Glift: Generic, Efficient Random-Access GPU Data Structures

Aaron Lefohn
University of California, Davis

Collaborators
- Joe Kniss
  University of Utah
- Robert Strzodka
  Stanford University
- Shubhabrata Sengupta
  University of California, Davis
- John Owens
  University of California, Davis

Problem Statement
- **Goal**
  - Simplify creation and use of random-access GPU data structures for graphics and GPGPU programming
- **Contributions**
  - Abstraction for GPU data structures
  - Glift template library

Compute vs. Bandwidth: 2005 Update
- **Float4 perfect cache hit**

Based on data from http://graphics.stanford.edu/projects/gpubench/results/

Compute vs. Bandwidth: 2005 Update
- **Float4 sequential (streaming) read**

Based on data from http://graphics.stanford.edu/projects/gpubench/results/

CPU Software Development
- **Benefits**
  - Algorithms and data structures expressed in problem domain
  - Decouple algorithms and data structures
  - Code reuse
GPU Software Development

• Problems
  • Code is tangled mess of algorithm and data structure access
  • Algorithms expressed in GPU memory domain
  • No code reuse

GPU Data Structure Abstraction

• What’s Missing?
  • Standalone abstraction for GPU data structures for graphics or GPGPU programming

CPU (C++) Example

typedef boost::multi_array<float, 3> array_type;
array_type srcData(boost::extents[10][10][10]);
array_type dstData(boost::extents[10][10][10]);

... initialize data ...

for (size_t z = 1; z < 10; ++z) {
  for (size_t y = 1; z < 10; ++y) {
    for (size_t x = 1; z < 10; ++x) {
      dstData[z][y][x] = srcData[z-1][y-1][x-1];
    }
  }
}

We Want To Transform This...

void main( uniform VMem3D data,
            AddrIter3D iter,
            out float result )
{
  float3 va = iter.addr();
  return srcData.vTex3D(va - float3(1,1,1));
}

Overview

• Motivation
• Abstraction
• Glift template library
• Conclusions
Building the Abstraction

- **Goals**
  - Enable easy creation of new structures
  - Minimal efficient abstraction of GPU memory model
  - Separate data structures from algorithms
  - Clarify characteristics of GPU-compatible structures
  - Encourage efficiency

Building the Abstraction

- **Approach**
  - Bottom-up, working towards STL-like syntax
  - Identify common patterns in GPU papers and code
  - Inspired by
    - STL, Boost, STAPL, A. Stepanov
    - Brook

Previous GPU Data Structure Abstractions

- **Brook**
  - Virtualizes CPU/GPU interface for 1D - 4D arrays
- **Sh**
  - Virtualizes 1D arrays and CPU/GPU data access

What is the GPU Memory Model?

- **CPU interface**
  - `glTexImage`    malloc
  - `glDeleteTextures` free
  - `glTexSubImage`    memcpy   GPU -> CPU
  - `glGetTexSubImage*`    memcpy   CPU -> GPU
  - `glCopyTexSubImage`    memcpy   GPU -> GPU
  - `glBindTexture`  read-only   parameter bind
  - `glFramebufferTexture` write-only   parameter bind

* Does not exist. Emulate with `glReadPixels`

What is the GPU Memory Model?

- **GPU Interface (shown in Cg)**
  - `uniform samplerND`    parameter declaration
  - `texND(tex, addr)`    random-access read
  - `streamND(tex)*`    stream read

* Does not exist, but is a useful construct for efficiency reasons

GPU Data Structure Abstraction

- **Concepts**
  - Physical memory
  - Virtual memory
  - Address translator
  - Iterators
    - Address iterators
    - Element iterators
Physical Memory

- Native GPU textures
  - Choose based on algorithm efficiency requirements
    - 1D
      - Read-write, linear, 4096 max size
    - 2D
      - Read-write, bilinear, 4096² max size
    - 3D
      - Read-only, trilinear, 512³ max size
    - Cube
      - read-write, bilinear, square, array of six 2D textures
    - Mipmaps
      - Additional (multiresolution) dimension to address

Virtual Memory

- Virtual N-D address space
  - Choose based on problem space of algorithm
  - Defined by physical memory and address translator

Address Translator

- Mapping between physical and virtual addr
  - Core of data structure
  - Select based on virtual and physical domains and memory/compute efficiency requirements of algorithm
  - Small amount of code defines all required CPU and GPU memory interfaces

Address Translator Examples

- Examples
  - ND-to-2D
  - 3D-to-2D tiled “flat 3D textures”
  - Page table
  - Grid of lists
  - Hash table
  - Sitemap

Address Translator Classifications

- Representation
  - Analytic / Discrete
- Compute Consistency
  - Uniform vs. non-uniform
- Memory Complexity
  - O(1), O(log N), O(N), ...
- Total / Partial
  - Complete vs. sparse
- Compute Complexity
  - O(1), O(log N), O(N), ...
- One-to-one / Many-to-one
  - Uniform vs. adaptive
Iterators

- Separate algorithms and data structures
  - Minimal interface between data and algorithm
  - Algorithms traverse elements of generic structures
  - Required for GPGPU use of data structure

- Two types of iterators
  - Address iterators
    - Iterator value is N-D address
    - GPU interpolants (Brook iterator streams)
  - Element iterators
    - Iterator value is data structure element
    - C/C++ pointer, STL iterator, Brook streams

Which Element Iterators?

