



*n*VIDIA™

Vertex Program Modules

Erik Lindholm
erikl@nvidia.com

Vertex Program Modules

- Some interesting case studies. These programs are fairly optimized, but better is possible. Can always add/subtract power series terms for higher/lower accuracy. Make sure you optimize the coefficients for the number of terms.
- Cross product
- LOG
- EXP
- COS/SIN
- 4x4 LUT
- 4x4 Matrix Inversion
- MANDELBROT



NVIDIA™

Cross Product

- $\begin{vmatrix} i & j & k \\ R0.x & R0.y & R0.z \\ R1.x & R1.y & R1.z \end{vmatrix} = (R0.y * R1.z - R1.y * R0.z)i +$
- $(R0.z * R1.x - R1.z * R0.x)j +$
- $(R0.x * R1.y - R1.x * R0.y)k;$
- Or $(R0.yzx * R1.zxy - R1.yzx * R0.zxy)$
- **MUL R2,R0.yzxw,R1.zxyw;**
- **MAD R2,-R1.yzxw,R0.zxyw,R2**



NVIDIA

LOG Algorithm

- Reduce input to exponent and mantissa
- Power series evaluation using mantissa
- Add exponent to power series result
- $\text{Log}_2(m \cdot 2^e) = \log_2(m) + e$
- Note that most ops are scalar, would be cheap to do 4 log() in parallel.



nVIDIA™

LOG Constants

Power series used here is

$$\begin{aligned}\log_2(1.0+x) = & c[1].x*x + c[1].y*x^2 + c[1].z*x^3 + \\ & c[1].w*x^4 + c[0].x*x^5 + c[0].y*x^6 + \\ & c[0].z*x^7 + c[0].w*x^8; \quad /* 0 \leq x \leq 1 */\end{aligned}$$

C[0] 2.41873696e-01, -1.37531206e-01, 5.20646796e-02, -9.31049418e-03

C[1] 1.44268966e+00, -7.21165776e-01, 4.78684813e-01, -3.47305417e-01

C[2] 1.0, 0.0, 0.0, 0.0

C simulation gives a max absolute error of less than 1.8e-7 for the Mantissa input.



NVIDIA

LOG

```
LOG R0,v[2].x;  
ADD R0.y,R0.y,-c[2].x;          /* subtract 1.0 */  
MAD R0.w,c[0].w,R0.y,c[0].z; /* start power series */  
MAD R0.w,R0.w,R0.y,c[0].y;  
MAD R0.w,R0.w,R0.y,c[0].x;  
MAD R0.w,R0.w,R0.y,c[1].w;  
MAD R0.w,R0.w,R0.y,c[1].z;  
MAD R0.w,R0.w,R0.y,c[1].y;  
MAD R0.w,R0.w,R0.y,c[1].x;  
MAD R0,R0.w,R0.y,R0.x;        /* exponent add */
```



NVIDIA

EXP Algorithm

- Reduce input to integer and fraction
- Power series evaluation using fraction
- Multiply (2^{integer}) with power series result
- $\text{Exp2}(i + f) = (2^i) * \text{exp2}(f)$
- Note that most ops are scalar, would be cheap to do 4 exp() in parallel.



nVIDIA™

EXP Constants

Power series used here is

$$\text{exp2}(x) = 1.0 / (c[5].x + c[5].y * x + c[5].z * x^2 + \\ c[5].w * x^3 + c[4].x * x^4 + c[4].y * x^5 + \\ c[4].z * x^6 + c[4].w * x^7); \quad /* 0 \leq x \leq 1 */$$

C[4] 9.61597636e-03, -1.32823968e-03, 1.47491097e-04, -1.08635004e-05

C[5] 1.00000000e+00, -6.93147182e-01, 2.40226462e-01, -5.55036440e-02

C Simulation gives a max absolute error of less than 3.8e-7 for the fractional input.



nVIDIA™

EXP

```
EXP R0,v[2].x;  
MAD R0.w,c[4].w,R0.y,c[4].z; /* start power series */  
MAD R0.w,R0.w,R0.y,c[4].y;  
MAD R0.w,R0.w,R0.y,c[4].x;  
MAD R0.w,R0.w,R0.y,c[5].w;  
MAD R0.w,R0.w,R0.y,c[5].z;  
MAD R0.w,R0.w,R0.y,c[5].y;  
MAD R0.w,R0.w,R0.y,c[5].x;  
RCP R0.w,R0.w;  
MUL R0,R0.w,R0.x;          /* multiply in 2^i */
```



nVIDIA

Power Series Variant

- Another option is to use DP4 ops to build up a power series. Below is a 16 term power series.
- DST R0,X,X;
- MUL R0,R0.xzyw,R0.xxy; /* 1,x,x^2,x^3 */
- DP4 R1.x,R0,c[ABCD];
- DP4 R1.y,R0,c[EFGH];
- DP4 R1.z,R0,c[IJKL];
- DP4 R1.w,R0,c[MNOP];
- MUL R0,R0,R0;
- MUL R0,R0,R0; /* 1,x^4,x^8,x^12 */
- DP4 R0,R0,R1;



