High Performance Production-Quality Fluid Simulation via NVIDIA's QuadroFX

July 2003

Chris Batty
Frantic Films
cbatty@franticfilms.com

Mark Wiebe
Frantic Films
mwiebe@franticfilms.com

Ben Houston
Frantic Films
bhouston@franticfilms.com

Introduction

The creation of a GPU-based equivalent to our CPU-based preconditioned conjugate gradient solver took about 1 month from start to finish. Initial results were underwhelming but after some creative optimization a 50% speed-up was achieved in comparison to our fastest CPU-based version of the pressure projection step. This speed-up was achieved even though it proved impossible to implement the optimal preconditioner on the current NVIDIA GPU architecture.

Background

Frantic Films, Canada's world class visual effects production house, has developed internally, for competitive advantage, a state of the art fluid simulation system. While the design and implementation of the fluid simulation system has been successful, its performance on modern CPUs is underwhelming. Achieving realistic looking fluid effects via physically accurate computation fluid dynamics has proved to be incredibly computationally demanding.

Figure 1: The fluid simulation system

After a site visit by NVIDIA's Studio Relations group in early December 2002 and some developer brainstorming shortly thereafter, it became apparent that we could harness some of the floating point horse power of NVIDIA's newest GPU's in service of our computational needs.

Pressure Projection Step

The calculation of each frame by our fluid simulator requires the execution of almost two dozen quite heterogeneous steps. From a practical standpoint it was not feasible to spend the effort to accelerate each step thus we focused on the most expensive bottleneck step: the pressure projection step.

The pressure projection step is responsible for enforcing the incompressible nature of fluid by relating every piece of the liquid body to every other piece via physical quantity known as pressure. The mathematical formulation is simply $\nabla^2 \mathbf{p} = \nabla \cdot \mathbf{u}$. In our case only **u** and the boundary conditions are known and thus the task is to solve for **p**. It is established that the first order approximation to this problem generates results that are satisfactory for our purposes. Thus we can reformulate the above as the common linear system **Ax**=**b**, where **A** represents the Laplacian operator, **b** represents the

divergence of **u** and **x** is unknown. The bulk of the computations required to complete the pressure projection step are involved in the calculation of the unknown **x** that satisfies this relation.

The step is computationally intensive because of the size of these variables. The elements of matrix **A** thus compose a system of linear equations which relate, via first order methods, each voxel of the simulation grid to its six neighbors (while applying the appropriate boundary conditions where necessary). Our production-quality simulations, in order to capture the necessary fluid details, require at least a million voxels. Thus the width of the square matrix A is at least one million and the number of non-zero elements it must represent is seven times that. Because we represent the boundary conditions with second order accuracy within the overall first order approximation we require each element to have 32 bits of floating point precision. This means that the memory requirement alone of the non-zero matrix elements is at least 28 MB.

Preconditioned Conjugate Gradient Method

Because of the size of the matrix A the method of solution via inversion is not infeasible. The alternative solution method that we have explored is the iterative *preconditioned conjugate gradient* approach. The preconditions conjugate gradient method, as the name suggests, consists of two parts: a matrix preconditioner and the conjugate gradient hill climbing algorithm.

The conjugate gradient hill climbing algorithm moves towards the solution of the linear system by repeatedly applying simple forward solution methods.

There are many different matrix preconditioners that one can use with the conjugate gradient hill climbing algorithm. These preconditions are used to transform the gradient landscape of the problem space so that the shorter paths to the solution are emphasized thus reducing the number of iterations needed for convergence.

In our work with our CPU-based conjugate gradient solver we experimented with both the *diagonal preconditioner* and the *incomplete Cholesky preconditioner*. The diagonal preconditioner was incredibly simple to implement while the incomplete Cholesky preconditioner involves a significant degree of interleaved read and writes. The diagonal precondition also differed in regards to the incomplete Cholesky preconditioner in that it required on average 3 times as many iterations of the conjugate gradient algorithm before achieving a similar degree of solution convergence.

It was this specific algorithm, the preconditioned conjugate gradient method, which we decided to move over onto NVIDIA's GPU architecture.

GPU Algorithm Implementation

The first step towards creating the GPU-based version of the preconditioned conjugate gradient algorithm was getting the data into texture memory. This was achieved via the use of the OpenGL extensions NV_float_buffer. We setup our 32-bit floating point render targets via the WGL_ARB_pbuffer and WGL_ARB_render_texture extensions.

