# Brook Spec v0.2

Ian Buck

October 31, 2003

## 0.1 What is Brook?

Brook is an extension of standard ANSI C and is designed to incorporate the ideas of data parallel computing and arithmetic intensity into a familiar, efficient language. The general computational model, referred to as *streaming*, provides two main benefits over traditional conventional languages:

1. *Data Parallelism*: Allows the programmer to specify how to perform the same operations in parallel on different data.

2. *Arithmetic Intensity*: Encourages programmers to specify operations on data which minimize global communication and maximize localized computation.

## 0.2 Streams

Streams provide a method for communicating data between memory, kernels, and stream operators.

**Streams and Stream Elements**
The purpose of declaring streams is to express a collection of objects which can be operated on in parallel. The collection of objects is referred to as a *stream* while each object is a *stream element*. The stream variable is a *reference* to a collection of stream elements. The type of the stream variable determines the stream element type but not necessarily the number of elements contained in the stream.

Example of stream declarations,

```
float a<>;                // 1
float b<>[3][2];          // 2
```

Line 1 declares a stream of floats. Line 2: Declares a stream of 3x2 arrays of floats. The <> specifies that these variables are stream variables.

**Stream Shape**
Streams can also be declared with a shape information to be used in conjunction with the stream operators.

Examples:

```
float b <300, 200, 100>;  // 1
mytype c <,400>;          // 2
float d <>;               // 3
```

Line 1 declares a stream with shape 300 by 200 by 100, indicating the stream has 6,000,000 elements. The Line 2 specifies a N by 400 stream. The upper most dimension does not need to be specified and will be computed by the runtime environment based on the number of elements in the stream. However, the number of elements in the stream must contain exactly as many elements such that it fills the array permitting an integer number the upper most dimension (400, 800, 1200 elements as in stream b above). If a kernel function or stream operator violates this rule, an exception will be generated. Line 3 specifies an one-dimensional.

Placement of the stream <> specifier must appear immediately after the variable name. Further, the <> specifier can only appear once within the stream declaration. Typedefs are permitted however the expanded type of the declared variable must follow the same rules.

For example:

```
float a<>;          // OK. Stream of floats
float b<>[3][2];    // OK. Stream of float array[3][2]
float c[2]<>;       // Illegal.  Array of streams
float d<><>;        // Illegal.  Stream of streams
typedef float mytype[3][2];
mytype e<>;         // OK. Stream of float array[3][2]
typedef float mystreamtype <>[3,2];
mystreamtype f;     // OK. Stream of float array[3][2];
mystreamtype g<>;   // Illegal.  Stream of streams
mystreamtype h[2];  // Illegal.  Array of streams
```

Arrays of streams nor stream of streams is currently not supported in Brook but may be supported in the future. A function cannot return a stream type.

**Restrictions**

Streams are *not* associated with a particular region of memory. Rather, they behave as connectors between arrays, kernels, and stream operators. Therefore, they have the following restrictions:

- Individual stream elements are not accessible outside of kernel functions.
- Use of the "=" assignment operator on stream variables outside of kernel functions is illegal.
- Static initializers are not permitted on stream variables.

**Passing streams to non-kernel functions**

When stream variables are passed to non-kernel functions, they are passed by reference. If the function changes the contents of a stream variable, these changes are reflected in the caller.

## 0.3 Loading and Storing streams

Streams are loaded and stored from memory using the stream operators `streamRead`and `streamWrite`operators.

```
streamRead  (stream variable,
             source array,
             starting array offset of first dimension,
             size from first dimension,
             starting offset of second dimension,
             size from second dimension,
             ...)
Examples:

float s<>;
float a[7];
float b[3][5];

streamRead (s, a, 0, 7);          // 1
streamRead (s, b, 0, 3, 0, 5);    // 2
streamRead (s, a, 3, 2);          // 3
streamRead (s, b, 1, 2, 1, 2);    // 4
```

Line 1 assigns stream s to include the entire contents of the array a. Line 2 assigns s to the entire contents of array b. Line 3 assigns stream s to a subregion of the array a including only elements a[3], a[4]. Line 4 assigns s to a subregion of array b including only elements (b[1][1], b[1][2], b[2][1], b[2][2]). The order of the elements in the stream matches the order of the elements in memory as defined by C array memory layout rules, i.e. row-major layout (i.e. b[0][0], b[0][1], b[0][2], ...).

