Looking at OpenCL Assembly Code

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License

How to Extract the OpenCL Assembly Language

size_t size;
status = clGetProgramInfo( Program, CL_PROGRAM_BINARY_SIZES, sizeof(size_t), &size, NULL);
PrintCLError( status, "clGetProgramInfo (1):" );

unsigned char * binary = new unsigned char [ size ];
status = clGetProgramInfo( Program, CL_PROGRAM_BINARIES, size, &binary, NULL );
PrintCLError( status, "clGetProgramInfo (2):" );

FILE * fpbin = fopen( CL_BINARY_NAME, "wb" );
if( fpbin == NULL )
{
    fprintf( stderr, "Cannot create '%s'
", CL_BINARY_NAME );
}
else
{
    fwrite( binary, 1, size, fpbin );
    fclose( fpbin );
}
delete [] binary;

This binary can then be used in a call to clCreateProgramWithBinary()
typedef float4 point;
typedef float4 vector;
typedef float4 color;
typedef float4 sphere;

constant float4 G = (float4) ( 0., -9.8, 0., 0. );
constant float DT = 0.1;
constant sphere Sphere1 = (sphere) (-100., -800., 0., 600. );

kernel void Particle(  global point * dPobj,  global vector * dVel,  global color * dCobj )
{
    int gid = get_global_id( 0 ); // particle #
    point p = dPobj[gid];
    vector v = dVel[gid];

    point pp = p + v*DT + .5*DT*DT*G; // p'
    vector vp = v + G*DT; // v'

    dPobj[gid] = pp;
    dVel[gid] = vp;
}
particles.cl, III

```cpp
vector Bounce( vector in, vector n )
{
    n.w = 0.;
    n = normalize( n );
    vector out = in - 2. * n * dot( in.xyz, n.xyz );
    out.w = 0.;
    return out;
}

vector BounceSphere( point p, vector v, sphere s )
{
    vector n;
    n.xyz = fast_normalize( p.xyz - s.xyz );
    n.w = 0.;
    return Bounce( in, n );
}
```

NVIDIA OpenCL Assembly Language Sample

```
ld.global.v4.f32 {%f188, %f189, %f190, %f191}, [%r1]; // load dPobj[ gid ]
ld.global.v4.f32 {%f156, %f157, %f158, %f159}, [%r2]; // load dVel[ gid ]
mov.f32 %f17, 0f3DCCCCCD; // put DT (a constant) → register f17
fma.rn.f32 %f248, %f156, %f17, %f188; // (p + v*DT).x → f248
fma.rn.f32 %f249, %f157, %f17, %f189; // (p + v*DT).y → f249
fma.rn.f32 %f250, %f158, %f17, %f190; // (p + v*DT).z → f250
mov.f32 %f18, 0fBD48B43B; // .5 * G.y * DT * DT (a constant) → f18
mov.f32 %f19, 0f00000000; // 0., for .x and .z (a constant) → f19
add.f32 %f256, %f248, %f19; // (p + v*DT).x + 0. → f256
add.f32 %f257, %f249, %f18; // (p + v*DT).y + .5 * G.y * DT * DT → f257
add.f32 %f258, %f250, %f19; // (p + v*DT).z + 0. → f258
mov.f32 %f20, 0fBF7AE148; // G.y * DT (a constant) → f20
add.f32 %f264, %f156, %f19; // vx + 0. → f264
add.f32 %f265, %f157, %f20; // vy + G.y * DT → f265
add.f32 %f266, %f158, %f19; // vz + 0. → f266
```
Things Learned from Examining OpenCL Assembly Language

• The points, vectors, and colors were typedef’ed as float4’s, but the compiler realized that they were being used as float3’s, and didn’t bother with the 4th element.

• The floatn’s were not SIMD’ed. (We actually knew this already, since NVIDIA doesn’t support vector operations in their GPUs.) There is still an advantage in coding this way, even if just for readability.

• The function calls were all in-lined. (This makes sense – the OpenCL spec says “no recursion”, which implies “no stack”, which would make function calls difficult.)

• Defining G, DT, and Sphere1 as constant memory types was a mistake. It got the correct results, but the compiler didn’t take advantage of them being constants. Changing them to type const threw compiler errors because of their global scope. Changing them to const and moving them into the body of the kernel function Particle did result in compiler optimizations.

• The sqrt(x^2+y^2+z^2) assembly code is amazingly involved. I suspect it is an issue of maintaining highest precision. Use fast_sqrt(), fast_normalize(), and fast_length() when you can.

• The compiler did not do a good job with expressions-in-common. I had really hoped it would figure out that detecting if a point was in a sphere and determining the unitized surface normal at that point were mostly the same operation, but it didn’t.

• There is a 4-argument fused-multiply-add instruction (d = a*b + c, one instruction in hardware). The compiler took great advantage of it.

Fused Multiply-Add

Many scientific and engineering computations take the form:

\[ d = a + (b*c); \]

A “normal” multiply-add compilation would handle this as:

\[ \text{tmp} = b*c; \]
\[ d = a + \text{tmp}; \]

A “fused” multiply-add does it all at once, that is, when the low-order bits of B*C are ready, they are immediately added into the low-order bits of A at the same time the higher-order bits of B*C are being multiplied.

Consider a Base 10 example: 789 + (123*456)

\[
\begin{array}{c}
123 \\
x 456 \\
\hline
738 \\
615 \\
\hline
492 \\
\hline
+ 789 \text{ Can start adding the 9 the moment the 8 is produced!} \\
\hline
56,877
\end{array}
\]

Note: “Normal” A+(B*C) ≠ “FMA” A+(B*C)