High Level Languages for GPUs

Ian Buck
NVIDIA

High Level Shading Languages

- **Cg**, **HLSL**, & **OpenGL Shading Language**
  - **Cg**:
  - **HLSL**:
  - **OpenGL Shading Language**:

Compilers: CGC & FXC

- HLSL and Cg are syntactically almost identical
  - Exception: Cg 1.3 allows shader “interfaces”, unsized arrays
- **Command line compilers**
  - Microsoft’s FXC.exe
    - Compiles to DirectX vertex and pixel shader assembly only
  - NVIDIA’s CGC.exe
    - Compiles to everything
    - cgc -profile ps_2_0 myshader.cg
  - Can generate very different assembly!
    - Driver will recompile code
  - Compliance may vary

Babelshader

- Converts between DirectX pixel shaders and OpenGL shaders
- Allows OpenGL programs to use DirectX HLSL compilers to compile programs into ARB or fp30 assembly.
- Enables fair benchmarking competition between the HLSL compiler and the Cg compiler on the same platform with the same demo and driver.

GPGPU Languages

- **Why do want them?**
  - Make programming GPUs easier!
    - Don’t need to know OpenGL, DirectX, or ATI/NV extensions
    - Simplify common operations
    - Focus on the algorithm, not on the implementation
- **Sh**
  - University of Waterloo
    - http://libsh.org
- **Brook**
  - Stanford University
    - http://graphics.stanford.edu/projects/brookgpu

GPGPU

Sh Features

- Implemented as C++ library
  - Use C++ modularity, type, and scope constructs
  - Use C++ to metaprogram shaders and kernels
  - Use C++ to sequence stream operations
- Operations can run on
  - GPU in JIT compiled mode
  - CPU in immediate mode
  - CPU in JIT compiled mode
- Can be used
  - To define shaders
  - To define stream kernels
- No glue code
  - Declare parameters
  - Declare textures
- Memory management
  - Automatically uses puffers and/or uberbuffers
  - Textures are shadowed and act like arrays on both the CPU and GPU
  - Textures can encapsulate interpretation code
  - Programs can encapsulate texture data
- Program manipulation
  - Introduction
  - Uniform/varying conversion
  - Program specialization
  - Program composition
  - Program concatenation
  - Interface adaptation
Sh Fragment Shader

```c
fsh = SH_BEGN_PROGRAM("gpu:fragment") { 
    ShInputNormal3f nv; // normal (VCS)
    ShInputVector3f lv; // light-vector (VCS)
    ShInputVector3f vv; // view vector (VCS)
    ShInputColor3f ec; // irradiance
    ShInputTexCoord2f u; // texture coordinate
    ShOutputColor3f fc; // fragment color

    vv = normalize(vv);
    lv = normalize(lv);
    nv = normalize(nv);
    ShVector3f hv = normalize(lv + vv);
    fc = kd(u) * ec;
    fc += ks(u) * pow(pos(hv|nv), spec_exp);
} SH_END;
```

Streams and Channels

- **ShChannel<element_type>**
  - Sequence of elements of given type
- **ShStream**
  - Sequence of channels
  - Combine channels with &:
    ```c
    ShStream s = a & b & c;
    ```
  - Refers to channels, does not copy
  - Single channel also a stream
- Apply programs to streams with `<<`
  ```c
  ShStream t = (x & y & z);
  s = p << t;
  ```

Stream Processing: Particles

```c
// SETUP (define particle state update kernel)
p = SH_BEGN_PROGRAM("gpu:stream") { 
    ShInOutPoint3f Ph, Pt;
    ShInOutVector3f V;
    ShInputVector3f A;
    ShInputAttrib1f delta;
    Pt = Ph;
    A = cond(abs(Ph(1)) < 0.05, ShVector3f(0.,0.,0.), A);
    V += A * delta;
    V = cond((V|V) < 1., ShVector3f(0., 0., 0.), V);
    Ph += (V + 0.5*A)*delta;
    ShAttrib1fmu(0.1), eps(0.3);
    for (i = 0; i < num_spheres; i++) {
        ShPoint3f C = spheres[i].center;
        ShAttrib1f r = spheres[i].radius;
        ShVector3f PhC = Ph - C;
        ShVector3f N = normalize(PhC);
        ShPoint3f S = C + N*r;
        ShAttrib1f collide = ((PhC|PhC) < r*r)*((V|N) < 0);
        Ph = cond(collide,
          Ph - 2.0*((Ph - S)|N)*N, Ph);
        ShVector3f Vn = (V|N)*N;
        ShVector3f Vt = V - Vn;
        V = cond(collide,
          (1.0 - mu)*Vt - eps*Vn, V);
    }
    under = Ph(1) < 0.;
    Ph = cond(under,
      Ph * ShAttrib3f(1.,0.,1.), Ph);
    ShVector3f Vn = V * ShAttrib3f(0.,1.,0.);
    ShVector3f Vt = V - Vn;
    V = cond(under,
      (1.0 - mu)*Vt - eps*Vn, V);
    Ph(1) = cond(min(under,(V|V)<0.1),
      ShPoint1f(0.), Ph(1));
    ShVector3f dt = Pt - Ph;
    Pt = cond((dt|dt) < 0.02, Pt +
      ShVector3f(0.0, 0.02, 0.0), Pt);
} SH_END;
```

