Brook+ Data Types

- Important for all data representations in Brook+
  - Streams
  - Constants
  - Temporary variables
- Brook+ Supports
  - Basic Types
  - Short Vector Types
  - User-Defined Types

Basic Data Types

- Basic data types
  - float 32-bit floating point
  - double 64-bit floating point
  - int 32-bit signed integer
  - unsigned int/uint 32-bit unsigned integer

  e.g. Stream<float> f;
  Stream<uint> i;
  Stream<double> d;
- Literal Constants for above types

  float a = 0.2f;
  double d = 0.3;
  int c = 10;

Basic Data Types

- Unsupported types (Reserved for future use – currently keywords)
  - char
  - unsigned char
  - short
  - unsigned short
  - long
  - unsigned long
  - ulong
  - long long

Vector Types

- Short vectors (2 to 4 elements) inherited from 3D graphics
  - traditionally used to represent positions (x, y, z, w) or color (r, g, b, a)
  - GPU registers are 4-component registers, 128-bit per register
  - names built by appending the count to the type

    e.g. int2, float4
  - doubles are limited to up to 2 elements

Vector Types – Initialization &Operations

- C++ constructor-style initialization
  float4 a = float4(1.0f, 2.0f, 3.0f, 4.0f);
  int2 b = int2(1, 3);
- 8 Access to individual fields is through structure member syntax:
  .x, .y, .z, .w
  int k = b.x + b.y * b.y;
- Arithmetic operations on operands of vector types are supported
  - equivalent to applying the operator to each component individually
  - handy for quick performance tuning and data compaction
    float4 b = float4(1.0f, 0.0f, 10.0f, 5.0f);
    float4 k1 = a + b;

Vector Types - Usage

Original Kernel

```
void main()
{
   unsigned int count = 1000;
   Stream<float> a(1, &count),
   b(1, &count), c(1, &count);
   ......
   sum(a, b, c);
}

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

Vectorized Kernel

```
void main()
{
   unsigned int count = 1000/4;
   Stream<float4> a(1, &count),
   b(1, &count), c(1, &count);
   ......
   sum(a, b, c);
}

kernel
void sum(float4 a<>,
   float4 b<>, out float4 c<>)
{
   c = a + b;
}
```

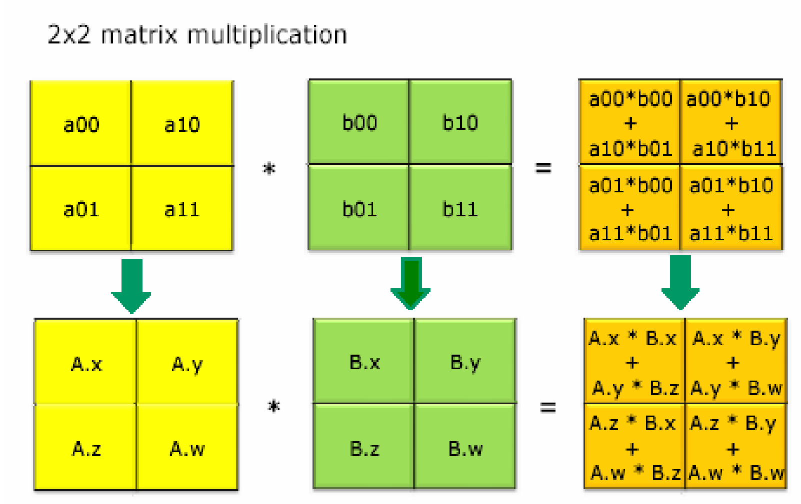
Vector Types – Swizzle operations

- Reordering components of a short vector for certain operations
  - Nice shorthand for developers
  - Optimization hint to compiler
- Brook+
  - Use Swizzle to reorder elements in operands
    e.g. temp = input.yxzx + temp;
  - Use Swizzle to mask elements in output
    e.g. output.xw = input.x;
- C – Cannot use vector instructions without reordering explicitly
- float temp[4] = {0, 1, 2, 3}; // yxzx
  temp[0] += input.y; temp[1] += input.x;
  temp[2] += input.z; temp[3] += input.x;

Swizzle Examples

```
float4 pos = float4(3.f, 5.f, 2.f, 1.f);
float value1 = pos.x;        // value1 is 3.f
float2 vec0 = pos.xy;        // vec0 is {3.f, 5.f}
float4 vec1 = pos.yzxx       // vec1 is {5.f, 2.f, 3.f, 3.f}
vec1.x = pos.z;              // vec1 is {2.f, 2.f, 3.f, 3.f}
```

Swizzle usage



Swizzle usage

- Originally A * B would be written as

```
float4 mult;
mult.x = A.x * B.x + A.y * B.z;
mult.y = A.x * B.y + A.y * B.w;
mult.z = A.z * B.x + A.w * B.z;
mult.w = A.z * B.y + A.w * B.w;
```

- Using swizzles, A * B would be written as

