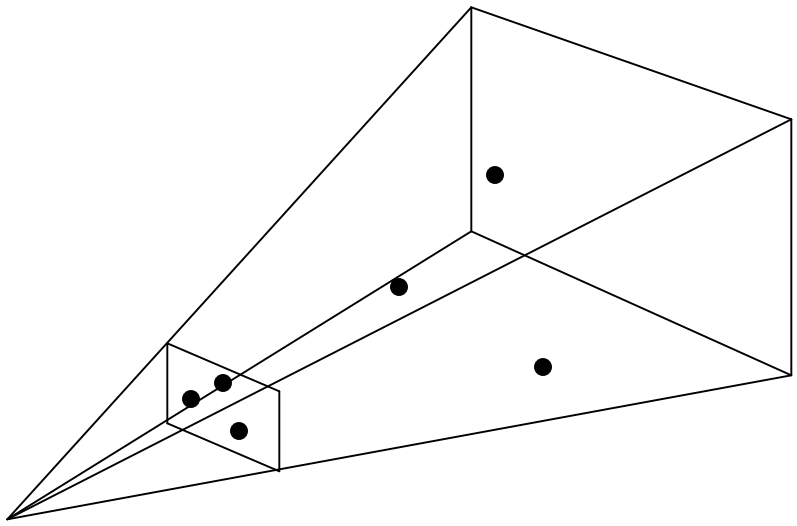
Rapidmind

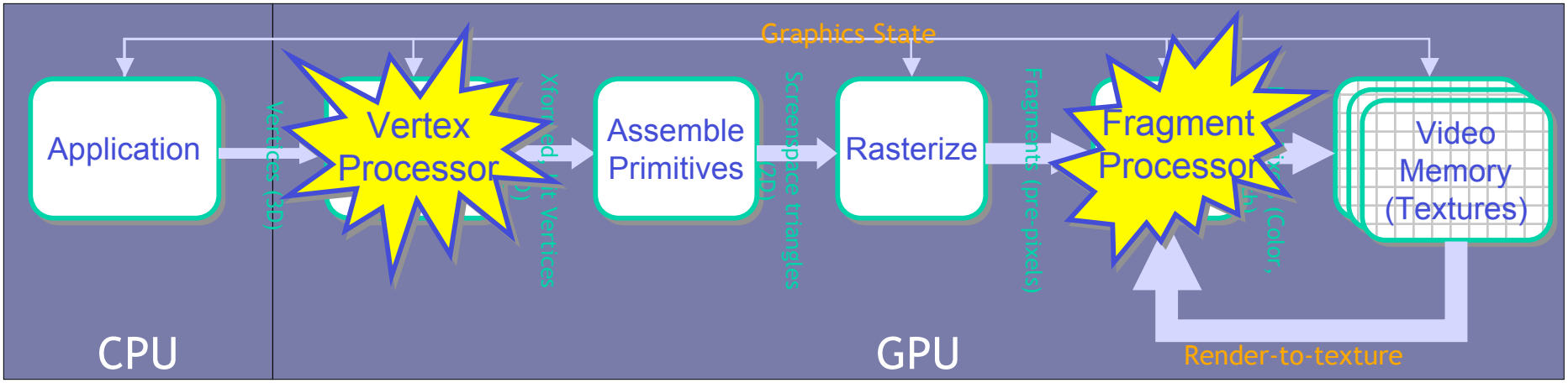
Background on GPGPU

GPU Tutorial

- Goal: 3-D image -> 2-D image
- 2 main stages:
  - Convert 3-D coordinates to 2-D windows
    - Vertex processing
  - Fill in 2-D windows
    - Fragment processing

