matrix multiply cases, so we decided to stay with hand coded assembly for this project. The use of pixel shader 2.0 or greater is necessary in the case of this simple shader not because of any special instructions or even the number of instructions, but because lower pixel shader versions automatically clamp their final result to [0,1]. We would like to use the entire floating-point range.

3.2 Addition

Addition is very similar to assignment. Because we have the limitation that the destination texture may not be the same as either of the source textures, we need to code both a general add and an accumulate (‘+=’) operation. We only cover the binary version here, because the accumulate version is the same as the above assignment with additive blending with the existing render target turned on.

```
void Matrix::add(Matrix & a, Matrix & b)  {
if (a.nRows != b.nRows || a.nCols != b.nCols)
    throw "matrix dimensions don’t agree";

if      (this == &a)    {   add(b); return;     }
else if (this == &b)    {   add(a); return;     }

resize(a.nRows, a.nCols);

if (a.nRows * a.nCols == 0) return;

d3dDevice->SetRenderTarget(0,mathSurface);
d3dDevice->BeginScene();
d3dDevice->SetVertexShader( vertexShaders[VS_SINGLE_TEX_QUAD] );
d3dDevice->SetPixelShader( pixelShaders[PS_ADD] );
d3dDevice->SetTexture(0,a.mathTexture);
d3dDevice->SetTexture(1,b.mathTexture);
d3dDevice->SetStreamSource( 0, quadVertexBuffer, 0, sizeof(MathVertex) );

float TexcoordBiasW = (1.0f/cols2TextureWidth(nCols))  * 0.5f;
float TexcoordBiasH = (1.0f/rows2TextureHeight(nRows)) * 0.5f;

float consts[4 * 2] = {
    0.5, -0.5, 0.5, 1,
    0.5+ TexcoordBiasW,  0.5 + TexcoordBiasH, 0  , 0  };

d3dDevice->SetVertexShaderConstantF(0, consts, 2);
d3dDevice->DrawPrimitive( D3DPT_TRIANGLEFAN, 0, 2 );
d3dDevice->EndScene();
}
```

There are only a few places where the above differs from the assignment code. First, we need to check if the dimensions of the two source textures match, otherwise the addition operation is mathematically undefined. We also check if one of the source operands is the same as the destination, and call the special case accumulate code in this case. The second texture is also assigned to the second texture sampler. We use the same vertex shader as before.

The pixel shader is a different one, but not much more complicated; it simply performs additive blending of the two source textures:

```
//PS_ADD    out = tex0 + tex1
ps_2_0
dcl_2d s0
dcl_2d s1
dcl t0
texld r0, t0, s0
texld r1, t0, s1
add r0, r0, r1
mov oC0, r0
```

3.3 Multiplication

Writing a general matrix multiply is a bit more challenging because unlike addition, it doesn't reduce to mere image blending. Figure 1 shows the schematic for our matrix multiply procedure.

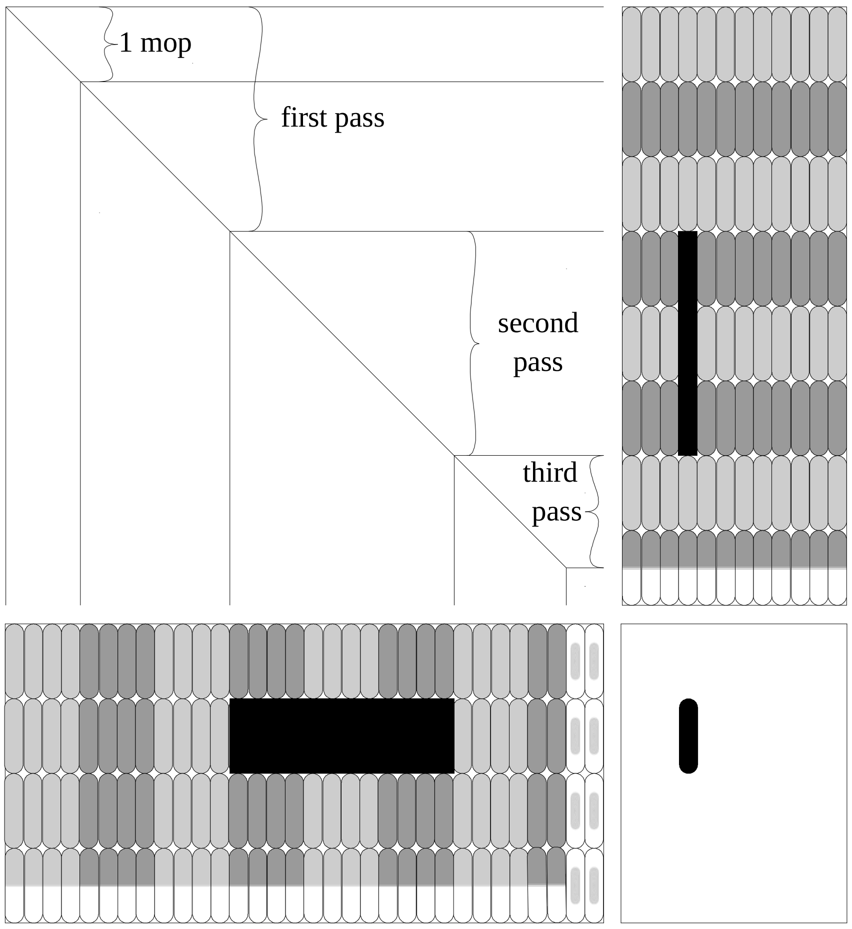

Figure 1: Schematic of matrix multiply

The texture corresponding to the left operand matrix A is shown on the left side. The texture of the right side operand matrix B is at the top right. C, the result matrix, is shown at the bottom right. C = A B should hold after the operation completes.

By the definition of matrix multiplication, the number of columns in A have to equal the number of rows in B. We call this range of q numbers the *inner* dimension. Finally, the number of rows in A is equal to the number of rows in C and the number of columns in B is equal to the number of columns in C. We call these the *outer* dimensions.

In our Figure 1 example matrix A is 14×30 and matrix B is 30×12. The 4×1 submatrices stored by the textures in a single pixel are shown as ovals. Because the matrices' heights are not exactly divisible by four, the last two elements of the last row of pixels are unused, indicated by their white color. Note that the texture representing A is only 30 pixels wide. The last two columns of white ovals with gray markings represent samples read in by the pixel shader outside of the [0,1] texture coordinate range: these virtual texels need to read as zero. They are necessary so our pixel shader can always work with blocks of 4 texels, even if the input matrix sizes are not exact multiples of four.

As any pixel shader, the matrix multiply code has to emit a single (partially computed) pixel from the pixel shader. Each pixel stores four values, each of which is a dot product between a row vector of A and a column vector of B. Both of these vectors have q elements, where q is 30 in our example. Thus, at each pixel we need to perform four of these dot products, which is the same as a 4×q matrix-vector multiplication. Because q may be quite large, our GPU may not be able to sample all the $1\frac{1}{4}$ q texels necessary in one pass due to pixel shader instruction count limits. Thus we need to decompose this operation into a set of smaller operations depending on our instruction count limits.