```
float4 mult = A.xxzz * B.xyxy + A.yyww * B.zwzw;
```

User defined types

- Basic Types and Short Vectors can be aggregated using user defined structs

- Syntax for declaration and data access is same as C structs

```
typedef struct PairRec
{
    float row;
    float column;
} Pair;
kernel void foo(float c, Pair p<>, out float b<>)
{
    b = c * p.row + p.column;
}
```


User defined types

- Streams of structs are internally translated into multiple streams
  - In previous example, Pair p<> will be translated into 2 float streams
  - Data processed on the host-side during Stream::read()
    - One-time Overhead
    - Might be better to use float2 Stream in this case

Type checking

- brcc performs strong type-checking
  - Operands must have same
    - Variable type
    - Number of components (short vectors)
    - Constant type (literal constants)
  - Implicit conversions are NOT allowed
    - Initializations
    - Assignments (LHS and RHS should have same type)
    - Arithmetic Expressions
- Type-promotion not done with strong type-checking

- Relational operations on vector types use only x component

Standard Streams

- Standard streams passed using open brackets - <>
  - Positions of Input and output elements are implicit and predictable

    e.g.

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

Stream rank

- Need to be specified explicitly during creation/allocation
- Up to 3-dimensional streams supported

```
int dim2[2] = {width, height};
Stream<float> str2D(2, dim2);
int dim3[3] = {width, height, depth};
Stream<float> str3D(3, dim3);
```

- Rank not specified when used in Kernel

```
kernel void(float str2D<>, float str3D<>, out str1D<>);
```

- Stream Dimensions and Ranks should match for well-defined results
  - 1.2 and prior supports Stream resize
  - 1.3 runtime will issue a warning/error on mismatch

Arbitrary Reads - Gather

- Implicit addressing mechanism is restrictive for many interesting applications
  - Desirable to arbitrarily address streams
  - Access is called a gather operation in stream terminology

- Gather streams
  - Can be arbitrarily addressed
  - Are not subject to alignment like input streams – can have arbitrary dimensions relative to output stream
- Declared using square brackets - [], as kernel arguments
  - Declaration and indexing 'within the kernel' uses C-array syntax
  - Stream declaration outside the kernel does not change
  - Need to specify dimensionality like C arrays, i.e. [] for 1D and [][] for 2D streams

Gather streams

```
kernel void simpleGather( int index<>, float data[],
                          out float result<> )
{ result = data[index]; }
...

Stream<int> indices(1, &count);
Stream<float> data(1, &count);
Stream<float> result(1, &count);
...
simpleGather( indices, data, result );
// result[i] = data[indices[i]];
```

Gather stream declaration

Gather stream read

No change in Stream allocation

Gather streams

- Dimensionality needs to be specified in kernel arguments, e.g. `[][]` for 2D and `[][][]` for 3D gather
- Indexed using C-style indexing

```
kernel void gather_direct_2D(float2 index, float a[][],
                             out float b<>)
{
    b = a[index.y][index.x];
}
```

2D Gather

Index using a C-style indexing

- Can have arbitrary number of gather operations in a kernel
  - Also referred to as *arrays* for obvious reasons
  - Index should be of integer type with C-style numbering (0..)
  - `float` indexing allowed for backward compatibility – might be deprecated in future releases

Array dimensions

- Array Allocation order (same when using vector types)
  unsigned int dims[4] = {x, y, z, w};
- Array Reference order (same when using scalar types)
  A[w][z][y][x];
- Use of gather streams is more popular in real-life applications
  - General-purpose
  - Better control over memory access patterns

Instance()

- In C, applications explicitly control the current (x, y) index being processed

```
for(int y = 0; y < N; y ++)
      for(int x = 0; x < M; x ++)
            float temp = A[y][x];
```

- In Brook+, kernels are executed for each element in output stream implicitly over the execution domain
- Querying the current kernel instance being processed is useful in various cases
- instance() intrinsic returns an int4 vector for current kernel instance
  - For a given stream rank, invalid components are set to 0
  - Similar to pthread_self() with a spatial context

Stream outputs

- C – Writing to arrays is very flexible

```
out0[x][y] = value;
out1[y][x] = value2;
```

- Write to output streams on GPUs has been quite restricting
  - Traditionally restricted to writing pixel values to a single color buffer
  - Output position of the output result was also fixed
- Brook+ - Write to output stream at the implicit position

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


Multiple outputs

- Legal to output multiple streams from a kernel
  - Simply specify multiple out streams, e.g.

