CS 380 - GPU and GPGPU Programming
Lecture 18+19: GPU Texturing 2;
Stream Computing and GPGPU

Markus Hadwiger, KAUST
Reading Assignment #11 (until April 30)

Read (required):

• Programming Massively Parallel Processors book, Chapter 6 (Performance Considerations)

• CUDA 6.0 C Programming Guide
  Appendix G: Compute Capabilities

  Study the different memory access requirements for different compute capabilities!

• Chapter 39: Parallel Prefix Sum (Scan) with CUDA,
  GPU Gems 3 book

GPU Texturing

Rage / id Tech 5 (id Software)
Texture Mapping

2D (3D) Texture Space
   | Texture Transformation
2D Object Parameters
   | Parameterization
3D Object Space
   | Model Transformation
3D World Space
   | Viewing Transformation
3D Camera Space
   | Projection
2D Image Space

Kurt Akeley, Pat Hanrahan
Texture Mapping Polygons

Forward transformation: linear projective map

\[
\begin{bmatrix}
x \\
y \\
w
\end{bmatrix} = \begin{bmatrix}
a & b & c & s \\
d & e & f & t \\
g & h & i & r
\end{bmatrix}
\]

Backward transformation: linear projective map

\[
\begin{bmatrix}
s \\
t \\
r \\
w
\end{bmatrix} = \begin{bmatrix}
a & b & c & x \\
d & e & f & y \\
g & h & i & w
\end{bmatrix}^{-1}
\]

Kurt Akeley, Pat Hanrahan
Incorrect attribute interpolation

Linear interpolation

$A' \neq A$!
Linear interpolation

Compute intermediate attribute value

- Along a line: \[ A = aA_1 + bA_2, \quad a+b=1 \]
- On a plane: \[ A = aA_1 + bA_2 + cA_3, \quad a+b+c=1 \]

Only projected values interpolate linearly in screen space (straight lines project to straight lines)

- \( x \) and \( y \) are projected (divided by \( w \))
- Attribute values are not naturally projected

Choice for attribute interpolation in screen space

- Interpolate unprojected values
  - Cheap and easy to do, but gives wrong values
  - Sometimes OK for color, but
  - Never acceptable for texture coordinates
- Do it right

Kurt Akeley, Pat Hanrahan
Linear Perspective

Correct Linear Perspective

Incorrect Perspective

Linear Interpolation, Bad
Perspective Interpolation, Good

Kurt Akeley, Pat Hanrahan
Perspective Texture Mapping

Linear interpolation in object space:

\[
\frac{ax_1 + bx_2}{aw_1 + bw_2} \neq a \frac{x_1}{w_1} + b \frac{x_2}{w_2}
\]

Linear interpolation in screen space:

\[
a = b = 0.5
\]

Vienna University of Technology
Early Perspective Texture Mapping in Games

DOOM (id Software, 1993)
Early Perspective Texture Mapping in Games

Quake (id Software, 1996)
Perspective-correct linear interpolation

Only projected values interpolate correctly, so project $A$
- Linearly interpolate $A_1/w_1$ and $A_2/w_2$
Also interpolate $1/w_1$ and $1/w_2$
- These also interpolate linearly in screen space
Divide interpolants at each sample point to recover $A$
- $(A/w) / (1/w) = A$
- Division is expensive (more than add or multiply), so
  - Recover $w$ for the sample point (reciprocate), and
  - Multiply each projected attribute by $w$

Barycentric triangle parameterization:
\[
A = \frac{aA_1/w_1 + bA_2/w_2 + cA_3/w_3}{a/w_1 + b/w_2 + c/w_3}
\]
\[a + b + c = 1\]

Kurt Akeley, Pat Hanrahan
Perspective Texture Mapping

- Solution: interpolate (s/w, t/w, 1/w)
- \((s/w) / (1/w) = s\) etc. at every fragment

Heckbert and Moreton
Perspective-Correct Interpolation Recipe

\[ r_i(x, y) = \frac{r_i(x, y)/w(x, y)}{1/w(x, y)} \]

(1) Associate a record containing the \( n \) parameters of interest \((r_1, r_2, \cdots, r_n)\) with each vertex of the polygon.

(2) For each vertex, transform object space coordinates to homogeneous screen space using \( 4 \times 4 \) object to screen matrix, yielding the values \((xw, yw, zw, w)\).

(3) Clip the polygon against plane equations for each of the six sides of the viewing frustum, linearly interpolating all the parameters when new vertices are created.

(4) At each vertex, divide the homogeneous screen coordinates, the parameters \( r_i \), and the number 1 by \( w \) to construct the variable list \((x, y, z, s_1, s_2, \cdots, s_{n+1})\), where \( s_i = r_i/w \) for \( i \leq n \), \( s_{n+1} = 1/w \).

(5) Scan convert in screen space by linear interpolation of all parameters, at each pixel computing \( r_i = s_i/s_{n+1} \) for each of the \( n \) parameters; use these values for shading.

Heckbert and Moreton
Projective Texture Mapping

- Want to simulate a beamer
  - ... or a flashlight, or a slide projector
- Precursor to shadows
- Interesting mathematics:
  2 perspective projections involved!
- Easy to program!
Projective Texture Mapping
Projective Shadows in Doom 3
What about homogeneous texture coords?

Need to do perspective divide also for projector!

(s, t, q) \rightarrow (s/q, t/q) for every fragment

How does OpenGL do that?

Needs to be perspective correct as well!

