

High-level GPGPU Programming

Takeshi Hakamata

Outline

- Motivations
- High-level languages for GPGPU
- Examples
- Comparison

Motivations

- Writing a GPGPU program is cumbersome
- Need to use a graphics API for non-graphics purpose
 - FBO or pbuffers for writing data to textures
 - Defining a texture to represent a stream
- Tying shader variables with C++ variables
 - Defining uniform variables

High-level languages for GPGPU

- Sh
 - Support both GPGPU and graphics programming
- Brook for GPUs
 - Emphasize on GPGPU
- Scout
 - Mostly for scientific visualization

Sh

- Embedded domain specific language (EDSL)
- Implemented as C++ class library
- Can be used to define shaders and stream kernels
- Backend: CPU and GPU

Sh

- Code between `SH_BEGIN_PROGRAM()` and `SH_END`
 - Compiled to a shader
 - Execution is delayed (retained mode)
- If computation on Sh classes are done outside `BEGIN/END`
 - Immediately executed (immediate mode)

Sh Example

```
#include <sh/sh.hpp>
#include <iostream>

using namespace std;
using namespace SH;

int main()
{
    shInit();

    ShProgram prg = SH_BEGIN_PROGRAM("gpu:stream") {
        ShInputAttrib3f a;
        ShOutputAttrib3f b;
        b = a + ShAttrib3f(42.0, 42.0, 42.0);
    } SH_END;

    float data[] = { 1.0, 0.5, -0.5 };
    ShHostMemoryPtr mem_in = new ShHostMemory(sizeof(float) * 3, data, SH_FLOAT);
    ShChannel<ShAttrib3f> in(mem_in, 1);

    float outdata[3];
    ShHostMemoryPtr mem_out = new ShHostMemory(sizeof(float) * 3, outdata, SH_FLOAT);
    ShChannel<ShAttrib3f> out(mem_out, 1);

    out = prg << in;

    mem_out->hostStorage()->sync();
    float* results = static_cast<float*>(mem_out->hostStorage()->data());
    cout << "out = (" << results[0] << ", " << results[1] << ", " << results[2] << ")" << endl;
}
```

Sh Example: Particle System

```
// Specifiy the generic particle update program, later it will
// be specialized for the the actual particle update.
ShProgram particle = SH_BEGIN_PROGRAM("stream") {
    ShInOutPoint3f  SH_DECL(pos);
    ShInOutVector3f SH_DECL(vel);
    ShInputVector3f SH_DECL(acc);
    ShInputAttrib1f SH_DECL(delta);
    // clamp acceleration to zero if particles at or below ground plane
    acc = cond(abs(pos(1)) < 0.05, ShVector3f(0.0, 0.0, 0.0), acc);
    // integrate acceleration to get velocity
    vel += acc*delta;
    // integrate velocity to update position
    pos += vel*delta;
    // parameters controlling the amount of momentum tranfer on collision
    ShAttrib1f mu(0.3);
    ShAttrib1f eps(0.6);
    // check if below ground level
    ShAttrib1f under = pos(1) < 0.0;
    // clamp to ground level if necessary
    pos = cond(under, pos*ShAttrib3f(1.0, 0.0, 1.0), pos);
}
```


Sh Example: Particle System

```
// modify velocity in case of collision
ShVector3f veln = vel*ShAttrib3f(0.0, 1.0, 0.0);
ShVector3f velt = vel - veln;
vel = cond(under, (1.0 - mu)*velt - eps*veln, vel);
pos(1) = cond(min(under, (vellvel) < 0.1), ShPoint1f(0.0f), pos(1)); // clamp to
    the edge of the plane, just to make it look nice
pos(0) = min(max(pos(0), -5), 5);    pos(2) = min(max(pos(2), -5), 5);
} SH_END;
```

Inside the idle callback, update the streams.