Streams

- Collection of records requiring similar computation
  - particle positions, voxels, FEM cell, ...
    ```c
    Ray r<200>;
    float3 velocityfield<100,100,100>;
    ```
- Similar to arrays, but...
  - Index operations disallowed: position[i]
  - Read/write stream operators
    - `streamRead (r, r_ptr);`
    - `streamWrite (velocityfield, v_ptr);`

Brook: General Purpose Streaming Language

- **Stream programming model**
  - GPU = streaming coprocessor
- **C with stream extensions**
- **Cross platform**
  - ATI & NVIDIA
  - OpenGL & DirectX
  - Windows & Linux
Kernels
• Functions applied to streams
  - similar to for_all construct
  - no dependencies between stream elements

```c
kernel void foo (float a<>,
                float b<>,
                out float result<>) {
  result = a + b;
}
float a<100>;
float b<100>;
float c<100>;
foo(a,b,c);
for (i=0; i<100; i++)
c[i] = a[i]+b[i];
```

Kernels
• Kernel arguments
  - input/output streams

```c
kernel void foo (float a<>,
                float b<>,
                out float result<>)
  {
    result = a + b;
  }
```

Kernels
• Kernel arguments
  - input/output streams
  - gather streams

```c
kernel void foo (...,
                float array[])
  {
    a = array[i];
  }
```

Kernels
• Kernel arguments
  - input/output streams
  - gather streams
  - iterator streams

```c
kernel void foo (...,
                iter float n<>)
  {
    a = n + b;
  }
```

Kernels
• Kernel arguments
  - input/output streams
  - gather streams
  - iterator streams
  - constant parameters

```c
kernel void foo (...,
                float c)
  {
    a = c + b;
  }
```

Kernels
• Ray triangle intersection

```c
kernel void krnIntersectTriangle(Ray ray<>,
                                  Triangle tris[],
                                  RayState oldraystate<>,
                                  GridTrilist trilist[],
                                  out Hit candidatehit<>){
  float $d$, $e$, $f$;
  float $X$, $Y$, $Z$;
  if(oldraystate.state.y > 0) {
    idx = trilist[oldraystate.state.w].trinum;
    edge1 = tris[idx].v1 - tris[idx].v0;
    edge2 = tris[idx].v2 - tris[idx].v0;
    pvec = cross(ray.d, edge2);
    det = dot(edge1, pvec);
    inv_det = 1.0/|det|;
    tvec = ray.o - tris[idx].v0;
    candidatehit.data.y = dot(tvec, pvec) * inv_det;
    qvec = cross(tvec, edge1);
    candidatehit.data.z = dot(ray.d, qvec) * inv_det;
    candidatehit.data.x = dot(edge2, qvec) * inv_det;
    candidatehit.data.w = idx;
  } else {
    candidatehit.data = float4(0,0,0,-1);
  }
}
```
Reductions

- Compute single value from a stream
  - associative operations only

  ```c
  float sum(float a, float r) {
    r += a;
  }
  ```

  ```c
  float a[100];
  float r;
  sum(a, r);
  ```

- Multi-dimension reductions
  - stream “shape” differences resolved by reduce function

  ```c
  float sum(float a, float r) {
    r += a;
  }
  ```

  ```c
  float a[20];
  float r[5];
  sum(a, r);
  ```

Stream Repeat & Stride

- Kernel arguments of different shape
  - resolved by repeat and stride

  ```c
  kernel void foo(float a, float b, out float c) {
    float x[20];
    float y[10];
    foo(a, b, c);
  }
  ```

  ```c
  float a[20];
  float b[10];
  foo(a, b, c);
  ```

Matrix Vector Multiply

```c
kernel void mul(float a, float b, out float c) {
  result = a * b;
}
```
Running Brook

• BRT\_RUNTIME selects platform
  - CPU Backend: BRT\_RUNTIME = cpu
  - OpenGL ARB Backend: BRT\_RUNTIME = ogl
  - DirectX9 Backend: BRT\_RUNTIME = dx9

Runtime

• Accessing stream data for graphics aps
  - Brook runtime api available in C++ code
  - autogenerated .hpp files for brook code

    brook::initialize( "dx9", (void*)device );
    // Create streams
    fluidStream = stream::create<float4>( kFluidSize, kFluidSize );
    normalStream = stream::create<float3>( kFluidSize, kFluidSize );
    // Set a handle to the texture being used by
    // the normal stream as a backing store
    normalTexture = (IDirect3DTexture9*)
      normalStream->getIndexedFieldRenderData(0);
    // Call the simulation kernel
    simulationKernel( fluidStream0, fluidStream0, controlConstant,
                     fluidStream1 );

Applications

ray-tracer
segmentation
flt edge detect
linear algebra

Evaluation

Compared against:
• Intel Math Library
• Atlas Math Library
• Cached blocked segmentation
• FFTW
• Waid SSE Ray-Triangle code

Efficiency

Brook version within 80% of hand-coded GPU version

Brook for GPUs

• Release v0.3 available on Sourceforge
• Project Page
  - http://graphics.stanford.edu/projects/brook
• Source
  - http://www.sourceforge.net/projects/brook
• Brook for GPUs: Stream Computing on Graphics Hardware
  Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, Pat Hanrahan