Dan Bentley

EE482C final project write-up

Conclusions:

I explored Conway's Game of Life as a Hello, World application for Stream Programming. Used this as a springboard to create the program as a test-case, write an introductory document about Stream Programming and explore issues in Stream Programming. I found that the Brook Run-Time was in need of test-cases for the system, that it's difficult to describe programming and that there are extensions to Brook which could be useful. In the rest of this report, I will discuss the prose and program I wrote but spend most of the time proposing language changes, ranging from the minor and useful to the grandiose and possibly useful.

What I did:

I wrote the Life program. I have one kludged-up version that ran on the system in the ee482 project context. I also have an untested version that relies on Brook's new copy-in, copy-out semantics (the language without references), but that I can prove works if copy-in, copy-out works. The reason for the limited scope of my programs as compared to my original intentions will be explained later. I also believe I found a bug in 1-D stenciling, but this is still under investigation by Ian.

I wrote a document that intends to introduce people to Stream Programming to the level that they understand the Game of Life, with all its stenciling and multi-dimensionality. I also came up with a way of describing the duality of Stream Programming (as expressed in the fact that StreamC/KernelC are written and compiled separately). I refer to the marshaling of streams as "communication" and the dealing with data in kernels as "computation." Using distinct terms to identify these concepts in a hierarchy (Stream Programming has children communication and computation) can be useful to having an effective lexicon for discussion. These terms may not be

the most appropriate (in that communication also refers to instruction-level scheduling), but if they do not fit I would lobby for the introduction of other terms.

I designed language proposals. One of these Ian has already agreed to, and the rest have not yet been proposed to him. I will not describe them later, but for now I mention it for the sake of completeness of describing what I did.

What I learned:

I thought my project would be useful to new programmers, in getting them up to speed on Stream Programming. It turns out that it's also useful to Brook, by testing areas of it that might otherwise go unused. Also, by making small test-cases, it is easy to identify problems that might creep in. Thus, not only is it true that if a programmer can get Life to run, he is half-way to doing anything with Stream Programming, it is also true that if a run-time system can run Life, it is half-way towards doing anything with Stream Programming. As the SSS group attempts to bring Brook up in multiple implementations, I think having such tests will be immensely useful.

Explaining concepts, such as the duality of Stream Programming, without well-defined terms is difficult. As, more and more, complex Stream Programming has to be described to a larger and larger audience, it will be necessary to devote some time and energy to determining how best to not only do Stream Programming, but teach it as well.

Finally, I gained some experience programming with Brook. Even in just my slight experience, I feel that I learned much. This led to my proposals for language extensions, and I hope taught me much about nascent and dynamic programming languages in general.

Language Proposals:

Sorted from mundane to grandiose:

Allow FileStreams to be created from FILE*'s.

This allows stdin/stdout/stderr to be used as FileStreams, and brings Brook back in line with the C philosophy on Files. It seems imperative to me to stay as close to C as possible so that programmer's conceptions are still useful, and porting applications from C remains as painless as possible.

Requesting Position in a Stream.

I propose a function resembling int GetIndexInStream(int dimension) that would, within a kernel, return the index of the element being processed in the given dimension. Thus, if there are actions that should take place at a border (for instance, printing out a newline every nth element), it is easy to determine. This type of construct should be easy to compile efficiently, but would be a pain without such a feature.

Sequential kernels.

I think there should be a way to specify maintained state through applications of a kernel. Specifically, variables declared static within the body of a kernel persists through iterations. Also, the function endKernel() would stop further iterations of this kernel (e.g., if I only want to process elements until I find some sentinel value). I realize that this was already debated, but I think that by restricting use of it to requiring explicit declaration, it will be easier to code certain idioms which would otherwise be difficult. For instance, with this proposal, I could very easily make a stream which contained the first ten integers by writing the kernel:

```
kernel void
MakeTenInts( out ints nums) {
        static int a = 0;
        if(a >= 10) endKernel();
        nums = a++;
}
```

I am not sure how I would write such a function in current Brook. This proposal is not as harmful as it might seem, because the above function could be parallelized to run on different clusters at the same time, as could many such examples. That would require a mature optimizing compiler, but it seems possible and is an elegant way to express what you want to do.