NVIDIA

COS/SIN Algorithm

- Reduce input to normalized range 0.0-1.0 (0.0-2PI)
- Classify input into 3 ranges:
 - 0.0 - 0.25, 0.25 – 0.75, 0.75 – 1.0
- Reduce input for each range to +/-0.25 by translation. Use negative coefficients if needed.
- Evaluate power series for three ranges in parallel
- Select correct range with dp3
- SIN is just a shifted COS



nVIDIA™

COS/SIN Constants

Power series used here is

$$\begin{aligned}\cos(x) = & c[4].zwzw + c[4].xyxy * x^2 + \\ & c[3].zwzw * x^4 + c[3].xyxy * x^6 + \\ & c[2].zwzw * x^8 + c[2].xyxy * x^{10};\end{aligned}$$

/* -0.25<=x<=0.25 */

C[0] 0.0,0.5,1.0,0.0

C[1] 0.25,-9.0,0.75,1.0/(2*PI)

C[2] 24.9808039603,-24.9808039603, -60.1458091736, 60.1458091736

C[3] 85.4537887573, -85.4537887573a6, -64.9393539429, 64.9393539429

C[4] 19.7392082214, -19.7392082214, -1.0,1.0

C Simulation gives a max absolute error of less than 1.8e-7 for the normalized input.



NVIDIA

COS

```
MUL R1.x,c[1].w,v[2].x;    /* normalize input */
EXP R1.y,R1.x;             /* and extract fraction */
SLT R2.x,R1.y,c[1];        /* range check: 0.0 to 0.25*/
SGE R2.yz,R1.y,c[1];       /* range check: 0.75 to 1.0 */
DP3 R2.y,R2,c[4].zwzw;     /* range check: 0.25 to 0.75 */
ADD R0.xyz,-R1.y,c[0];     /* range centering */
MUL R0,R0,R0;
MAD R1,c[2].xyxy,R0,c[2].zwzw; /* start power series */
MAD R1,R1,R0,c[3].xyxy;
MAD R1,R1,R0,c[3].zwzw;
MAD R1,R1,R0,c[4].xyxy;
MAD R1,R1,R0,c[4].zwzw;
DP3 R0,R1,-R2;             /* range extract */
```



NVIDIA

SIN

```
MAD R1.x,c[1].w,v[2].x,-c[1].x;    /* only difference from COS */
EXP R1.y,R1.x;
SLT R2.x,R1.y,c[1];
SGE R2.yz,R1.y,c[1];
DP3 R2.y,R2,c[4].zwzw;
ADD R0.xyz,-R1.y,c[0];
MUL R0,R0,R0;
MAD R1,c[2].xyxy,R0,c[2].zwzw;
MAD R1,R1,R0,c[3].xyxy;
MAD R1,R1,R0,c[3].zwzw;
MAD R1,R1,R0,c[4].xyxy;
MAD R1,R1,R0,c[4].zwzw;
DP3 R0,R1,-R2;
```



NVIDIA™

4x4 LUT Algorithm

- Do a lerped 4x4 lookup from constants
- Assume repeat mode on u,v. Replicate data set to make repeat cheaper.
- Easy to have larger v size
- Harder to have u size larger than 4
- Clamp mode is very similar. Clamp source (u,v) using MIN/MAX and replicate last row instead of first row of constant blocks.



NVIDIA™

4x4 LUT Constants

```
C[0] 1.0, 0.0, 0.0, 0.0      /* extraction masks */
C[1] 0.0, 1.0, 0.0, 0.0
C[2] 0.0, 0.0, 1.0, 0.0
C[3] 0.0, 0.0, 0.0, 1.0
C[4] 1.0, 0.0, 0.0, 0.0      /* repeat: copy of first row */

C[ 8] 0.0, 0.2, 0.2, 0.0);    /* LUT data */
C[ 9] 0.2, 0.8, 0.8, 0.2);
C[10] 0.2, 0.8, 0.8, 0.2);
C[11] 0.0, 0.2, 0.2, 0.0);
C[12] 0.0, 0.2, 0.2, 0.0);    /* repeat: copy of first row */

C[32] -0.5/4.0, -0.5/4.0, 0.0, 1.0 /* centering */
C[33] 4.0, 4.0, 0.0, 0.0          /* scaling */
```