Trick: interpolate (s/w, t/w, r/w, q/w)

(s/w) / (q/w) = s/q etc. at every fragment

Remember: s, t, r, q are equivalent to x, y, z, w in projector space! \rightarrow r/q = projector depth!
Multitexturing

- Apply multiple textures in one pass
- *Integral* part of programmable shading
  - e.g. diffuse texture map + gloss map
  - e.g. diffuse texture map + light map
- Performance issues
  - How many textures are free?
  - How many are available
Example: Light Mapping

- Used in virtually every commercial game
- Precalculate diffuse lighting on static objects
  - Only low resolution necessary
  - Diffuse lighting is view independent!
- Advantages:
  - No runtime lighting necessary
    - VERY fast!
  - Can take global effects (shadows, color bleeds) into account
Light Mapping

Original LM texels

Bilinear Filtering
Light Mapping Issues

- Why premultiplication is bad…

   - Full Size Texture (with Lightmap)
   - Tiled Surface Texture plus Lightmap

→ use tileable surface textures and low resolution lightmaps
Example: Light Mapping

- Precomputation based on non-realtime methods
  - Radiosity
  - Ray tracing
    - Monte Carlo Integration
    - Path tracing
    - Photon mapping
Light Mapping

Lightmap

mapped
Light Mapping

Original scene

Light-mapped
Ambient Occlusion

- Special case of light mapping
- Cosine-weighted visibility to environment modulates intensity:

\[ A_p = \frac{1}{\pi} \int_{\Omega} V_{p,\omega}(N \cdot \omega) \, d\omega \]

- Darker where more occluded
- Option: “per object” lightmap
  - Allows to move object
Ambient Occlusion

Model/Texture: Rendermonkey
Stream Computing and GPGPU
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

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 the 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 Computer 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
- Can be chained together

Courtesy John Owens
Why Streams?

• Exposing parallelism
  • Data parallelism
  • Task parallelism

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 = 2*b;
}

for(each a)
{
  ???
}

• 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

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, ...

```c
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

```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);
```

```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

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

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

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

kernels

• Ray Triangle Intersection

```
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].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

```c
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);
```

```
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

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>41</th>
<th>59</th>
<th>26</th>
<th>53</th>
<th>58</th>
<th>97</th>
<th>93</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>23</td>
<td>84</td>
<td>62</td>
<td>64</td>
<td>33</td>
<td>83</td>
<td>27</td>
<td>95</td>
</tr>
<tr>
<td>2</td>
<td>88</td>
<td>41</td>
<td>97</td>
<td>16</td>
<td>93</td>
<td>99</td>
<td>37</td>
<td></td>
</tr>
<tr>
<td>51</td>
<td>5</td>
<td>82</td>
<td>9</td>
<td>74</td>
<td>83</td>
<td>94</td>
<td>45</td>
<td></td>
</tr>
<tr>
<td>92</td>
<td>30</td>
<td>78</td>
<td>16</td>
<td>40</td>
<td>62</td>
<td>86</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>89</td>
<td>98</td>
<td>62</td>
<td>80</td>
<td>34</td>
<td>82</td>
<td>53</td>
<td>42</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>70</td>
<td>6</td>
<td>79</td>
<td>82</td>
<td>14</td>
<td>80</td>
<td>86</td>
<td></td>
</tr>
<tr>
<td>51</td>
<td>32</td>
<td>82</td>
<td>30</td>
<td>66</td>
<td>47</td>
<td>9</td>
<td>38</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>84</th>
<th>64</th>
<th>83</th>
<th>97</th>
</tr>
</thead>
<tbody>
<tr>
<td>88</td>
<td>97</td>
<td>93</td>
<td>99</td>
<td></td>
</tr>
<tr>
<td>98</td>
<td>80</td>
<td>82</td>
<td>86</td>
<td></td>
</tr>
<tr>
<td>70</td>
<td>82</td>
<td>82</td>
<td>86</td>
<td></td>
</tr>
</tbody>
</table>

SIGGRAPH 2004
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

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

```c
float a<20>;
float 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];
```
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);
```

```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];
```
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,b,c);
```

...
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);
```
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

```c
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
What is Brook+?

**Brook** is an extension to the C-language for stream programming originally developed by Stanford University.

**Brook+** is an implementation by AMD of the Brook GPU spec on AMD's compute abstraction layer with some enhancements.
Example

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

int main(int argc, char** argv)
{
    int i, j;
    float a<10, 10>;
    float b<10, 10>;
    float c<10, 10>;

    float input_a[10][10];
    float input_b[10][10];
    float input_c[10][10];

    for(i=0; i<10; i++)
        for(j=0; j<10; j++)
            input_a[i][j] = (float) i;
        input_b[i][j] = (float) j;

    streamRead(a, input_a);
    streamRead(b, input_b);
    sum(a, b, c);
    streamWrite(c, input_c);
    ...
}
```

Kernels – Program functions that operate on stream elements

Streams – collection of data elements of the same type which can be operated on in parallel.

Brook+ access functions
Brook+ Compiler

Converts Brook+ files into C++ code. Kernels, written in C, are compiled to AMD’s IL code for the GPU or C code for the CPU.
Brook+ Runtime

IL code is executed on the GPU. The backend is written in CAL.
**Brook+ Features**

Brook+ is an extension to the Brook for GPUs source code.

Features of Brook for GPUs relevant to modern graphics hardware are maintained.

Kernels are compiled to AMD’s IL.

Runtime uses CAL for the GPU backend.

Original CPU backend also included.
Brook+ Release

Brook+ package:
- Compiler and runtime binaries
- Source code and build environments
- Sample applications

Source code released under the BSD License.

Project will also reside on SourceForge.net.
Brook+ Moving Forward

Double precision - FireStream 9170

Mem-export (scatter)

Graphics API interoperability

Multi-GPU support

Other operating systems (Linux, Vista, 64-bit)
Thank you.