The offset and size arguments may be any computed value.

The shape information of the stream is not used for streamRead. All of the parameters in the streamRead call relates to the shape of the source array. The stream output is in the shape specifed by the declaration of the stream variable.

Restrictions:

- The number of offset/size parameters must match the number of dimensions from the source array.
- The offset + size must not exceed the size of the dimension.
- The base type of the array must match the type of the stream.

Violation of these restrictions results in a compile time error and or run-time exception

`streamWrite`follows a similar syntax.

```
streamWrite  (stream variable,
              destination array,
              starting offset of first dimension,
              size from first dimension,
              starting offset of second dimension,
              size from second dimension,
              ...)
Examples:

float s<>;
float a[7];
float b[3][5];

streamWrite (s, a, 0, 7);          // 1
streamWrite (s, b, 0, 3, 0, 5);    // 2
streamWrite (s, a, 3, 2);          // 3
streamWrite (s, b, 1, 2, 1, 2);    // 4
```

Line 1 stores the contents of stream s to the entire contents of the array a. Line 2 stores s to the entire contents of array b. Line 3,4 stores stream s to a subregion of the array similar stream Read. The order of the elements written matches the order of the array elements in memory as defined by C array memory layout rules. The restrictions for `streamWrite`are identical to `streamRead`with the added restriction that an exception is raised if the number of elements in the stream is less than the number of elements specified in the store operation.

In addition to `streamRead`and `streamWrite`, Brook also provides `streamReadAll`and `streamWriteAll`functions which reads or writes the entire array. For example:

```
streamWriteAll (s, a);
streamReadAll (s, a)
```

These calls are identical to their corresponding `streamRead`and `streamWrite`functions with the offset parameters set to zero and the length equal to the size of the array dimension.

## 0.4 Brook code restrictions

Brook C Subset:

Brook code must be contained in a separate compilation unit (file or set of files) which obeys certain restrictions. Brook functions may be called by arbitrary external code (ie .c files), and may call arbitrary external functions, subject to restriction 6 below.

1. Disallowed keywords:

   Certain keywords are disallowed, with various reasons:

```
asm             // machine dependent
goto            // complicates control flow analysis and
                           transformation for little gain
volatile        // only required for multi-threading
static         // see below
```

2. Disallowed static storage-class variables:

   Static storage-class variables are disallowed so as to prevent visible side-effects from external code. This precludes global and static variables.

3. Heavily restricted pointers:

   Pointers are heavily restricted so as to limit aliasing. In practice, these restrictions allow pointers to be used only for passing variables ”by reference” to functions.

   – Variables with pointer types may only be declared as function arguments.

   – The only allowed expression which can produce a pointer type is: &variable, and the result of this expression must be passed directly to a function without application of further operators.

   – The only operators allowed on pointers are: * and `->`

   – Pointers to pointers and pointers to functions are not allowed.

4. Recursion is disallowed:

   Recursive function calls are disallowed so as to allow inlining and interprocedural analysis.

5. Precise exceptions are not supported:

   The exception model for Brook code is not yet defined, but it is not precise.

6. Assumed properties of external code:

   Certain restrictions are imposed on external code so as to ensure that aliasing and array sizes are known.

   – External code that calls a Brook function is assumed to pass unaliased pointers and arrays to the Brook function, and not to access any storage reachable under the Brook restrictions using those pointers and arrays until the Brook function returns.

   – A Brook function called from external code must specify the size of all dimensions of any array arguments. For example,

```
void mybrookfunction (float a[300][200][100],  // OK
                      float b[][200][100],     // Illegal
                      ...)
```

   – An external function called from within a Brook function is assumed to access storage reachable under the Brook restrictions using the arrays and pointers passed to the functions until the function returns. With the exception of this use, automatic storage used by Brook code is assumed not to be accessed by external code.