NVIDIA™

4x4 LUT 1/3

```
ADD R0,v[2],c[32]; /* adjust sampling point */
EXP R1.y,R0.x;      /* repeat mode on u */
MOV R1.x,R1.y;
EXP R1.y,R0.y;      /* repeat mode on v */
MUL R4,R1,c[33];    /* scale by size */
EXP R5.y,R4.x;      /* u fraction */
MOV R5.x,R5.y;
EXP R5.y,R4.y;      /* v fraction */
```



nVIDIA™

4x4 LUT 2/3

```
ARL A0.x,R4.y;          /* v base address */
MOV R0,c[A0.x+8];        /* first u,v quad read */
MOV R1,c[A0.x+9];        /* second u,v quad read */
ARL A0.x,R4.x;          /* u base mask address */
DP4 R2.x,R0,c[A0.x+0];   /* mask off (u,v) */
DP4 R2.y,R1,c[A0.x+0];   /* mask off (u,v+1) */
DP4 R2.z,R0,c[A0.x+1];   /* mask off (u+1,v) */
DP4 R2.w,R1,c[A0.x+1];   /* mask off (u+1,v+1) */
```



NVIDIA

4x4 LUT 3/3

```
ADD R4.xy,R2.zwzw,-R2.xyxy;      /* start bilerp */  
MAD R4.xy,R5.x,R4.xyxy,R2.xyxy;  
ADD R4.z,R4.y,-R4.x;  
MAD R0,R5.y,R4.z,R4.x;          /* final sample */
```



NVIDIA™

4x4 Matrix Inversion

- Invert a full 4x4 matrix stored in constant memory locations `c[0]-c[3]`
- Output the full 4x4 inverse matrix into R8-R11



NVIDIA™

4x4 Matrix Inversion Code 1/4

```
MOV R4,c[0];      /* move source matrix into regs */  
MOV R5,c[1];  
MOV R6,c[2];  
MOV R7,c[3];
```



nVIDIA™

4x4 Matrix Inversion Code 2/4

```
MUL R0,R6.wyzx,R7.zwyx;      /* generate first half of matrix */
MUL R1,R4.wyzx,R5.zwyx;
MUL R2,R6.wxzy,R7.zwxy;
MUL R3,R4.wxzy,R5.zwxy;
MAD R0,R6.zwyx,R7.wyzx,-R0;
MAD R1,R4.zwyx,R5.wyzx,-R1;
MAD R2,R6.zwxy,R7.wxzy,-R2;
MAD R3,R4.zwxy,R5.wxzy,-R3;
DP3 R8.x,R5.yzwx,R0;
DP3 R9.x,R4.yzwx,-R0;
DP3 R10.x,R7.yzwx,R1;
DP3 R11.x,R6.yzwx,-R1;
DP3 R8.y,R5.xzwy,-R2;
DP3 R9.y,R4.xzwy,R2;
DP3 R10.y,R7.xzwy,-R3;
DP3 R11.y,R6.xzwy,R3;
```



NVIDIA™

4x4 Matrix Inversion Code 3/4

```
MUL R0,R6.wxyz,R7.ywxz;      /* generate second half of matrix */
MUL R1,R4.wxyz,R5.ywxz;
MUL R2,R6.zxyw,R7.yzxw;
MUL R3,R4.zxyw,R5.yzxw;
MAD R0,R6.ywxz,R7.wxyz,-R0;
MAD R1,R4.ywxz,R5.wxyz,-R1;
MAD R2,R6.yzxw,R7.zxyw,-R2;
MAD R3,R4.yzxw,R5.zxyw,-R3;
DP3 R8.z,R5.xywz,R0;
DP3 R9.z,R4.xywz,-R0;
DP3 R10.z,R7.xywz,R1;
DP3 R11.z,R6.xywz,-R1;
DP3 R8.w,R5.xyzw,-R2;
DP3 R9.w,R4.xyzw,R2;
DP3 R10.w,R7.xyzw,-R3;
DP3 R11.w,R6.xyzw,R3;
```



NVIDIA™

4x4 Matrix Inversion Code 4/4

```
DP4 R7.w,R8,R4;    /* determinant */
RCP R7.w,R7.w;      /* reciprocate determinant */
MUL R8,R8,R7.w;     /* multiply into matrix */
MUL R9,R9,R7.w;
MUL R10,R10,R7.w;
MUL R11,R11,R7.w;
```



NVIDIA™

Mandelbrot Algorithm

- $(X+iY) = (x+iy)*(x+iy) + (x_0+iy_0)$
- $X = x*x - y*y + x_0$
- $Y = 2xy + y_0$
- Final magnitude is $X*X + Y*Y$
- Send one vertex per pixel
- Image has 21 iterations.