- Type of iterator defines
  - Permission
    - Read-only, write-only, read-write
  - Access region
    - Single, neighborhood, random
  - Traversal
    - Forward, backward, parallel range

Two types of iterators

- Address iterators
  - Iterator value is N-D address
  - GPU interpolants (Brook iterator streams)

- Element iterators
  - Iterator value is data structure element
  - C/C++ pointer, STL iterator, Brook streams

Element Iterators

- CPU and GPU iterators
  - Wider range of CPU iterator types (less restricted)
  - GPU iterators define GPGPU computation domain
  - Possibly more GPU iterator types as machine model evolves...

Simple Example

- CPU (C++) 3D array

```c++
typedef boost::multi_array<float, 3> array_type;
array_type srcData( boost::extents[10][10][10] );
array_type dstData( boost::extents[10][10][10] );
... initialize data ...
for (size_t z = 1; z < 10; ++z) {
  for (size_t y = 1; y < 10; ++y) {
    for (size_t x = 1; x < 10; ++x) {
      dstData[z][y][x] = srcData[z-1][y-1][x-1];
    }
  }
}
```

Example : GPU Shader Factorization

```glsl
float3 main( uniform samplerRECT data,
            uniform float2 winSize,
            uniform float3 sizeConst3D,
            float2 winPos : WPOS ) : COLOR {
  float3 hereAddr3D = getAddr3D( winPos, winSize, sizeConst3D );

  float3 neighborAddr = hereAddr3D - float3(1, 1, 1);
  return texRECT(data, getAddr2D(neighborAddr3D, winSize, sizeConst3D) );
}
```
Example : Glift Components

```
float3 getAddr3D( float2 winPos, float2 winSize, float3 sizeConst3D ) {
    float3 curAddr3D;
    float2 winPosInt = floor(winPos);
    float addr1D = winPosInt.y * winSize.x + winPosInt.x;
    addr3D.z  = floor( addr1D / sizeConst3D.z );
    addr1D   -= addr3D.z * sizeConst3D.z;
    addr3D.y  = floor( addr1D / sizeConst3D.y );
    addr3D.x  = addr1D - addr3D.y * sizeConst3D.y;
    return addr3D;
}

float2 getAddr2D( float3 addr3D, float2 winSize, float3 sizeConst3D ) {
    float addr1D = dot( addr3D, sizeConst3D );
    float normAddr1D = addr1D / winSize.x;
    return float2(frac(normAddr1D) * winSize.x, normAddr1D);
}
```

Example : GPU Shader with Glift

```
void main( uniform VMem3D data,
    uniform float2 winSize,
    uniform float3 sizeConst3D,
    float2 winPos : WPOS ) : COLOR
{
    float3 hereAddr3D   = getAddr3D( winPos, winSize, sizeConst3D );
    float3 neighborAddr = hereAddr3D - float3(1, 1, 1);
    return texRECT(data, getAddr2D(neighborAddr3D, winSize, sizeConst3D) );
}
```

Example : GPU C++ Code with Glift

```
vec3i origin(0,0,0);
vec3i size(10,10,10);
ArrayGpuND<vec3i,vec1f> srcData( size );
ArrayGpuND<vec3i,vec1f> dstData( size );
... initialize dataPtr ...
srcData.write( origin, size, dataPtr );
gpu_range_iterator it = dstData.gpu_range(origin, size);
it.bind_for_read( iterCgParam );
dstData.bind_for_read( srcCgParam );
dstData.bind_for_write( COLOR0, myFrameBufferObject );
mapGpu( it );
```

Additional Benefits of Abstraction

- Multiple PhysMem with same AddrTrans
  - “Unlimited” amount of data in structures
- Multiple AddrTrans with one PhysMem
  - “reinterpret_cast” physical memory
- Continuous memory layout
  - Efficient stream processing of PhysMem or AddrTrans

Overview

- Motivation
- Abstraction
- Glift template library
- Conclusions

Glift Components

```
Application
    Container Adaptors
    VirtMem
       PhysMem       AddrTrans
       C++ / Cg / OpenGL
```
Glift Design Goals

- Generic implementation of abstraction
- As efficient as hand-coding
- Unified C++ and Cg code base
- Easily extensible
- Incrementally adoptable
- Easy integration with Cg/OpenGl

C++/Cg Integration

- Each component defines C++ and Cg code
  - C++ objects have Cg struct representation
  - Stringified Cg parameterized by C++ templates
- Cg “template” instantiation
  - Insert generated Glift source code into shader
  - glift::cgInstantiateParameter
- All other compilation/loading/binding identical to standard shader

More Glift Examples....