   – streams variables are not accessible outside of Brook programs.

## 0.5 Kernel functions

Kernel functions are user specified functions which are executed over the set of input streams to produce elements onto the set of output streams.

Kernels operate *implicitly* over the entire set of input streams. The kernel will execute once for each element in the input stream(s). The run-time will guarantee that all input streams are the same length. If a kernel is called with input streams of differing lengths, the kernel will immediately fault, producing no output.

Kernels functions are declared similar to standard C-functions. An example kernel is shown below:

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

The parameter 'a' is the input stream, 'b' the output stream, 'p' a constant parameter. The body of the kernel foo will be executed for each input element of 'a', producing a new stream, 'b', which contains the a set floats which are p larger than the corresponding 'a' elements. The keyword `kernel`is a *function specifier* (similar to the `inline`keyword) used to specify that this is a kernel function, not a normal C function. The following sections define the particulars of kernel functions and their differences to normal C functions.

## 0.6 Kernel arguments

The arguments to kernel functions can be of any of the following types. Note that the arguments can appear in any order in the kernel declaration, not necessarily in the order outlined below. There can be any number of different arguments of the different types.

### 0.6.1 Input Streams

Input streams are specified with the <> specifiers similar to stream declarators, i.e. `int a<>`. The shape information is optional and does not effect the computation.

The type of the input can be any valid stream type, including native types (int, float, char, ...) or user defined types (structs, typedefs, ...). Inside the body of the kernel, the type of the variable is equivalent to the specified type in the argument declaration without the <> specifiers (in our example, the parameter a is of type int inside the kernel). Input streams are passed by value so any assignments to the input stream variable performed inside the body of the kernel do not modify the source stream. In general, the input stream is *not* modified by the kernel, the length and values remain unchanged.

Only a single element from the input stream is visible from within a kernel function invocation. There is no mechanism for examining multiple stream elements from the input streams (no "peek" or "pull" functions).

### 0.6.2 Fixed Output Streams

Fixed output streams are specified with the <> specifiers and `out`keywords, i.e. `out int b<>`. The `out`keyword must appear before any other type information. Fixed output stream arguments contain the results of a kernel function. The kernel body assigns values to the out stream variable(s). When the kernel functions returns, the value of the out variable(s) is placed onto the fixed output stream. The kernel function always produces a single output element onto the stream regardless of whether the kernel performed an assignment or not (though the complier should provide warnings if not all code paths perform an assignment to the variable). Therefore the total number of elements produced on fixed output streams is always equal to the number of input stream elements.

The order of the elements in the output streams mimics the input stream element order as discussed below.

Similar to input streams, the type of the fixed output stream variable inside the kernel is equal to the type of the stream elements, specified in the argument declaration. Reads from output streams are allowed however the initial value is undefined. The programmer cannot rely on the undefined initial values of output streams, and the compiler should provide warnings if reads are performed in the kernel code.

Any values associated with the output stream prior to the calling of the kernel are not retained in the stream. The output stream length is set to empty when passed to a kernel. The output stream variable is a new stream. This is also true for variable output streams below.

### 0.6.3 Variable Output Streams

Variable output streams are similar to fixed output streams except that the kernel function can produce zero or more output elements from within the kernel function. Variable output streams are specified with the `vout`keyword, i.e. `vout[100]` stream int c+. (Same keyword and typing rules as the previous stream types apply). The value 100 is the *max push count* which is an optional parameter which indicates that this kernel function will produce no more

than 100 elements for each input element processed. Producing more than this number generates undefined results. The max push count argument must be a *constant-expression* (compiler must be able to compute this value at compile time). Producing an element on a variable output stream requires an explicit call to the `push`operator. First, the kernel function assigns a value to the variable similar to fixed output streams. These assignments however do not produce an element on to the stream, nor is the variable automatically placed onto the output stream as per fixed output streams. Rather, the function must perform a push operation which copies the contents of the variable the output stream as shown below.

```
void kernel foo (int a<>,
                 vout[100] int c<>) {
  int i;
  for (i=0; i<a; i++) {
      c = i;
      push(c);
  }
}
```

In the above example, the variable output stream c results in a variable number of output elements based on the contents of the stream a. The contents of the argument is not modified by the push and the kernel is free to read and write from variable with the same caveats as fixed output streams. The push operator can only be used inside kernels and on variable output streams.

