Brook: A Streaming Programming Language

I. Buck

8 October 2001

1 Motivation

1.1 Why design our own language?

- Modern processors can be limited by communication rather than computation. However, communication or dependency information can often be difficult to determine at compile time and therefore limit parallelism. Communication and dependency information should be explicit within Brook to alleviate this problem.
- Scientific computing can benefit greatly from parallel computing. However, porting large applications can be difficult if not impossible for unconventional languages. Brook should cater to the scientific computing community and be simple to port from existing C applications.
- Straightforward to properly parallelize the computation. The programmer should have a clear understanding of what portions of the calculations are going to be parallelized by the system. Both data and task parallelism should be fairly intuitive.
- Retargetable. Mappable to a variety of hardwares including existing stream machines (a.k.a Imagine), SMP systems, and future graphics hardware. It should also be clear to the programmer how their code may be mapped to hardware.
- In many ways a good language should reflect the underlying hardware model. With the advancement of streaming computational hardware, a new language is required to properly map desired computation to this new hardware.
- Furthermore this language is designed to be a target for higher application specific languages. The language is designed to be a single retargetable language which many higher order languages can be compiled down to in an effienent way.

2 Streams, Stream Functions, and Kernels Functions

2.1 What is a Stream?

Brook introduces a new datatype called a **Stream**. A stream is simply a collection of data which can be operated on in parallel. Each stream element consists of a record of values. The layout of these records within the hardware memory is hidden from the programmer to accommodate for the variety of hardware implementations.

2.2 Stream Functions

The elements of a stream can only be accessed via a stream function. A stream function is a function which is applied to the elements of a stream. A function can take multiple stream inputs and have multiple stream outputs.

2.3 Kernel Functions

A kernel function is a type of stream function which operates in parallel over the elements of a stream. The operations allowed in kernel functions are limited in order to insure parallel operation.

3 Programming Model

3.1 Design Goals

One of the basic design goals for Brook is to minimize the complex code analysis work which has often been needed for generating parallel code. In all the constructs presented below, the compiler should not be required to analyze the code content of stream or kernel functions in order to distribute the computation. Rather, this information is presented in the function argument declaration. This makes the compilers job much easier and hopefully provide a more chances for parallization.

3.2 Native Types

The syntax of Brook is designed to be similar to C with a few additions. A native vector library is provided which offers mathematical types and operations common to scientific computing. These types include: *Vectors*: vec2f (2 component, IEEE standard floating point), vec3f (3 component), vec4f, vec2d (double precision), vec3d, vec4d, vec2i (32 bit integer), vec3i, vec4i; *Matrices*: mat2f (2x2 floating point matrix), mat3f, mat4f, mat2d, mat3d, mat4d, mat2i, mat3i, mat4i. These are limited in size with the largest being mat4f and mat4i which is a 4x4 matrix of 32-bit values.

```
vec4f a = {...};
vec4f b;
mat4f M = {...};

b = M * a;
```

Figure 1: Transforming vector a by matrix M and storing the result in b.

```
\\ Stream definitions
stream Vertex {
  vec4f pos;
  vec4f color;
  vec3f normal;
  vec2f texture;
}

\\ Stream declarations
Vertex vtx;
Vertex triangles [][3];
vec3fs framebuffer[1024][1024];
```

Figure 2: Defining and declaring a stream. Defining streams is similar to defining a C struct.

Standard mathmatical operations such as computing the dot product or performing a matrix transform are expressed in a C-like style as shown in figure 1. The compiler converts these operations to the most efficient hardware implementation. Since many modern architectures support vector-like operations, these native types can be compiled directly to these functional units.

3.3 Stream Declaration

A stream consists of a collection of elements which can be declared similar to C structs. Each element can contain any native C type (float, int, double) or vector type (vec3f, mat4d). Arrays are allowed although no pointers can exist within a stream declaration. An example stream definition and declaration is shown in figure 2. Streams can also be declared with one of the the native stream types. Native stream types include all the native scalar types, the naming is simply the native type with an "s" appended: vec4fs, mat3is, floats, ints.