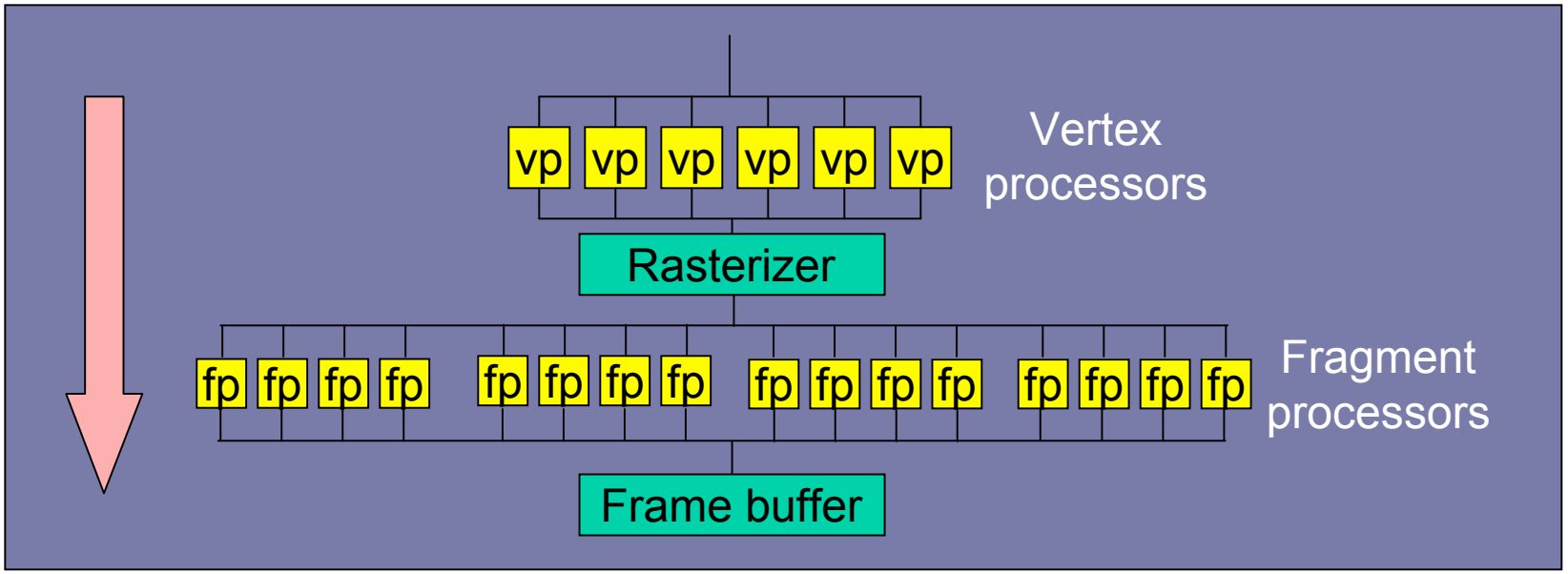


GPU Hardware Pipeline



GPU Parallelism

- Parallelism @ vertex and fragment calculations

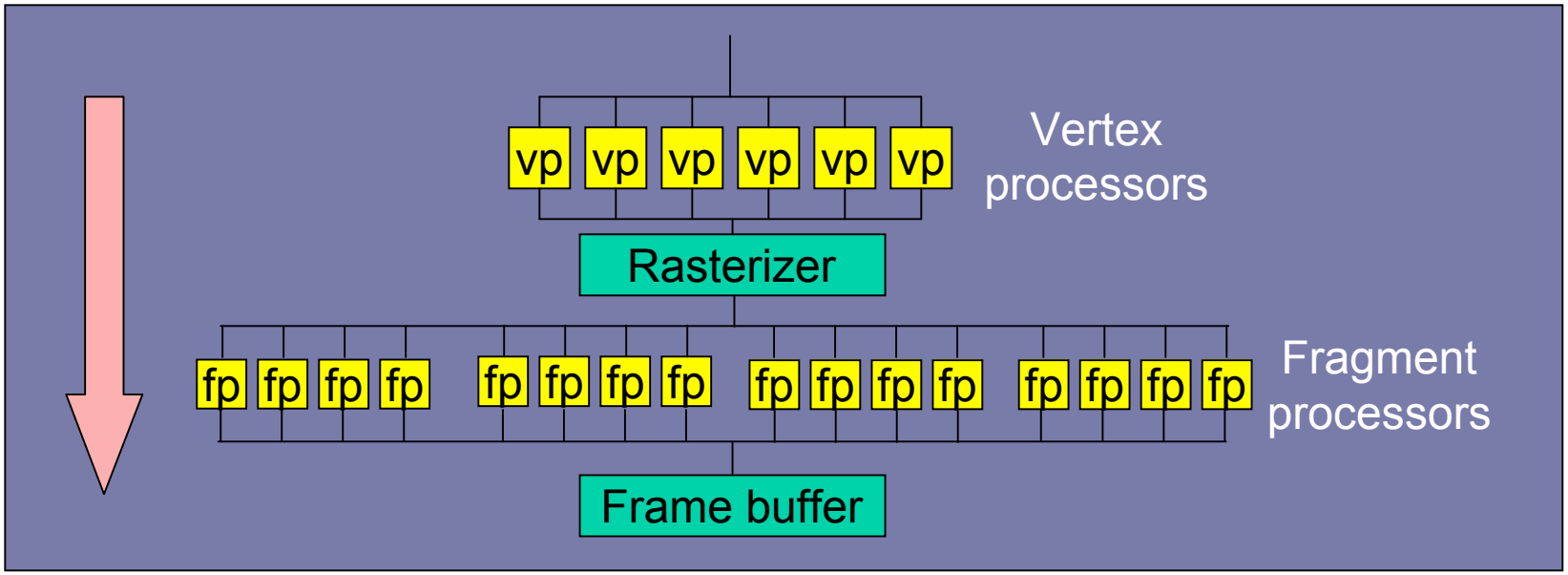


GPU Programmability

- Vertex and fragment processors can be programmed
- Shader = programs written for vertex and fragment calculations
- Vertex shaders = transformation, lighting
- Fragment shaders = texture, color, fog

GPU SIMD

- Vertex processors all run SAME shader program
- Fragment processor all run SAME shader program



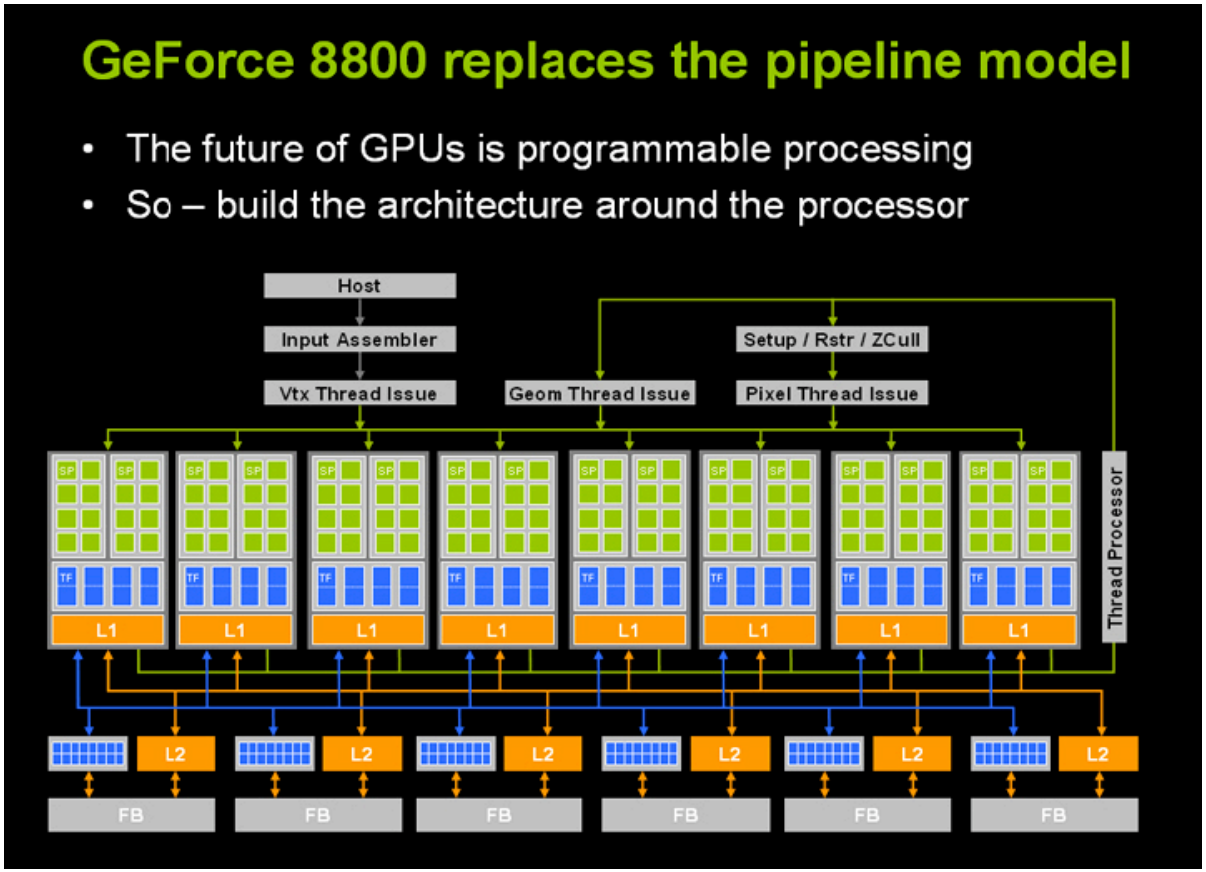
GPU Drawbacks

- No integer data operands
- No integer operations
  - e.g. bit shift, AND, OR, XOR, NOT
- No double precision arithmetic
- Unusual programming model