```
kernel void multi_out(float a<>, out float b<>,
out float c<>, out float d<>)
{
    b = a * a + a;
    c = a * a - a;
    d = a * a * a;
}
```

- Results undefined if dimensions of output streams differ
  - Dimensions of first output stream in argument list is used for domain execution

Multiple output streams

- Current AMD GPUs limit maximum outputs to 8
  - Brook+ allows exceeding the GPU limits by using a multi-pass algorithm
    - Existing kernel is split into multiple kernels
    - Each sub-kernel is executed in a serial manner

  e.g. Following kernel is legal in Brook+

```
kernel void foo(float inp<>,
out float b1<>, out float b2<>, out float b3<>, out float b4<>,
out float b5<>, out float b6<>, out float b7<>, out float b8<>,
out float b9<>, out float b10<>)
{
    b1 = inp;   b2 = inp;
    b3 = inp;   b4 = inp;
    b5 = inp;   b6 = inp;
    b7 = inp;   b8 = inp;
    b9 = inp;   b10 = inp;
}
```


Arbitrary writes - Scatter

- Legal to output to arbitrary address from a kernel
- Output stream needs to be declared and indexed as a scatter stream using the C array syntax, []
  - Similar to gather streams

  e.g. Following kernel computes an output steam based on the addresses given in the index stream

```
kernel void
scatter(float index<>, float a<>, out float4 b[])
{
    b[index] = a;
}
```

Scatter

- Currently scatter operations are slower than normal output streams
  - GPU requires stream elements to be 128-bit aligned (notice the use of float4 type in previous example)
  - Brook+ supports all data types using appropriate conversions
- Only works on R670 and later. Does not work on R600 (HD 2900)
- Allows arbitrary number of writes in a kernel

```
kernel void
multiple_writes(float a<>, out float4 b[])
{
    int2 ind = instance().xy;
    b[ind.x] = a;
    b[ind. y] = a;
}
```

- Order of writes not guaranteed in case of same output address for multiple kernel instances

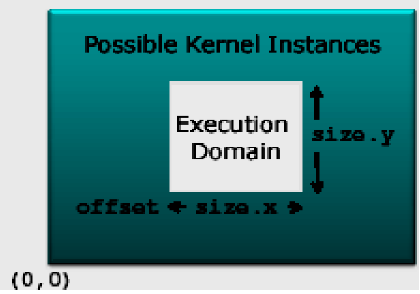
Domain of Execution

- Corresponds to the number of kernel instances created by the runtime
- Maps directly to number of elements in output stream
- However, if output stream is a scatter stream
  - No necessary correlation between output stream dimensions and domain of execution

    E.g. output domain might be larger than execution domain
- Kernel Interface allows explicit control over execution domain
  - brcc generated Kernel prototype derives from KernelInterface class with () operator overridden

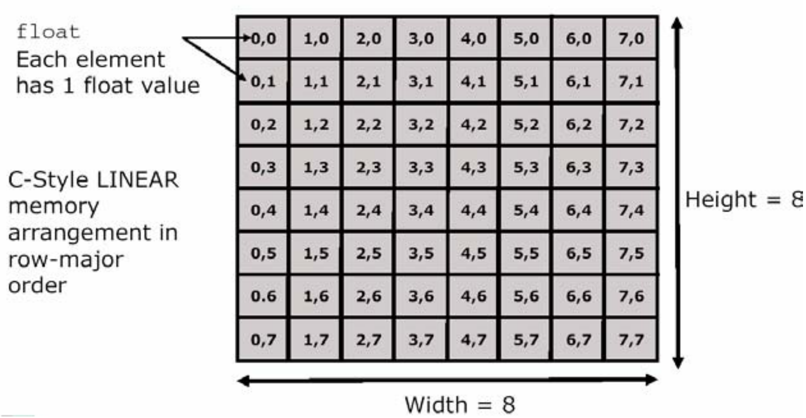
KernelInterface class