The order of the elements in the output stream is preserved. For example if the first input element causes three pushes of values a,b,c and the second input element produces d,e, the values appear fully ordered in the output stream (a,b,c,d,e).

**Unbounded Variable Streams**

The max push count parameter is optional but it is understood that there may result in less efficient compiled code if not specified. Furthermore, initial compilers may choose not support unbounded variable streams. To specify unbounded variable streams, the max push count is simply not specified, i.e. `vout[] stream float d`.

### 0.6.4 Non-stream Arguments

Kernels can also be passed non-stream arguments similar to regular C-functions. These are passed by value and any modifications made to these variables inside the kernel body are not observable between kernel invocations. These passed by value semantics includes arrays which would normally be passed by reference in C. The dimensions of the array must be fully specified in argument string, i.e. `int a[3][4][19]`, not `int a[][4][19]` (a legal C array type). The programmer should, whenever possible, specify the `const`qualifier on non-stream arguments to improve kernel performance.

## 0.7 Kernel Execution

Kernel functions are called in the same method as normal C function however the stream arguments should be declared streams.

Executing the kernel function:

```
void kernel foo (int a<>, float b<>,
                 out int c<>, int val,
         vout[10] float d<>) {...body...}
```

performs the following pseudo code:

```
Def: s[i] := i-th element of stream s (zero based)
Def: length(s) := number of elements of stream s

if (length(a_stream) != length(b_stream))
  EXCEPTION;
initial_val = val;
```

```
n = length(a_stream);
d_pos = 0;
length(c_stream) = n;
for (i=0;i<n;i++) {
  Set a = a_stream[i];
  Set b = b_stream[i];
  c,d = undefined value;
  val = initial_val;
  Execute (body) {
    if (push(d) is called) then
      d_stream[d_pos++] = d;
  }
  c_stream[i] = c;
}
length(d_stream) = d_pos;
```

If the lengths of the input streams are of different lengths, a run-time exception is generated.

The user is free to reuse stream variables for inputs and outputs assuming no two output arguments are the same variable (i.e. `foo(a,b,val,c,c);`). If a variable is specified as both an input argument and an output argument, the collision is handled as if the variables were different (no read/write collisions occur).

## 0.8 Kernel Body Code

The body of the kernel code is regular C code with the following restrictions.

1. No global memory reads or writes
2. No nested kernel function calls
3. No pointer or address operators (,&)
4. No library function calls (printf, malloc, etc.)
5. No goto operations.
6. No function pointers

The kernel body is free to call user defined functions as long as the compiler can verify that these functions also observe these restrictions and is not recursive. (Should these helper functions also be declared kernel functions with no stream arguments?)

## 0.9 Return Values

Currently, a kernel function must return void.

## 0.10 Reductions

Brook provides support for parallel reductions on streams. A reduction is an operation from a stream to a single value with the special property that the operation can be defined by a single, two-input operator that is both associative and commutative. Given this property, the elements of the stream can be treated as an unordered set and the operator applied to combine those elements in any order until it yields a single value.

Reductions can be performed from within kernel functions via *reduction variables* or via user specified *reduction functions*. Below is a simple reduction which computes the sum of a float stream using a kernel reduction variable.

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

The reduction variable, `result`, is specified with the `reduce`keyword. A kernel function is permitted to have multiple reduction variables declared in the argument string. This example updates the value of the reduction variable with the incoming stream of floats to produce the sum of all the elements in the input stream `a`. Though the reduction can be perceived as sequential, Brook does not guarantee the order of operations on reductions. Though addition is both communitive and associative theoretically, due to the IEEE fp32 limitations, the reduction result is an *approximation* of the mathematical result.