Each vector was represented by a 2D texture with the data laid out left to right and wrapping at the end of each line. The matrix representing the Laplacian operator, since its total size often exceeded a maximum texture size limit, was represented as 7 separate textures, one for each of its row's seven non-zero elements.

We decided to write each of the mathematic operations that are required in a conjugate gradient iteration in Cg. Since our chosen platform and language, Microsoft's .NET platform and C#, did not support any Cg interfaces natively, we had to write our own. Luckily, this did not prove too difficult.

The conjugate gradient algorithm required the implementation in Cg of the following mathematical operations:

vector-vector element-wise addition	$\mathbf{u} + \mathbf{v}$
vector-scalar multiplication	$\alpha\mathbf{u}$
matrix-vector multiplication	$\mathbf{Ax}$
vector-vector dot product	$\mathbf{u} \cdot \mathbf{v}$

In our CPU-based implementation the matrix-vector multiplication required the most computational time followed by the non-trivial matrix transform of the incomplete Cholesky preconditioner. The GPU-based implementation of the conjugate gradient algorithm was similar to the CPU-based implementation in that the matrix-vector multiplication was still the most time consuming. Interestingly, it proved difficult to implement the vector-vector dot product operation on the GPU and thus on the GPU, unlike our CPU-based implementation, this step required significant computational time.

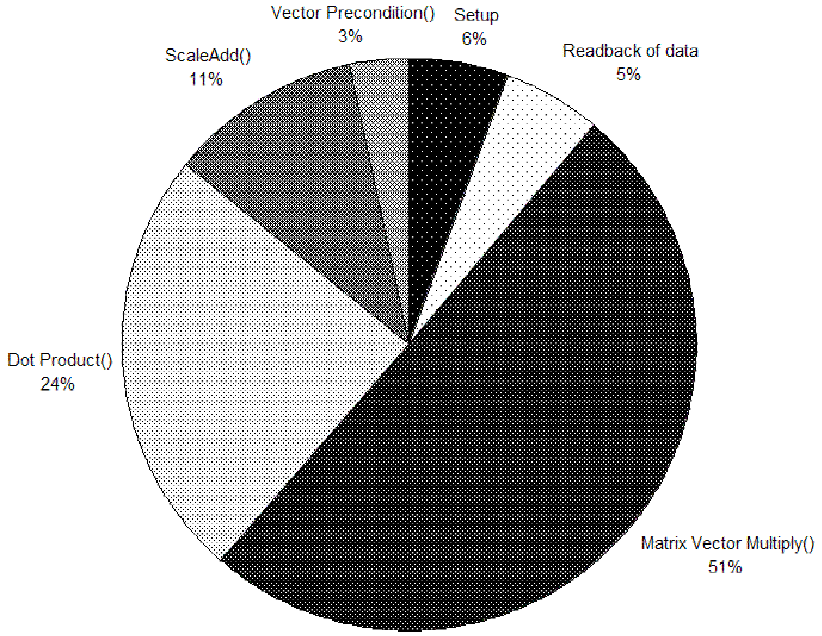

Figure 2: Computational time breakdown of GPU-based implementation conjugate gradient algorithm