GPU Improvement

- NVIDIA GeForce G80 – unified pipeline and shader
- CUDA – Computer Unified Device Architecture
- Unified stream processors
  - Vertices, pixels, geometry, physics
  - General purpose floating point processors
  - Scalar processor rather than vector processor

NVIDIA GeForce 8800

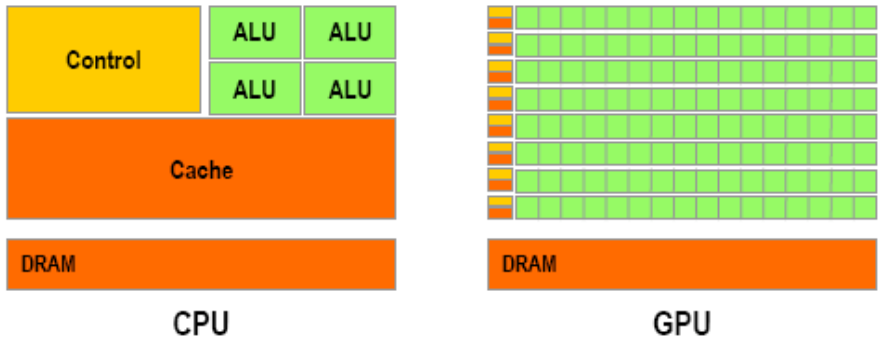


Facts and Motivations

- **GPUs are fast…**
  - 3.0 GHz Intel Core2 Quad (QX6850):
    - Computation: 96 GFLOPS peak
    - Memory bandwidth: 21 GB/s peak
    - Price: $1100 (chip)
  - NVIDIA GeForce 8800 GTX:
    - Computation: 330 GFLOPS observed
    - Memory bandwidth: 55.2 GB/s observed
    - Price: $550 (board)
- **GPUs are getting faster, faster**
  - CPUs: 1.4× annual growth
  - GPUs: 1.7×(pixels) to 2.3× (vertices) annual growth

Why Are GPUs So Fast?

- GPU originally specialized for math-intensive, highly parallel computation
- So, more transistors can be devoted to data processing rather than data caching and flow control



Problem: GPGPU

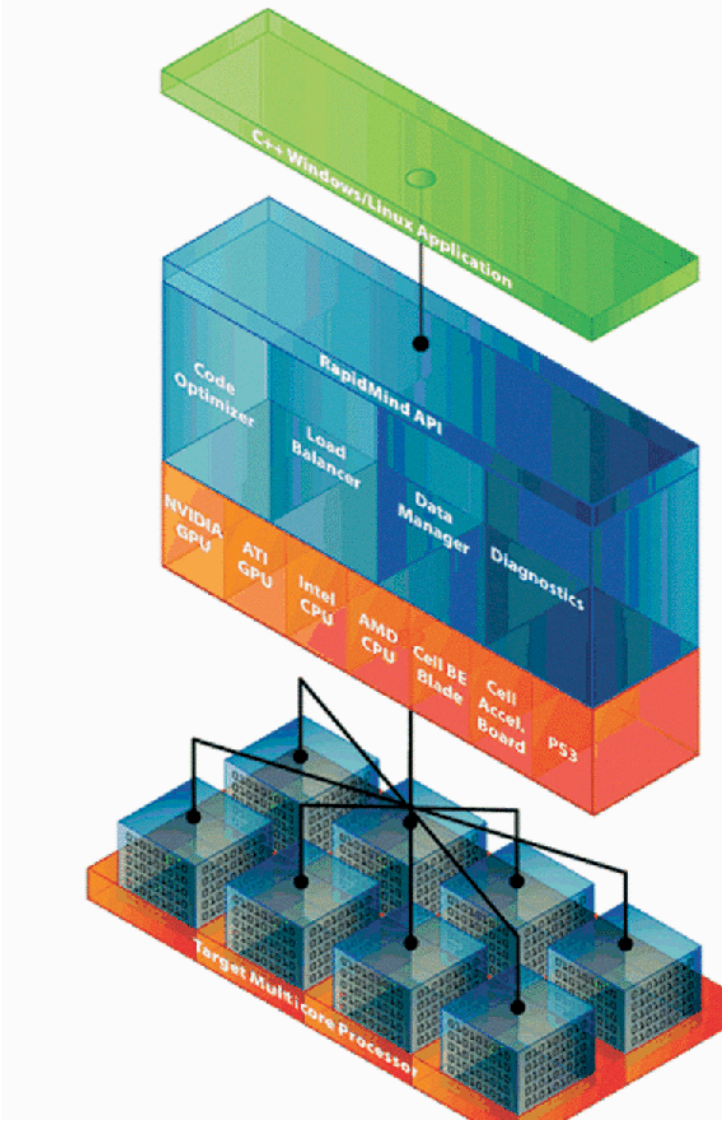
- OLD: *GPGPU* – trick the GPU into general-purpose computing by casting problem as graphics
  - Turn data into images (“texture maps”)
  - Turn algorithms into image synthesis (“rendering passes”)
- Promising results, but:
  - Tough learning curve, particularly for non-graphics experts
  - Potentially high overhead of graphics API
  - Highly constrained memory layout & access model
  - Need for many passes drives up bandwidth consumption

- New GPGPU: Many high level tools are available for use
  - Rapidmind, Peakstream(now acquired by google), CUDA …

Platform overview and Programming model

Platform overview

- RapidMind is a development and runtime platform that enables single threaded, manageable applications to fully access multi-core processors.
- With RapidMind, developers continue to write code in standard C++ and use their existing skills, tools and processes.
- The RapidMind platform then parallelizes the application across multiple cores and manages its execution.



Platform overview

- **API**
  - Intuitive, integrates with existing C++ compilers, and requires no new tools or workflow
- **Platform**
  - Code Optimizer analyzes and optimizes computations to remove overhead
  - Load Balancer plans and synchronizes work to keep all cores fully utilized
  - Data Manager reduces data bottlenecks
  - Logging/Diagnostics detects and reports performance bottlenecks
- **Processor Support Modules**
  - x86 processors from AMD and Intel
  - ATI/AMD and NVIDIA GPUs
  - Cell Blade, Cell Accelerator Board, PS3

SIMD (Single Instruction Multiple Data)

- All parallel execution units are synchronized
  - they respond to a single instruction from single program counter
- Operates on Vectors of data all of the same type
  - member elements of vector must have same meaning for parallelism to be useful
- Achieves *data level parallelism*

SPMD (Single Program Multiple Data)

- A subcategory of MIMD (Multiple Instruction Multiple Data)
- Tasks are split up and run simultaneously on different processors with different input data
- Processors run program at independent points as opposed to the lockstep execution of SIMD
- Usually refers to message passing vs shared memory

GPU SIMD/SPMD

- The processors all share the same program counter and pipeline.
  - When processor 1 is at instruction 23, all the processors are at instruction 23.
- The limited support for control flow:
  - Each processor has it's own execution mask that can conditionally be executed for one instruction.
  - Thus if you have a loop starting at instruction 10 and ending with a conditional branch on instruction 23 then; if just one processor has to continue looping but all 127 other processors are ready to leave the loop they (the 127) will be masked off from executing until the single processor has finally exited the loop.