Streams can be declared as a simple set such as the *vtx* example. The length of the stream is unspecified, allowing it to grow or shrink as needed. Streams can also be declared as arrays, giving them a fixed size as in the *framebuffer* example. The third example, *triangles*, the first dimension of the stream is undefined allowing for a variable number of elements in that dimention.

```
void
PrintVertexPos (Vertex vtx) {
  printf (``pos: %f %f %f\n'', vtx.pos[0], vtx.pos[1], vtx.pos[2]);
}

void main (void) {
  Vertex v;
  \\ Do some calculations on v
  ....
  \\ Print the Vertex stream
  PrintVertexPos (v);
}
```

Figure 3: A simple stream function.

Do we allow variable number of undefined dimentions or only one?

The layout of streams in memory is not exposed to the programmer. The only way to access stream elements is through stream functions.

3.4 Stream Functions

Stream functions are functions which process elements of a stream. These functions are defined much like a C function. The inputs of a stream function are elements from a stream as well as any constant values. Figure 3 illustrates a simple function which prints the position field of a Vertex stream.

Invoking the stream function *PrintVertexPos* causes the function to be called on each element within the stream *v*. The element passed into the stream function is both readable as well as writable. General stream functions provide a convenient method for unrestricted access to elements within a stream since they operate similarly to C functions. They are capable of conditionals, looping, local variables, accessing global memory, perform function calls, etc. However, they are serial operations and are executed on a single processor or host CPU. For parallel performance, the programmer should use a special type of stream function called a **kernel function**.

3.5 Kernel Functions

General stream functions such as *PrintVertexPos* are not parallelized by the compiler nor do they use special streaming hardware within the host system. In order for stream functions to take advantage of parallel hardware the programmer should use a kernel function.

Kernel function are a subset of stream functions which allow the use parallel hardware units to operate on a stream in parallel. In order to allow the kernel function to be parallelized, restrictions are placed on the types of operations it is able to perform.

```
kernel void
vtransform (Vertex vtx, Vertex out tvtx, mat4f matrix) {
  tvtx.pos = matrix * vtx.pos
}
```

Figure 4: A simple kernel function.

Figure 3.5 shows is an example of a kernel function which applies a matrix transform to each element of vertex stream.

The **kernel** keyword signals to the compiler that this is a kernel function which should be run in parallel on the hardware. Kernel functions, in general, can perform normal C-like operations: Local variables, conditionals, and loops. However certain restrictions are enforced when defining kernels:

1. Global variables are not visible within kernels. Only the arguments of kernel are accessible.
2. Function calls are permitted to other kernel functions only. This prevents the kernels from calling system functions, memory allocation functions, etc.
3. Stream elements can only be accessed read only or write only. Read modify write is not allowed. By default, arguments are considered to be read only, unless the **out** or **outm** keyword is used. (More on these keywords below.)

These restrictions are checked by the compiler if the *kernel* keyword is specified at beginning of the function declaration. One of the key restrictions for kernel function is the read or write only arguments. This allows the compiler to easily detect communication between kernels and build a basic flow graph for the computation. The compiler is also able to schedule the execution of stream functions, both kernel and general stream functions.

3.6 1-N Kernels

In the *vtransform* example, there is a one to one mapping from input elements to output elements. One vertex is passed in from vtx and one vertex is outputted to tvtx. This is the defined behavior for the **out** keyword.

Brook also supports multiple outputs to a stream. Figure 5 is a kernel for doing x-major line rasterization where two endpoints of a line are input and a variable number of fragments are output.

In this example, the *lineraster* program can output multiple fragments which are determined by the line length. The fragment stream is passed in via the argument list with the **outm** keyword. This tells the compiler that the kernel may output a variable number of elements to stream *f*, including no elements at all. To output to the stream, the kernel uses the **push** function which outputs the current value of f to the stream.

To summarize, there are two different types of keywords for outputs.

```
stream Line {
  vec2i a;
  vec2i b;
}

stream Fragment {
  vec2i pos;
}

kernel void
lineraster (Line l, Fragment outm f) {
  int dx = abs (l.a.x - l.b.x), dy = abs (l.a.y - l.b.y);
  int p = 2 * dy - dx;
  int twoDy = 2 * dy, twoDyDx = 2 * (dy - dx);
  int xEnd;
  vec2i v;

  /* Determine which point to use as start, which as end */
  if (l.a.x > l.b.x) {
    v = l.b;
    xEnd = l.a.x;
  } else {
    v = l.a;
    xEnd = l.b.x
  }

  /* Write the first fragment */
  f.pos = v;
  push(f);

  /* Bresenham loop */
  while (v.x < xEnd) {
    v.x++;
    if (p < 0) p += twoDy;
    else { v.y++; p += twoDyDx; }

    f.pos = v
    push(f);
  }
}
```

Figure 5: A xmajor line rasterization kernel which demonstrates the multiple element output from a kernel using the push operator.

