Compiling Brook To StreamC

Abhishek Das Mattan Erez Jayanth Gummaraju

Computer Systems Laboratory

EE Department

Stanford University

CA 94305, USA

1 Introduction

Stream processing is becoming an area of active research in computer architecture. Many applications, including media and signal processing, image compression, and scientific applications can be speeded up by several orders of magnitude by processing them with stream processors[1][6]. However, compiling the applications efficiently in order to exploit the underlying architectural capabilities of stream processors is still an unsolved problem.

Brook[2] and StreamC[5] are examples of two languages that are used to write stream applications. Brook is a general purpose streaming language that is capable of targeting any stream processor. Brook is very similar to C, making it easy to code and also cast other existing applications in C-like language to Brook. StreamC, on the other hand, is a special purpose streaming language that is specific to Imagine [1]. Applications written in StreamC are optimized to run efficiently on Imagine. Hence, an automatic compilation of Brook to StreamC will help to write a stream application in a high level language, and, at the same time, run an optimized version of the application on Imagine.

The objective of this project is three-folds. First, develop a compiler framework for automatically compiling Brook to StreamC. Second, exercise the compiler framework on a scientific application (StreamMD) in order to test the correctness of the compiler. Finally, add optimizations to the compiler, in order to produce StreamC code that is optimized to run on the Imagine hardware.

The following list, gives the major milestones that have been reached:

- Developed a meta-compiler that automatically compiles Brook into StreamC code. Due to the short duration of the project, we only compiled commonly occurring constructs in Brook and also identified the main issues to compile constructs that were not handled. In particular, we handled stream and kernel declarations, reduction variables, different types of kernel calls including calls with Filestreams and reduction variables, non-kernel calls, and several syntax issues specific to StreamC.
- Tested the correctness of the meta-compiler with several toy programs and StreamMD. The StreamMD code has all the aforementioned constructs and in addition has a single operator (viz. SelfProduct) , making it an ideal candidate for testing the meta-compiler. The StreamC code produced by meta-compiling StreamMD[1], compiles and runs successfully on Imagine!
- Optimized the SelfProduct operator and handled stripmining. The self product operator takes a stream as input and generates an expanded stream containing all possible pairs of elements (that commute) of the original stream as output. By stripmining the stream to be expanded, we eliminated the need to create and store such an expanded stream. We also handled stripmining in a generalized case, by stripmining the streams within a region indicated by the brook program.

The rest of the report is organized as follows. Section 2 describes the basic compiler framework that we developed. Section 3 discusses the implementation of stream related transformations. Section 4 elaborates on how reduction variables were handled. Section 5 describes stripmining both in the generalized scenario and in the SelfProduct operator. Section 6 gives the major conclusions from our work. Finally, Section 7 discusses the future enhancements possible to our work.

2 Compiler Framework

Figure 1 shows the process of compiling a program written in Brook to StreamC in order to run on the Imagine hardware. First, Brook is compiled to the intermediate representation (IR) of gcc

[1] The kernels were hand-coded for testing purposes.

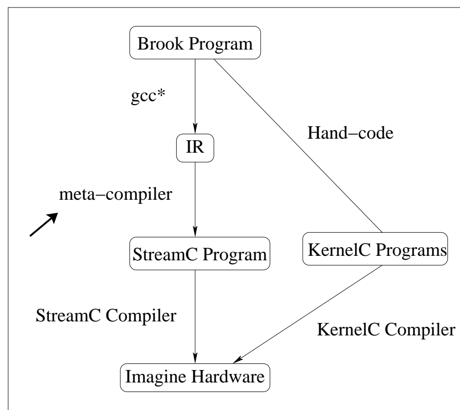

Figure 1: **From Brook to Imagine Hardware**

by using a gcc compiler with enhancements to handle Brook specific constructs. Next, a meta-compiler[3] translates the IR into an optimized StreamC code. The kernels in Brook are translated by hand into KernelC[7]. Finally, the StreamC/KernelC code is tested by compiling and running on the Imagine hardware simulator. For our project, we modified the meta-compiler and also hand-coded kernels.

The meta-compiler is a wrapper around primitives that manipulate gcc's IR, which is also known as Abstract Syntax Tree (AST). Each node in AST is a tree by itself, representing statements, function calls, prototypes, etc. The *mc_tree* primitives, which manipulate the nodes of the AST, are wrappers around *cs_tree* primitives which are gcc's primitives. The *mc_tree* primitives provide a convenient framework to translate a program written in Brook to any other language.