Our atomic pixel shader operation is a 4×4 matrix-vector multiplication, where the 4×4 matrix is fetched from A and the 4×1 vector from B. We refer to this atomic multiply as a *MOP* for 'matrix operation'. We need to perform $\lceil \frac{q}{4} \rceil$ of these MOPs per pixel, and accumulate the results, in order to obtain the final result for an output pixel. We pack as many of these MOPs into our pixel shader as possible.

In our example we assume a hypothetical pixel shader that can perform no more than three of these MOPs in a single pass. In general we define the macro `numMOpsPerFragment` as the number of MOPs that can fit into a pixel shader. For ps 2.0, we managed to fit six of them.

If the hypothetical example shader can do three MOPs per pass, and we need a total of $\lceil \frac{30}{4} \rceil = 8$ MOPs for the final result, we need to perform $\lceil \frac{8}{3} \rceil = 3$ additive passes, as indicated in the figure. Ps 2.0 would only need two passes.

As an example, we have highlighted a pixel in the destination texture. The pixel shader that emits this pixel as part of the second additive pass samples the darkened 15 texels from A and B.

Of course the outer dimensions we don't have to worry about, they are taken care of by the inherent parallel processing of the GPU in the form of adjacent pixels, just like in the assignment and addition shaders. Now that we have covered the theory, we present the implementation:

```
void Matrix::multiply(Matrix & a, Matrix & b)   {
```

As usual we need to check if we're trying to render a texture onto itself. Here we do not have a backup plan so we simply report an error:

```
if (this == &a || this == &b)
    throw "can’t operate inplace -- not supported by D3D.";
```

If the matrix dimensions do not agree, matrix multiplication is undefined:

```
if (a.nCols != b.nRows)
    throw "matrix dimensions don’t agree";

resize(a.nRows, b.nCols);
if (nRows * nCols == 0)     return;
```

First we compute a few constants depending on the input sizes and the number of instructions permitted in the pixel shader. We will render `numQuads` quads aligned with the destination surface, with additive shading. We compute `numQuads` with the formulas given above.

```
const unsigned numQuads =
roundUpDivide(rows2TextureHeight(b.nRows),numMOpsPerFragment);
```

Like we did for assignment and addition, we compute texture coordinate bias values to ensure that texels are sampled at their centers. Here we have two input textures so we need to do this twice:

```
const float TexcoordBiasW = (1.0f/cols2TextureWidth(nCols))  * 0.5f;
const float TexcoordBiasH = (1.0f/rows2TextureHeight(nRows)) * 0.5f;

const float TexcoordBiasAW = (1.0f/cols2TextureWidth(a.nCols))  * 0.5f;
const float TexcoordBiasBH = (1.0f/rows2TextureHeight(b.nRows)) * 0.5f;
```

A single pixel shader performs several MOPs. We supply it with texture coordinates for the first five samples corresponding to the first MOP, but only provide texture coordinate increments relative to the first five, which can be used by the pixel shader to compute the texture coordinates of the subsequent samples. `tcMOpIncrementBH` is the height of a texel in B, which the amount the pixel shader has to seek down in the texture to get to the pixel used for the next MOP.

```
const float tcPixelBH = 2 * TexcoordBiasBH;
const float tcMOpIncrementBH = tcPixelBH;
```

The second increment we need is that of the texture coordinates between the additive passes. These will be used by the vertex shader, as we will not pass explicit texture coordinates to minimize the size of our vertex buffer.

```
const float tcPassIncrementBH = numMOpsPerFragment * tcPixelBH;
```

The same constants are also computed for the other input texture:

```
const float tcPixelAW = 2 * TexcoordBiasAW;
const float tcMOpIncrementAW = 4 * tcPixelAW;
const float tcPassIncrementAW = numMOpsPerFragment * tcMOpIncrementAW;
```

The meaning of the vertex and pixel shader constants will become clear when we look at the shaders:

```
float vconsts[] = {
    0.5 + TexcoordBiasW,  0.5 + TexcoordBiasH, 0 + TexcoordBiasBH, 0,
    0 + TexcoordBiasAW, tcPixelAW + TexcoordBiasAW,
    2 * tcPixelAW + TexcoordBiasAW, 3 * tcPixelAW + TexcoordBiasAW,
    tcPassIncrementBH, tcPassIncrementAW, 0, 0
    };
float pconsts[] = {
    1 * tcMOpIncrementAW, 0, 0,0,   //2 mops
    0, 1 * tcMOpIncrementBH, 0,0,
    2 * tcMOpIncrementAW, 0, 0,0,   //3 mops
    0, 2 * tcMOpIncrementBH, 0,0,
    3 * tcMOpIncrementAW, 0, 0,0,   //4 mops
    0, 3 * tcMOpIncrementBH, 0,0,
    4 * tcMOpIncrementAW, 0, 0,0,   //5 mops
    0, 4 * tcMOpIncrementBH, 0,0,
    5 * tcMOpIncrementAW, 0, 0,0,   //6 mops
    0, 5 * tcMOpIncrementBH, 0,0,
    };

d3dDevice->SetRenderTarget(0,mathSurface);
d3dDevice->BeginScene();
d3dDevice->SetVertexDeclaration( vertexDeclaration2 );
d3dDevice->SetVertexShader( vertexShaders[VS_MULT_1] );
d3dDevice->SetPixelShader( pixelShaders[PS_MULT_0] );
d3dDevice->SetTexture(0,a.mathTexture);
d3dDevice->SetTexture(1,b.mathTexture);
d3dDevice->SetStreamSource( 0, quadsVertexBuffer, 0,TINYVERTEX_SIZE );
d3dDevice->SetVertexShaderConstantF(1, vconsts, 3);
d3dDevice->SetPixelShaderConstantF(0, pconsts, 2 * numMOpsPerFragment);
```

The vertex buffer contains a triangle list in the following format:

```
    /*  x   y    quadIndex
    { -1, -1,       0   },
    { +1, -1,       0   },
    { +1, +1,       0   },

    { -1, -1,       0   },
    { +1, +1,       0   },
    { -1, +1,       0   },

    { -1, -1,       1   },
    { +1, -1,       1   },
    { +1, +1,       1   },

    { -1, -1,       1   },
    { +1, +1,       1   },
    { -1, +1,       1   },

    ....

    { -1, -1,       99  },
    { +1, -1,       99  },
    { +1, +1,       99  },

    { -1, -1,       99  },
    { +1, +1,       99  },
    { -1, +1,       99  },
    */
```

