University of Central Florida

Stream Computing using Brook+

School of Electrical Engineering and Computer Science
University of Central Florida

Slides courtesy of P. Bhaniramka

Outline

- Overview of Brook+
- Brook+ Software Architecture
  - Compiler
  - Runtime
- Brook+ Kernel Development
  - Performance analysis
  - Performance Optimization
  - Debug

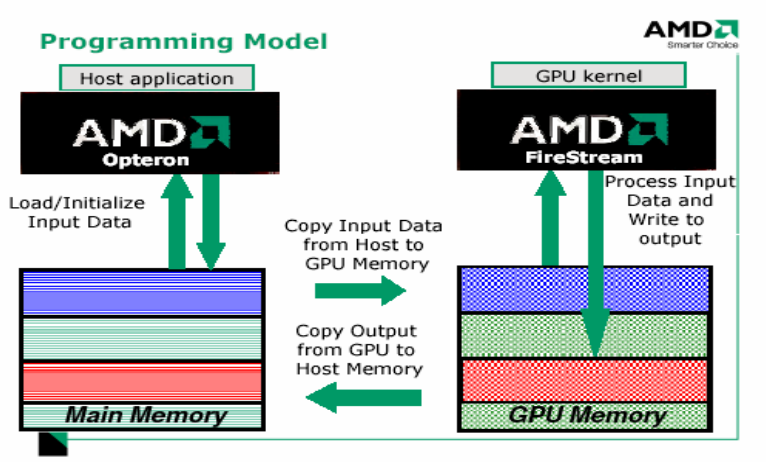
Brook+

- Brook is an extension to the C-language for stream computing
  - Syntax to represent and manipulate Streams
  - Syntax for kernel specification
  - Originally developed by Stanford U.
- Brook+ is AMD's implementation of the Brook GPU spec on AMD's CAL (compute abstraction layer)
  - Fixes
  - Optimizations
  - Many changes in 1.3 release

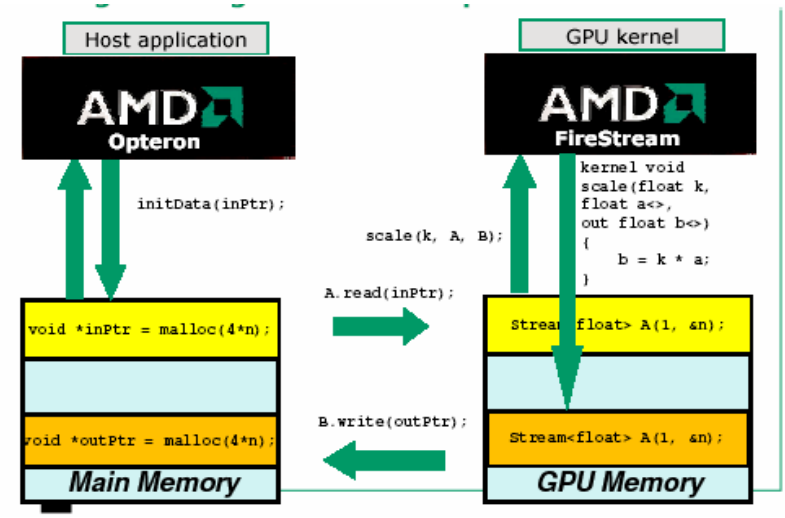
Design Goals

- Open Standards, Open Source
  - Existing initiative with multiple users
  - Cross-platform support
- Simple and easy to use
  - Attractive for application development and platform adoption
  - Low-level tools accessible for further enhancements
- Portability and Performance
  - BrookGPU worked with multiple backends (DirectX9, OpenGL, CPU running on both ATI and nVidiaGPUs)
- Interoperability with other tools
  - Necessary for production quality software
  - Fits in the Stream SDK stack
  - C++ application compatibility (1.3 release)

Programming model



Programming model: example



Programming model

- Brook+ programs consists of
  - Host side code
    - Application data management
    - Stream allocation and data transfer
    - Computation (sequential)
  - GPU-side Kernel (Data parallel code)
    - Read input streams
    - Perform parallel computations
    - Write output streams