```
class KernelInterface
{
        public:
           void domainOffset(uint4 offset); // e.g. (1024, 1024, 0, 0);
           void domainSize(uint4 size);     // e.g. (512, 512, 1, 1);
           ..............
};
......
scatter.domainOffset(offset);
scatter.domainSize(size);
scatter(index, a, b);
......
```

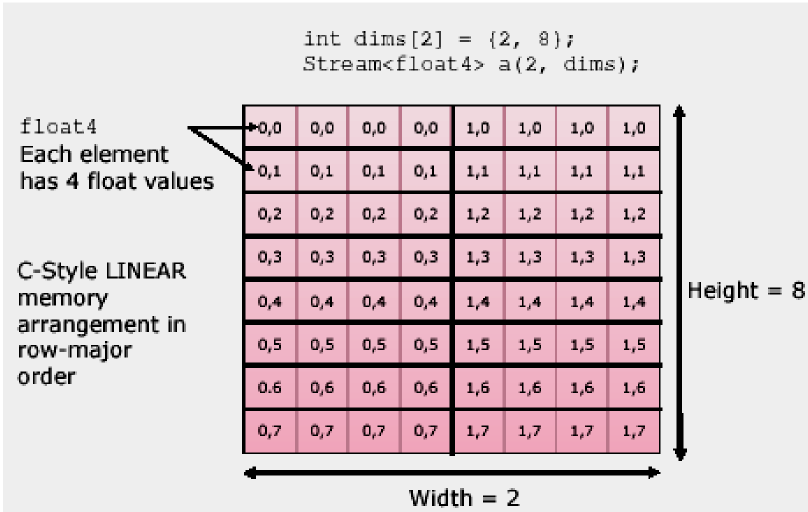


Streams: Data layout

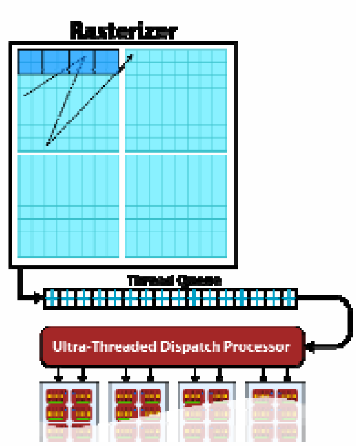
```
int Width = 8, Height = 8;
float a<Height, Width>;
```



Data layout of streams of vector types

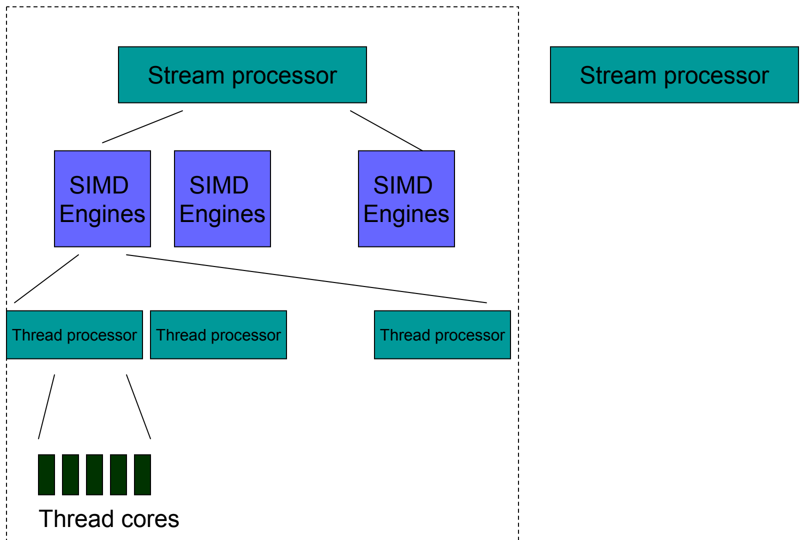


Thread Creation:

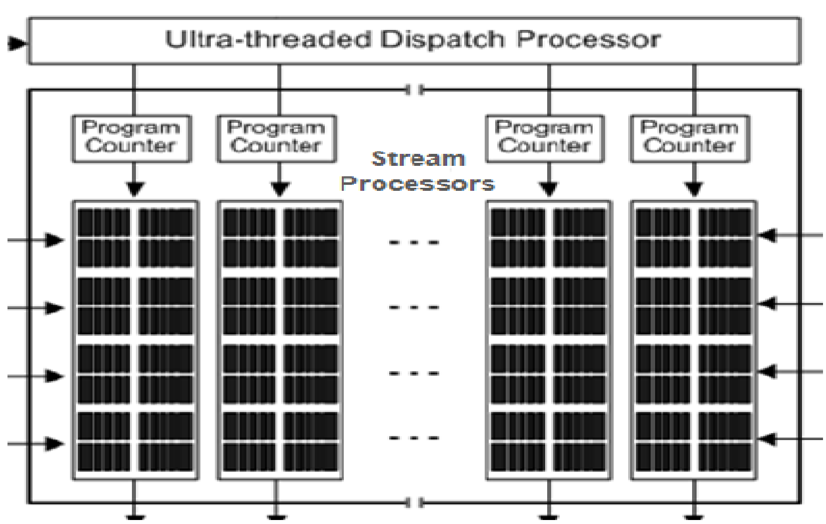


- Threads are created during rasterization of domain
- Wavefront = ∑ Quads
- Quad = 2x2 Threads
  - Non-active threads in a domain => Idle SPUs
- Order of thread creation = rasterization order (no disclosed)
  - Based on multiples of 8x8 blocks = size of wavefront (64-threads)

Thread Execution: Hardware Structure



Thread execution: RV770 structure



Thread execution: thread processing

- All thread processors within a SIMD engine execute the same instruction for each cycle
- In a thread processor, up to four threads can issue 4 VLIW instructions over four cycles
- RV 670 and RV 770, 16 thread processors execute the same instructions, with each thread processor processing four threads => appear to be a 64-wide SIMD engine called a *wavefrount*.
- SIMD engines operate independently of each other

Flow control

- Branches
  - If a kernel has a branch with two paths, wavefront executes first and then second
  - Branch Granularity = Number of threads executed during a branch
  - Branch Granularity = Wavefront size
- Loops
  - Total execution time for wavefront = Time of longest loop
- Other SIMDs execute independently in case of branches and loops

Constants

- All non-stream kernel arguments are treated as constants
  - Value remains same for all kernel instances in the domain
  - Only basic and short vector data types are supported
  - All other operations like swizzle are valid

