Reading Assignment #9 (until Nov 4)

Read (required):

- **Brook for GPUs: Stream Computing on Graphics Hardware**
  Ian Buck et al., SIGGRAPH 2004
  http://graphics.stanford.edu/papers/brookgpu/

Read (optional):

- **The Imagine Stream Processor**
  Ujval Kapasi et al.; IEEE ICCD 2002

- **Merrimac: Supercomputing with Streams**
  Bill Dally et al.; SC 2003
  https://dl.acm.org/citation.cfm?doid=1048935.1050187
Types of Parallelism

Bit-Level Parallelism (70s and 80s)
- Doubling the word size 4, 8, 16, 32-bit (64-bit ~2003)

Instruction-Level Parallelism (mid 80s-90s)
- Instructions are split into stages → multi stage pipeline
- Superscalar execution, …

Data Parallelism
- Multiple processors execute the same instructions on different parts of the data

Task Parallelism
- Multiple processors execute instructions independently
1990s Fixed function graphics-pipeline used for more general computations in academia (e.g., rasterization, z-buffer)

2001 Shaders changed the API to access graphics cards

2004 Brook for GPUs changed the terminology

Since then:

- ATI’s Stream SDK (originally based on Brook)
- NVIDIA’s CUDA (together with Brook developers)
- OpenCL (platform independent)
- GLSL Compute Shaders (platform independent)
Stream Programming Abstraction

Goal: SW programming model that matches data parallelism

Streams
• Collection of data records
• All data is expressed in streams

Kernels
• Inputs/outputs are streams
• Perform computation on streams (each data record is processes independently)
• Can be chained together

Courtesy John Owens
Why Streams?

- Exposing parallelism
  - Data parallelism
  - Task parallelism

```c
for(i = 0; i<size; i++)
{
    a[i] = 2*b[i];
}

for(i = 0; i<size; i++)
{
    a[i] = a[i+1]*2;

for(each a, b)
{
    a[i] = 2*b[i];
}

for(each a)
{
    a = 2*b;
}
```

- Multiple stream elements can be processed in parallel
- Multiple tasks can be processed in parallel
- Predictable memory access pattern
- Optimize for throughput of all elements, not latency of one
- Processing many elements at once allows latency hiding
Brook for GPUs: Stream Computing on Graphics Hardware

Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan

Computer Science Department
Stanford University
domain specific solutions

Application

Graphics API

GPU

map directly to graphics primitives

requires extensive knowledge of GPU programming
building an abstraction

general GPU computing question
– can we simplify GPU programming?

– what is the correct abstraction for GPU-based computing?

– what is the scope of problems that can be implemented efficiently on the GPU?
contributions

- Brook stream programming environment for GPU-based computing
  - language, compiler, and runtime system

- virtualizing or extending GPU resources

- analysis of when GPUs outperform CPUs
GPU programming model

each fragment shaded independently

- no dependencies between fragments
  - temporary registers are zeroed
  - no static variables
  - no read-modify-write textures
- multiple “pixel pipes”
GPU = data parallel

each fragment shaded independently
  - no dependencies between fragments
    - temporary registers are zeroed
    - no static variables
    - no read-modify-write textures
  - multiple "pixel pipes"

data parallelism
  - support ALU heavy architectures
  - hide memory latency

[Torborg and Kajiya 96, Anderson et al. 97, Igehy et al. 98]
Brook language

stream programming model
  – enforce data parallel computing
    • streams
  – encourage arithmetic intensity
    • kernels
design goals

- general purpose computing
  GPU = general streaming-coprocessor
- GPU-based computing for the masses
  no graphics experience required
  eliminating annoying GPU limitations
- performance
- platform independent
  ATI & NVIDIA
  DirectX & OpenGL
  Windows & Linux
Brook language

C with streams

• streams
  – collection of records requiring similar computation
    • particle positions, voxels, FEM cell, ...

    Ray r<200>;
    float3 velocityfield<100,100,100>;

  – data parallelism
    • provides data to operate on in parallel
Brook language

kernels

- kernels
  - functions applied to streams
    - similar to for_all construct

```
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

- kernels arguments
  - input/output streams

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

- kernels arguments
  - input/output streams
  - gather streams

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

- kernels arguments
  - input/output streams
  - gather streams
  - iterator streams

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

- kernels arguments
  - input/output streams
  - gather streams
  - iterator streams
  - constant parameters

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

• Ray Triangle Intersection

```
kernel void krunIntersectTriangle(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].trilnum;
    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

- reductions
  - compute single value from a stream

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

- reductions
  - compute single value from a stream

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

float a<100>;
float r;
sum(a,r);
```

```c
r = a[0];
for (int i=1; i<100; i++)
    r += a[i];
```
reductions

- reductions
  - associative operations only
    \[(a+b)+c = a+(b+c)\]
  - sum, multiply, max, min, OR, AND, XOR
  - matrix multiply
  - permits parallel execution
Brook language reductions

- multi-dimension reductions
  - stream “shape” differences resolved by reduce function
Brook language

reductions

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

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

    ```
    float a<20>;
    float r<5>;
    sum(a,r);
    ```
Brook language

reductions

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

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

float a<20>;
float r<5>;

sum(a,r);

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];
Brook language

reductions

- multi-dimension reductions
  - stream "shape" differences resolved by reduce function

```c
reduce void sum (float a<>,
                 reduce float r<>)

    r += a;

}

float a<20>;
float r<5>;
sum(a,r);
```

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];
Brook language

stream repeat & stride

- kernel arguments of different shape
  - resolved by repeat and stride
Brook language
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);
```
Brook language

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[0] , b[0] , c[0])
foo(a[2] , b[0] , c[1])
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])
```

April 6th, 2004
Brook language

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);

April 6th, 2004
Brook language

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) ;

April 6th, 2004
system outline

brcc
- source to source compiler
- generate CG & HLSL code
- CGC and FXC for shader assembly
- virtualization

brt
Brook run-time library
- stream texture management
- kernel shader execution
eliminating GPU limitations

treating texture as memory
  - limited texture size and dimension
  - compiler inserts address translation code

```cpp
float matrix<8096,10,30,5>;
```
applications

ray-tracer

segmentation

fft edge detect

linear algebra
summary

GPU-based computing for the masses

bioinformatics
simulation
rendering
statistics
Thank you.

- John Owens
- Ian Buck et al.
- AMD