Generalized Streams

Bill Dally
September 20, 2001

1 Summary

Streams in the sense of Imagine or the StreaMIT language are one dimensional constructs. This is adequate to represent one-dimensional data sets as well as unordered data sets, but leads to clumsyness when handling data sets with higher dimensionality or irregular (graph-structured) data sets.

In this memo we look at how streams can be extended to handle these more general cases. We look at how this might be expressed in a stream language and also at how access to these generalized streams might be efficiently implemented in a streaming supercomputer.

2 Syntax

We start by looking at streams in Imagine StreamC/KernelC and then will address how to generalize them. In StreamC/KernelC we declare a stream by specifying its type and length in a greatly simplified form:

```
stream <type> aStream(size) ;
```

In a kernel we access a stream with in and out operations:

```
kernel foo(istream<type> a, istream<type> b, ostream<type> c)
{
  a >> x ; b >> y ;
  c << x+y ;
}
```

Note that this doesn't even allow us to address forward or backward in the linear dimension of the stream.

To extend the semantics of streams, we propose adding a shape and a template to a stream declaration.

```
stream <type> aStream<shape> <template> ;
```

The shape can specify an array with an arbitrary number of dimensions or can specify an irregular graph with an arbitrary number of neighbors. The template specifies which stream elements are accessed with this element in a

kernel. The first example below shows a 3-D array stream in which the six immediate neighbors are accessible. The second example shows a graph stream where four neighbors are accessible. The four names in the template must be pointer fields of type `bType`.

```
stream aType arrayStream[1024][512][200]
  {[-1][0][0],[1][0][0],[0][-1][0],[0][1][0],[0][0][-1],[0][0][1]} ;
stream bType graphStream {left, right, up, down} ;
```

Once streams are declared in this manner, the neighbors declared in the template can be accessed by kernels.

```
kernel jacobi(istream double as, ostream double bs)
{
  double temp ;
  /* compute using this element and its template */
  temp = const * (as[0][0][0] +
                  as[-1][0][0] + as[1][0][0] +
                  as[0][-1][0] + as[0][1][0] +
                  as[0][0][-1] + as[0][0][1]) ;
  as >> ; /* pop this element off the stream */
  bs << temp ; /* put result on output stream */
}
```

The array references within the kernel are relative to the current stream element.

We need to decide how to handle boundary conditions. The easiest approach is to pad the arrays so that the templates never walk off the edge of the array.

Another issue is how to type check the templates of arguments. If all stream indices are constants - as in the above example, it is easy to statically check that the kernel does not reference outside the template of the stream. With variable and even array-valued indices, however, this will not be easy to check and an explicit template declaration in the kernel argument list is needed.

3 Implementation

There are two major implementation challenges. The first is to sequence the streams from memory to maximize the re-use of stream elements. The second is to manage the communication across clusters to access neighbor stream elements.

A brute-force approach to the sequencing problem is to treat every use as a separate stream. In the example above, the kernel `jacobi` would accept seven input streams and generate one output stream. While this works, its clearly suboptimal.

A slightly better approach is to convert the template to a set of one-dimensional streams and then use inter-cluster communication to access these streams. In the `jacobi` example, this cuts the number of streams to five - one that includes `[-1][0][0], [0][0][0],` and `[1][0][0]` and one stream each for each of the other template elements. Here we get re-use along the first index, but no re-use on the other two.

We can improve upon this by blocking the computation and putting a row buffer (and array buffer) into the kernel. For example, if we process the 3-D array in 32×32×32 blocks (32K elements per block), we can put a 32-element row buffer and a 1K-element 2-D array buffer in the kernel to delay elements in the higher two dimensions. With this approach we need only read one stream into the kernel except on block boundaries where a boundary stream will need to be accessed.

For a graph-structured stream, there is little alternative to the brute force approach. However, an appropriate ordering of references should increase the probability of re-use in the stream cache.