The current version of the meta-compiler, although suitable for simple transformations, has several limitations that we plan to address in the future. First, there is no explicit construct that keeps record of all the dependencies between kernel and scalar code. Such a construct would be very useful to perform splitting and merging.Second, the meta-compiler is not suitable to perform sophisticated optimizations like kernel scheduling, liveness analysis of variables and streams, etc. Third, the error checking primitives are not robust to handle all possibilities. Finally, the meta-compiler is still in its initial phase of testing. Hence, there are potentially several bugs that need to be fixed.

3 Stream Related Transformations

The syntax for declaring and allocating memory for streams in Brook and StreamC are as follows:

- In Brook:

  typedef "stream" datatype brooktype;

  brooktype streamname;

- In StreamC:

  im_stream<*typename*> NAMED(streamname);

  streamname = newStreamData<*typename*>(*streamlength*);

In order to transform the Brook stream declarations into StreamC, two issues need to be considered. First, the Brook *stream* type has to be derived and converted to the corresponding StreamC *stream* type. Second, the length of the stream has to be known before its usage.

3.1 Transforming Datatypes

The *datatype* specified in Brook is a C built-in data type. The StreamC type is same as the C built-in datatypes, but with a preceding *im_*. The metacompiler allows us to look at the type of a *mc_tree* node by calling *mc_type* on it. By looking at the *mc_type* and using accessors such as *cs_type_is_int*, we were able to do the type conversion. Our own accessor function written for this purpose was *char* mc_return_canonical (mc_tree)*.[2]

Handling datatypes of records, however, is much more complicated. StreamC doesn't allow the user to operate on records since any computation on records should be handled in imagine, i.e the kernels. However, Brook doesn't classify record operations as kernel specific. In order to allow this, we needed to have the same structures available to StreamC as structs. This was handled by placing the records in KernelC header files, while duplicating them as structs in StreamC header files.

[2] In order to handle reduction of expanded streams, we needed to get the type of the stream being expanded and not the type of the expanded stream.

3.2 Handling Stream Length

The second issue is to determine the length of the stream. Although the concept of streams is well understood, the language syntax for streams has not been permanently etched. Brook being a general purpose high-level stream programming language, treats streams as an infinite collection of records, which can be expanded or reduced dynamically. However, StreamC is not as flexible as Brook; the length of the streams has to be known before operating on it. Hence, the challenge was to make sure that streams are allocated the required amount of memory before being used.

The meta-compiler that we used has just one pass. Hence, the streamlength has to be determined at use-time and the memory needs to be allocated just before the use. There are two scenarios by which the stream is first *used* (or initialized).

- Reading files into streams.
- Generating streams through kernels.

3.2.1 Handling Filestreams

I/O in Brook is handled differently than in StreamC. Instead of loading/saving streams from a specially formatted file, Brook allows arbitrary I/O handling by way of fileStreams. When compiling Brook to StreamC we had two options for dealing with files:

- Exploit the StreamLoadFile/StreamSaveFile operators of StreamC. This requires:
  - Converting all the files to the specific StreamC format;
  - Modify streamSaveFile to handle mixed data-type records).
  - Extract all computations from the file-kernel and create a kernel for it in the StreamC program.
- Convert the Brook kernel into a StreamC function that reads from a file into a stream.

We decided to use the latter approach due to its simplicity.

Handling input Filestreams:

The input file-stream is accessed as a regular file using *fscanf* to read elements and then store them

into the output stream. The kernel exits when the end-of-file is reached. A while loop is inserted into the kernel to read elements from the files and append them to a Vector. Once the vector is full (end-of-file reached) a new stream is created and its length is set to the number of records loaded. Finally, a streamLoadVect is performed to save the data into the new stream.

Handling output file streams:
Here the input streams are used to determine how many times to execute the kernel body, and fprintf is used to store elements to the file. The stream is first saved to a vector, and then a for loop iterates over the vector storing the records to the file using the fprintf from the Brook code.

3.2.2 Handling Kernel Generated Streams

In order to handle streams generated by kernels, we took advantage of Brook's fixed I/O rate feature in kernels, i.e for every input element a fixed/static number of output elements are generated. This allows us to determine the streamlength of the steam being generated in terms of the lengths of the input streams to the kernels. As our first step, we didn't analyze the kernels to determine the I/O rate and rather assumed that for each input element only one output element is generated.

Another feature of brook kernels is that the kernels iterate over the input elements for the number of number of times equal to the length of the shortest input stream. For this, we needed to add a new StreamC function *STREAMC_min*. *STREAMC_min* takes an arbitrary number of streams as input and returns the length of the shortest stream.