Since our GPU-based implementation of the algorithm spent the bulk of its time calculating the matrix-vector multiplication we spend significant efforts on optimization its performance. In this regards we were quite successful. Our sparse matrix representation, since it contains only seven non-zero elements per row, lead to some interesting solutions. We were able to convert the usual "multiplication" and "modulus" indexing operations into quickly "add" and "fraction" operations. We were also able to fully exploit the 4 element per clock tick instructions in our multiplication routine. The optimize Cg source listing for the matrix-vector multiplication and its corresponding assembler code is listed below:

```
fragout_float main (
     vf30 In,
     uniform samplerRECT mat0,
     uniform samplerRECT mat1,
     uniform samplerRECT mat2,
     uniform samplerRECT mat3,
     uniform samplerRECT mat4,
     uniform samplerRECT mat5,
     uniform samplerRECT mat6,
     uniform samplerRECT vector,
     uniform float width,
     uniform float invWidth,
     uniform float3 XOff,
     uniform float3 YOff,
     uniform float3 ZOff
     )
{
    fragout_float O;

    float2 skewPos;
    float2 skewCoords = In.TEX0.xy;
    skewCoords.y = skewCoords.y - 0.5 +
            skewCoords.x * invWidth;

    float4 mainVec =
         f4texRECT(vector, In.TEX0.xy);
    //add the centre diagonal values
    O.col = mainVec *
         f4texRECT(mat3, In.TEX0.xy);

    //New Positive Y-offset
    skewPos = skewCoords + YOff.xy;
    skewPos.x = frac(skewPos.y) * width;
    O.col += f4texRECT(vector, skewPos) *
         f4texRECT(mat5, In.TEX0.xy);

    //New Negative Y-offset
    skewPos = skewCoords - YOff.xy;
    skewPos.x = frac(skewPos.y) * width;
    O.col += f4texRECT(vector, skewPos) *
         f4texRECT(mat1, In.TEX0.xy);

    //New Positive Z-Offset
    skewPos = skewCoords + ZOff.xy;
    skewPos.x = frac(skewPos.y) * width;
    O.col += f4texRECT(vector, skewPos) *
         f4texRECT(mat6, In.TEX0.xy);

    //New Negative Z-offset
    skewPos = skewCoords - ZOff.xy;
    skewPos.x = frac(skewPos.y) * width;
    O.col += f4texRECT(vector, skewPos) *
         f4texRECT(mat0, In.TEX0.xy);

    //New Positive X-offset
    skewPos = skewCoords + XOff.xy;
    skewPos.x = frac(skewPos.y) * width;
    float4 temp = f4texRECT(mat4, In.TEX0.xy);
    O.col.a += temp.a *
         f4texRECT(vector, skewPos).r;
    O.col.rgb += temp.rgb * mainVec.gba;

    //New Negative X-offset
    skewPos = skewCoords - XOff.xy;
    skewPos.x = frac(skewPos.y) * width;
    temp = f4texRECT(mat2, In.TEX0.xy);
    O.col.gba += temp.gba * mainVec.rgb;
    O.col.r += temp.r *
        f4texRECT(vector, skewPos).a;

    return O;
}
```

```
septdiagmul.cg
!!FP1.0
# NV_fragment_program generated by NVIDIA Cg compiler
# cgc version 1.1.0003, build date Mar  4 2003
12:32:10
# command line args: -profile fp30
#vendor NVIDIA Corporation
#version 1.0.02
#profile fp30
#program main
#semantic main.mat0
#semantic main.mat1
#semantic main.mat2
#semantic main.mat3
#semantic main.mat4
#semantic main.mat5
#semantic main.mat6
#semantic main.vector
#semantic main.width
#semantic main.invWidth
#semantic main.XOff
#semantic main.YOff
#semantic main.ZOff
#var float4 In.WPOS : $vin.WPOS : WPOS : 0 : 1
#var float4 In.COL0 : $vin.COL0 : COL0 : 0 : 1
#var float4 In.COL1 : $vin.COL1 : COL1 : 0 : 1
#var float4 In.TEX0 : $vin.TEX0 : TEX0 : 0 : 1
#var float4 In.TEX1 : $vin.TEX1 : TEX1 : 0 : 1
#var float4 In.TEX2 : $vin.TEX2 : TEX2 : 0 : 1
#var float4 In.TEX3 : $vin.TEX3 : TEX3 : 0 : 1
#var float4 In.TEX4 : $vin.TEX4 : TEX4 : 0 : 1
#var float4 In.TEX5 : $vin.TEX5 : TEX5 : 0 : 1
#var float4 In.TEX6 : $vin.TEX6 : TEX6 : 0 : 1
#var float4 In.TEX7 : $vin.TEX7 : TEX7 : 0 : 1
#var samplerRECT mat0 :  : texunit 0 : 1 : 1
#var samplerRECT mat1 :  : texunit 1 : 2 : 1
#var samplerRECT mat2 :  : texunit 2 : 3 : 1
#var samplerRECT mat3 :  : texunit 3 : 4 : 1
#var samplerRECT mat4 :  : texunit 4 : 5 : 1
#var samplerRECT mat5 :  : texunit 5 : 6 : 1
#var samplerRECT mat6 :  : texunit 6 : 7 : 1
#var samplerRECT vector :  : texunit 7 : 8 : 1
#var float width :  :  : 9 : 1
#var float invWidth :  :  : 10 : 1
#var float3 XOff :  :  : 11 : 1
#var float3 YOff :  :  : 12 : 1
#var float3 ZOff :  :  : 13 : 1
#var float4 col : $vout.COL : COL : -1 : 1
#var float depth : $vout.DEPR : DEPR : -1 : 1
DECLARE width;
DECLARE invWidth;
DECLARE XOff;
DECLARE YOff;
DECLARE ZOff;
TEX R0, f[TEX0].xyxx, TEX5, RECT;
TEX R1, f[TEX0].xyxx, TEX7, RECT;
ADDR R2.x, f[TEX0].y, -{0.5}.x;
MADR R2.x, f[TEX0].x, invWidth.x, R2.x;
MOVR R3.x, f[TEX0].xyxx;
MOVR R3.y, R2.x;
ADDR R2.y, R3.xyxx, YOff.xyxx;
FRCR R2.w, R2.y;
MULR R2.w, R2.w, width.x;
MOVR R2.x, R2.w;
TEX R2, R2.xyxx, TEX7, RECT;
TEX R4, f[TEX0].xyxx, TEX3, RECT;
MULR R0, R2, R0;
MADR R0, R1, R4, R0;
ADDR R2.y, R3.xyxx, -YOff.xyxx;
FRCR R2.w, R2.y;
MULR R2.w, R2.w, width.x;
ADDR R4.y, R3.xyxx, ZOff.xyxx;
FRCR R3.w, R4.y;
MULR R3.w, R3.w, width.x;
MOVR R2.x, R2.w;
TEX R2, R2.xyxx, TEX7, RECT;
TEX R5, f[TEX0].xyxx, TEX1, RECT;
MADR R0, R2, R5, R0;
ADDR R2.y, R3.xyxx, -ZOff.xyxx;
FRCR R2.w, R2.y;
MULR R2.w, R2.w, width.x;
MOVR R4.x, R3.w;
TEX R4, R4.xyxx, TEX7, RECT;
TEX R5, f[TEX0].xyxx, TEX6, RECT;
MADR R0, R4, R5, R0;
```

```
ADDR R4.y, R3.xyxx, XOff.xyxx;
ADDR R3.y, R3.xyxx, -XOff.xyxx;
FRCR R3.w, R4.y;
MULR R3.w, R3.w, width.x;
MOVR R2.x, R2.w;
TEX R2, R2.xyxx, TEX7, RECT;
TEX R5, f[TEX0].xyxx, TEX0, RECT;
MADR R0, R2, R5, R0;
MOVR R4.x, R3.w;
TEX R2.x, R4.xyxx, TEX7, RECT;
TEX R4, f[TEX0].xyxx, TEX4, RECT;
MADR R2.x, R4.w, R2.x, R0.w;
MOVR R0.w, R2.x;
MADR R2.xyz, R4.xyzx, R1.yzwy, R0.xyzx;
MOVR R0.xyz, R2.xyzx;
FRCR R2.x, R3.y;
MULR R2.x, R2.x, width.x;
MOVR R3.x, R2.x;
TEX R2.w, R3.xyxx, TEX7, RECT;
TEX R3, f[TEX0].xyxx, TEX2, RECT;
MADR R1.xyz, R3.yzwy, R1.xyzx, R0.yzwy;
MOVR R0.yzw, R1.xxyz;
MADR R1.x, R3.x, R2.w, R0.x;
MOVR R0.x, R1.x;
MOVR o[COLR], R0;
END
# 56 instructions, 6 R-regs, 0 H-regs.
# End of program
62 lines, 0 errors.
```

