

Brook QuickSpec

I. Buck

April 24, 2002

0.1 What is a Stream?

A stream can be thought of as a variable length list of references to data elements. When an element is placed into a stream, the data is copied to a heap and a reference to the data is stored in the stream. When the stream is read back, the stream is dereferenced to obtain the data. Most of the stream operators, like `streamStride` or `streamDomain`, manipulate references rather than the location of the actual data. By assigning a stream variable to another results in both streams pointing to the same data. Note that its also is possible therefore to create a stream containing multiple pointers to the same data (as in the `streamProduct` operator). The stream data is garbage collected with the stream variables.

0.2 Stream Declaration

Outside of kernel functions, stream variables are abstract types, making it illegal to examine or assign the contents of a stream variable. Inside kernel function however, the stream variables can be examined. Kernel functions are the only way to gain access to the values within a stream.

Currently to declare a stream type, use the “typedef stream” operator as shown below. Pointer types are not allowed in streams.

Declaration examples:

```
typedef stream float floats;
```

A stream of floats

```
typedef stream float floatarray[10];
```

An stream of array[10] floats.

```
typedef stream float *floats;
```

Illegal. Stream types cannot contain pointers.

By replacing `float` by any typedef'ed name, a developer can stream most user defined types.

0.3 Stream Manipulation Functions/Operators

(Similar to IMAGINE StreamC)

```
s = t;
```

s is a reference to stream t.

```

streamCat(s, t, u);
    s is a reference to t concatenated with u. Types of s, t, u must match.
streamRotate(s, t, 1);
    s is a reference to t rotated right by 1 element.
streamRotate(s, t, -8);
    s is a reference to t rotated left by 8 elements.
streamProduct(s, t, u);
    s is a stream of type [2] which is a reference of all combinations of each element
    of t with each element of u.
streamSelfproduct(s, t);
    s is a stream of type [2] consisting of all combinations of each element of t with all
    other elements of t. No repeats: set of  $t[i], t[j] (i \neq j)$ 
streamCopy(s, t);
    s is a copy of the data in t.
streamGather(s, addrs);
    s is a stream of elements from dereferencing a stream of memory addresses.
streamIndex(s, t, indicies);
    Returns a stream of references to the elements of t in the order given by the stream
    of indicies.
streamScatter(s, addrs);
    Stores elements of stream s to the memory addresses contained in stream addrs.
streamDump(s, addr);
    Stores the stream into main memory starting at addr.
streamLoad(s, addr, len);
    Loads len elements from main memory into a stream.
streamStride(s, t, 3, 16);
    s is a reference to a stream derived from taking 3 elements of t then skipping 16
    elements the taking 3 elements ...
streamReverse(s, t);
    s is a reference which is the reverse order of t.
streamBitReverse(s, t, 2);
    s is a reference to a stream consisting of elements of t in with bit width 2 reversed
    order.
streamZero(s);
    Zeros all the data in a stream.
streamSetLength(s, 100);
    Set the length of a stream, truncating or growing if necessary. New elements have
    undefined values.
unsigned int len = streamGetLength(s);
    Returns the number of element in s.

```

0.4 Multidimensional Streaming

Though streams are conceptually one dimensional lists, they can be manipulated as if they were higher dimensional arrays. For example a stream could represent a 2D matrix by storing the values in row major order. Brook contains native support for manipulating streams in multidimensional ways. By default all streams are 1-D.

```
streamShape(s, 2, 100, 200);
```

Applies a shape to the stream for determining neighbor information. In this example, *s* is a 2 dimensional stream, 100 in the x direction, 200 in the y. These dimensions do not change the layout of the data in any way. Rather is it stored as part of the stream state (along with length, type, etc.) and referenced by the following operators.

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

s is a reference to elements of *t* starting at the start offset for the dimension and up to, but not including, end. 2 refers the number of dimensions of *s* which was specified by the shape command. Next are the range pairs for each dimension. In this example, *t* will contain *s*[5..99][1..89].

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

4 elements of *s* grouped into a single element of *t*. `streamGroup` takes a variable number of arguments. *t* is the output. *s* is the input stream. The next argument is the number of dimensions of *s* (redundant). For each dimension, there is a boundary mode argument which defines what to do on the boundary of the dimension. This can be `STREAM_BOUNDS_CLAMP` (use a specified value for the boundary values), `STREAM_BOUNDS_PERIODIC` (periodic boundary condition), or `STREAM_BOUNDS_HALO` (do not include any elements which span the boundary). The boundary mode argument is followed by the number of elements to group together in that dimension.

```
streamStencil(t, s, 1, STREAM_BOUNDS_HALO, -1, 2);
```

Similar to group however each element of *t* contains a value from *s* and all of its neighbors in *s*. The arguments work the same way except each dimension has a left and right component determining how many elements either side of element of *s* should be included in each element of *t*.