nVIDIA™

Mandelbrot Constants

C[0] 1.0,-1.0, 0.0, 0.5

C[1] 0.2, 1.0, 0.4, 0.0



nVIDIA™

Mandelbrot

/* init */

MUL R0,v[0].xyzy,c[0].xxxy; */* R0 = (x0,y0,0.0,-y0) */*

MUL R4,R0.xzyw,c[0].xxww; */* R4 = (x0,0.0,0.5*y0,-0.5*y0) */*

/* iterate many times (by repeating block) */

MAD R1,R0.xyxx,R0.xyyw,R4; */* x*x+x0,y*y,xy+y0/2,-xy-y0/2 */*

ADD R0.xyw,R1.xzww,-R1.ywwz; */* x*x-y*y+x0,2xy+y0,0.0,-2xy-y0 */*

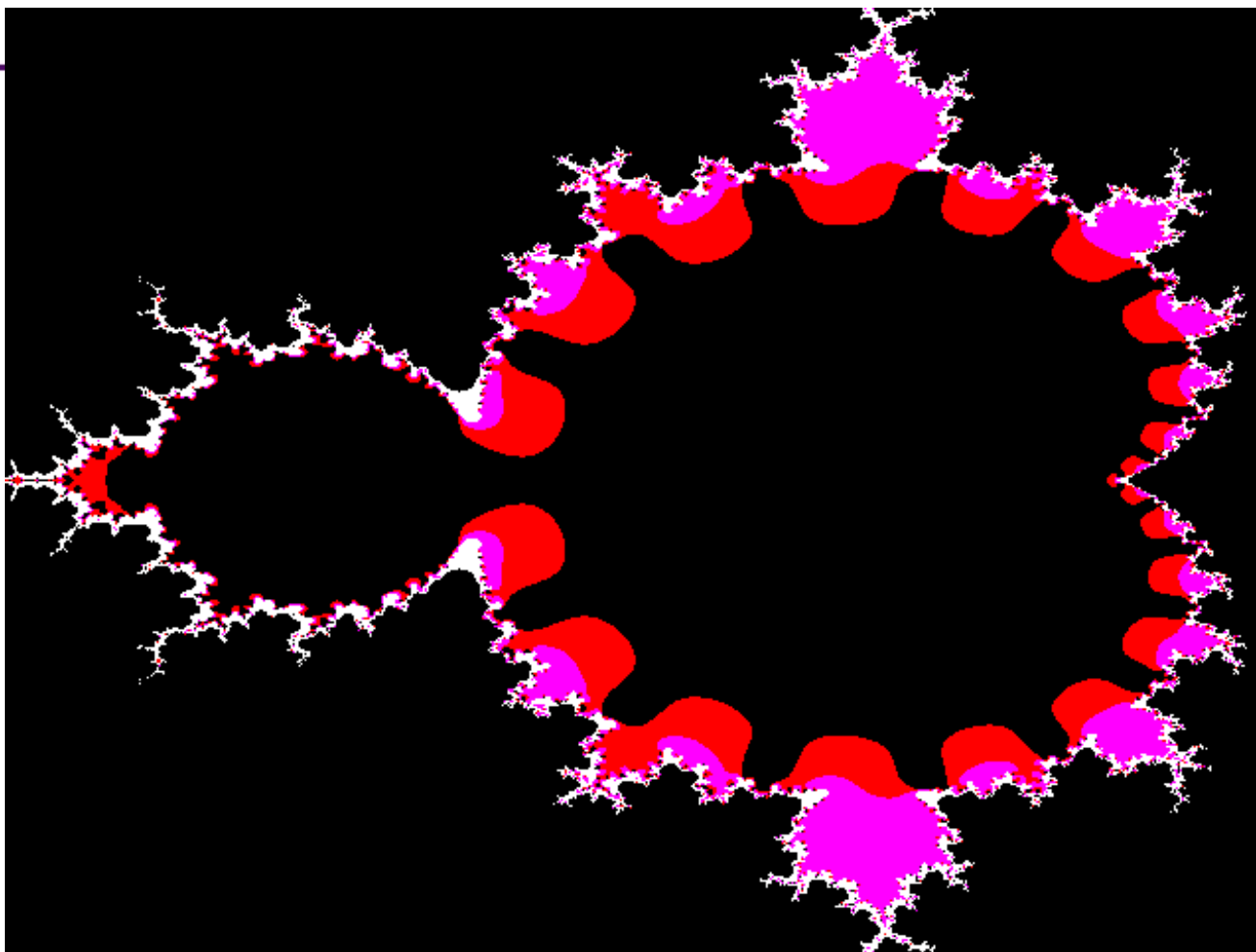
/* color output */

DP3 R1.w,R0,R0; */* x*x + y*y + 0*0 */*

SGE o[COL0].xyz,R1.w,c[1];



nVIDIA



nVIDIA™