The first two numbers are the 2D clip space coordinates of the vertex, as before. We have also added a value that is the index of the quad that the vertex belongs to in the sequence. The vertex shader will use this index value for texture coordinate generation. Because the data is so simple, we pack each vertex into a 32 bit word, and use the `D3DDECLTYPE_UBYTE4` data type. As the bytes are unsigned, we add two to the coordinates, storing –1 as 0 and 1 as 2. Finally, we render 2 `numQuads` of these triangles:

```
d3dDevice->DrawPrimitive( D3DPT_TRIANGLELIST, 0, 2 );

if (numQuads > 1)
    {
    d3dDevice->SetRenderState( D3DRS_ALPHABLENDENABLE,   TRUE );
    d3dDevice->DrawPrimitive( D3DPT_TRIANGLELIST, 6, 2 *  (numQuads - 1));
    d3dDevice->SetRenderState( D3DRS_ALPHABLENDENABLE,   FALSE );
    }
d3dDevice->EndScene();
}
```

On to the shaders. The vertex shader's job is to 'decompress' the very frugal quantity of data from the vertex buffer, and generate decent texture coordinates. Note how we have submitted all our rendering passes after the first in a single `DrawPrimitive()` call, so there is no room to perform any state changes between quads. The vertex shader has to use the quad index from the vertex buffer to tell which pass is being performed.

```
vs_1_1
dcl_position v0
def c0, 0.5, -0.5, 0.5, 1
def c4, -1, -1, 0, 0
```

Because we have encoded the input vertex coordinates as unsigned bytes, we map them to signed values by subtracting one.

```
add r3.xy, v0.xy, c4.xy //map from [0,2] to [-1, 1]
mov oPos.xy, r3.xy      //emit pos
mov oPos.zw, c0.zw
```

We start the texture coordinate generation by taking the vertex coordinate as the starting point, and inverting the vertical axis – this is the same as in the previous shaders.

```
mov r0.xy, c1.xy        //transform viewport axes to tex uv axes
mad r0.xy, r3.xy, c0.xy, r0.xy
```

Next, we need to compute the U texture coordinates for texture A, and the V texture coordinates for texture B. These depend on which pass we are in: the pass index is stored in `v0.w`. This is multiplied by `tcPassIncrementAW` and `tcPassIncrementBH` respectively, which are constants computed above, and stored in `c3`.

```
mul r1, v0.w, c3.zzxz   //can't 'mad' as it would reference 2 consts in 1 instr
add r1, r1, c1
mul r2, v0.w, c3.yyyy
add r2, r2, c2
```

Finally, we emit the five texture coordinates needed for the first MOP of the pixel shader. The V coordinates of texture A and the U coordinate of texture B are simply stretched along with the quad to map linearly over the entire destination surface. Even though it would be trivial to compute the four texture coordinates of A in the pixel shader itself, we choose to do as much of this work as possible in the vertex shader. This way we avoid bumping up against the very restrictive pixel shader instruction count limits, and the dependent texture sampling limits in particular.

```
mov oT0.x, r2.x
mov oT1.x, r2.y
mov oT2.x, r2.z
mov oT3.x, r2.w

mov oT0.y,  r0.y
mov oT1.y,  r0.y
mov oT2.y,  r0.y
mov oT3.y,  r0.y
mov oT4.x, r0.x
mov oT4.y, r1.z
```

All the matrix element arithmetic is done in the pixel shader. We have made the pixel shader generic in the sense that it is made up of as many MOPs as it is possible to execute at once on the target architecture, which is six in ps.2.0. When new hardware becomes available that supports newer pixel shader versions, getting a performance boost should only be a matter of duplicating some additional MOP blocks in the shader, and incrementing the ps version declaration. Our ps.2.0 implementation uses 30 `texld` instructions of the maximum 32, and is thus very close to optimal.

Inputs to the pixel shader are the registers `t0`...`t3`, the texture coordinates of four horizontally adjacent pixels in A, `t4`, the texture coordinate for texture B, and a large set of constants: `c0.x` holds the texture coordinate increment needed to move four pixels to the left in A, while `c1.y` has the increment needed to move one pixel down in B. `c2` and `c3` are two times `c0` and `c1`, respectively. `c4` and `c5` are the same values times three and so on. Because we have many constant registers available and few instruction slots, it is good to precompute these values.

```
    ps_2_0
    dcl t0.xyzw
    dcl t1.xyzw
    dcl t2.xyzw
    dcl t3.xyzw
    dcl t4.xyzw
    dcl_2d s0
    dcl_2d s1
```

To perform the first MOP we fetch the needed data,

```
    texld r0, t0, s0
    texld r1, t1, s0
    texld r2, t2, s0
    texld r3, t3, s0
    texld r4, t4, s1
```

and execute the 4×1 matrix vector multiply. The result is held in `r5`.

```
    mul r5, r4.xxxx, r0
    mad r5, r4.yyyy, r1, r5
    mad r5, r4.zzzz, r2, r5
    mad r5, r4.wwww, r3, r5
```

If we had defined `numMOpsPerFragment` as 1 above, we would just write `r5` to `oC0`, and be done. However, we have not yet exhausted the capacities of the pixel shader, so we keep going:

```
#if numMOpsPerFragment >= 2
```

The texture coordinates are adjusted to correspond to the next set of inputs:

```
    add r6, t0, c0
    add r7, t1, c0
    add r8, t2, c0
    add r9, t3, c0
    add r10, t4, c1
```

Then we sample the textures as before. Note, however that we now use registers `r6` through `r10` instead of `r0` through `r4`. This is because ps.2.0 does not allow that we sample a texture into any one register more than four times, so the destination registers have to be rotated.

```
    texld r6, r6, s0
    texld r7, r7, s0
    texld r8, r8, s0
    texld r9, r9, s0
    texld r10, r10, s1
```

We accumulate the result of the second matrix-vector product with the first:

```
    mad r5, r10.xxxx, r6, r5
    mad r5, r10.yyyy, r7, r5
    mad r5, r10.zzzz, r8, r5
    mad r5, r10.wwww, r9, r5
#endif
```

MOPs three to six are identical save for the register rotation we mentioned:

```
#if numMOpsPerFragment >= 3
    add r0, t0, c2
    add r1, t1, c2
    add r2, t2, c2
    add r3, t3, c2
    add r4, t4, c3

    texld r0, r0, s0
    texld r1, r1, s0
    texld r2, r2, s0
    texld r3, r3, s0
    texld r4, r4, s1

    mad r5, r4.xxxx, r0, r5
    mad r5, r4.yyyy, r1, r5
    mad r5, r4.zzzz, r2, r5
    mad r5, r4.wwww, r3, r5
#endif
#if numMOpsPerFragment >= 4
    add r6, t0, c4
    add r7, t1, c4
    add r8, t2, c4
    add r9, t3, c4
    add r10, t4, c5

    texld r6, r6, s0
    texld r7, r7, s0
    texld r8, r8, s0
    texld r9, r9, s0
    texld r10, r10, s1

    mad r5, r10.xxxx, r6, r5
    mad r5, r10.yyyy, r7, r5
    mad r5, r10.zzzz, r8, r5
    mad r5, r10.wwww, r9, r5
#endif
#if numMOpsPerFragment >= 5
    add r0, t0, c6
    add r1, t1, c6
    add r2, t2, c6
    add r3, t3, c6
    add r4, t4, c7

    texld r0, r0, s0
    texld r1, r1, s0
    texld r2, r2, s0
    texld r3, r3, s0
    texld r4, r4, s1

    mad r5, r4.xxxx, r0, r5
    mad r5, r4.yyyy, r1, r5
    mad r5, r4.zzzz, r2, r5
    mad r5, r4.wwww, r3, r5
#endif
#if numMOpsPerFragment >= 6
    add r6, t0, c8
    add r7, t1, c8
    add r8, t2, c8
    add r9, t3, c8
    add r10, t4, c9

    texld r6, r6, s0
    texld r7, r7, s0
    texld r8, r8, s0
    texld r9, r9, s0
    texld r10, r10, s1

    mad r5, r10.xxxx, r6, r5
    mad r5, r10.yyyy, r7, r5
    mad r5, r10.zzzz, r8, r5
    mad r5, r10.wwww, r9, r5
#endif
mov oC0, r5
```

