Looking at OpenCL Assembly Code

mjb@cs.oregonstate.edu

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

How to Extract the OpenCL Assembly Language

```c
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( )

particles.cl, I

```c
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. );
```

particles.cl, II

```c
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;
}
```
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 ); 
}

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 4\text{th} element.
- The float3's were not SIMD'ed. (We actually knew this already, since NVIDIA doesn't supported 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 \text{const} threw compiler errors because of their global scope. Changing them to \text{const} and moving them into the body of the kernel function Particle did result in compiler optimizations.
- The \text{sqrt}(x^2+y^2+z^2) assembly code is amazingly involved. I suspect it is an issue of maintaining highest precision. Use \text{fast\_sqrt()}, \text{fast\_normalize()}, and \text{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 \text{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.