More powerful than regular SIMD, but not have overhead on control flow.

GPU SIMD cont

- Sub grouping reduces this impact as each subgroup has it's own program counter, set of masks and processors. If the loop scenario occurs then only the processors in the group are affected - thus say in a sub group of 32 processors, 1 loops and the other 31 are masked off. The remaining processors in the other subgroups are not affected.
- Note, it is believed that it is a feature of G80 to make it more suitable for GPGPU. Not very clear that GLSL can make use of that or not.

Rapidmind SPMD

- Allows control flow in the kernel program
- More powerful than SIMD
- Example code:

```
Program p;
    p = RM_BEGIN {
    In<Value3f> a, b;
    Out<Value3f> c;

    Value3f d = f(a, b);
            RM_IF ( all( a > 2.0f ) ) {
                    c = d + a * 2.0f;
        } RM_ELSE {
                    c = d - a * 2.0f;
        } RM_ENDIF;
    } RM_END;
```

- The control flow can be converted to corresponding control flows in GLSL, but the overhead on control flow (due to hardware) still exists

Just in time compilation

- Converting program definition into OpenGL codes at runtime
- Program algebra : operations on the programs (discussed later)
- Two modes : retained mode / intermediate mode

Just in time compilation

- First, it decides which "backend" should be responsible for the program executions.
  - Backends form the connection between the RapidMind platform and a particular piece of target hardware, E.g Cell BE, OpenGL-based GPUs, and a fallback backend.
- Once a suitable backend has been chosen (a process that is generally instantaneous), it is asked to execute the program under the given conditions.
  - The first time this is done generally causes the program object to be compiled for that particular backend, similar to the way a JIT environment behaves. Once a program has been compiled, it is not recompiled.
  - This runtime compilation mechanism is powerful, as the generated code is optimized for the exact conditions it is being run under.

Retained mode and intermediate mode

- Every operation has two implementations. In immediate mode, when you ask for two numbers to be added, the computation is performed and the result returned at that time.
- At retained mode, all the operations switch from performing a computation to recording a computation.
  - All the operations you specify in a particular instance of retained mode are collected together into a sequence of operations forming a program.
  - In retained mode it looks like you are writing operations in C++, but those operations are really compiled at runtime into a program by the compiler portion of Rapidmind that targets several GPU and CPU backends.
- The true power comes into play when immediate mode and retained mode are mixed.
  - Variables with Rapidmind types declared inside a retained mode program belong to that program. Variables declared outside (i.e. in immediate mode) belong to the host application.
  - If you use a host variable from a program, it becomes a uniform parameter of that program.
  - In other shading languages you would have to explicitly manage the relationship between host variables and program variables, incurring lots of inconvenient glue code (which can sometimes be as long as the shaders you are writing). In Rapidmind, updating a uniform variable becomes as easy as assigning to it in C++. In essence the C++ scope rules are being used to manage the relationships between host code and shader code. This powerful idea extends to all other forms of C++ abstraction, enabling you to completely use functions, classes, namespaces and other forms of C++ modularity.

  - Note: these is from the discussion on libSh, but we believe that Rapidmind share the same feature.

Language Syntax

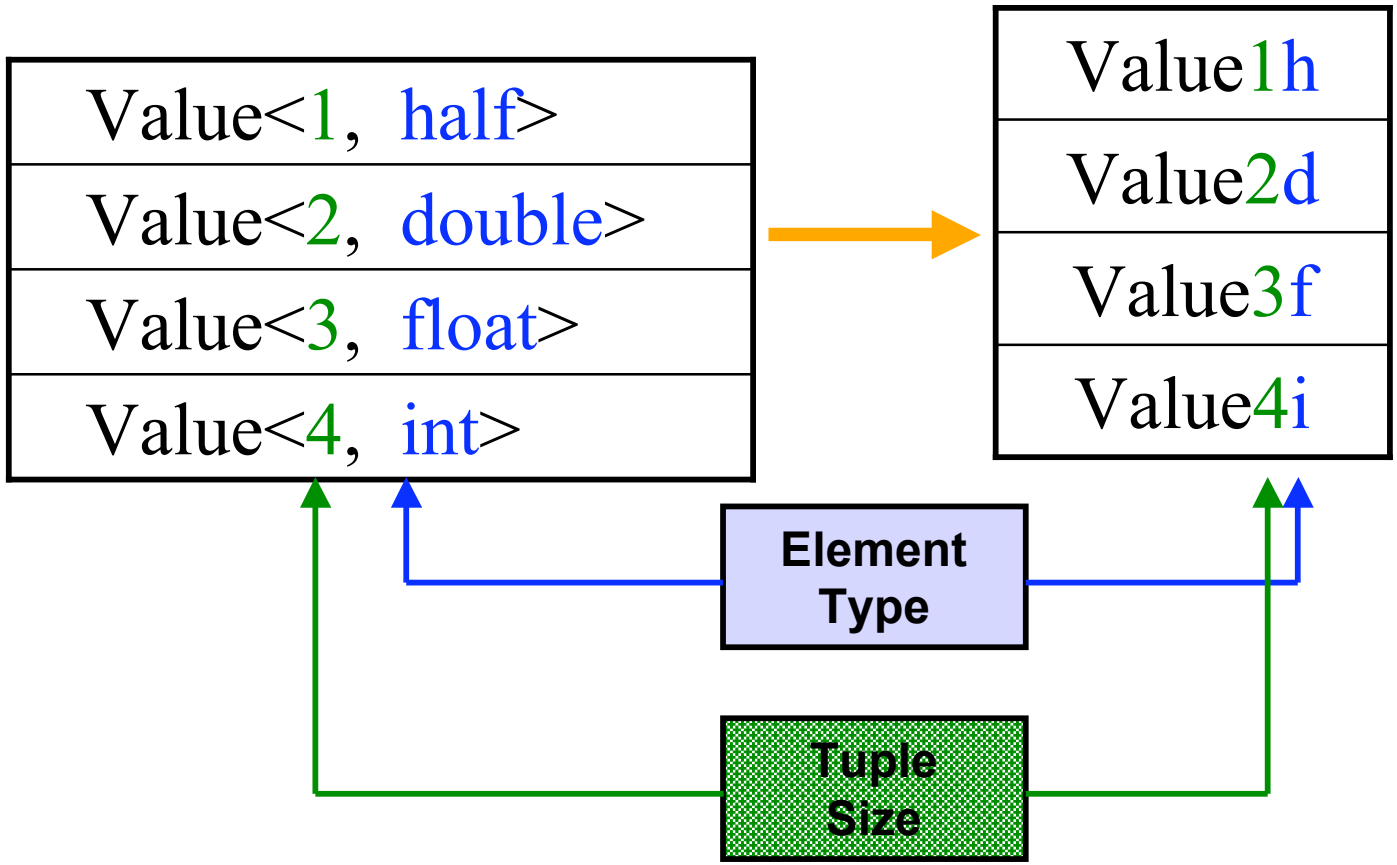
Key Concepts

- ***Vocabulary*** for parallel programming
  - Set of nouns (types) and verbs (operations)
  - Added to existing standard language: ISO C++
- A *language* implemented as a C++ *API*
  - For specifying data-parallel computation
- A data-parallel programming language
  - Embedded inside C++