There are a few additional details to be mentioned: because a pixel shader operates on 4×(4 `numMOpsPerFragment`) submatrices, only input matrices with dimensions that are multiples of 4 `numMOpsPerFragment` are handled trivially. Other matrix sizes perform extra work because the pixel shading involving the last column-block of A and last row-block of B read in zeros and perform redundant computations. We even have to do work to ensure that indeed zeros get read in, and not undefined values. First, we set the texture coordinate mapping mode to black border color. Unfortunately not all GPUs support this feature. To support these GPUs we would either need to change the way we store the matrices in the surfaces so that the edge texels are not used, set the edge pixels to black, and use clamp mode, or restrict ourselves to matrices with row and column counts that are multiples of 4 `numMOpsPerFragment`. Finally, the pixel shader does a lot of redundant work processing input matrices with the inner dimension significantly smaller than 4 `numMOpsPerFragment`. Of course such small matrices are best processed on the CPU anyway to avoid the overhead of creating and reading back textures.

3.4 Transposed Multiplication

In practice we rarely need to compute transpose of a matrix as such, but often need to multiply the transpose of a matrix with another matrix. We implement a transposed multiply operation to be able to do this. The operation we now describe implements C := A^T B, where A,B, and C are still defined as in the last section. The operation C := A B^T is also useful, but its implementation would be very similar to this one, so we will omit it. As we will see later, this operation will end up to be more costly than the plain multiply. For this reason, it may be worth it to implement a simple transpose operation C := A^T as well, even though this operation can be inefficiently emulated using this code with B = **1**. Such an operation would be a clear win if a sequence of multiplications were needed with a certain transposed matrix, and perhaps even in general.

A trivial CPU implementation of the transposed multiply code would simply exchange the row and column indexing of A, but this is not so easy on the GPU because we have packed several matrix elements into a single pixel, so the transpose has to happen on two levels: The atomic 4×4 matrix pixel shader operation has to be transposed, and the ordering of these sub matrices also needs to be reversed. An indication of this added complexity is that this time we only managed to fit four transposed MOPs into our ps.2.0 pixel shader, as opposed to six for the plain multiply. Because this new constant is different from the previous, we define it as `numMTOpsPerFragment`.

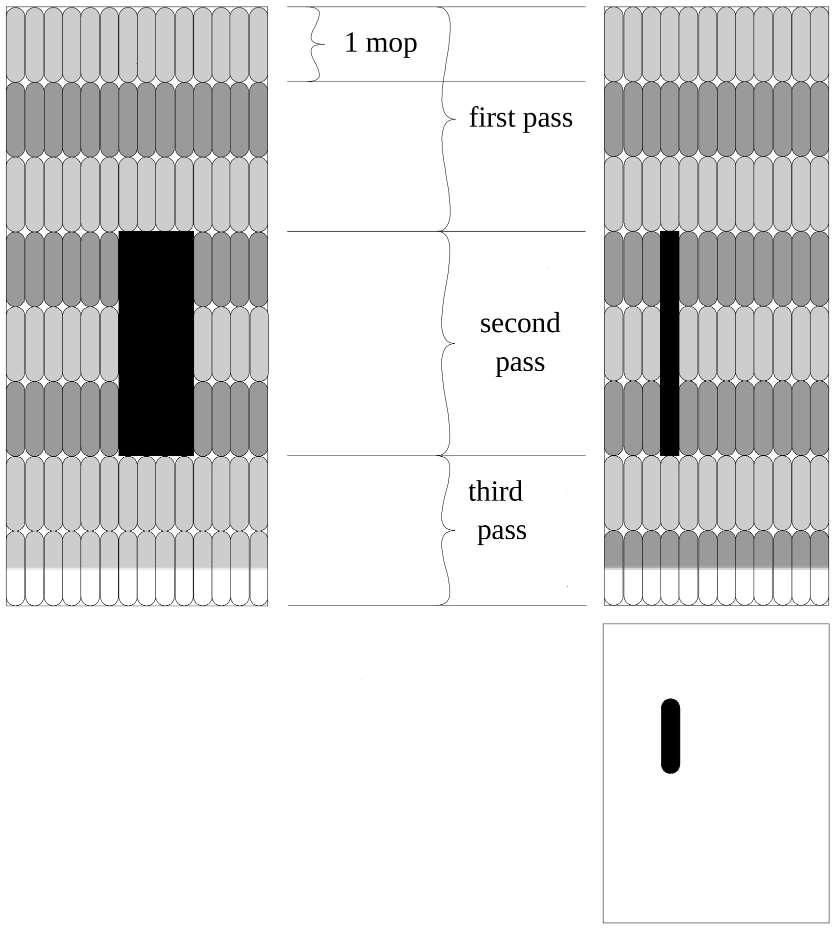

Figure 2: Schematic for transposed matrix multiply

We again provide a diagram to visualize this algorithm in Figure 2. This is exactly the same problem as given in Figure 1, with the matrix A now provided in a transposed form. To compute C as before, we transpose A while we perform the multiply. The matrix B and C are unchanged. The darkened regions again show the texels sampled to compute the contribution of the second pass to the black output pixel. Note that in matrix A, the region consists of three vertically stacked MOPs, each of which has four texels in a horizontal row. Our pixel shader will now be stepping four times to the left before resetting the horizontal offset and taking a step downward. The pixel shader has to move the starting texture coordinate of A downwards between passes.