```
kernel void
scale(float k1, float2 k2, float4 k4,
float a<>, out float b<>)
{
   b = a * k1 + k2.y + k4.w;
}
```


Cached arrays

- GPUs have special caches for small read-only arrays (constant buffers)
  - Used for various constants to be used in computation
- Cached arrays can be used with brcc and runtime
  - Kernel hints on array size – specified using literal constants for array size

```
kernel void cachedArray(float data[1024],
                                   out float result<>)
```

  - Runtime allows directly passing the array pointer to kernel invocation routine

```
float *data = getConstantData();
cachedArray(data, outStream);
if(outStream->error())
{
  cerr << "Error in output stream: "
             << outStream->errorLog() << endl;
}
```

Cached Arrays

- Can result in significant performance improvements for specific applications like
  - Kernel filters
  - Transfer functions
- GPU restriction on maximum cache size apply
  - 4096 elements per cached array
  - Max 10 cached arrays allowed by Brook+
- Array size should be completely defined at compile time

```
kernel void test(float data[1024][], out float result<>)
```

  - ERROR: Problem with Array variable declaration:…

Reduction - Motivation

C

```
// Declare array
float mat[50];

// Iterate over output elements
float sum = 0.0;
for(int i = 0; i < 50; i++)
  sum += mat[i];
```

- Common in various applications like financial algorithms, numerical simulations, etc.
  - E.g. Average the set of values computed from random samples to get one price

Reduction

- Provide a data-parallel method for computing a single value from a set of records
  - Examples of reduction operations include maximum, median, arithmetic sum, matrix product, etc
  - Output stream has a single element or lesser elements compared to input stream
- Operation required to be associative and commutative, i.e.

  (a b) c = a (b c)

  a b = b a
- Output specified using reduce keyword
- Kernel uses reduce modifier instead of kernel keyword
- Legal to read and write to reduce parameter during computation

Reduction example

C

```
// Declare array
float mat[50];

// Iterate over output elements
float sum = 0.0;
for(int i = 0; i < 50; i++)
  sum += mat[i];
```

Brook+

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

unsigned int dim1 = 50;
Stream<float> mat(1, &dim1);
float res;
unsigned int dim2 = 25;
Stream<float> res2(1, &dim2);
// reduce to single element
sum(mat, res); // res = ∑ mat[i]
// reduce number of elements
sum(mat, res2);
// res2[i] = mat[2 * i] + mat[2 * i + 1]
```

Reduction

- Number of input elements used to produce output elements determined by number of output elements
  - Implemented using multiple passes over the kernel as needed
  - If single element along a dimension, all elements are combined

=> Dimension Reduction, e.g.

uint in[2] = {100, 200};

uint out = 100;

Stream<float> s(2, &in) reduced to

Stream<float> t(1, &out)

Valid to pass scalar element as output stream to runtime

– If stream, factor between dimensions is used

=> Partial Reduction, e.g.

uint out2[2] = {100, 50};

