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
    • fxc /Tps_2_0 myshader.hlsl
  - 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**

http://graphics.stanford.edu/~danielrh/babelshader.html

- 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.sourceforge.net
  http://libsh.org

• Brook
  Stanford University
  http://brook.sourceforge.net
  http://graphics.stanford.edu/projects/brookgpu
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 pbuffers 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**
  - Introspection
  - Uniform/varying conversion
  - Program specialization
  - Program composition
  - Program concatenation
  - Interface adaptation
Sh Fragment Shader

```sh
fsh = SH_BEGIN_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 &:
    ```
    ShStream s = a & b & c;
    ```
  - *Refers* to channels, does *not* copy
  - Single channel also a stream

- **Apply programs to streams with <<**
  ```
  ShStream t = (x & y & z);
  s = p << t;
  (a & b & c) = p << (x & y & z);
  ```
// SETUP (define particle state update kernel)

```
p = SH_BEGIN_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; ShAttrib1f mu(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);
    }
}
```

ShAttrib1f 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);
}

// define state stream
ShStream state =
    (pos & pos_tail & vel);
// curry p with state and parameters
ShProgram update =
    p << state << gravity << delta;

...
Stream Processing: Particles
Brook: General Purpose Streaming Language

• Stream programming model
  - GPU = streaming coprocessor

• C with stream extensions

• Cross platform
  - ATI & NVIDIA
  - OpenGL & DirectX
  - Windows & Linux
Streams

• Collection of records requiring similar computation
  - particle positions, voxels, FEM cell, ...

  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);
Kernels

- Functions applied to streams
  - similar to for_all construct
  - no dependencies between stream elements

```cpp
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

    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 idx, det, inv_det;
    float3 edge1, edge2, pvec, tvec, qvec;
    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 = 1.0f/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

reduce void sum (float a<>,
    reduce float r<>)
  
    r += a;
}

float a<100>;
float r;

sum(a,r);

\[
\begin{align*}
  r &= a[0]; \\
  \text{for} & \ (\text{int} \ i=1; \ i<100; \ i++) \ \r
  r &=+ \ a[i];
\end{align*}
\]
Reductions

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

```c
reduce void sum (float a<>,
    reduce float r<>)
{
    r += a;
}
```

```c
float a<20>;
floát r<5>;
sum(a,r);
```
```c
for (int i=0; i<5; i++)
    r[i] = a[i*4];
for (int j=1; j<4; j++)
    r[i] += a[i*4 + j];
```
Stream Repeat & Stride

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

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

float a<20>;
float b<5>;
float c<10>;
foo(a,b,c);
```

```
foo(a[0],  b[0],  c[0])
foo(a[2],  b[0],  c[1])
foo(a[4],  b[1],  c[2])
foo(a[6],  b[1],  c[3])
foo(a[8],  b[2],  c[4])
foo(a[10], b[2],  c[5])
foo(a[12], b[3],  c[6])
foo(a[14], b[3],  c[7])
foo(a[16], b[4],  c[8])
foo(a[18], b[4],  c[9])
```
Matrix Vector Multiply

kernel void mul (float a<>, float b<>,
    out float result<>) {
    result = a*b;
}

reduce void sum (float a<>,
    reduce float result<>) {
    result += a;
}

float matrix<20,10>;
float vector<1, 10>;
float tempmv<20,10>;
float result<20, 1>;

mul(matrix,vector,tempmv);
sum(tempmv,result);

\[
\begin{bmatrix} M \\ V \end{bmatrix} \begin{bmatrix} V \\ V \end{bmatrix} = \begin{bmatrix} T \end{bmatrix}
\]
Matrix Vector Multiply

kernel void mul (float a<>, float b<>,
    out float result<>) {
    result = a*b;
}
reduce void sum (float a<>,
    reduce float result<>) {
    result += a;
}

float matrix<20,10>;
float vector<1, 10>;
float tempmv<20,10>;
float result<20, 1>;

mul(matrix,vector,tempmv);
sum(tempmv,result);
Running Brook

• Compiling .br files

Brook CG Compiler
Version: 0.2  Built: Jul 24 2005, 11:36:29
brcc [-hvndktyAN] [-o prefix] [-w workspace] [-p shader ]
    [-f compiler] [-a arch] foo.br

-h    help (print this message)
-v    verbose (print intermediate generated code)
-n    no codegen (just parse and reemit the input)
-d    debug (print cTool internal state)
-k    keep generated fragment program (in foo.cg)
-t    disable kernel call type checking
-y    emit code for 4-output hardware
-A    enable address virtualization (experimental)
-N    deny support for kernels calling other kernels
-o prefix    prefix prepended to all output files
-w workspace workspace size (16 - 2048, default 1024)
-p shader    cpu/ps20/ps2a/ps2b/arb/fp30/fp40 (can specify multiple)
-f compiler favor a particular compiler (cgc / fxc / default)
-a arch assume a particular GPU (default / x800 / 6800)
Running Brook

- **BRT_RUNTIME selects platform**
  - CPU Backend: \( \text{BRT\_RUNTIME} = \text{cpu} \)
  - OpenGL ARB Backend: \( \text{BRT\_RUNTIME} = \text{ogl} \)
  - DirectX9 Backend: \( \text{BRT\_RUNTIME} = \text{dx9} \)
Runtime

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

```cpp
brook::initialize( "dx9", (void*)device );

// Create streams
fluidStream0 = stream::create<float4>( kFluidSize, kFluidSize );
normalStream = stream::create<float3>( kFluidSize, kFluidSize );

// Get 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
- fft edge detect
- linear algebra
- SAXPY
- SGEMV
Evaluation

Compared against:
- Intel Math Library
- Atlas Math Library
- Cached blocked segmentation
- FFTW
- Wald 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