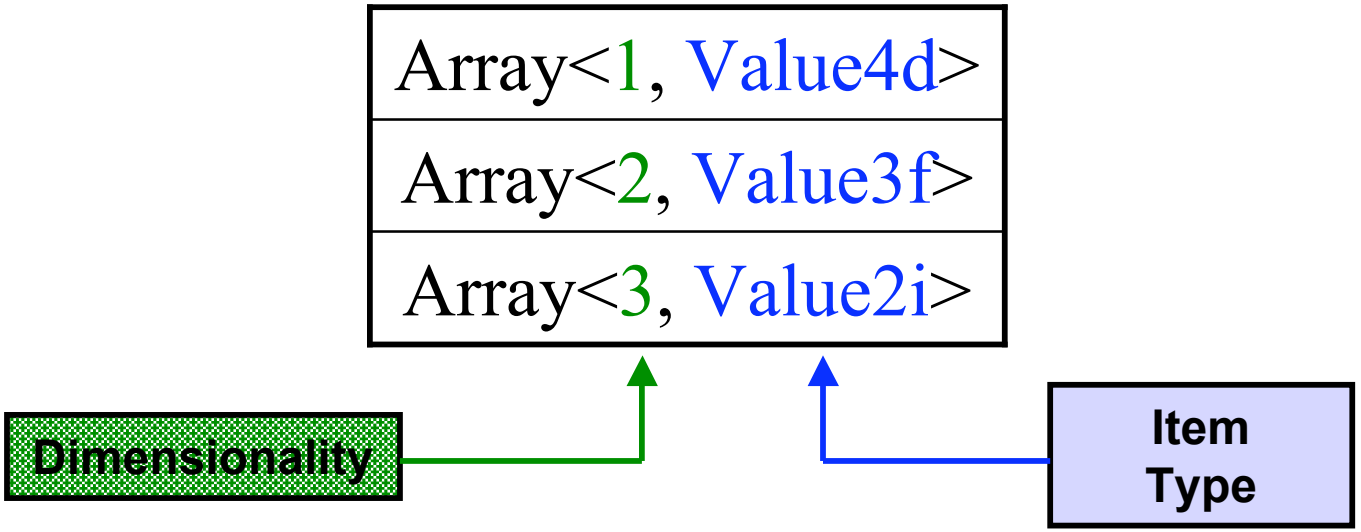
Nouns: Basic Types

Purpose	Type
Container for fixed-length data	Value
Container for variable-sized multidimensional data	Array
Container for computations	Program

Values


Value<1, half>
Value<2, double>
Value<3, float>
Value<4, int>
Value1h
Value2d
Value3f
Value4i
Element Type
Tuple Size

Arrays

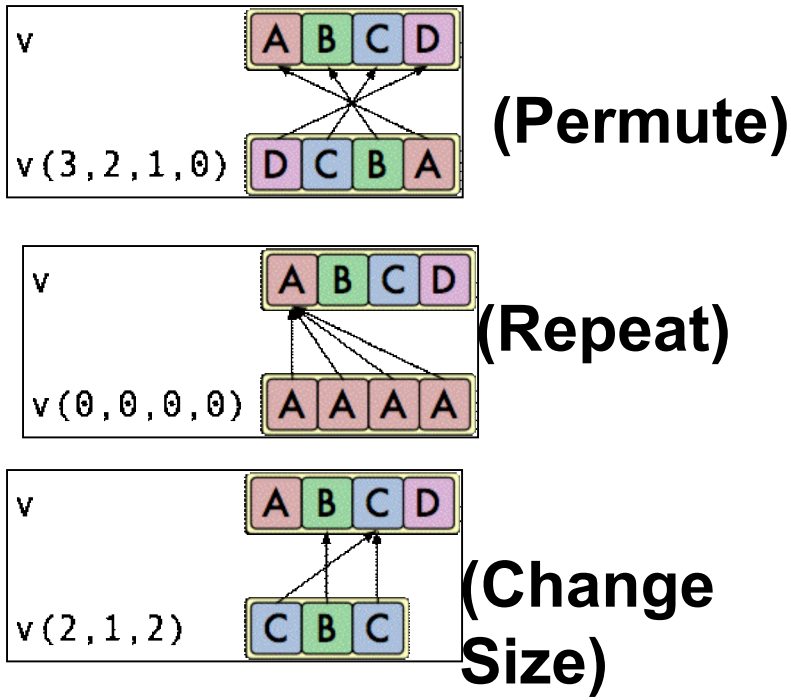

Array<1, Value4d>
Array<2, Value3f>
Array<3, Value2i>
Dimensionality
Item Type

Verbs: Operators

- RapidMind provides all the same arithmetic operations offered by C++.
  - +, -, *, /, %, etc.
- Writemasking: a[0]=(b+c)[1]
  - computes the sum of values b and c, extracts component 1 from the intermediate result, and writes it to component 0 of a.
- Swizzling:
  - v(3, 2, 1, 0)
  - v(0, 0, 0, 0)
  - v(2, 1, 2)

Verbs: Swizzling

- Swizzling refers to the extraction, rearrangement, and possible duplication of elements of value types.
  - Value4f v
  - v(3, 2, 1, 0)
  - v(0, 0, 0, 0)
  - v(2, 1, 2)



Verbs: Functions

- Can declare functions in the usual way:

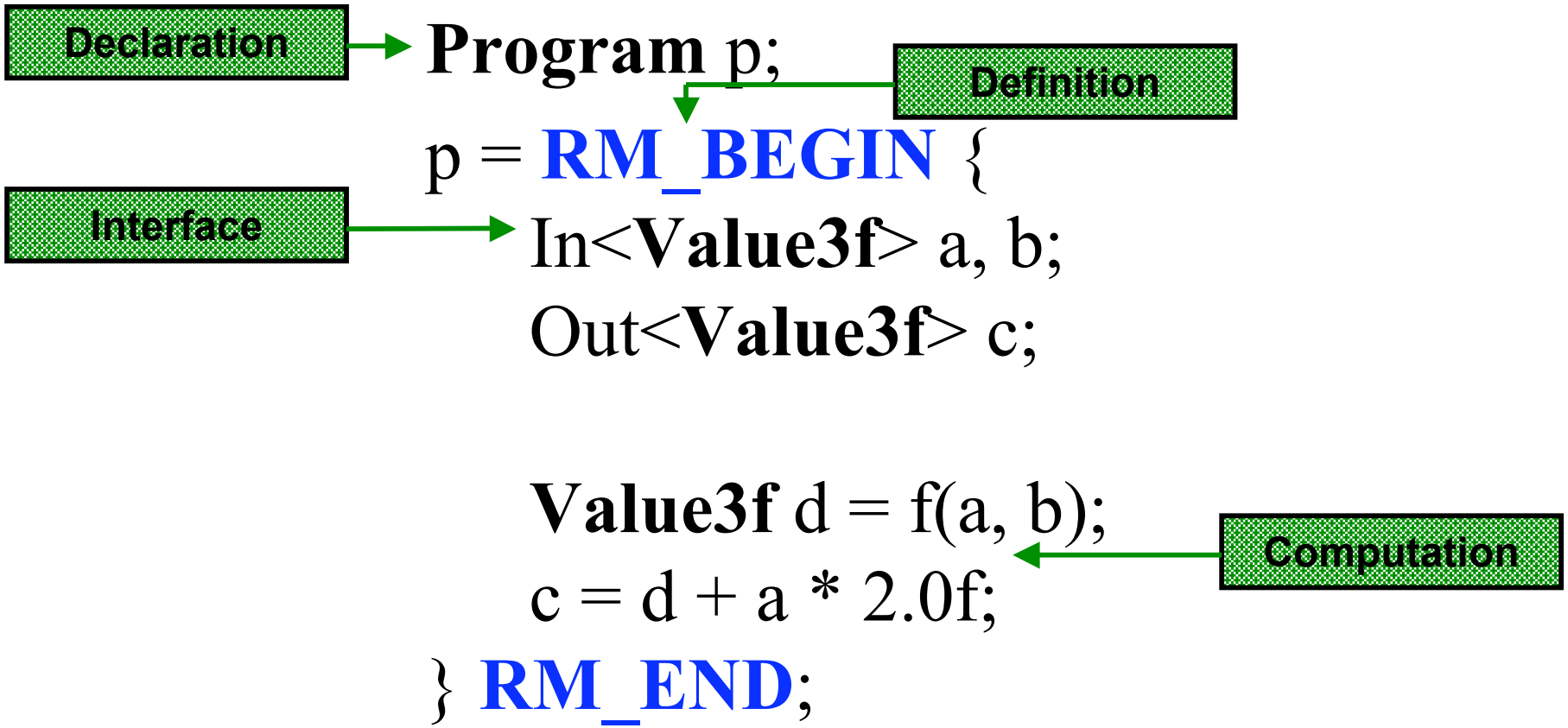
```
Value3f reflect (Value3f v, Value3f n) {
        return Value3f( 2.0*dot(n, v)*n – v );
}
```

- Standard Library
  - Matrix and Geometric Operations:
    cross product, dot, inner dot, normalize, etc.
  - Trigonometry: sin, cos, tan, asin, acos, atan, etc.
  - Exponential and Logarithms: exp, log, pow, etc.
  - Interpolation: linear interpolation
  - etc.

Programs

- **Immediate mode**
  - Execute operations on RapidMind types on host
  - Acts like a standard matrix-vector library
- **Retained mode**
  - Enter retained mode with **RM_BEGIN**, exit with **RM_END**
  - Record operations on RapidMind types
  - Store operations in **Program** object
  - Compile captured operations for coprocessor
    - Dynamic compilation

Program Definition



Control Flow

- Since C++ control structures always execute in immediate mode, special constructs are required for retained-mode control structures.
- RM_IF, RM_ELSE, RM_ENDIF
- RM_WHILE, RM_ENDWHILE
- RM_DO, RM_UNTIL
- RM_FOR, RM_ENDFOR
- RM_BREAK, RM_CONTINUE, RM_RETURN

Control Flow Example

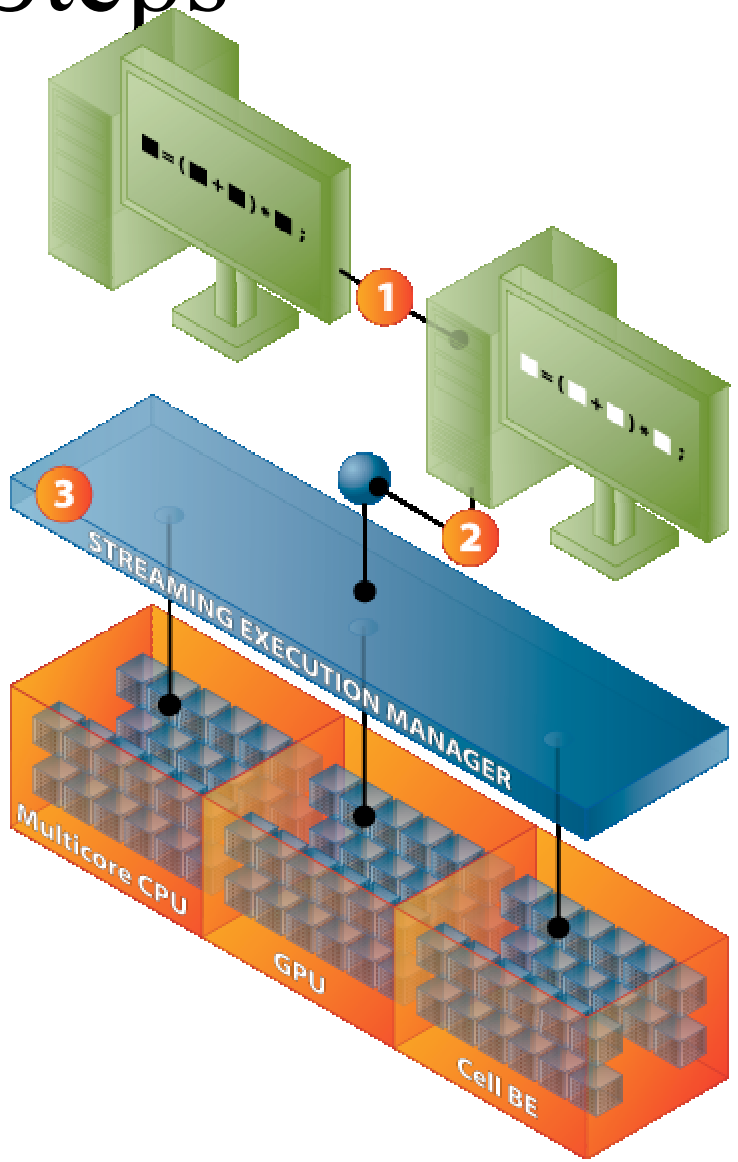
```
Program p;
p = RM_BEGIN {
      In<Value3f> a, b;
      Out<Value3f> c;

      Value3f d = f(a, b);
      RM_IF ( all( a > 2.0f ) ) {
            c = d + a * 2.0f;
      } RM_ELSE {
            c = d - a * 2.0f;
      } RM_ENDIF;
} RM_END;
```

//all: logical and of all componets

Conversion Steps

1. Replace types: developers replace numerical types with the equivalent RapidMind platform types
2. Capture computations: Application is captured, recorded, and dynamically compiled to a program object by RapidMind platform during runtime.
3. Stram execution: RapidMind platform manage parallel execution of program objects on the target hardware.