Stream<float> s(2, &in) reduced to Stream<float> t<2, &out2>

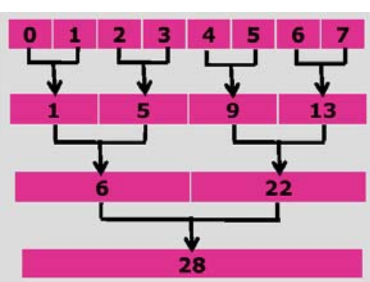
Calling a reduce kernel

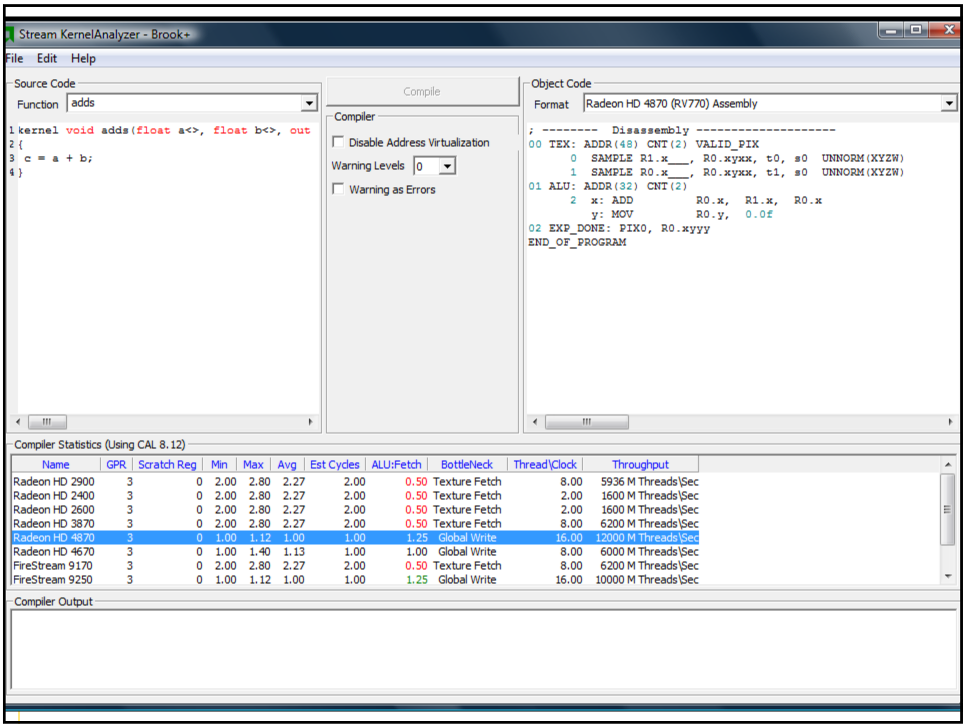
- Apply the kernel to *n* input elements to compute one output elements, where *n=reduceFactor*

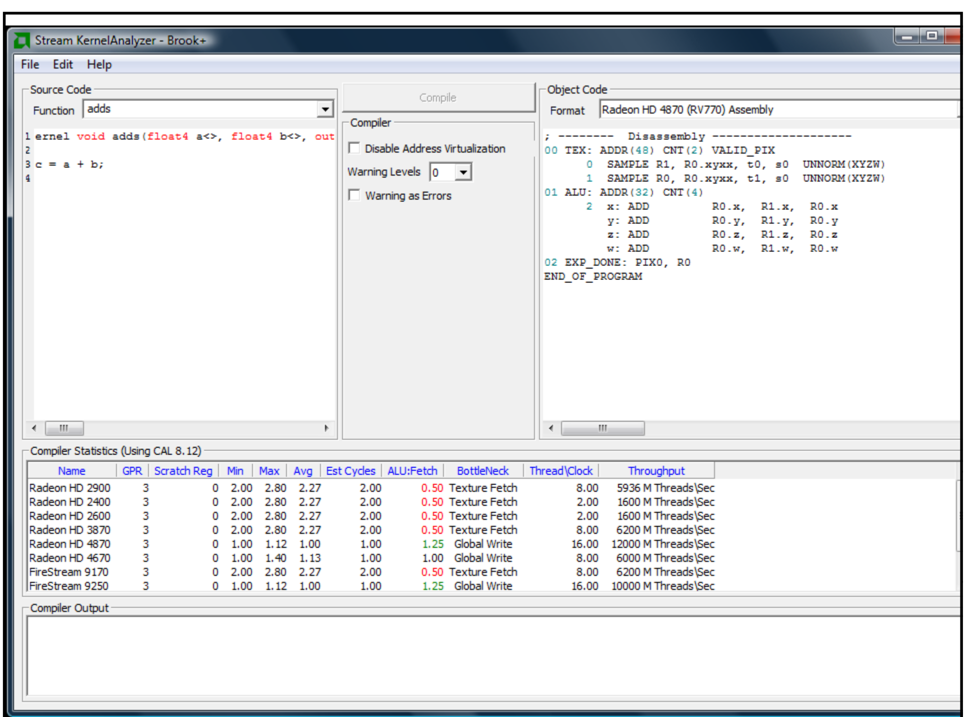
reduceFactor * |output stream| = |input stream|