An important issue was to make sure that the stream-lengths were a multiple of the number of clusters in Imagine, because this is an artifact of the Imagine processor (SIMD clusters). We handled this by adding a macro *STREAMC_LEN* to convert stream-lengths to the nearest multiple of NUM_CLUSTERS. A warning message is generated if the streamlength is not a multiple of NUM_CLUSTERS.

It is worthwhile to note that there is an important overhead associated with our approach. In order to get the lengths of the input streams, a StreamC function *getLength()* has to be used. This function has significant overhead of Imagine-to-host transfer. This overhead increases with the number of input streams to the kernel.

4 Reductions

Often values need to be computed to gather information about a stream computation. Such variables are called reduction variables in Brook. They are arguments passed into the kernel functions which can be identified by the keyword *reduce*. Reductions are not only restricted to native C types. Reductions involving stream elements as reduction variables allow for operations such as matrix multiplication. In this section, we discuss how to handle reduction variables of the native types. In the next section, we will discuss stream elements as reduction variables in the context of *SelfProduct*, a stream operator in brook.

Reduction variables are supported in StreamC and KernelC in the form of micro-controller(*uc*) variables. In the metacompiler, such variables can be identified by the annotations on the mc_tree corresponding to the formal parameters. If the reduce variable that is detected is of the native C type, we need to create a corresponding uc variable and assign it the same value as the reduce variable. Also, after the kernel computation is over, the value of the uc variable is assigned back to the reduce variable. The StreamC syntax for these operations is as follows:

- im_uc<im_*type*>*uc_variable = reduce_variable;*
- reduce_variable = ucread(uc_variable);

Brook supports more complex data types for reductions, such as arrays and structs. Such cases can be handled by creating uc variables for each of the fields of the complex type, such that they correspond to Imagine types. The metacompiler preserves the entire tree structure for such complex types, making it trivial to handle such cases. If the size of the arrays is not known at compile time, this method cannot be used. We are yet to figure out a good way of handling such cases.

Another issue we encountered was in handling the scope of such uc variables created. We faced the same issue in declaring variables to get stream lengths. This was handled by maintaining a hash-table data-structure, which keeps track of all variables created thus far. This helps in avoiding redeclarations. But this fails in the case when the variable is declared inside a scope and used again outside it. We added a counter to handle such cases.

5 Strip-mining

We know that an application can become memory bandwidth limited when it is unable to keep its working set in the register file. A streamC application faces the same bottleneck when it can't fit all its streams in the SRF or when there are a lot of sequential memory operations [3], thus not allowing the program to harness the fruits of producer-consumer locality. streamC allows the programmer to take advantage of the explicit bandwidth hierarchy in Imagine. The idea is to amortize the cost of memory operations over a lot of computations in the clusters, i.e execute many kernels with the streams already available in the SRF, and thus avoiding the memory transfers. The DLP existing in the applications allows us to operate on each stream element independent of the other elements. Taking advantage of the DLP, instead of getting the entire stream in the SRF, we get a *strip* of the stream in the SRF and operate a sequence of kernels on it, such that none of the intermediate streams spill out of the SRF, thus creating more producer-consumer locality. This idea is called *strip-mining*.

As discussed earlier, brook is a high-level stream language, which is architecture-independent, and hence hides the underlying bandwidth hierarchy. While transforming a brook application into a streamC application, we wanted to allow the user to take advantage of the strip-mining optimization. However, in the general case, streamlengths are known only at run time and hence the working set that will fit in the SRF. This as an interesting research question, which will include some low-level analysis like live-range and kernel-flow analyses. Since streamC has a smart profiler, *Istream*, which gives feedback about the streamlength that can fit in the SRF, we decided to take advantage of it and shelf the working set analysis for the time-being.

However, this forced us to ask the Brook programmer to explicitly outline the segment of code which will need to be strip-mined. We added the following two constructs in Brook, which look like function calls and denote the beginning and end of the segment respectively:

- STRIP_BEGIN(...);
- STRIP_END();

[3] A stream created by a kernel is stored into the memory and loaded again for the next kernel

The **...** denotes that there can be 0 or more arguments passed to *STRIP_BEGIN()*. Each of these arguments correspond to the streams that need to be strip-mined. If there is a stream operation involving any of these streams, all the streams which are part of the stream operation are strip-mined. If no argument is passed, all the streams are strip-mined. These two constructs convert into a for-loop in streamC with the segment of code in between as the loop body.