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

The array elements of *t* are ungrouped into a flat stream of *s*.

```
streamClampValue(s, void *);
```

Sets the value for the clamp. The data pointed to by the second argument is copied and stored as part of the stream state.

0.5 Global Memory functions

These functions allow the user to allocate memory which can be accessed by the kernel functions by passing a pointer. If a kernel is executed across multiple parallel nodes, the data is replicated on each node. Allocated data is read-only inside of kernel functions.

```
void * streamGlobalAlloc (unsigned int size)
```

Allocates size bytes for passing allocated memory to kernel functions.

```
void streamGlobalFree (void *)
```

Frees memory allocated by `streamKernelAlloc`.

0.6 Kernel functions

Kernel functions are special type of stream functions which have certain restrictions to ensure parallel execution. Kernel functions are declared as normal C functions but with the “kernel” keyword before the function name.

Restrictions

1. No global memory reads or writes. (Except for GlobalAlloc memory)
2. Arguments can be read only, write only, or reduce.
3. No nested kernel function calls.

Example:

```
void kernel printfloats (stream float *s) {
    printf ("%f\n", *s);
}
```

In the above example, the printfloats example is executed on every element of the input stream. In general, each execution of the kernel function removes one element from each input stream. Execution completes if there are no more remaining input elements or “outfixed” outputs (see below) available.

0.7 Kernel Arguments

Arguments considered read only unless we use one of the following keywords:

1. out: Specifies argument write only. System will automatically shrink or grow the stream as needed.
2. mout: Specifies multiple writes. Outputs must be explicitly issued using the **push** command which produces another element on this element stream.
3. outfixed: Same as out however the system does not modify the length of the stream.
4. reduce: Specifies reducible argument. Only associative operations will work correctly.

0.8 Return Values

Currently kernel function cannot return streams.

0.9 Reductions

These are user defined reduction functions for more control over the reduction tree. A reduction function must takes at least two arguments which are the two elements to reduce. Additional arguments cannot be input streams.

```
void
func (out stream_type ret, stream_type *a,
     stream_type *b, ...)
{
    ... compute ret from a, b.
}
```

```

stream_type *s;
type p;
REDUCE (func, &p, s);

```

The syntax of these reduction functions hasn't been nailed down. Its not clear yet what is the cleanest way to represent this operation.

Reduction arguments can be used to implement a simple reduction. For example:

```

void kernel foo (floats s, reduce float *max) {
*max = *max > s ? *max | s;
}

float max;
stream float *s;
foo(s, &max);

```

The max pointer (which is not a stream) is assigned the largest value of the stream s.

The reduce keyword can be used in conjunction with streaming arguments as well.

```

void kernel vectoradd (reduce floats a,
                      floats b) {
    a += b;
}

```

Since a stream can contain multiple references to the same data element. The reduce keyword allows for safely modifying these values. Below is an example of a simple n-body simulation.

```

typedef stream float vec3f[3];
typedef stream float vec3f_pair[3][2];

void kernel forcecalc (vec3f_pair p,
                      reduce vec3f_pair f[2]) {
    vec3f force;
    ... Caculate force from p ...
    vector_add(f[0], force);
    vector_subtract(f[1], force);
}

void main () {
    vec3f p;
    vec3f f;
    vec3f_pair p_pair;
    vec3f_pair f_pair;
}

```

```

... Load stuff into p ...
... Initialize f ...

/* Do force calculation */
p_pair = streamSelfProduct(p);
f_pair = streamSelfProduct(f);
forcecalc (p_pair, f_pair);
}

```

0.10 Special Streams

FileStream: Just like a FILE type in C. Usage:

```

void
kernel loaddata (stream float *a, FileStream fp) {
    brfscanf (fp, "%f", a);
}

FileStream fp = brfopen(`myfile.txt`, `rt`);
loaddata(a, fp);

```

Use brfscanf, brfprintf for FileStream IO. Same syntax as fscanf and fprintf.

1 Change Log

4/24/2002:

Changes for Dally Streaming class. Removed pointer notion of streams. The reason for this was due to the multidimensional operators which were causing complex pointer chasing to get at the results.

Added multidimensional operators added boundary condition modes.

Changed stencil method to match group operator. This make stencil a like the group operator which helps the metacompiler and reduces the amount of state needed per stream. Issue: what about writing to stencils?

Commented out return values for kernels. Its unclear when these will be implemented.

Eliminated reducible operators since it seems that as long as the programmer understands that the operation is only assured to work if its associativity. Plus the metacompiler is going to choke on these.

Added brfopen function since we no longer support C++ in the metacompiler.