Operation	Execution Count	Total Time	Average Time
Setup	51	36.28 s	711.36 ms
Readback of Data	51	34.62 s	678.88 ms
Matrix Vector Multiply	9600	322.16 s	33.56 ms
Dot Product	19891	156.24 s	7.85 ms
Scale-Add	28749	69.25 s	2.40 ms
Vector Precondition	9600	22.11 s	2.30 ms

The final results of the GPU-based implementation as compared to the CPU-based implementation are acceptable. Our fastest GPU-based implementation was 50% faster than our fastest CPU-based implementation. This is in spite of the fact we were unable to implement the optimal matrix preconditioner and thus had to make due with one that required three times as many iterations of the conjugate gradient algorithm. This improvement was also in spite of our inability to optimize the vector-vector dot product operation. The final timing results of our CPU-based and GPU-based implementations are listed in the table below:

	AMD Athlon 1800+	NVIDIA Quadro FX 1000	NVIDIA Quadro FX 2000
Diagonal Preconditioned	32.0s	12.3s	10.3s
Incomplete Cholesky Preconditioned	16.5s	n/a	n/a

Conclusion

It is not that difficult to convert CPU-based implementations of computationally intensive algorithms to the GPU via the new high level shader languages such as Cg. Because modern GPUs have significantly more floating point throughput than modern CPU such conversions have the potential of resulting in significant speed ups. In our case this speed-up was achieved in regards to some components of our general algorithm, i.e. the matrix-vector multiplication, but because of some limitations imposed by the GPU were we had to suffer penalties in other areas. It is likely that the limitations we encountered will be removed in future generations of GPUs and their supporting driver libraries.