Pushing onto Neighbor's Streams

Instead of using a push architecture for operations such as add&store, it may be possible to use, on the high-level language level the idea of streams to obviate the need for such an idea. By instead offering a high-level function to push into another element's stream of queued values, the need would be obviated to act directly on its state. Instead, one could push values into its queue and then introduce another kernel to empty the queue of outstanding requests on each element. This is most definitely my roughest idea, but would come across as something like:

```
kernel void
DoSomething(cell_s c) {
        int i, max, value = f(c);
        if(someCondition) {
                for(i = 0; i < max; i++) {
                        neighbors[i].value += value
                }
        }
}
```

becomes:

```
DoSomething(cell_s c) {
```

```
        int value = f(c);
        if(someCondition) {
                for(i = 0; i < max; i++) {
                        push(neighbors[i].processQ, value);
                }
        }
}
```

followed by a call to:

```
kernel void
ProcessQueues(cell_s c) {
        ProcessValues(c.processQ, &c.value); //Note, uses a recursive kernel as described below, but could use just a normal for loop.
}
```

This could be useful because the push function could be heavily optimized, and then the passing of information and the updating of state could be separated. I think that for the problems of irregular grids, this is a convenient way to express the passing of information between neighbors and the computation based on that information.

Recursive Kernels

My final and largest proposal is to make Stream Programming recursive. This would mean being able to have streams passed into kernels, then kernels could call other kernels. This may not be the most efficient suggestion from a hardware point of view, but I think from a

language design stand-point, it is the most elegant. Also, by making a required use of a keyword to acknowledge the potential performance hit. Also, I think it may be possible to have different levels of kernels map to different levels of parallelism. For instance, each element in the highest level stream may take minutes or seconds to process, and deserves its own node in the cluster. Within that kernel, each element in the second level stream could be allocated its own cluster. On the lowest level, each record may have a stream of int's that were passed to it by neighbors that it should add up. Because this stream is likely to be small, it may just be parallelized to the level of different ALU's. My basic point is that streams and kernels are basically the best foreach construct. They make ordering not matter, and by using reduction variables, the compiler may optimally schedule application of dependencies.

This proposal would also require an operator like CM-Lisp's dot operator, which would allow streams to be passed into kernels and stored like other types, unevaluated/uniterated. The example in my presentation is still useful. In this case, imagine that at the beginning of a program I want to initialize the neighbors in an unstructured mesh where the neighbors themselves are stored as a stream. This kernel might look something like:

```
kernel void InitNeighbors(cell_s board, uniter cell_s firstNeighbors) {
        board.neighbors = firstNeighbors;
}
```

Conclusions

Brook is a nascent language and as it is developed, I sincerely believe that it will become a great first-generation truly high-level Stream Programming Language. I hope that some of these proposals will influence the language, even if only in making people think. Because one of these proposals has already been tentatively accepted, I feel like this project has made something that approached a difference. It purposely contains no performance numbers, because the

performance that it seeks to optimize is that of the programmer, the beginner and the developer of new Stream Programming systems.

Future work on this project, which may actually be taken up, involves continuing to write test cases for features of Brook that may be easy to debug, maintaining the introduction to Brook and expanding it, and continuing to work with language proposals, including eventually implementing them. For instance, I think that my introduction to Brook could, after revision and possibly addition of tools such as how to install Brook and create the Makefile, serve as a solid introduction to Stream Programming for use in future classes or the general public. I think that some of these language proposals could be useful, and after refinement implementation of them might not be out of reach of a person. I think that perhaps the biggest part of future work is attacking the design of Brook not from a perspective of what is efficient or immediately needed, but what is elegant and will make Brook a language that is comfortable to program in 5 years from now, when the state of Stream Programming is, for better or worse, much different and not nearly as tied to Imagine.

Prose Document

Stream Processing is a new way of considering and describing computational problems. In contrast to traditional programming's single level of control, Stream Processing both allows and requires a user both to marshal data between computational kernels and define the kernels themselves. Such an approach is well-suited to both sides of the human-computer interaction for certain classes of problems: The computers can compile and execute it more efficiently, and the programmers can comprehend and construct it faster. To explore Stream Processing, we will use the Brook Programming Language.

The fundamental data type of Brook is a Stream. A Stream is a collection of Records and can be operated on in parallel. From this description, the duality of Stream Processing becomes apparent: "Computation", as embodied in kernels, acts on the data of a Stream program, whereas "communication" connects the inputs and outputs of the different kernels to make a useful program. It is only with the combination of these two elements that a Stream Program can be written.

For the moment, we will forego further discussion on Stream Programming in general and furnish examples. Brook is a high-level Stream Programming Language based on the C Programming Language. Brook introduces a new data type to C: a stream, which is a variable-length ordered set of data elements that can be operated on in parallel. Before creating a stream, the type of the stream must first be defined. To create a stream type that is a stream of C's native "int" type, one would write:

```
typedef stream int ints;
```

Unlike C, however, Stream Programming's first program must necessarily be something more complicated than "Hello, World." To showcase Brook, we will first present a solitary kernel. This kernel, add2streams, takes two streams of "int's", as defined above (a and b) as input and produces one as output (c, as shown by the keyword out).

```
kernel void add2streams(ints a, ints b, out ints c) {
        c = a + b;
}
```