The C++ code for the operation is quite similar to the vanilla multiply:

```
void Matrix::multiplyAT(Matrix & a, Matrix & b) {
if (this == &a || this == &b)
    throw "can’t operate inplace -- not supported by D3D.";
if (a.nRows != b.nRows)
    throw "matrix dimensions don’t agree";
resize(a.nCols, b.nCols);
if (nRows * nCols == 0)     return;

const unsigned numQuads =
roundUpDivide(rows2TextureHeight(b.nRows),numMTOpsPerFragment);

const float TexcoordBiasW = (1.0f/cols2TextureWidth(nCols))  * 0.5f;
const float TexcoordBiasH = (1.0f/rows2TextureHeight(nRows)) * 0.5f;
const float TexcoordBiasAW = (1.0f/cols2TextureWidth(a.nCols))  * 0.5f;
const float TexcoordBiasABH = (1.0f/rows2TextureHeight(a.nRows)) * 0.5f;
```

We compute bias values as usual above, and the offsets for texture B are also unchanged below:

```
const float tcPixelBH = 2 * TexcoordBiasABH;
const float tcMOpIncrementBH = tcPixelBH;
const float tcPassIncrementBH = numMTOpsPerFragment * tcPixelBH;
```

The offsets for matrix A are now in both the horizontal and vertical directions:

```
const float tcPixelAW = 2 * TexcoordBiasAW;
const float tcPixelAH = 2 * TexcoordBiasABH;
const float tcMOpIncrementAH = tcPixelAH;
const float tcPassIncrementAH = numMTOpsPerFragment * tcMOpIncrementAH;
```

There is an additional issue in transposed multiply that did not show up before. Previously it was always proper for the vertex shader to simply linearly map vertex coordinates from the range [1, -1] to the texture coordinate range [0,1] to define the U or V texture coordinates of an input texture. Now, however, the U dimension of texture A is mapped vertically, and the V dimension horizontally. If A is not square, and there are unused components in the bottom row of the destination texture because its height is not a multiple of four, the mapping has to be adjusted. We map the vertex range [1,-1] to [0, `quotient`], where `quotient` is computed below. In effect we virtually round up the texture size to the nearest multiple of four. The rest of the code should be familiar.

```
const unsigned awidth = cols2TextureWidth(a.nCols);
unsigned modW = awidth % 4;
if (modW != 0) modW = 4 - modW;
const float quotient = (awidth + modW)/(float)awidth;
const float halfQuot = quotient * 0.5f;

float vconsts[] = {
    0.5, - halfQuot, 0.5, 1,
    0.5 + TexcoordBiasW,  0.5 + TexcoordBiasH, 0 + TexcoordBiasABH, 0,
    0   + TexcoordBiasABH, 0, 0, 0,
    0, q + TexcoordBiasAW, 0, 0,
    tcPassIncrementBH, tcPassIncrementAH, 0, 0
    };

float pconsts[] = {
    tcPixelAW, 0, 0,0,
    0, 1 * tcMOpIncrementBH, 0,0,
    0, 1 * tcPixelAH, 0,0,
    0, 2 * tcMOpIncrementBH, 0,0,
    0, 2 * tcPixelAH, 0,0,
    0, 3 * tcMOpIncrementBH, 0,0,
    0, 3 * tcPixelAH, 0,0,
    0, 4 * tcMOpIncrementBH, 0,0,
    0, 4 * tcPixelAH, 0,0,
    0, 5 * tcMOpIncrementBH, 0,0,
    0, 5 * tcPixelAH, 0,0,
    };

d3dDevice->SetRenderTarget(0,mathSurface);
d3dDevice->BeginScene();
d3dDevice->SetVertexDeclaration( vertexDeclaration2 );
d3dDevice->SetVertexShader( vertexShaders[VS_MULT_T] );
d3dDevice->SetPixelShader( pixelShaders[PS_MULT_T] );
d3dDevice->SetTexture(0,a.mathTexture);
d3dDevice->SetTexture(1,b.mathTexture);
d3dDevice->SetStreamSource( 0, quadsVertexBuffer, 0, TINYVERTEX_SIZE );
d3dDevice->SetVertexShaderConstantF(0, vconsts, 5);
d3dDevice->SetPixelShaderConstantF(0, pconsts, 1 + 2 * (numMTOpsPerFragment - 1) );
d3dDevice->DrawPrimitive( D3DPT_TRIANGLELIST, 0, 2 );
```

```
if (numQuads > 1)
    {
    d3dDevice->SetRenderState( D3DRS_ALPHABLENDENABLE,   TRUE );
    d3dDevice->DrawPrimitive( D3DPT_TRIANGLELIST, 6, 2 * (numQuads - 1) );
    d3dDevice->SetRenderState( D3DRS_ALPHABLENDENABLE,   FALSE );
    }
d3dDevice->EndScene();
}
```

The vertex shader code first extracts and emits the vertex position, as the straight multiply did:

```
vs_1_1
dcl_position v0
def c5, -1, -1, 0, 0
add r3.xy, v0.xy, c5.xy
mov oPos.xy, r3.xy
mov oPos.zw, c0.zw
```

The vertex to texture coordinate mapping is now done twice, because as discussed above, texture A's texture coordinate range is no longer always [0,1], while B's still is. These four instructions could be optimized into fewer, but the vertex shader is no bottleneck here.

```
mov r0.xy, c1.xy
mad r0.xy, r3.xy, c0.xy, r0.xy

mov r1.xy, c3.xy
mad r1.xy, r3.xy, c0.xy, r1.xy
```

The code to add offsets to the texture coordinates is the same as in the plain multiply, except it works along different dimensions for A:

```
mul r3, v0.w, c4.zzxz
add r3, r3, c1
mul r2, v0.w, c4.yyyy
add r2, r2, c2
```

Note that unlike before, we only emit two texture coordinates. We were not able to optimize the pixel shader in this case by precomputing more texture coordinates here.

```
mov oT0.x, r1.y
mov oT0.y, r2.x
mov oT1.x, r0.x
mov oT1.y, r3.z
```

Below we have the last shader presented in this article. You will notice that after we fetch the first sample from A, we keep nudging the texture coordinates to the right to fetch the next three samples. The last `texld` samples the 4-vector from B.

```
    ps_2_0
    dcl t0.xyzw
    dcl t1.xyzw
    dcl_2d s0
    dcl_2d s1
    texld r0, t0, s0
    add   r4, t0, c0
    texld r1, r4, s0
    add   r4, r4, c0
    texld r2, r4, s0
    add   r4, r4, c0
    texld r3, r4, s0

    texld r4, t1, s1
```

The transposed multiply can be accomplished with four `dp4`-s, the result goes to `r5`:

```
    dp4 r5.x, r4, r0
    dp4 r5.y, r4, r1
    dp4 r5.z, r4, r2
    dp4 r5.w, r4, r3
#if numMTOpsPerFragment >= 2
```

To execute the next MOP, we push the original `t0` downward in A by adding `c2`, and then again sampling four consecutive pixels. We rotate the sampling destination registers so we avoid getting a 4th order dependent read error in the shader compiler as long as possible.

```
    add   r0, t0, c2
    texld r6, r0, s0
    add   r0, r0, c0
    texld r7, r0, s0
    add   r0, r0, c0
    texld r8, r0, s0
    add   r0, r0, c0
    texld r9, r0, s0

    add r1, t1, c1
    texld r10, r1, s1

    dp4 r6.x, r10, r6
    dp4 r6.y, r10, r7
    dp4 r6.z, r10, r8
    dp4 r6.w, r10, r9
    add r5, r5, r6
#endif
#if numMTOpsPerFragment >= 3
```