### 0.10.1 Kernel Reduction Variables

Operations on reduction variables are limited to permit the compiler the opportunity to parallelize the computation. Reading the value of a reduction variable directly inside of a kernel function is not permitted. In addition, if the reduction variable is a complex type (structs, unions, arrays, etc.), access to members (by use of the "." or "[]" operators) is also forbidden. Reduction variables can only be updated via predefined *reduction operators* or user defined *reduction functions*.

The predefined reduction operators consist of the C assignment operators `+=`, `*=`, `|=`, `&=`, and `^=`. These operators may be used to update reduction variables only if supported by reduction variable type. For example,

```
void kernel sum (float a<>,
 struct mystruct s<>,
                 reduce float r,
 reduce struct mystruct p) {
  r +=  a;  // OK: ''float += float'' is defined.
  p +=  s;  // Illegal: += is not defined on structs.
}
```

The reduction operators provide a mechanism for performing simple reductions on native types inside kernels. In contrast, reduction functions (defined below) allow the programmer to perform reductions on complex data types and apply a user-defined operations rather than the limited predefined operators. Both reduction operators and reduction functions may be performed multiple times inside kernels. The only restrictions are that the same reduction operation or function must appear within the kernel function per reduction variable. Furthermore, at least one reduction must be performed outside of the conditional logic.

For example,

```
void kernel sum (float a<>,
 float b<>,
    float c<>,
                 reduce float r,
    reduce float s,
 reduce float t) {
  r +=  a;
  if (b > 0.0f)
    r +=  b;     // OK:  r += appears earlier
  if (c < 1.0f)
    t += c;      // Illegal: all t updates in conditionals
  r *=  c;       // Illegal: r += appeared earlier.
  s *=  b;       // OK
  s +=  b;       // Illegal: s *= appeared earlier.
}
```

### 0.10.2 Reduction functions

Reduction functions are user specified reduction operators which can be called from within a kernel function or directly from the Brook code to perform a reduction directly on a stream.

Reduction functions only support reductions which are both associative and commutative. The compiler must be able to compute the reduction in any order. For example, computing sum of a stream is both commutative and associative:

$a + b + c + d = (a + b) + (c + d) = c + a + d + b$

Examples of legal reductions are sum, product, min/max, *OR*, *AND*, and *XOR* bit operations. Examples of invalid reductions include subtraction and divide. Note that the compiler may not attempt to validate that the reduction function is legal. The programmer is asserting to the compiler that the reduction function is valid. Specifying an invalid reduction results in undefined behavior.

Reduction functions are specified similar to kernel functions with few additional restrictions. Below is a sample reduction function which computes the sum of a float stream.

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

The `reduce`keyword before the function name is a *function specifier* which indicates that the function `sum` is a reduction function, not a regular C function. The reduction function updates the value of the reduction variable. The `reduce`keyword indicates which argument of the reduction function contains the value to be updated. All reduction functions must have at least two arguments. One argument must be the reduction variable indicated with the `reduce`keyword. The other argument, specified with the `<>` specifiers, must be the value used to update the reduction variable. Furthermore, the types of the `reduce`and the stream arguments must match. Any optional remaining arguments are constants (non-stream and non-`reduce`arguments) which can be used by the reduction function. These arguments are passed-by-value similar to kernel functions; any assignments the non-`reduce`argument (including the stream argument) are not observed between reduction function invocations or the calling kernel function.

Return types, like kernels, must be `void`

If a reduction function is called directly from the Brook control code, it behaves similar to a kernel function call. The stream argument is the input stream which is reduced to produce a single result stored in the reduce variable. For example,

```
void reduce sum (float a<>, reduce float result) {
  result = result + a;
}
float total = 0.0f;
sum(myfloats, total);
```

The variable `total` after the sum function call contains the sum of all the elements in the stream. Note that it is important to initialize the `reduce`value *before* calling the reduction function since this value is the initial value of the reduction.