1. **out**: One output element is generated per input element. **push** operator is not required.

2. **outm**: Zero to n elements generated for an input element. Each element must be explicitly output using the **push** function.

4 Array Accesses

In the previous examples, streams have been declared as simple sets. For computations which require convolution operations the programmer may want to query the neighboring values within a stream.

To allow for this, Brook supports neighborhood addressing. Figure 6 demonstrates a 3x3 convolution kernel which performs a simple blur at given a rate. For each element, it computes a new value based on the weighted sum of itself with the value above, below, left, and right.

In this example, the argument list to the kernel function includes the relative addressing information. *Convolve* takes a stream input in the form of the array. x and y are **offset variables** which contain the current offset of *grid_in* which is being processed by the kernel function. Following the offset variables is the **range information**. In this example, the kernel declares that it access one element left and right of x (*[x:-1,1]*) and one element above and below of y (*[y:-1,1]*). The compiler uses this information to know what parts of *grid_in* must be defined before calling the function. The range can be specified for both outputs and inputs if needed. Accessing the elements of the array is done through relative offsets from the offset variables.

In some cases, a kernel may only access the *x,y* value of an array and not require any other values. This is similar to the range *[x:0,0]*. In this case, the range information does not need to be specified, as shown in the *SetValues* kernel function. The *:0,0* is not required and the zero range is assumed.

Finally, a kernel may require absolute addressing within an array. This is the case if the offset into the array is computed by a function and is not known at compile time. The range is therefore the entire array. This is specified by not indicating a "*" for the range variables. Figure 4 declares function argument that can be accessed anywhere in the x direction but only one element above or below in the y direction. When accessing *grid_anyx_in*, the x component is absolute instead of relative.

The programmer should try to specify the minimum range necessary for the function to operate. If all the offset variables are specified as absolute then the full array must be defined in order for the function to be executed. This can limit the task parallelism of the kernel.

```
stream Flow {
  float t;
}

kernel void
SetValues (Flow out grid_out[x][y]) {
  grid_out.t = x + y;
}

kernel void
Convolve (Flow grid_in[x:-1,1][y:-1,1], Flow out grid_out,
  float rate) {

  float k = 1.0f - (rate*4.0f);
  float a, b, c, d;
  float t = grid_in.t;

  /* Check the boundry condition */
  a = (x==0)      ? t : grid_in[-1][0].t;
  b = (x==XMAX) ? t : grid_in[+1][0].t;
  c = (y==0)      ? t : grid_in[0][-1].t;
  d = (y==YMAX) ? t : grid_in[0][+1].t;

  grid_out.t =
    t * k +
    a * rate + b * rate +
    c * rate + d * rate;
}

void main (void) {
  Flow grid[1024][1024];

  /* initialize grid */
  SetValues(grid);

  /* Perform the convolution */
  Convolve (grid, grid, 0.1f)
}
```

Figure 6: A relative addressing example. Note that in this example, the *grid* stream in passed in as both the input and output of the kernel. This is allowed however the restriction within the kernel still remains, each argument is read only or write only.

```
kernel void
Convolve (Flow grid_anyx_in[x:*,*][y:-1,1], Flow out grid_out, float rate) {
....
```

Figure 7: Example of absolute range array. Here the x component of grid_anyx_in is an absolute address.

```
kernel float
Divergence (Flow grid_in[x:-1,1][y:-1,1]) {

  float a, b, c, d;
  float t = grid_in.t;
  float div;

  a = (x==0)   ? t : grid_in[-1][0].t;
  b = (x==XMAX)? t : grid_in[+1][0].t;
  c = (y==0)   ? t : grid_in[0][-1].t;
  d = (y==YMAX)? t : grid_in[0][+1].t;

  return abs(t-a) + abs(t-b) +
         abs(t-c) + abs(t-d);
}

void main (void) {
Flow grid[XMAX][YMAX];
  const float rate = 0.1;
  float diverge;
  do {
    Convolve (grid, grid, rate);
    diverge >?= Divergence (grid);
  } while (diverge > 5.0f);
}
```

Figure 8: Example of a reduction operation. The kernel function returns a floating point number. The $>?=$ operator returns the maximum value returned from the Divergence function.