Conversion Example

```
#include <cmath>
float f;
float a[512][512][3];
float b[512][512][3];
float func(
 float r, float s
) {
 return (r + s) * f;
}
void func_arrays() {
 for (int x = 0; x<512; x++) {
  for (int y = 0; y<512; y++) {
   for (int k = 0; k<3; k++) {
    a[y][x][k] =
     func(a[y][x][k],b[y][x][k]);
   }
  }
 }
}
```

1. Replace Types

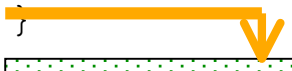
```
#include <cmath>
float f;
float a[512][512][3];
float b[512][512][3];
float func(
 float r, float s
) {
 return (r + s) * f;
}
void func_arrays() {
 for (int x = 0; x<512; x++) {
  for (int y = 0; y<512; y++) {
   for (int k = 0; k<3; k++) {
    a[y][x][k] =
     func(a[y][x][k],b[y][x][k]);
   }
  }
 }
}
```

```
#include <rapidmind/platform.hpp>
using namespace rapidmind;
Value1f f;
Array<2, Value3f> a(512, 512);
Array<2, Value3f> b(512, 512);
Value3f func(
 Value3f r, Value3f s
) {
 return (r + s) * f;
}
```

2. Capture Computations

```
#include <cmath>
float f;
float a[512][512][3];
float b[512][512][3];
float func(
 float r, float s
) {
 return (r + s) * f;
}
void func_arrays() {
 for (int x = 0; x<512; x++) {
  for (int y = 0; y<512; y++) {
   for (int k = 0; k<3; k++) {
    a[y][x][k] =
     func(a[y][x][k],b[y][x][k]);
   }
  }
 }
}
```

```
#include <rapidmind/platform.hpp>
using namespace rapidmind;
Value1f f;
Array<2, Value3f> a(512, 512);
Array<2, Value3f> b(512, 512);
Value3f func(
 Value3f r, Value3f s
) {
 return (r + s) * f;
}
```



```
void func_arrays() {
 Program func_prog = RM_BEGIN {
  In<Value3f> r, s;
  Out<Value3f> q;
  q = func(r, s);
 } RM_END;
 . . .
}
```

3. Parallel Execution

```
#include <cmath>
float f;
float a[512][512][3];
float b[512][512][3];
float func(
 float r, float s
) {
 return (r + s) * f;
}
void func_arrays() {
 for (int x = 0; x<512; x++) {
  for (int y = 0; y<512; y++) {
   for (int k = 0; k<3; k++) {
    a[y][x][k] =
     func(a[y][x][k],b[y][x][k]);
   }
  }
 }
}
```

```
#include <rapidmind/platform.hpp>
using namespace rapidmind;
Value1f f;
Array<2, Value3f> a(512, 512);
Array<2, Value3f> b(512, 512);
Value3f func(
 Value3f r, Value3f s
) {
 return (r + s) * f;
}

void func_arrays() {
 Program func_prog = RM_BEGIN {
  In<Value3f> r, s;
  Out<Value3f> q;
  q = func(r, s);
 } RM_END;
 a = func_prog(a, b);
}
```

Usage Summary

- Usage
  - Include platform header
  - Link to runtime library
- Data
  - Tuples
  - Arrays
  - Remote data abstraction
- Programs
  - Defined dynamically
  - Execute on coprocessors
  - Remote procedure abstraction

More details
on operators and functions

Operations on Values

- Modules
  - Arithmetic Operations *e.g* + - *sum product mad*
  - Trigonometric and Exponential Functions
  - Interpolation and Approximation *e.g lerp poly*
  - Geometry *e.g cross dot*
  - Logical and Comparison Functions *e.g* < > *any all*
  - Discontinuities *e.g min max abs*
  - Miscellaneous Functions *e.g cast join*

Accessing Array Contents

- operator[] :non-normalized coordinates
- operator() : normalized coordinates

Arrays, Accessors and References

- There are three flavors of RapidMind arrays: Array objects, ArrayAccessor objects and ArrayRef objects.
- An Array object represents a "physical" array containing user-defined or computed data.
- To refer to some portion of an array's data, the ArrayAccessor class is provided.
- Sometimes it is useful to be able to refer to an array whether it is an ArrayAccessor or an Array. The ArrayRef class exists for this purpose

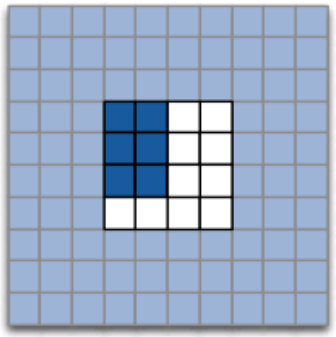
Accessors

- **offset(A,n)**
  - Drop first n elements of A
- **take(A,n)**
  - Drop all but first n elements of A
- **slice(A,i,j)**
  - Extract subarray from i to j, inclusive
- **stride(A,k)**
  - Extract every kth element
- **shift(A,k)**
  - shift k element

Return instance of **ArrayAccessor** type

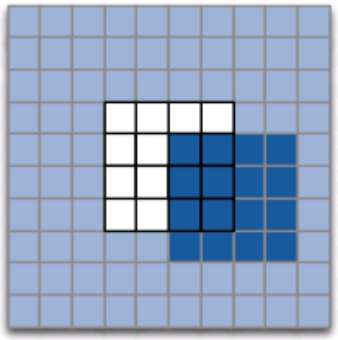
– *References* subarray “view”, does not copy

Accessors



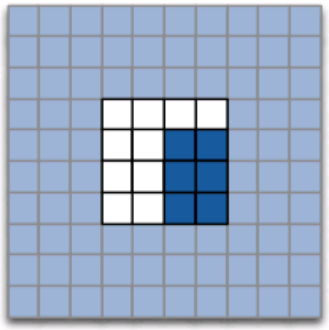
take(A, 2, 3)

The take access pattern generator.