If a reduction function is called from within a kernel function, the stream argument may be any computed value from within the kernel body, not necessarily a stream input to the kernel. The `reduce`argument must be one of the reduction variables specified in the kernel argument string. Furthermore, per reduction variable, any additional arguments to the reduction function *must* be declared `const`variables or constant expressions inside the kernel body and *must* be the *same* arguments for every use within the kernel. For example,

```
void reduce mul_with_factor (float a<>,
                             reduce float result,
                             int factor) {
  result = result * factor * a;
}

void kernel foo (float s<>, reduce float total,
 const int a, int b) {
  mul_with_factor (s, total, b); // Illegal. b not a const,
  mul_with_factor (s, total, a);  // OK
  mul_with_factor (s+3.0f, total, a); // OK
  mul_with_factor (s, total, 3.0f); // Illegal, ``a'' previously used.
}
```

### 0.10.3 Reduction Function Body

The reduction function's body must observe all the same restrictions as kernel functions. (See the kernel function section for more details.) However, the reduction function can read and write the reduction variable freely including use of the "." operator or the array index operator which is forbidden on reduction variables inside of the kernel function. Whatever the value of the reduction variable is at the end of the reduction function is the computed reduced value. Modifications to the non-reduce variables are not preserved reflected in the calling the kernel function (passed-by-value semantics) or between reduction function invocations. This includes arrays which are normally passed by reference in C.

It should be noted that the compiler must compute partial reductions and reorder the computation in order to execute in parallel. Because of this, the argument values may not always be a value from kernel function but rather a partial result. (This is why the argument types must match.) This permits the compiler to perform the reduction in parallel. Therefore, the programmer cannot rely on the values of the reduce variable inside the kernel functions.

### 0.10.4 Order of operations

Brook does not define the order of reduction operations and the compiler is free to reorder the operations to execute in parallel. Therefore, operations which are not precisely associative and commutative, like IEEE floating point, may result in slightly different results with different compilers or architectures. Brook does guarantee that the result is repeatable, i.e. the same compiled program produces the exact same result on the same architecture (in the same morph configuration, if applicable) between application runs.

## 0.11 Stream Operators

The following function will be help define the some of the operators however it is not an actual stream operator.

```
  streamPatternRead (
  T1 dst<>, T2 scr<>, int num\_dim,
  int lobound0, int hibound0, int stride0,
  int group0_left, int  group0_right,
  BoundryInfo boundry0_left, T2 lclampvalue (optional),
  BoundryInfo boundry0_right, T2 rclampvalue (optional),

  int lobound1, int hibound1, int stride1,
  int group1_left, int group1_right,
  BoundryInfo boundry1_left, T2 lclampvalue (optional),
  BoundryInfo boundry1_right, T2 rclampvalue (optional),
...)}
      int s0, s1, ..;
      for (s0 = lobound0; s0 < hibound0; s0 += stride0) {
        for (s1 = lobound1; s1 < hibound1; s1 += stride1) {
          ..
            T1 temp[group0][group1]..;
            int g0, g1, ..;
            for (g0 = group0_left; g0 < group0_right; g0++) {
              for (g1 = group1_left; g1 < group1_right; g1++) {
                ..
                  if (s0 + g0 >= 0 && s0 + g0 < dim0 &&
                      s1 + g1 >= 0 && s1 + g1 < dim1 &&
                      ..) {
                    temp[g0][g1].. = src[s0 + g0][s1 + g1]..;
                  } else {
                    if (s0 + g0 < 0)
                      switch(boundry0_left)  {
                        NULL:
                        raise_exception();
```

```
                        CLAMP:
                        temp[g0][g1]... = lclampvalue;
                        continue;
                        HALO:
                        goto halo_bail;
                        PERIODIC:
                        while (s0+g0<0) g0 += dim0;
                      }
                    if (s0 + g0 >= dim0)
                      switch(boundry0_right)  {
                        NULL:
                        raise_exception();
                        CLAMP:
                        temp[g0][g1]... = rclampvalue;
                        continue;
                        HALO:
                        goto halo_bail;
                        PERIODIC:
                        while (s0+g0>dims) g0 -= dim0;
                      }
                     ... repeat for s1,g1,dim1 & s2,g2,dim2 & ...
                     // Perform periodic write
                     temp[g0][g1].. = src[s0 + g0][s1 + g1]..;
                    }
                  }
                }
              }
              push_onto_stream(dst,temp);
halo_bail:
          }
        }

    streamDomain(t, s, 2, 5, 100, 1, 90);
```

t is a copy of elements of s starting at the start offset for the dimension and up to, but not including, end. 2 refers the number of shape dimensions of s which was indicated by the shape specifiers. Next are the range pairs for each dimension. In this example, t will contain s[5..99][1..89]. An exception is generated if the operator attempts to read outside of s.