To explain: add2streams will be run for as long as there are elements left in either a or b. Each application of the kernel get its own integer taken from a and b (even if not used), and the value output to the stream c will be the final assignment to the local variable c. One important aspect of Brook highlighted in this example is that Brook is a functional Stream Programming language. Intentionally, there is no way to have variables that persist across applications of the kernel. Instead, Brook introduces the concepts of reduction variables. A reduction variable is passed as a variable, normally a pointer, and can be both read and written to. The order that the reads and writes occur in is undefined, so any use must be associative for the results to be consistent. Take as an example of reduction variables the following kernel add2streamsWithSum which again takes two streams and produces one, as well as storing into the memory pointed to by the parameter sum (initially zero) the sum of all elements in the two streams.

```
kernel void add2streamsWithSum(ints a, ints b, out ints c, reduce int *sum) {
        c = a + b;
        *sum = *sum + c;
}
```

These kernels, however, are only the computation half of describing a Stream Program. Instead of "Hello, World", the first Stream program will be Conway's Game of Life. This program is substantial enough to demonstrate many of Brook's features specifically and recurring elements of stream programming in general, while still being simple enough to be understood. Our application will be run with the command:

life <infile> <size> <generations> <outfile>

It will read in its beginning world and output its end world, both to files specified on the command line. Also specified on the command line will be the number of generations to run and the size of the board (in one dimension, it is assumed to be a square board). The world will run for the number of generations specified. We will present the program text, and then explain the new constructs (hopefully their names and usage will make them clear enough).

```
typedef stream int cell_s;
typedef stream int **neighborhood;

kernel void
LoadLifeBoard( out cell_s s, FileStream f) {
        char d;
        brfscanf(f, "%c ", &d);
        s = d;
}

kernel void
StoreLifeBoard(out FileStream f, cell_s s) {
```

```
        brfprintf(“f, “%c “, s);
}

kernel void
Generation(out cell_s newBoard, neighborhood board) {
        int tally;
        tally = board[0][0] + board [0][1] + board[0][2] + board[1][0] + board [1][2] + board
[2][0] + board[2][1] + board[2][2];
        if(board[1][1]) { //cell is alive
                newBoard = (tally == 2 || tally == 3);
        } else { //cell is dead
                newBoard = (tally == 3);
        }
}

main(int argc, char **argv) {
        cell_s board;
        neighborhood boardp “3,3”;
        FileStream infile = brfopen(argv[1], “rt”);
        FileStream outfile = brfopen(argv[4], “w”);
        int size, generations;

        sscanf(argv[2], “%d”, &size);
        sscanf(argv[3], “%d”, &generations);

        LoadLifeBoard(board, infile);
```

```
        while(generations--) {
                streamShape(board, 2, size, size);
                streamStencil(boardp, board, STREAM_STENCIL_CLAMP, 2, -1, 1, -1, 1);
                Generation(board, boardp);
        }

        StoreLifeBoard(outfile, board);
}
```

Most of this program should be readable by a proficient C programmer. In fact, the only aspects that should be unfamiliar are those dealing with multi-dimensional streams and stenciling. These two concepts are key to Brook's status as a high-level stream programming language.

Multi-dimensional streaming allows a program to view this stream of elements as existing in multiple dimensions by a call to streamShape. In Life, the call to streamShape declares that the stream board is 2-dimensional, with dimensions of size and size (since it is a square board). Multi-dimensionality means little by itself, but in combination with stenciling it makes trivial operations that would otherwise be difficult.

The concept of stenciling allows a programmer to pass into a kernel both the element being processed and a defined neighborhood. In the Game of Life, for example, we stencil the stream board to create the stencil boardp. We use clamping, which sets values requested from outside the stream (on the edges) to a fixed value (of 0). We then specifiy that we want the clamping to be done in 2 dimensions and pass in 2 arguments for each dimension: how far before and after this element we want to read. When we declared boardp, the quoted numbers at the end declared it to be of type neighborhood, a stencil, that is a 3-by-3 matrix. When we call

streamStencil, we said that we wanted a stencil in two dimensions that included elements from one before to one after (inclusive) of the element under consideration in each dimension.

In the kernel Generation, the input is one stencil per application of the kernel and the output one cell of the new board. The meaning of this kernel code, then, is to first add up all the neighbors (the matrix starts at 0,0 for the lower left-hand corner, so in this case 1,1 is the actual element). The rest of the generation code is to calculate the new state as per the rules of Conway's Game of Life, explained elsewhere.

At this point, the reader should have some idea of the syntax, semantics and philosophy of Brook specifically and Stream Programming in general. Stream Programming can be very useful for tasks that are largely calculating new values of elements based on the previous values of analogous elements. With new processors, these Stream Programs should also offer great performance improvements over traditional languages on traditional hardware. Hopefully armed with this knowledge of a better Life, the reader can make his or her own Stream Programs, leading to a better life.