Appendix A

The following is a calculation of the theoretical floating point operations per second (FLOP/s) possible on our three computational platforms. These calculations assume that it is useful to simply calculate multiplication after multiplication with no data access or storage requests in between:

Computational Platform	**Theoretical Maximum GFLOP/s**
AMD Athlon 1800+	1.5 GHz * 1 FLOP/s = 1.5 GFLOP/s
NVIDIA Quadro FX 1000	0.3 GHz * 8 pipelines * RGBA = 9.6 GFLOP/s
NVIDIA Quadro FX 2000	0.4 GHz * 8 pipelines * RGBA = 12.8 GFLOP/s

We can see that from this standpoint it is clear that the Quadro FX chip out performs the Athlon significantly in regards to floating point operations. Unfortunately, we were unable to realize the full speed up because of the slowness of the dot product operation.

Appendix B

The source code and assembly output for the main dot product Cg program.

SumReduce.cg

Cg Source Code

```
fragout_float main(
                vf30 In,
                uniform samplerRECT tex
                )
{
    fragout_float O;
       float2 pos = 2 * (In.TEX0.xy - 0.5) + 0.5;

       O.col = f4texRECT(tex, pos) +
               f4texRECT(tex, pos + float2(1, 0)) +
               f4texRECT(tex, pos + float2(0, 1)) +
               f4texRECT(tex, pos + float2(1, 1));

    return O;
}
```

Assembly Output

```
sumreduce.cg
!!FP1.0
# NV_fragment_program generated by NVIDIA Cg compiler
# cgc version 1.1.0003, build date Mar  4 2003  12:32:10
# command line args: -profile fp30
#vendor NVIDIA Corporation
#version 1.0.02
#profile fp30
#program main
#semantic main.tex
#var float4 In.WPOS : $vin.WPOS : WPOS : 0 : 1
#var float4 In.COL0 : $vin.COL0 : COL0 : 0 : 1
#var float4 In.COL1 : $vin.COL1 : COL1 : 0 : 1
#var float4 In.TEX0 : $vin.TEX0 : TEX0 : 0 : 1
#var float4 In.TEX1 : $vin.TEX1 : TEX1 : 0 : 1
#var float4 In.TEX2 : $vin.TEX2 : TEX2 : 0 : 1
#var float4 In.TEX3 : $vin.TEX3 : TEX3 : 0 : 1
#var float4 In.TEX4 : $vin.TEX4 : TEX4 : 0 : 1
#var float4 In.TEX5 : $vin.TEX5 : TEX5 : 0 : 1
#var float4 In.TEX6 : $vin.TEX6 : TEX6 : 0 : 1
#var float4 In.TEX7 : $vin.TEX7 : TEX7 : 0 : 1
#var samplerRECT tex :  : texunit 0 : 1 : 1
#var float4 col : $vout.COL : COL : -1 : 1
#var float depth : $vout.DEPR : DEPR : -1 : 1
ADDR R0.xy, f[TEX0].xyxx, -{0.5}.x;
MADR R0.xy, {2}.x, R0.xyxx, {0.5}.x;
ADDR R1.xy, R0.xyxx, {1, 0}.xyxx;
TEX R1, R1.xyxx, TEX0, RECT;
TEX R2, R0.xyxx, TEX0, RECT;
ADDR R1, R2, R1;
ADDR R2.xy, R0.xyxx, {0, 1}.xyxx;
TEX R2, R2.xyxx, TEX0, RECT;
ADDR R2, R1, R2;
ADDR R0.xy, R0.xyxx, {1, 1}.xyxx;
TEX R0, R0.xyxx, TEX0, RECT;
ADDR o[COLR], R2, R0;
END
# 12 instructions, 3 R-regs, 0 H-regs.
# End of program
21 lines, 0 errors.
```

Appendix C

The steps of our optimization of the Cg routines.

1) We attempted to use byte textures rather than floating point textures, in hopes of accelerating texture accesses, and reducing memory usage on the card. However, using byte textures required that we map from the [-128, 127] range to [0,1] and back again in the Cg code. Upon converting back to floating point textures we achieved an immediate speed up of approximately 3s (from 15.2 to 12.3).