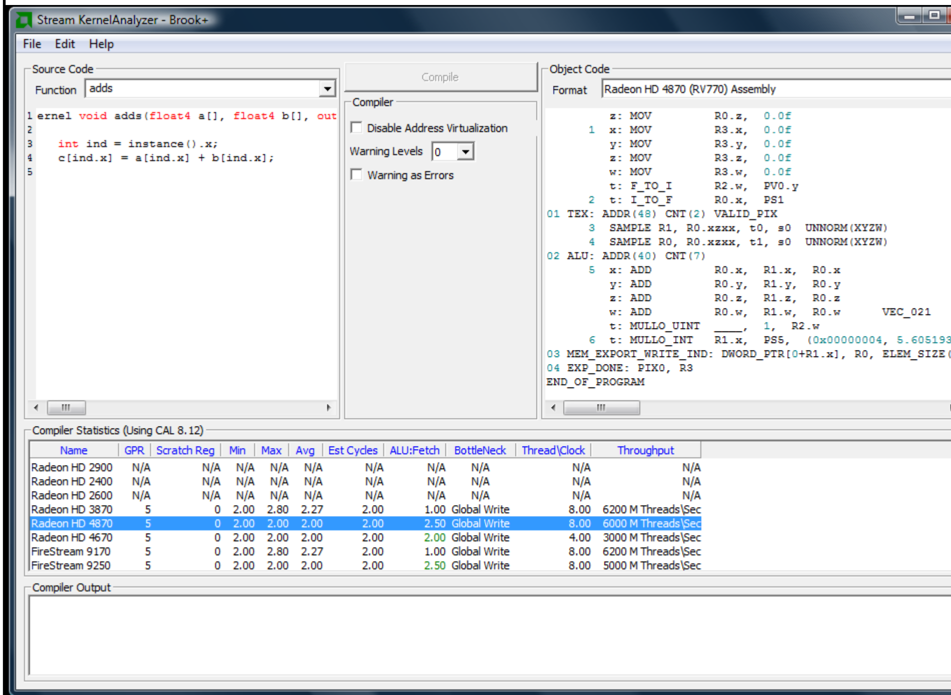
```
reduce void
sum (float a<>, reduce float c<>)
{
        c = c + a;        // c += a;
}
```

Apply the above 1-dimension rule to each dimension



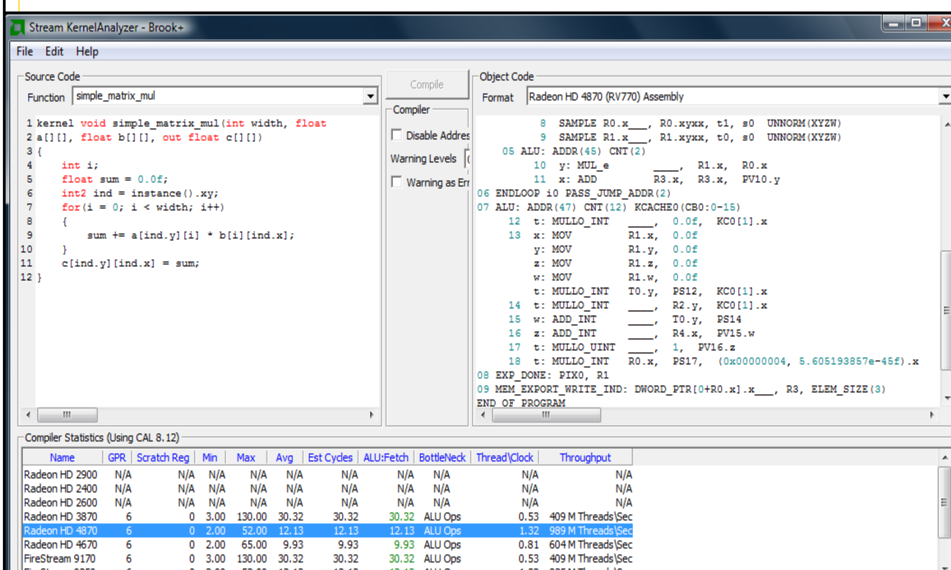

Stream KernelAnalyzer - Brook+
File Edit Help
Source Code
Function adds
1 kernel void adds(float a<>, float b<>, out
2 {
3 c = a + b;
4 }
Compile
Compiler
Disable Address Virtualization
Warning Levels 0
Warning as Errors
Object Code
Format Radeon HD 4870 (RV770) Assembly
; -------- Disassembly --------------------
00 TEX: ADDR(48) CNT(2) VALID_PIX
0 SAMPLE R1.x___, R0.xyxx, t0, s0 UNNORM(XYZW)
1 SAMPLE R0.x___, R0.xyxx, t1, s0 UNNORM(XYZW)
01 ALU: ADDR(32) CNT(2)
2 x: ADD R0.x, R1.x, R0.x
y: MOV R0.y, 0.0f
02 EXP_DONE: PIX0, R0.xyyy
END_OF_PROGRAM
Compiler Statistics (Using CAL 8.12)
Name GPR Scratch Reg Min Max Avg Est Cycles ALU:Fetch BottleNeck Thread\Clock Throughput
Radeon HD 2900 3 0 2.00 2.80 2.27 2.00 0.50 Texture Fetch 8.00 5936 M Threads\Sec
Radeon HD 2400 3 0 2.00 2.80 2.27 2.00 0.50 Texture Fetch 2.00 1600 M Threads\Sec
Radeon HD 2600 3 0 2.00 2.80 2.27 2.00 0.50 Texture Fetch 2.00 1600 M Threads\Sec
Radeon HD 3870 3 0 2.00 2.80 2.27 2.00 0.50 Texture Fetch 8.00 6200 M Threads\Sec
Radeon HD 4870 3 0 1.00 1.12 1.00 1.00 1.25 Global Write 16.00 12000 M Threads\Sec
Radeon HD 4670 3 0 1.00 1.40 1.13 1.00 1.00 Global Write 8.00 6000 M Threads\Sec
FireStream 9170 3 0 2.00 2.80 2.27 2.00 0.50 Texture Fetch 8.00 6200 M Threads\Sec
FireStream 9250 3 0 1.00 1.12 1.00 1.00 1.25 Global Write 16.00 10000 M Threads\Sec
Compiler Output