Equivalent streamPatternRead operation:

```
    streamPatternRead(t, s, 2,
                      5, 100, 1, 0, 1, NULL, NULL,
                      1, 90, 1, 0, 1, NULL, NULL)

    streamGroupEx(t, s, 1,
                  -1, STREAM\_GROUP\_PERIODIC,
                   3, STREAM\_GROUP\_PERIODIC);}
```

4 elements of s grouped into a single element of t. streamGroupEx takes a variable number of arguments. $t$ is the output stream. $s$ is the input stream. The number of dimensions of shape of s is specified next (in this case only 1). This must be the same as the specified shape of s or 1 if not specified. Next is the left boundary of the group followed by a boundary mode argument which defines what to do passing the boundary in any dimension. This can be STREAM_GROUP_CLAMP (use a specified value for the boundary values), STREAM_GROUP_PERIODIC (periodic boundary condition), or STREAM_GROUP_HALO (do not include any elements which span the boundary).

If STREAM_GROUP_CLAMP is selected, after each dimension, a pointer is passed indicating the value to use in clamping for that dimension.

Equivalent streamPatternRead operation:

```
  streamPatternRead(t, s, 1,
                    0, t.length, 3-(-1), -1, 3, PERIODIC, PERIODIC)
```

A simplified version of streamGroupEx is also available:

```
streamGroup(t, s, STREAM_GROUP_PERIODIC, 1, 4);
```

which is equivalent to:

```
streamGroupEx(t, s, 1, 0, STREAM_GROUP_PERIODIC, 4, _GROUP_PERIODIC);
```

```
streamStencilExt(t, s, 1,
                -1, STREAM\_BOUNDS\_PERIODIC,
                 3, STREAM\_BOUNDS\_PERIODIC);
```

Similar to group however each element of t is a value from s and all of its neighbors in s, producing overlapping regions. The arguments work the same way as streamGroupEx. The equivalent streamPatternRead call is:

```
  streamPatternRead(t, s, 1,
                    0, t.length, 1, -1, 3, PERIODIC, PERIODIC)
```

Likewise, there is a simplified streamStencil function which assumes the same boundary conditions for all dimensions, however still requires both a left and right offset of for the stencil shape, i.e.:

```
  streamStencil(t, s, STREAM\_BOUNDS\_PERIODIC,
                1, -1, 3);
```

```
streamFlatten(t, s);
```

streamFlatten acts like a cast, converting a stream consisting of compound array types (ie. float[50][30]) into non-array types (i.e. float). The elements of stream s are unpacked in a row major order *across stream elements of s based on the shape of s*.

```
streamStride(t, s, 2, 3, 16, 2, 5)
```

t is a stream derived from taking 3 elements of s then skipping 16 elements the taking 3 elements..., in the left most dimension *and* taking two in the next dimension, skipping 5, etc.

The equivalent streamPatterRead call is:

```
streamPatternRead(t, s, 2,
                  0, s.length[0], 16, 0, 3, NULL, NULL,
                  0, s.length[1], 5,  0, 2, NULL, NULL);
```

```
streamReplicate(t, s, 1, 3);
```

t contains the elements of s replicated 3 times. i.e. s=1,2,3,4 t=1,1,1,2,2,2,3,3,3,4,4,4. Multidimensional replicates are also supported.

```
streamRepeat(t, s, 1, 3);
```

t contains the elements of stream s repeated 3 times. i.e. s=1,2,3,4 t=1,2,3,4,1,2,3,4,1,2,3,4. Multidimensional repeats are also supported.

```
streamCat(t, a, b);
```

streamCat: Sets t to be the concatenation of the two streams a and b.

```
streamMerge(t, a, b, 2, 5, 9);
```

streamMerge sets the output stream t to be the input stream a but with the elements starting at the offset (5,9) replaced by the contents of stream b up to the length of stream b.

```
streamGatherOp(t, index_stream, array, STREAM_GATHER_FETCH);
```

streamGatherOp performs a indirect read on the array using the index_stream to produce a stream of fetch values (t). The type of the index_stream must be integer and the array type must match the stream element type. If the array is multidimensional, the index_stream provides a linear offset into the array (based on C row-major array layout).