2) Another bottleneck of our algorithm was calculating texture offsets in our matrix-vector multiplication routine. Several offsets must be calculated from the "current" position in the vector. These offsets must handle the wrapping of the textures from one line onto the next. Our initial naive implementation required both division and modulus operations for the calculation of each offset. By replacing this code with a "skewed" coordinate system, we were able to minimize these operations and achieve a speedup of around 3s from (24s to 21s). We also replaced the fmod operation with an equivalent that required a multiply by a pre-computed inverse rather than a divide.

Note that if GL_WRAP was available on the floating point textures we used, the fmod would be unnecessary, since the y-component would continue to be correct as we offset it, and the x-coordinate would wrap automatically.

To summarize the skewed system, given (x, y) we find (x, y - 0.5 + x / width). Then we can simply add our offsets (offset, offset/width) to the skewed coords, and then essentially find x = fmod(x, width).

3) Another flaw in our original implementation was creating float2 offsets in our matrix-vector multiply from individual float's that were passed into the fragment program. By passing in pre-constructed float2's we achieved a surprising speedup of nearly a second.

4) Our early test application began each run of the Cg algorithm (solving Ax = b) with the initial guess of x set equal to b. By replacing this with the solution for x on the previous time-step, we achieved an improvement of around 1 second.

5) By recognizing that the y-value in our "skewed" coordinate scheme implicitly contains the x-coordinate, we replaced our "pseudo" fmod equivalent with frac(y) * width, which gives the same result. This scheme breaks down with very large textures because the accuracy of the fractional part of the y-value decreases the larger the texture gets.

6) Our first version of the reduce operation for dot-product conceptually divided a texture into four components, and summed 4 corresponding element together. These elements were thus a distance of at least texturewidth/2 apart, which meant that the texture cache could not work effectively. By modifying it to use groups of 4 adjacent pixels, we saw a few tenths of a second speed up.

7) In order to get accurate timings for each component of the Cg algorithm, we added glFinish's after each operation. (Note that the glFinish's must be placed before the pbuffer that is being written to is deactivated, since each pbuffer possesses its own GL context.) By removing these glFinish's we get about a 0.7s improvement, although we lose the ability to measure individual operation times accurately.

8) A related issue we faced was that our initial timings pin-pointed dot-product readback as the major culprit in slowing down the program. However, this was only due to caching on the part of the card. When the code arrived at a glReadPixels, it was forced to complete all the operations up to that point, and thus appeared to be extremely slow. This lead us to change our code to perform the scalar

operations of the Cg algorithm on the card, using 1x1 pbuffers, in order to eliminate readback as much as possible. Naturally, this modification yielded little improvement, because readback wasn't as large a problem as we'd thought. Reverting to doing scalar operations in software gave us about a couple seconds improvement.

9) We experimented with loading our matrix into one large texture rather than 7 smaller textures. The idea was to optimize use of the texture cache, by Several issues made this unworkable. First, with large matrices, we run the risk of hitting the maximum texture size (on the Quadro FX 2000/1000 this is 2048x2048). Second, we must again deal with calculating offsets which introduces extra computation. Our trick from point 5 becomes more difficult to apply because of the large texture size. The offset issue would be resolved if we could use GL_WRAP, but again, since we used non-power-of-two textures we cannot use this option. It would be possible to use power-of-two textures along with byte-textures (see 1 above), although it would be a waste of space. We would lose the speed benefit of floating point textures so the trade-off is unlikely to be worth it.

10) Also for the dot-product, by stopping the repeated reduction fragment program calls in the card earlier on and reading back a small number of pixels, (around 16), we found that we achieved a minor speed improvement. Our original version continued reduction down to a single pixel, and the overhead of these calls negated the benefit of reducing the number of pixels to read back.

About Frantic Films

Frantic Films is headed by an experienced management team in President **Chris Bond**, Vice-President **Ken Zorniak** and CEO **Jamie Brown**. Together, they have assembled cutting-edge technology and top-tier talent to create a company that is dedicated to delivering precise, detailed and visually stunning effects to clients such as Warner Bros., Paramount Pictures, 20th Century Fox, and Regent Entertainment, to name a few.

Contact Frantic Films At:

Suite 300 - 70 Arthur Street
Winnipeg, Manitoba, Canada R3B 1G7
tel: (204) 949-0070
fax: (204) 949-0050

http://www.franticfilms.com/