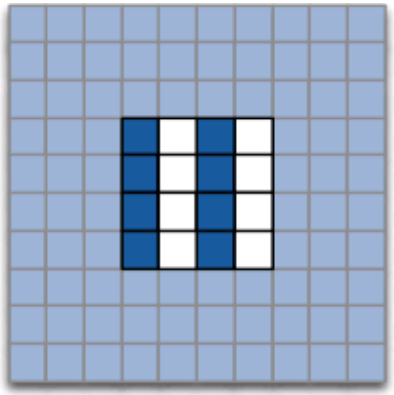
shift(A, 2, 1)

The shift access pattern generator.



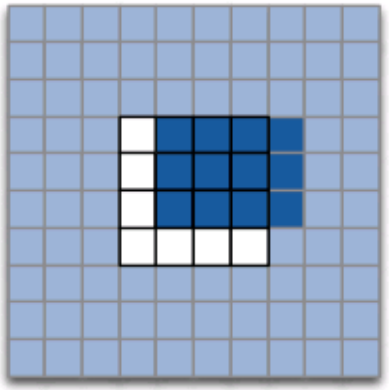
offset(A, 2, 1)

The offset access pattern generator.



stride(A, 2, 1)

The stride access pattern generator.



slice(A, 1, 0, 4, 2)

The slice access pattern generator.

Virtual Arrays

- grid
  - Generate a grid
  - Useful for providing parameters for the program
  - E.g grid(4,4) will generate a grid A with A(i,j)=(i,j)

Collective operations

- A collective operation is any operation that acts on an entire array in such a way that any element of the output can depend on any element of the input.
- Build-in collective operations
  - min max sum product…
- Customize your own collective operations
  - reduce

```
  // This reduction program sums all the first components, takes the min of all the
  // second elements, and takes the max of all the third elements of the input values.
Program p = RM_BEGIN {
  Value3f input1;   // an element from 'grades'
  Value3f input2;   // another element from 'grades'
  Value3f output = join(input1(0)+input2(0), min(input1(1),input2(1)), max(input1(2),input2(2)));
} RM_END

  // Assume that each component of a Value3f in this array has the same value.
Array<1,Value3f> grades;

Array<1,Value3f> grade_states = reduce(p,grades);
```

Program Manipulation

- Combination:
  - Program “algebra” to combine programs into new programs
  - Can use to modify interfaces to existing programs
  - Can use to specialize existing programs
- Partial evaluation:
  - Can bind inputs one at a time
  - Can convert inputs to non-local variables and vice versa
- Introspection:
  - Can analyze program interface and performance at runtime
  - Use for self-tuning libraries

Program Algebra

- **Algebra:**
  - Set of objects
  - Set of operators
  - Closed
- ***Objects*:**
  - Programs
- ***Operators****:*

– Functional composition:

p << q

– Concatenation:

combine(p,q)

Applications of the Program Algebra

- Interface adaptation
  - Reordering
  - Packing/unpacking
  - Input or output type conversion
- Specialization
  - Discard unneeded outputs
  - Eliminates unnecessary computation
- Pipelining
  - Combine producer/consumer programs into one:

A = (p << q << r)(B);

- Implement pipeline as single data-parallel task

Partial Evaluation

- Can bind only some inputs of a program, not all
- Binding gives a new program with fewer inputs
  - If bind only 1 input of an *n* input program
  - Get back program with *n*-1 inputs
- Partial evaluation provides
  - Flexibility
  - Interface adaptation
  - Optimization opportunities
- Two kinds of binding:
  - Tight: uses **()**
  - Loose: uses **<<;** is invertible using **>>**

Tight Binding

- Tight binding:

  Program q = p(A);

- Execution can be deferred
- When eventually executes:
  - Uses value of **A** in effect at time of *binding*
  - Compiler can use actual value of **A** to optimize code

Loose Binding

- Loose binding:
  Program q = p << A;
- Execution can be deferred
- When eventually executes:
  - Uses value of **A** in effect at time of *execution*
  - Value of **A** can be used to parameterize execution
- **A** acts like a non-local variable
- Can do unbinding via >>

HelloWorld: Vector addition

```
#include <rapidmind/platform.hpp>

using namespace rapidmind;

int main()
{
  // General initialization of the platform
  rapidmind::init();
  // use_backend("glsl");
  Array< 1 , Value1f> input1( 10000 );
  Array< 1 , Value1f> input2( 10000 );
  Array< 1 , Value1f> output( 10000 );

  // Access the internal arrays where the data is
     stored
  float * input_data1 = input1.write_data();
  float * input_data2 = input2.write_data();

  for ( int i = 0 ; i < 10000 ; ++i )
  {
        input_data1[ i ] = i;
        input_data2[ i ] = i * 2;
  }

// 2. Performing computation
  // The stream program that will be executed on the data
  Program prg = RM_BEGIN {
    In<Value1f> a;    // first input
    In<Value1f> b;    // second input
    Out<Value1f> c;   // output

    c = a + b;              // operation on the data
  } RM_END;

  // Execute the stream program
  output = prg(input1, input2);

  // 3. Showing results
  const float* results = output.read_data();

  for ( int i = 0; i < 10000 ; ++i )
  {
    std::cout      << "output[" << i << "] = ("
                   << results[ i ] << ")"
                   << std::endl;
  }

}
```

Reduction

- Many collective operators are already built-in, e.g. sum, product, min, max
- How reduction is implemented?

```
Program add_prog = RM_BEGIN {
  In< Valuelf > inl, in2;
  Out< Valuelf > out;
  out = inl + in2;
} RM_END;

for (int i = 0; i < log_width; ++i) {
  Array<1, Valuelf> pass = add_prog(stride(array, 2 ), stride(offset(array, 1 ), 2 ));
  array = pass;
}
```

- An log(N) algorithm

Q: The *values* are tuples like Value3f, why?

- A: Rapidmind starts from academic toolkit libsh, a GPGPU programming environment. The *Typedef*s for up to four elements of all basic types provided by the platform. These are natural in GPU programming and basic OpenGL concepts. OpenGL represents vertex coordinates, texture coordinates, normal vectors, and colors generically as tuples. Tuples can be thought of as 4-component vectors.

- Comments: Even on other architectures, the vector form can help make use of SIMD instructions
- Note: NVIDIA G80 has a scalar architecture.

Q:Rapidmind Versus CUDA

- CUDA is NVIDIA G80 exclusive, and free
- Rapidmind is based on OpenGL, can support almost all Graphics card, also has other backends, e.g cell, cc
- Performance: on G80 cards, CUDA will deliver better performance, as the programmer can have control on the small block of shared memory directly, also do not have overhead to go through OpenGL

Q: How is the parallel program scheduled to run?

- Single thread of control
- Data parallelism
- Race-free

- We can assume
  - Each stream execution of the *program object* is using a *fork and join* model, and a barrier is always put after the evoking the stream execution (function call)

Demo

Q/A