The `STREAM_GATHER_FETCH` parameter indicates that the Gather operation should simply read the value from the array and place it into the stream. Other functionality include `STREAM_GATHER_INTEGER_INC` perform an atomic increment on the stored array value every time that value is read from the array. Note, that this option is only valid for arrays consisting of a single integers. `STREAM_GATHER_FLOAT_INC` performs an IEEE fp32 version as well. An exception is generated if the gather accesses outside of the bounds of the array.

```
streamScatterOp(s, index_stream, array, STREAM_SCATTER_ASSIGN);
```

streamScatterOp performs an indirect write operation taking as input the stream s containing the data to scatter, the index_stream containing the offsets within the array to write the data. `STREAM_SCATTER_ASSIGN` performs a straightforward write of the stream data to the array. (What about collisions?) In addition to the ASSIGN operation, we can also perform `STREAM_SCATTER_INTEGER_ADD` and `STREAM_SCATTER_FLOAT_ADD` and for summing the value into the array. `STREAM_SCATTER_INTEGER_MUL` and `STREAM_SCATTER_FLOAT_MUL` and for multiplying the value into the array.

## 0.12 Q and A

Q: Why are unbounded variable output streams specified as `vout[]` instead of just `vout`?

A: The empty array brackets mimic the unspecified array bounds of C, meaning that the compiler should fill in the blank here. Also we want to emphasize to the programmer that they are not specifying information that would be useful to the compiler.

Q: Should push produced fully ordered output or unordered output?

A: Fully ordered. Certainly this is easier for the programmer to understand and there are ways to divide up the computation on the hardware to produced ordered output. Peter Mattson describes some methods for maintaining order:

1. The input stream is divided evenly among processors such that each processor gets a contiguous chunk. Each processor churns away on the chunk and produces another chunk of variable size. Chunks are combined in-order to produce the final stream. Both the division and combination block moves can be performed in parallel, computing the location of all output chunks is a single treesum.

2. Inputs are round-robin divided among processors and end of outputs for an input is tagged with a special bit. Outputs are then round-robin combined based on the bit. (All outputs from processor i up to the bit, then all outputs from processor i+1 up to the bit, ...)

3. Inputs are tagged, tags are propagated to the outputs, which are also tagged for each input, and the outputs are sorted by input tag then output for each input tag. For a merge sort, this is identical to 2.

Q: Why don't we allow reduction variables inside kernel function so that the programmer can simply update reduction variables inside of executing a kernel?

A: The main problem with this approach is that if we execute the kernel function in parallel, the programmer has not specified how to combine the partial reduction values into a single result. It may not be possible for even the smartest compiler to figure out how to do this merge from the kernel code.

Q: Why did we re-introduce reductions inside of kernel functions?

A: Its clear that performing reductions inside of kernels was very convenient to programmers. Furthermore, if we require that only the same reduction function/operator be used on a reduction variable it is fairly straight forward to convert between the two.

Q: What advantages are there to not fully specifing the stream element type vs the shape argument?

A: Flatten is clearly easier but does reshaping really make sence?

```
int a[300][200];
stream int b[100][100][3][2];
```

```
Group(b, a, 3, 2);

int c[][][3][2];
Group(c, a, 3, 2);

kernel foo (stream int a[20][23], stream out int b[3]
```

## 0.13 Acknowledgments

This specification was developed with the assistance of Mattan Erez, Stanford University, the Stanford Streaming Supercomputer project, Peter Mattson, Eric Schweitz, Kenneth Mackenzie, Reservoir Labs