Program structure – GPU kernel

- Written in separate file (usually .br)
- Compiled using brook+ compiler, brcc
- Generates GPU-specific binary code
- Generates header file to be included in host side for kernel invokation

```
kernel void scale(
                float k,
                float a<>,
                out float b<>)
{
   // Stream Read and Write done
   // during variable reference
   b = k * a;
}
OR
kernel void scale(
                float k,
                float a[][],
                out float b<>)
{
   // Same as above. Explicit
   // C-style array addressing
   int2 ind = instance().xy;
   b = k * a[ind.y][ind.x];
}
```

Program structure: host side

- Written in regular C/C++
- Compiled and linked using regular C/C++ compiler
  - Default is C++
  - C requires brcc flag
- Need to include brcc generated header

```
#include "scale.h"
// Host routine invoked to scale input
// buffer by constant value, k
bool scaleBuffer(
   float k,    // constant
   float *in, // buffer
   unsigned int dims[2]) // dimensions
{
   // Allocate in and out Streams
   Stream<float> in(2, dims);
   Stream<float> out(2, dims);
   // Initialize input
   s.read(in);
   // Launch GPU kernel
   scale(k, in, out);
   if(out.error()) // check for error
        return false;
   // Copy back result to buffer
   out.write(in);
   return true;
}
```

Brook+ Kernels

- Computational routines that operate on kernels
- One instance executed for each element in the domain of execution (a.k.a a thread)
  - Implicit: elements in the output stream
  - Explicit control: domainSize and domainOffset functions
- Internally compiled into GPU-specific binary
  - Invoked by host app. As a regular function call
  - Accept streams as read-only inputs and write-only outputs
  - Accept additional constants as read-only inputs

Kernel syntax

- Look like C routines with minor differences
  - Use of *kernel* keyword
  - Extensions for stream computing model
  - Restrictions for stream computing model

```
kernel void name(T arg0, T arg1, …, T argn-1);
e.g.,
kernel void sum(float a<>, float b<>, out float c<>)
{
    c = a + b;
}
```

Kernel syntax – Similarity with C

- Function declaration

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

- Built-in Types

  float, int, unsigned int, double

- Variable Declarations and Usage

  ```
  int k = 9;
  f = k;
  ```

- Block & Scoping rules

  ```
  int k;
  k = 5;
  int j; // Illegal
  ```

- Arithmetic Operators (+, *, /, etc.)

Kernel syntax – Similarity with C

- Assignment Expressions (=, +=, etc.)
- Array Declaration and Data Organization
  - kernel void bar(float b[][], out float c<>)
- Array Data Access
  - f = b[j][i];
- Comments (C++-style // and C-style /* */)
- Comparision (==, !=)
- Conditional expressions (? :)
- User defined functions (no recursive functions yet)
- A subset of C's flow constructs (do, while, for, if, break, continue)

  Not: goto, switch
- structs

Kernel syntax: extensions

- kernel Keyword for GPU routines
- Stream syntax (<>) with implicit addressing for reads and writes
  - kernel void foo(float a, float b<>, out float c<>)
  - c = a + b;
  - out Keyword for explicitly output streams
  - Implicit addressing,
  - read and write operations
- Strict type checking (implicit conversions not allowed)

  float a = 5.0; // Illegal. Need to explicitly use **5.0f**
- Vector data types and vector math operations (more details later)
- Function Overloaded Math routines (more details later)
- ……

Kernel syntax: restrictions

- Writing to static or global variables not allowed inside kernels
  - No data sharing among multiple kernel invocations (as of now)

```
float a; // global variable
kernel void bar(float b<>, out float c<>) {
  c = a * b; // Illegal. a not defined in local scope
}
```

- No memory allocation inside kernel
- No pointer arithmetic (as of now)
- Recursion is not allowed (as of now)