Brook for GPUs

- Programming language for parallel computers developed at Stanford
- Later adapted for GPGPU
- C with streaming extensions
- Compiler generates C++ and Cg shaders
- Can choose backend at runtime
- GPU and CPU backend

Brook for GPUs

- Two types of functions
 - General C like functions: compiled into C++ code
 - Computation kernels: compiled into Cg shaders
- Additional datatypes
 - Input/Output streams (float strm<100>)
 - Vectorized data (float2, float3, float4)

Streams

- Two operations
 - `streamRead(stream, array)`
 - `streamWrite(stream, array)`
- Can be multi-dimensional
 - `float3 velocityField<100, 100, 100>`
- Indexing of arbitrary locations disallowed
 - No `velocityField[i][j][k]`
 - Indexing of the current stream element is allowed with `indexof()` operator

Kernels

- Functions applied to streams
 - Similar to “for (all stream elements)” construct

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

```
float x<100>;  
float y<100>;  
float z<100>;  
foo(x, y, z);
```

```
for (int i = 0; i < streamSize; i++) {  
    result[i] = a[i] + b[i];  
}
```

Kernels

- Abstraction of fragment shaders
- Compiled into Cg shaders
- Functions supported by Cg are generally available
 - dot(), cross(), sin()/cos(), min()/max(), if-then-else, etc.
 - No support for integer types or bit fields
 - indexof() to get the position of an element in a stream

Reduction Kernels

- Reduce a stream to smaller one or even to a value
- Reduction kernels must be associative:
 - $(a \text{ op } b) \text{ op } c == a \text{ op } (b \text{ op } c)$

Example:

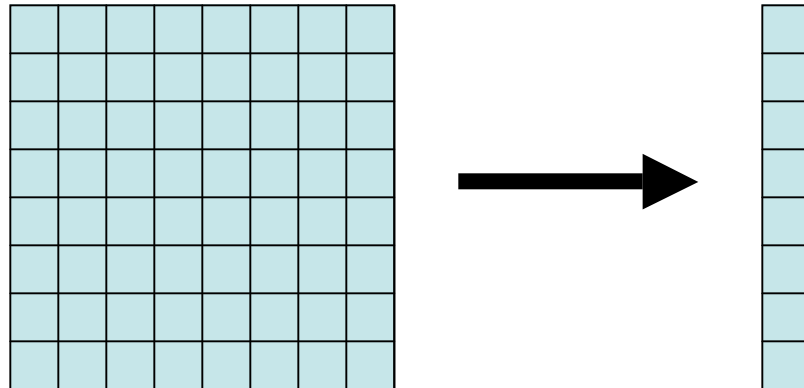
```
reduce sum(float inStream<>, reduce float result)
{
    result += inStream;
}
```

Reduction Kernels

- Reduce a matrix to a vector
- Output is also a stream

Example:

```
reduce sum(float inStream<>, reduce float result<>)  
{  
    result += inStream;  
}
```



Reduction Kernels

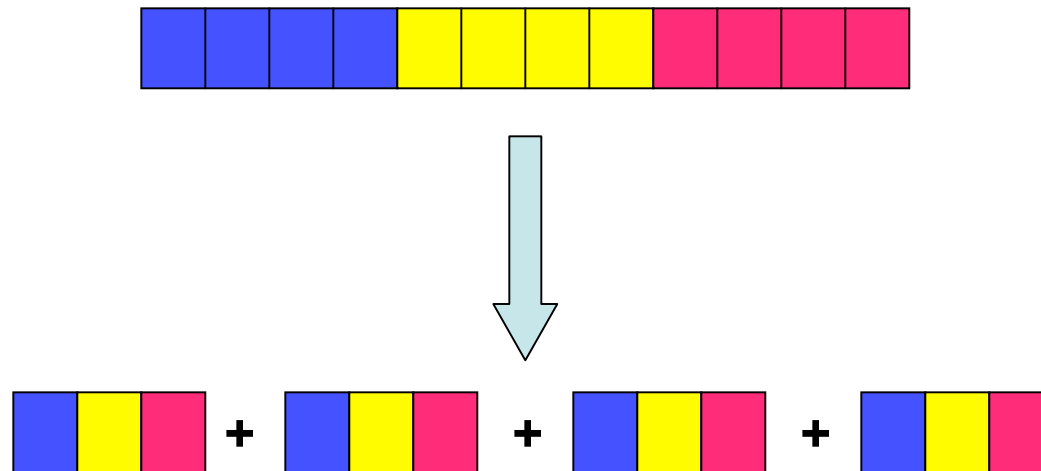
- Reduction to a different sized stream
 - reduce in multiple directions

Example:

float a<12>;

float b<3>;

sum(a, b);



General Functions

- Almost like C with few additional datatypes
 - Streams, float* types
- Generation of streams
- Call computation kernels
- Do ping-pong buffering

Runtime

- Both CPU code and GPU code are generated at compile time
- Backend can be configured at runtime by setting an environment variable
 - Available backends: OpenGL, DirectX, and CPU