5 Reduction Operations

5.1 Function reductions

Stream function includes a return value which can be used to gather information about the stream computation. Often with iterative scientific methods a particular computation is repeated until a system stabilizes. Figure 5.1 shows a kernel which returns the difference of a value amongst its neighbors. The main loop repeats the convolution kernel until the difference is below a certain threshold.

In this example, the $>? =$ operator returns the maximum value from all the Divergence kernel operations. This value is used as a loop conditional to see if we should continue convolving. Other reduction operators are as follows:

$+ =$	Sum of all values
$* =$	Product
$<? =$	Minimum
$>? =$	Maximum
$\mid =$	Logical OR
$\& =$	Logical AND
$\wedge =$	Logical XOR
$- =$	Negative of sum
$/ =$	Reciprocal of product

(Note: These are the same operators available in the C* language for the connection machine.) Note that all of these operators are associative which allows for parallel operation.

5.2 Reduction variables

For some reduction operations, returning a single native type isn't sufficient. For example, a user may want to keep track of the largest and smallest value while processing a stream.

For this, we allow **reduction variables**. Reduction variables are arguements passed into the kernel functions which can writen via the reduction operators. Reading is not permitted. Below is a simple example of computing the max and the min of a stream.

```
kernel void
maxmin(floats a, reduce float max, reduce float min) {
  max >?= a;
  min <?= a;
}

floats a;
... initialize a
max = min = 0;
maxmin(a, max, min);
```

The **reduce** keyword indicates to the compiler that this argument is a reduction variable. Reductions are not only restricted to native types, in this case floats. Reductions involving stream elements as reduction variables allow for operations such as matrix multiplies. An example of such an operation is described in the next section.

Clearly we want to allow user defined associative functions. How should this be represented? Allow "reduction" variables that are read/write? Do you permit non-associative reduction in kernels or are they forced to do stream functions instead of kernel functions

5.3 Array ordering

When using streaming arrays, the indexing variables present in the function prototype indicate how different arrays should be processed through a kernel function. For example, consider a large matrix-matrix multiply operation. The C code and equivalent Brook code are presented below.

C Code:

```
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    for (k=0; k<n; k++) {
      c[i,j] += a[k,j] * b[i,k];
    }
  }
}
```

Brook Code:

```
kernel void
matrixmult(reduce floats c[i,j], floats a[k,j], floats b[i,k]) {
  c += a*b;
}

floats a[n,n], b[n,n], c[n,n];
... initialize a, b, and c.
matrixmult(c,a,b);
```

In this example, Brook uses the indexing variables, i, j, k declared in the function prototype to establish which elements to a, b, c to be processed by the *matrixmult* function. All combinations of i, j, k with values of 0 to $n-1$ are processed by matrixmult. Since the compiler understands which elements of the streams are needed for the computation, its free to reorder the computation to best match the hardware. This example also demonstrates a reduction using stream elements as a reduction variable.

In general, the user needs a method to express dependency relationships between elements of arrays. In this proposed method, using the indexing variables

provides a mathmatical like representation of the dependency information. In the more general case, Brook allows complex expressions within the array indexes.

6 Stream Management

We would like to be able to do the following:

1. Exectute a kernel on a subset of a stream.
2. Provide better boundry cases. Cilindrical arrays for molecular dynamics.
3. More control over which elements of the stream gets processed. What is the syntax for running a kernel over a domain?
4. Others?

This part of the spec is incomplete.

7 Special Streams

There are some special stream types defined in Brook .

1. **FileStream**:A file stream which can be read or written with standard stdio.h functions like fscanf, fprintf, fread. Equivalent to a FILE pointer in C. These streams cannot be passed to kernel functions.
2. Others?

This part of the spec is incomplete.