In order to get strips of a stream, we take advantage of the stream derivation operations in streamC. We have assumed throughout that all streams are fixed-length streams with fixed size records, and hence didn't need to specify the attributes for stride, record length etc. A derived stream after being strip-mined looks as follows;

- im_stream<*im_type*> NAMED(streamname_stripmine) = streamname(stripmine_start, stripmine_start + strip_size);

stripmine_start is taken care of by the for-loop, incrementing it by *strip_size* after each iteration.

An important issue is to decide the optimum strip size for the streams. Again, we relied on the streamC profiler and started with a small *strip_size* to be on the safe side. We made sure that the *strip_size* be easily changed based on the profiler feedback. Thus, if there are two strip-mined loops, strip-size for both loops are made independent. All the stream operations inside the for-loop operate on *streamname_stripmine* and not *streamname*.

An example is as follows:

The following snippet of Brook code:

```
STRIP_BEGIN();
Mult(b, a);
Max(a, &m);
STRIP_END();
```

Is compiled into:

```
STREAMC_strip_size = STREAMC_STRIP_SIZE; //Set this to what profiler suggests
```

```
bool flag=true;
for(int stripmine_start=0; flag; ) {
    /* starts stripmining */ //;

    if_VARIABLE(stripmine_start==0) {
        int KERNEL_b_0_stream_length = b.getLength();
        KERNEL_opstream_length = STREAMC_min(
         KERNEL_b_0_stream_length);
    }
    im_stream<float_stream> NAMED(b_stripmine) = b(stripmine_start, stripmine_start + S
    im_stream<A_stream> NAMED(a_stripmine) = a(stripmine_start, stripmine_start + STREA
    Mult (b_stripmine, a_stripmine, a_stripmine);

    im_uc<im_float> uc_m_0_val;
    im_uc<im_int> uc_m_0_id;

    if_VARIABLE(stripmine_start==0) {
        int KERNEL_a_1_stream_length = a.getLength();
        KERNEL_opstream_length = STREAMC_min(
         KERNEL_a_1_stream_length);
    }
    a_stripmine = a(stripmine_start, stripmine_start + STREAMC_strip_size);
    uc_m_0_val = m.val;
    uc_m_0_id  = m.id;
    Max (a_stripmine, uc_m_0_id, uc_m_0_val);
    m.val = ucRead(uc_m_0_val);
    m.id = ucRead(uc_m_0_id);

    stripmine_start += STREAMC_strip_size;
```

```
        if_VARIABLE(stripmine_start == KERNEL_opstream_length)
          flag=false;
      } /* ends stripmining loop */ //;
```

5.1 SelfProduct and Product

Strip-mining in general has been discussed. However, in Brook, there are stream operations which result in the same problem of a big working set that spills out of the SRF. One such operation is *SelfProduct* on a stream, which creates all unique pairs of stream elements. Naturally, this causes an $O(N)$ size stream to expand into an $O(N^2)$ size stream. Any kernel operating on this expanded stream would require the entire $O(N^2)$ size stream to be in the SRF. In order to handle this problem, we again use strip-mining. Instead of creating all the pairs at once and storing them in memory, we perform some of the expansion using strip-mining through the SRF, and the rest using a similar approach for expanding within a kernel into local storage (the *scratchpad*).

We dynamically partition the non-expanded stream into a set of independent strips, and then use a for-loop to pass unique pairs of entire strips to the kernel. The modified kernel (currently the KernelC code is handled manually), internally expands the first, and smaller, of the input strips (*the base-strip*) by storing it in the scratchpad and looping over this entire stream for each element read from the second input (*the pair-stream*). We also created a second version of the computation kernel which handles the interactions within the base-strips. In conclusion, we require 5 for loops and two versions of the kernel to calculate an efficient self-product:

```
for (strip_start = 0; strip_start < stream_length; strip_start+=strip_length) {
  InteractWithinBase(stream(strip_start, strip_start+strip_length));

  // now to partition the strip into bases that fit in the scratchpad
  for (base_start=0; base_start < strip_length; base_start+=base_length) {
    // first we'll interact the bases between each other
    for (pair_start=base_start; pair_start < strip_length; pair_strip+=base_length) {
```

```
      InteractBetween(stream(strip_start+base_start, strip_start+base_start+base_length),
                      stream(strip_start+pair_start, strip_start+pair_start+base_length));
    }
  }

  // now we can interact between all unique strip pairs (breaking down into bases)
  for (pair_start=strip_start; pair_start < strip_length; pair_start+=strip_length) {
    for (base_start=0; base_start < strip_length; base_start+=base_length) {
      InteractBetween(stream(strip_start+base_start, strip_start+base_start+base_length),
                      stream(pair_start, pair_start+pair_length));
    }
  }
}
```

6 Conclusion

We started off with simple goals in mind and wanted to have a basic compiler infrastructure ready so as to extend it with more features in near future. One of the assumptions was to hand-code the kernels and automate only the StreamC part. In course of our work, we found the conversion of kernels to be easy enough to automate because of the stateless nature of kernels in brook. This allowed us to leave computations as they were, and the only interactions within clusters were handled using uc variables. This will be a great win for the brook designers.