A Simple Example

- Computing a dot product of two vectors
- Two vectors are represented as streams
- Dot product is done in a computation kernel

Dot Product

```
#define VECLEN 5

int main() {
    float v1Strm<VECLEN>;
    float v2Strm<VECLEN>;
    float prodStrm<VECLEN>;
    float v1[VECLEN] = {1.0f, 2.0f, 3.0f, 4.0f, 5.0f};
    float v2[VECLEN] = {5.0f, 4.0f, 3.0f, 2.0f, 1.0f};
    float dp;

    streamRead(v1Strm, v1);
    streamRead(v2Strm, v2);
    dp = dot(v1Strm, v2Strm);

    printf("v1 = ");
    printVec(v1);
    printf("v2 = ");
    printVec(v2);
    printf("dot product = %f\n", dp);

    return 0;
}
```

Dot Product

```
static float dot(float v1<>, float v2<>)  
{  
    float prodStrm<VECLLEN>;  
    float dp;  
  
    product(v1, v2, prodStrm);  
    sum(prodStrm, dp);  
  
    return dp;  
}  
  
kernel void product(float v1<>, float v2<>, out float prod<>)  
{  
    prod = v1 * v2;  
}  
  
reduce void sum(float p<>, reduce float res)  
{  
    res = res + p;  
}
```

Sparse Matrix-Vector Multiply

- Same algorithm as the previous lecture
- Non-zero entries are stored in the stream A
- Indices into non-zero entries are stored in the stream C.
- Use `gather()`, `mult()`, and `sum()` kernels

Sparse Matrix-Vector Multiply

Kernels

```
kernel void gather(float index<>, float x[NUM_ROWS+1], out float result<>) {  
    result = x[index];  
}
```

```
kernel void mult(float a<>, float b<>, out float c<>) {  
    c = a*b;  
}
```

```
reduce void sumRows(float nzValues<>, reduce float result<>) {  
    result += nzValues;  
}
```


Sparse Matrix-Vector Multiply

```
#define MAX_NZ_PER_ROW 5
#define NUM_ROWS      10
#define MAX_NZ        50

float Anz[MAX_NZ];
float cldx[MAX_NZ];
float x[NUM_ROWS];
float y[NUM_ROWS];

float AStrm<MAX_NZ>;      // nonzeros of A
float cldxStrm<MAX_NZ>;   // column indices
float productsStrm<MAX_NZ>;
float xGatherStrm<MAX_NZ>; // "gathered" version of x

float xStrm<NUM_ROWS>;
float yStrm<NUM_ROWS>;
```

Sparse Matrix-Vector Multiply

```
// Read in the matrix and the vector from file
```

```
// read in the matrix data
```

```
streamRead(AStrm, Anz);
```

```
streamRead(cldxStrm, cldx);
```

```
for (i=0;i<NUM_ROWS;i++)
```

```
    y[i] = 0.0f;
```

```
streamRead(xStrm, x);
```

```
// these four lines perform sparse matrix-vector multiply
```

```
streamRead(yStrm, y);
```

```
gather(cldxStrm, xStrm, xGatherStrm);
```

```
mult(AStrm, xGatherStrm, productsStrm);
```

```
sumRows(productsStrm, yStrm);
```

Sparse Matrix-Vector Multiply

```
streamWrite(yStrm, y);
```

```
printf("\n");
```

```
printf("result: y = Ax\n");
```

```
for (i=0;i<NUM_ROWS;i++)
```

```
    printf("%.3f\n", y[i]);
```

Scout

- C* (Thinking Machines Inc.) with extensions
- Command line compiler and GUI/IDE for visualization
- Can be used for data analysis (GPGPU)
- Rendering methods:
 - Volume rendering
 - Ray casting
 - Point rendering

Comparison

- Sh
 - Good for GPGPU and graphics
 - Class library is big
- Brook for GPUs
 - For GPGPU
 - Very simple (basically C plus stream types)
- Scout
 - For scientific visualization
 - Not open source?

Summary

- High-level programming hides details
 - Graphics API free
 - Stream programming
 - Same source code can be compiled to run on CPU or GPU
- However,
 - Not clear which programming constructs costs more
 - You need to use vector datatypes to exploit the full performance
 - Brook: float4
 - Sh: ShVector4f or ShPoint4f

For More Information

- Sh
 - <http://www.libsh.org>
- Brook for GPUs
 - <http://graphics.stanford.edu/projects/brookgpu>
- Scout
 - <http://www.gpgpu.org/articles/scout04.pdf>