The third and fourth blocks simply continue to follow this pattern.

```
    add   r4, t0, c4
    texld r0, r4, s0
    add   r4, r4, c0
    texld r1, r4, s0
    add   r4, r4, c0
    texld r2, r4, s0
    add   r4, r4, c0
    texld r3, r4, s0

    add r4, t1, c3
    texld r4, r4, s1

    dp4 r6.x, r4, r0
    dp4 r6.y, r4, r1
    dp4 r6.z, r4, r2
    dp4 r6.w, r4, r3
    add r5, r5, r6
#endif
#if numMTOpsPerFragment >= 4
    add   r0, t0, c6
    texld r6, r0, s0
    add   r0, r0, c0
    texld r7, r0, s0
    add   r0, r0, c0
    texld r8, r0, s0
    add   r0, r0, c0
    texld r9, r0, s0

    add r1, t1, c5
    texld r10, r1, s1

    dp4 r6.x, r10, r6
    dp4 r6.y, r10, r7
    dp4 r6.z, r10, r8
    dp4 r6.w, r10, r9
    add r5, r5, r6
#endif
```

Unfortunately the above conditional block is the last one that compiles with ps.2.0, because the first `texld` of the next block produces a 4th order texop error. This hand coded assembly code still manages to pack much more math into the pixel shader than the HLSL compiler managed.

```
#if numMTOpsPerFragment >= 5
    add   r7, t0, c8
    texld r0, r7, s0
    add   r7, r7, c0
    texld r1, r7, s0
    add   r7, r7, c0
    texld r2, r7, s0
    add   r7, r7, c0
    texld r3, r7, s0

    add r4, t1, c7
    texld r4, r4, s1

    dp4 r6.x, r4, r0
    dp4 r6.y, r4, r1
    dp4 r6.z, r4, r2
    dp4 r6.w, r4, r3
    add r5, r5, r6
#endif
    mov oC0, r5
```

3.5 Other Operations

From the operations described above, we create more by writing different variations, and writing macro operations that build on them. We briefly summarize them here:

```
float Matrix ::dot(Matrix & vec);               //this *= vec'
```

This is only defined if both operands are vectors. The dot product of two vectors a and b equals $a^T b$, so we can reuse the transposed multiply operation. This results in a temporary 1×1 texture whose red component we read out and return.

```
float Matrix::normSquared();
```

Only defined for a vector a, this simply calls `a.dot(a)`.

```
void Matrix::multiply(float c);                 //this *= c
```

Multiplication by a constant is implemented with a simple shader that does a multiplicative blend between the destination and a `c`-colored quad.

```
void Matrix::add(Matrix & b);                   //this += b
```

Unary accumulate is the `copy()` operation with additive blending with the rendertarget turned on.

```
void Matrix::max(Matrix & a, float ref);        //this = max(a, ref)
```

This operation is also similar to `copy()`, but also employs the `max` pixel shader opcode to compute the maximum of the corresponding matrix elements.

```
void Matrix::mad(Matrix & b, float c);          //this += b * c.
void Matrix::mad(Matrix & a,Matrix & b,float c);//this = a + b * c
void Matrix::mad(Matrix & a, Matrix & b);       //this += a .* b
void Matrix::madad(Matrix & a, Matrix & b, Matrix & c, Matrix & d);
//this = a + (b + c) .* d
```

Finally, all the different flavors of the *mad* (multiply add) operation are a combination of the add and constant multiply shaders. We use `.*` to denote array multiplication. We also implemented some initialization operations. To create a zero matrix we simply clear the texture to black. Identity matrices and other special matrices are best implemented by writing the appropriate data with the CPU. This is also how matrices are saved and loaded from file.

4. Applications

In this section we describe two high level algorithms that use the operations we described above. None of them reads back intermediate results (other than scalars) from the CPU, so all the real work still happens on the GPU as a sequence of render to texture operations. It would be possible to optimize both of them by writing special purpose *macro* shaders that combine several basic matrix operations to reduce the number of render to texture operations. We have done this to a small degree by implementing the multiply-add operations, but in general we would like to keep our operations small in number and reusable.

Both of the discussed methods are iterative. Iterative methods, in contrast to pivoting methods, are typically simple and perform a small number of matrix operations to converge to the desired result, rather than doing a number of scalar operations that are often difficult to vectorize.

4.1 Conjugate Gradients

The Conjugate Gradients algorithm was developed at the ETH Zurich in 1952 [4]. It is the most common iterative algorithm used to solve a system of linear equations of the form Ax = b, where the matrix A and the vector b are given, and the vector x is to be found. Although this algorithm has been extended to handle more general classes of matrices, we only deal with the simplest version that requires A to be symmetric positive definite.

Our implementation of the algorithm does not have any DirectX or shader code of its own. Instead, it uses the methods of the matrix class we created. The three-operand matrices are given:

```
Matrix & A = ...;
Matrix & x = ...;
Matrix & b = ...;

unsigned n = b.getNRows();
```

If the algorithm is used in a physics simulation context, is often desirable to warm start it with the solution of the problem in the previous simulation time step, with the hope that the solution of the current time step is nearby. If the size of the input vector x is compatible with the size of A, we assume that the user wants to warm start with x, otherwise we start with a first guess of zero:

```
if (x.getNRows() != n || x.getNCols() != 1)
    x.zeros(n, 1);
```

The algorithm uses three temporary vectors:

```
Matrix p, r, s;

p.copy(b);
r.copy(b);
float rr = r.normSquared();
s.multiply(A,p);
float t = p.dot(s);
float alpha = rr / t;
x.mad(p, alpha);
float rrnew = rr;
```

The conjugate gradients algorithm is proven to converge to the exact solution within n steps[1], though we could get an approximate solution with less iterations.

```
unsigned iter = n;

for (unsigned k = 2; k<=iter; k++)
    {
    r.mad(s, -alpha);
    rr = rrnew;
    rrnew = r.normSquared();
    float beta = rrnew / rr;
```

[1] This is only strictly true if we were using exact arithmetic. For a discussion of the convergence properties of conjugate gradients using floating point arithmetic, see [3].

```
    p.mad(r, p, beta);
    s.multiply(A,p);
    t = p.dot(s);
    alpha = rrnew / t;
    x.mad(p, alpha);
    }
```

The most expensive operation in the algorithm is the matrix-vector multiply M*p above. All other operations are vector operations. This makes the algorithm relatively 'light weight', and less suitable for demonstrating the number crunching abilities of the GPU. On the other hand a much more expensive algorithm would be less practical, and therefore this example is a good real world test of the GPU's applicability to real world linear algebra applications.