We learnt to automate SelfProduct and kernels operating on such expanded streams, optimizing each level of the bandwidth hierarchy visible in Imagine. We were also able to couple it with general strip-mining in StreamC. This optimization not only improved the performance, but also avoided wasting space for $O(N^2)$ sized streams.

The most important thing we learned from our work was that an application written in a high-level, architecture independent stream language like brook can be optimized to run on Imagine. Starting with simpler goals, we were able to achieve them and then set higher goals. From what

we learnt, we are not only able to foresee the future work required, but also the path leading to the same. We believe that in due course of time, we will be able to completely automate Brook to StreamC.

Apart from converting simple brook programs to StreamC programs and testing them, we also converted the StreamMD code in Brook to StreamC. We were able to run the StreamMD application, but since the Brook program itself has a few bugs, were not able to validate the results. Getting a compute-intensive application working in StreamC was in itself an achievement for us and our compiler.

7 Future Work

In our current implementation, we have handled only few Brook operators like self-product. Our immediate future work is to handle other operators like streamStencil, streamGroup, streamSetShape, etc. We also plan to handle the cartesian product operator using stripmining.

In the near future, we also plan to integrate several optimizations into the meta-compiler including software pipelining. Software pipelining of stream operations is a latency hiding technique, that overlaps the loads/stores of streams with the execution of kernels. Initially the Brook program will provide hints about which kernels to software pipeline, the depth of software pipelining, etc. Eventually, we plan to enhance the compiler to figure these out automatically.

We also want to look into splitting or merging Brook kernels. A trivial example of kernel splitting is the case, when function dealing with filestreams also operate on the stream elements. Since filestreams can only be handled in StreamC functions, we would like to split these kernel to create StreamC function and a KernelC kernel, which would handle the computations.

The simple manual compilation of Brook kernels into KernelC can be automated easily using the following procedure:

1. Simple syntax conversion such as += into a +, fsqrt() into sqrt(), and so on.

2. Convert IF statements to predication:

    - Calculate a predicate for each IF condition.

- Analyze the code for the scope of each condition and then protect writes to variables that affect the outputs with a series of predicates (val = select(func(pred1, pred2, ...), newval, oldval))

3. Wrap the translated Brook kernel code with one of several fixed templates for reading and writing from the input and output streams:

   - simple variable declarations, loop-stream constructs, and stream I/O in the case of most kernels, or
   - more complex code for dealing with self-product and reductions (this wrapper code is still independent from the actual work done by the kernel).
   - reductions across all clusters if a reduction variable should be outputed.

Note that the process described above only works for the type of kernels encountered in the StreamMD code, and will not work for kernels that use mout, loops, or stencils.

A very important optimization that must be performed for the StreamMD code is to convert some of the IF statements into conditional streams[4]. This process is not trivial and involves both KernelC and StreamC code.

In order to perform the compilation of kernels we cannot rely on the meta-compiler as is, and we will have to either use or develop a framework for basic-block control and data flow analyses.

References

[1] W. J. D. Brucek Khailany, S. Rixner, and et al. Imagine: Media processing with streams. In *IEEE Micro*, March/April 2001.

[2] I. Buck. Brook language specification, 2002. SSS Internal Document.

[3] B. Chelf. Brook metacompilation. SSS Internal Document, 2002.

[4] U. J. Kapasi, W. J. Dally, S. Rixner, P. R. Mattson, J. D. Owens, and B. Khailany. Efficient Conditional Operations for Data-parallel Architectures. In *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 159–170, December 2000.

[5] P. Mattson, U. Kapasi, J. Owens, and S. Rixner. Imagine programming system user's guide. Imagine internal document, 2001.

[6] J. D. Owens, W. J. Dally, S. Rixner, and et al. Polygon rendering on a stream architecture. In *SIGGRAPH*, pages 23–32, August 2000.

[7] S. Rixner, U. Kapasi, and et al. Imagine programming system user's guide. Imagine internal document, 2001.