- 4D array
- 3D sparse array
  - Sparse array implemented with a page table
- Stack

4D Array Declaration Example

- Build 4D array of vec3f values

```cpp
typedef PhysMemGPU<vec2i, vec3f> PMem2D;
typedef NdTo2DAddrTrans<vec4i, vec2i> Addr4to2;
typedef VirtMemGPU<Addr4to2, PMem2D> VMem4D;

vec4i virtSize( 10, 10, 10, 10);
vec2i physSize( 100, 100 );
PMem2D   pMem2D( physSize );
Addr4to2 addrTrans( virtSize, physSizse );
VMem4D   array4D( addrTrans, pMem2D );
```

4D Array Usage Example

- Interface similar to native texture

```cpp```
vec3f* data = ... initialize data ...
vec4i origin(0,0,0,0);

array4D.write( origin, virtSizee, data );
... array4D.bind_for_read( cgParam );
... array4D.bind_for_write( GL_COLOR_ATTACHMENT0 );
... array4D.read( origin, virtSize, data );
```

4D Array Shader Example

- Interface similar to native texture

```cpp```
float4 main( uniform VMem4D array4D,
            float4 addr ) : COLOR
{
    return 2.0f * array4D.vTex4D( addr );
}
```
Sparse 3D Array Declaration Example

- Build sparse 3D grid of vec4ub values

```cpp
typedef VirtPageTable<vec3i, vec3f, vec4ub, page_allocator> VMem3D;

vec3i virtSize(512, 512, 512);
vec3i physSize(128, 128, 128);
VMem3D sparse3D( virtSize, physSize );
```

Sparse 3D Array Usage Example

- Interface similar to native texture

```cpp
vec4ub* data = ... initialize data ...
vec3i origin(0,0,0);
vec3i size(20,20,20);

sparse3D.write( origin, virtSize, data );
... sparse3D.bind_for_read( cgParam );
... sparse3D.bind_for_write( GL_COLOR_ATTACHMENT0 );
... sparse3D.read( origin, size, data );
... gpu_range_iterator it = sparse3D.gpu_range(origin, size);
```

Sparse 3D Array Shader Example

- Element iterator interface (GPGPU)

```cpp
float4 main( ElementIter3D sparse3D ) : COLOR
{
   return sparse3D.value() / 2.0f;
}
```

GPU Stack Example

- Build stack of vec4ub values
  - Container adaptor atop 1D virtual array

```cpp
int maxSize = 10000;
glift::stack<vec4ub> gpuStack(maxSize);
glift::ArrayGpuND<vec1i, vec4ub> data(50);
... initialize data ...

gpuStack.push( data.gpu_range(0, 50) );
gpuStack.pop( data.gpu_range(0, 50) );
```

GPU Stack

- Push
  - Add N contiguous elements to “top”

- Pop
  - Remove N elements from “top”
**GPU Stack**

- **Pop**
  - Remove N elements from “top”

**More Examples**

- “Dynamic Adaptive Shadow Maps on Graphics Hardware”
  - SIGGRAPH 2005 Sketch
- “Octree Textures on Graphics Hardware”
  - SIGGRAPH 2005 Sketch

**Static Analysis of Generated Glift Code**

- Static instruction results
  - With Cg program specialization
    - 1D -> 2D: Glift 4, By-Hand 3, Brook 4
    - 3D page table: Glift 5, By-Hand 5, Brook 4
    - ASM: Glift 9, By-Hand 9
    - Octree: Glift 10, By-Hand 9
    - ASM + offset: Glift 10, By-Hand 9

  - Conclusion: Glift structures within 1 instr of hand-coded Cg

**Overview**

- Motivation
- Abstraction
- Glift template library
- Conclusions

**Summary**

- GPU programming needs data structure abstraction
  - More complex data structures and algorithms
  - Iterators clarify GPU memory access patterns
- Why programmable address translation?
  - Common pattern in many GPU apps
  - Small amount of code virtualizes GPU memory model
  - Data-parallel computing requires address space

- Glift template library
  - Generic C++/Cg implementation of abstraction
  - Easily integrates into OpenGL/Cg programming environment
  - Nearly as efficient as hand coding

**Summary**
Acknowledgements

- Craig Kolb, Nick Triantos NVIDIA
- Fabio Pellacini Cornell/Pixar
- Adam Moerschell, Yong Kil UCDavis
- Serban Porumbescu, Chris Co, ..... 
- Ross Whitaker, Chuck Hansen, Milan Ikits U. of Utah

- National Science Foundation
- Department of Energy

More Information

- Upcoming ACM Transactions on Graphics paper
  - “Glift: Generic, Efficient, Random-Access GPU Data Structures”
- Upcoming release of Glift template library
  - Watch www.gpgpu.org
- Google “Lefohn GPU”