Stream KernelAnalyzer - Brook+
File Edit Help
Source Code
Function adds
1 ernel void adds(float4 a<>, float4 b<>, out
2
3 c = a + b;
4
Compile
Compiler
Disable Address Virtualization
Warning Levels 0
Warning as Errors
Object Code
Format Radeon HD 4870 (RV770) Assembly
; -------- Disassembly --------------------
00 TEX: ADDR(48) CNT(2) VALID_PIX
0 SAMPLE R1, R0.xyxx, t0, s0 UNNORM(XYZW)
1 SAMPLE R0, R0.xyxx, t1, s0 UNNORM(XYZW)
01 ALU: ADDR(32) CNT(4)
2 x: ADD R0.x, R1.x, R0.x
y: ADD R0.y, R1.y, R0.y
z: ADD R0.z, R1.z, R0.z
w: ADD R0.w, R1.w, R0.w
02 EXP_DONE: PIX0, R0
END_OF_PROGRAM
Compiler Statistics (Using CAL 8.12)
Name GPR Scratch Reg Min Max Avg Est Cycles ALU:Fetch BottleNeck Thread\Clock Throughput
Radeon HD 2900 3 0 2.00 2.80 2.27 2.00 0.50 Texture Fetch 8.00 5936 M Threads\Sec
Radeon HD 2400 3 0 2.00 2.80 2.27 2.00 0.50 Texture Fetch 2.00 1600 M Threads\Sec
Radeon HD 2600 3 0 2.00 2.80 2.27 2.00 0.50 Texture Fetch 2.00 1600 M Threads\Sec
Radeon HD 3870 3 0 2.00 2.80 2.27 2.00 0.50 Texture Fetch 8.00 6200 M Threads\Sec
Radeon HD 4870 3 0 1.00 1.12 1.00 1.00 1.25 Global Write 16.00 12000 M Threads\Sec
Radeon HD 4670 3 0 1.00 1.40 1.13 1.00 1.00 Global Write 8.00 6000 M Threads\Sec
FireStream 9170 3 0 2.00 2.80 2.27 2.00 0.50 Texture Fetch 8.00 6200 M Threads\Sec
FireStream 9250 3 0 1.00 1.12 1.00 1.00 1.25 Global Write 16.00 10000 M Threads\Sec
Compiler Output



Name	GPR	Scratch Reg	Min	Max	Avg	Est Cycles	ALU:Fetch	BottleNeck	Thread\Clock	Throughput
Radeon HD 2900	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Radeon HD 2400	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Radeon HD 2600	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Radeon HD 3870	5	0	2.00	2.80	2.27	2.00	1.00	Global Write	8.00	6200 M Threads\Sec
Radeon HD 4870	5	0	2.00	2.00	2.00	2.00	2.50	Global Write	8.00	6000 M Threads\Sec
Radeon HD 4670	5	0	2.00	2.00	2.00	2.00	2.00	Global Write	4.00	3000 M Threads\Sec
FireStream 9170	5	0	2.00	2.80	2.27	2.00	1.00	Global Write	8.00	6200 M Threads\Sec
FireStream 9250	5	0	2.00	2.00	2.00	2.00	2.50	Global Write	8.00	5000 M Threads\Sec

Matrix Multiplication



Name	GPR	Scratch Reg	Min	Max	Avg	Est Cycles	ALU:Fetch	BottleNeck	Thread\Clock	Throughput
Radeon HD 2900	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Radeon HD 2400	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Radeon HD 2600	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Radeon HD 3870	6	0	3.00	130.00	30.32	30.32	30.32	ALU Ops	0.53	409 M Threads\Sec
Radeon HD 4870	6	0	2.00	52.00	12.13	12.13	12.13	ALU Ops	1.32	989 M Threads\Sec
Radeon HD 4670	6	0	2.00	65.00	9.93	9.93	9.93	ALU Ops	0.81	604 M Threads\Sec
FireStream 9170	6	0	3.00	130.00	30.32	30.32	30.32	ALU Ops	0.53	409 M Threads\Sec
FireStream 9250	6	0	2.00	52.00	12.13	12.13	12.13	ALU Ops	1.32	825 M Threads\Sec