Kernel syntax: restrictions

- Calling non-kernel functions from a kernel is illegal

```
void foo() { …. } // Non-kernel CPU routine
kernel void bar(float a, float b<>, out float c<>) {
…..
foo(); // illegal – calling CPU from the GPU!
}
```

- Kernels invoked from application have a void return type
  - Non-void kernels are meant for GPU local sub-routines
  - Callable from other kernels only

```
kernel float foo() { … }
kernel void bar(float a, float b<>, out float c<>) {
c = foo();
}
```

Kernel syntax: Argument

- Valid arguments
  - – Streams
- Outputs
  - Need Minimum 1 Output
  - Write-only (Initialize first to use as temporary variable)
    kernel void foo(**out float c<>**)
    kernel void bar(**out float c[]**)
- Inputs
  - Read-only
    kernel void foo(**float a<>**, out float b<>)
    kernel void bar(**float a[]**, out float b<>)
  - Constants – Single Variables or Fixed-size Arrays
    - Read-only
      kernel void foo(**float a**, out float c<>)
      kernel void foo(**float a[10]**, out float c<>)

Streams

- Collection of data elements of the same type that can be operated on in parallel
  - Essentially an array of elements of specified dimensions
  - Data organization is C-style in row major order
  - Can use structs for user-defined types
- Declaration similar to C arrays but with angular brackets
  - Associated with *name*, *type* and *shape*
    type name<n>; // 1D stream of type with n elements
    type name<n, m>; // 2D stream of type with n x m elements
    e.g.
    float a <10>; // 1D stream of 1 32-bit float value per element
    // with 10 elements
    double b <10,10>; // 2D stream of 1 64-bit double value per
    // element with 10x10 elements

Streams

- Kernel Usage Interface
  - Angular brackets for implicit addressing
    Declaration requires name and type
    type name<>; // Stream *name* of *type*
    Access requires variable usage only
    a = name;
  - C array-like Syntax for explicit addressing
    Declares requires name, type and rank
    type name[][]; // 2D Stream *name* of *type*
    type name[]; // 1D Stream *name* of *type*
    Access requires C-array-style reference
    a = name[j][i];

Steam: Host-side interface

```
namespace brook
{
    template<class T>
    class Stream // Templated C++ class
    {
    public:
        // Constructor
        Stream(unsigned short rank, unsigned int* dimensions);
        // Operators for reading and writing Stream contents
        void read(const void* ptr);
        void write(void* ptr) const;
        // Query errors and further information
        BRerror error();
        const char* errorLog();
        ……
    };
}
```

Streams: host-side interface

- Declaration, Allocation and De-allocation
  - Requires type, rank and dimensions

```
unsigned int n = 10000; // 1D double
Stream<double> s(1, &n);
```

OR

```
Stream<double> *s = new Stream<double>(1, &n);
unsigned int dims[2] = {1024, 1024}; // 2D float
Stream<float> s(2, dims);
```

OR

```
Stream<float> *s = new Stream<float>(2, dims);
```

  - Similarly for other ranks and types

Stream: host-side interface

- Data Initialization and Retrieval

void Stream::read(const void* ptr);

  - Initializes Stream content with data from given ptr
    - CPU to GPU data transfer
  - ptr must be atleast as big as Stream size, given by size of element * number of elements