4.2 Linear Complementarity Problem

The linear complementarity problem, while not as widely known as the problem of linear equation systems, is very useful for solving a wide variety of problems, including the dynamics of resting contact. Linear complementarity problems are a special kind of nonlinear programming problem, which in turn is a problem of constrained optimization. The LCP problem can be stated as:

$$x \geq 0$$

$$Ax + b \geq 0$$

$$x^T(Ax + b) = 0$$

As before, A and b are given, and x is to be found. We use the projected Jacobi method [8] for solving the problem, which is perhaps the simplest way to do so, though not necessarily the one with the best convergence properties. The projected Jacobi algorithm can be stated succinctly as the recursion:

$$x_{i+1} = \max(x_i + D(Ax_i + b), \vec{0})$$

Where D is defined as:

$$D = \varpi \text{ diagonal}(A)^{-1}$$

ω is a constant that steers convergence. Clever implementations of the algorithm tune this value while the solver runs to speed up convergence; we just use a fixed value. This algorithm again requires A to be symmetric positive definite.

As before, we first receive the matrices we are to operate on. Note that because d is a constant, it is also expected to be provided as an input. This time the number of iterations is also a mandatory input because with this algorithm, there is no guaranteed convergence for a certain number of iterations. In the code below we store the diagonal elements of the diagonal matrix D in a column vector d.

```
Matrix & x = ...;
Matrix & A = ...;
Matrix & d = ...;
Matrix & b = ...;
unsigned iter = ...;

unsigned n = b.getNRows();

Matrix w, t;
```

Here we again warm start the algorithm with the initial value of x if it exists, otherwise we start at zero:

```
if (x.getNRows() != n || x.getNCols() != 1)
    {
    x.zeros(n, 1);
    w.zeros(n, 1);
    }
else
```

```
    w.multiply(A, x);

t.madad(x, b, w, d);
x.max(t, 0);

for (unsigned k = 1; k<iter; k++)
    {
    w.multiply(A, x);
    t.madad(x, b, w, d);
    x.max(t, 0);
    }
```

The above loop implements the iteration presented above. Here too the most expensive operation is a matrix vector multiply.

5. Results

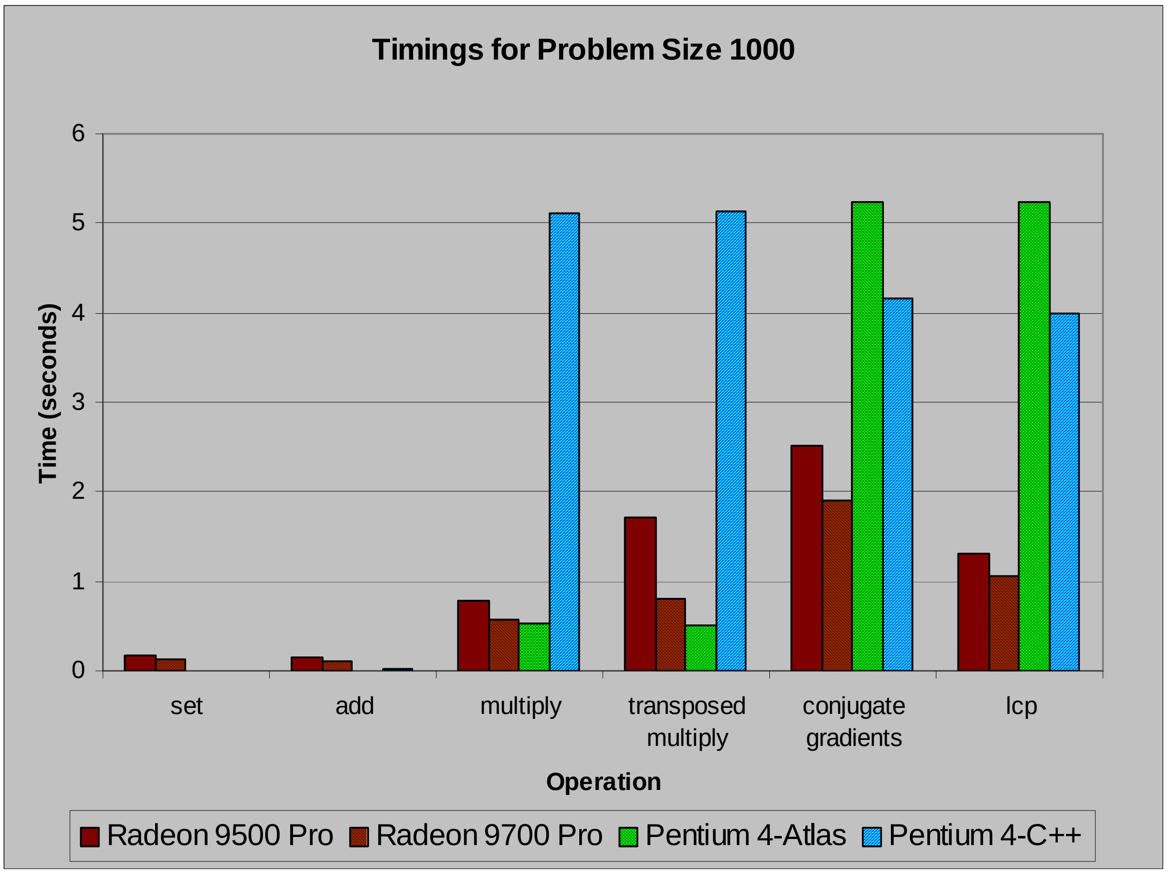

Figure 3: GPU vs. CPU performance for various operations involving a 1000 × 1000 matrix.

Figure 3 summarizes the running times for the matrix copy, add, multiply, transposed multiply, conjugate gradients, and projected Jacobi algorithms. We have profiled four configurations: Our GPU implementation running on a Radeon 9500 Pro that was plugged into a PC with an AMD Athlon 800 Mhz processor and 256 MB of RAM, and a Radeon 9700 Pro in a P4 2.4GHz, with 256MB RAM. The two CPU configurations are a simple C implementation by the author and a program using the ATLAS library. Both of the CPU configurations were timed on a 1.6GHz Intel Pentium 4 processor equipped PC with 256 MB RAM. We used a version of ATLAS optimized for Pentium 4 CPUs with 8 KB L1 cache, and 256 KB L2 cache; these specifications correspond to our test system. Note that while ATLAS gains a sizeable performance boost from this cache-size specific optimization, it means that the library has to be reconfigured for each target platform; this is somewhat impractical for interactive entertainment software that is to be distributed to end users. In comparison, a DirectX 9 program, which only indirectly interfaces with hardware, and thus exploits the video card's manufacturer specific driver optimizations, is more flexible.

All GPU timings represent the time elapsed between the call of the appropriate Matrix class method, and completion of the read back of the result matrix from video to system memory. Retrieving results to system memory is a significant portion of the time needed for copy and addition, but are negligible for the other operations. The C implementation is an order of magnitude slower than either the ATLAS or the GPU implementation, which fall in the same performance class. The GPU transposed multiply is slower than the straight multiply because it needs more rendering passes. The CPU implementations' speeds are the same for straight and transposed multiply, because here the difference boils down to a change in matrix indexing and computation order that does not even need to influence cache coherence.