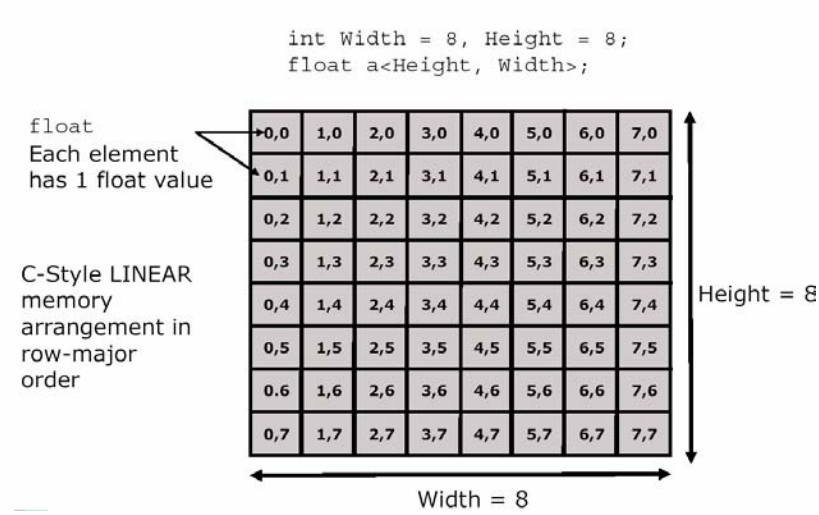
e.g.

```
unsigned int dims[2] = {1024, 1024}; // 2D float
Stream<float> s(2, dims);
# bytes = sizeof(float) * 1024 * 1024 = 4 MB
```

void Stream::write(void* ptr) const;

  - Initializes ptr content with data contained in Stream
    - GPU to CPU data transfer
  - ptr must be atleast as big as Stream size

Streams: Data layout



Stream: Error handling

- Error Query and Debugging

  BRerror Stream::error();

  – Used to check if an error occurred during processing of this Stream

  – Returns an error code (enum) for first error that occurred

    - BR_NO_ERROR if no error occurred
    - See ErrorCodes.h for full list of error codes
    - Error flag is cleared after the error() routine

  const char* Stream::errorLog();

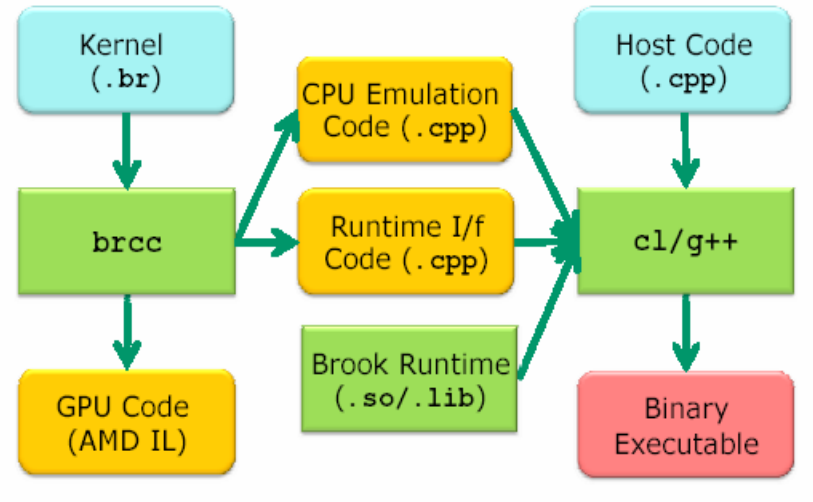
  – Returns NULL terminated char string with log messages

  – Information on errors from the first occurrence are accumulated

Brook+ Architecture

- Consists of two components
  - Brook+ Compiler, brcc
    - Generates GPU-specific Device Code
    - Generates CPU Emulation Code
    - Generates Compiler-Runtime Interface Code
    - Based on the cTool Open-source C parser
  - Brook+ Runtime
    - Responsible for
      - Launching pre-compiled Kernels
      - Low-level data management of Streams
      - Resource Virtualization
    - Supports multiple Runtime Backends
      - Provides common interface for each supported backend
      - Originally supported DirectX9, OpenGL and CPU backends
      - CAL backend added and supported by AMD

Brook+ development



Brook+ runtime

- Brook+ Runtime supports multiple backends
- Allows user to select a particular backend at runtime
  - Set environment variable, BRT_RUNTIME, to specify which backend to use
    BRT_RUNTIME=cpu for CPU emulation mode
    BRT_RUNTIME=cal for CAL backend (default with Brook+)
  - CPU backend is very useful for debugging kernels
  - CAL backend provides high performance (more details later)
    If specified backend fails to load, Brook+ defaults to CPU backend

Brook+ execution