The projected Jacobi algorithm for LCPs runs faster than conjugate gradients for a given problem size because the number of render to texture operations in the loop is much lower, and we never read back any intermediate values from the GPU, not even scalars. Conjugate gradients reads back a scalar value in `normSquared()` – this is the result of a dot product operation, and is retrieved from the 1×1 render target texture used in this case. Because it is only a single value it does not stress the limited texture memory to system memory bandwidth of PCs, but it still hurts performance because it forces the CPU and GPU to work together in a lockstep instead of working asynchronously. One could further optimize conjugate gradients by merging its sequence of vector operations into a single operation by writing a custom 'macro' pixel shader. This is left as an exercise for the reader.

Because LCP and conjugate gradients consist of a sequence of vector-vector and matrix-vector operations, ATLAS is unable to leverage its optimized matrix-matrix kernel, and instead adds significant overhead, loosing out to the inlined C version. For these two algorithms data caching is less of a bottleneck, because there are less numbers to work with. Instead, raw floating point performance is dominant, catapulting the Radeons into the lead.

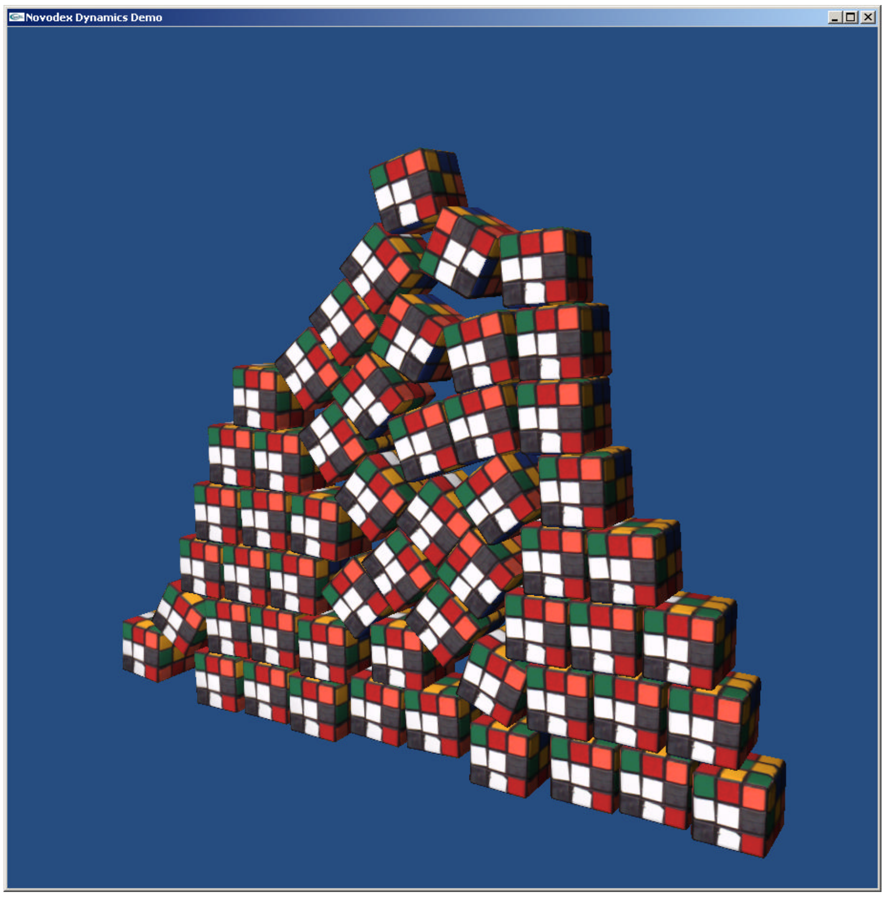

Figure 4: Collapsing wall of 60 cubes.

Figure 4 shows a practical application: a wall of 60 cubes collapsing. The simulation is performed by the projected Jacobi code. The simulation uses $\omega = -0.1$ and does 2n iterations, where n is the size of the input matrix. If the problem is expressed as a single dense matrix (with an initial size of 400×400), the simulation runs at 2 seconds per frame. If the problem is dynamically decomposed into small sub-problems, real-time performance is achieved. Of course, more advanced LCP algorithms (which are less suitable for GPU implementation) can achieve even better results, because of eventually lower storage overhead (sparse matrix) and much faster convergence.

6. Conclusion

In [6], Larsen and McAllister have benchmarked matrix multiplies on geForce3 GPUs, and have found that the GPU, working with byte values, achieves similar performance to the CPU using single precision floating point, effectively operating on four times as much data. They were pessimistic about the prospect of a four-fold increase in GPU performance, even if GPUs were to integrate floating point processing capabilities. Our results above indicate that this has indeed happened only two years later.

We have observed that the Radeon GPUs have outperformed optimized CPU code running on a mid-range PC when executing two important algorithms. Moreover, the performance penalty due to moving data to the GPU and back to main memory can be negligible compared to the overall cost of the computation when the problem size is sufficient. As a result, this additional source of computing power should not be ignored. Instead, algorithms must be found that can exploit the specific strengths of the GPU. Approaches that split the work between CPU and GPU and thus achieve maximum parallelism will make it possible to run simulations of previously untractable scales on low cost PCs.

7. Acknowledgements

Thanks to Wolfgang Engel, Tom Forsyth, and ATI and NVidia developer relations for help with different aspects of this project, and to Erwin Coumans, Mark Harris, Stephane Redon and Jan Paul van Waveren for valuable suggestions.

8. References

[1] Jeff Bolz, Ian Farmer, Eitan Grinspun and Peter Schröder, *Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid*, To appear in the proceedings of SIGGRAPH 2003.

[2] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson, *An extended set of FORTRAN Basic Linear Algebra Subprograms*, ACM Trans. Math. Soft., 14 (1988), pp. 1--17.

[3] Axel Facius, *Iterative Solution of Linear Systems with Improved Arithmetic and Result Verification*, Ph.D. Thesis, Universität Karlsruhe, July 2000.

[4] M. Hestenes and E. Stiefel, *Methods of conjugate gradients for solving linear systems*, J. Research Nat. Bur. Standards 49, 1952.

[5] Jens Krüger and Rüdiger Westermann, *Linear Algebra Operators for GPU Implementation of Numerical Algorithms,* To appear in the proceedings of SIGGRAPH 2003.

[6] E. S. Larsen and D. McAllister, *Fast Matrix Multiplies using Graphics Hardware,* SuperComputing 2001 Conference, Denver, CO, November 2001.

[7] C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh, *Basic Linear Algebra Subprograms for FORTRAN usage,* ACM Trans. Math. Soft., 5 (1979), pp. 308--323.

[8] K. G. Murty, *Linear Complementarity, Linear and Nonlinear Programming*, Helderman-Verlag, 1988.

[9] R. C. Whaley and J. Dongarra, *Automatically Tuned Linear Algebra Software,* SuperComputing 1998 Conference, Orlando, FL, November 1998.