SSS Compilation System

Mattan Erez

Version 0.1
April 15, 2002

Abstract

This document describes the SSS compilation system. The input is a program written in Brook [Buc02] and the compilation process involves three main phases: parsing of the Brook program, global analyses and optimizations resulting in a Stream Virtual Machine (SVM) [Lab02] program, and code generation to the target architecture.
Contents

1 Introduction ............................. 4
  1.1 Frontend Parsing ........................................... 4
  1.2 Global Passes ............................................. 4
  1.3 Code Generation ........................................... 4

2 Frontend .................................... 4

3 Global Phase .................................. 4
  3.1 Expand Brook Stream Operators .................................. 6
  3.2 Reduction Identification and Optimizations ............................ 7
  3.3 Persistent Storage Management ................................... 7
  3.4 Conditional Conversion ....................................... 8
  3.5 Split/Merge Kernels ......................................... 8
  3.6 Split Records ............................................. 8
  3.7 Data Partitioning and Replication ................................ 9
    3.7.1 Static Communication Patterns ............................... 9
    3.7.2 Dynamic Communication Patterns ................................ 9
  3.8 Task Partitioning ........................................... 9
  3.9 Synchronization ............................................. 10
  3.10 Stream Scheduling ......................................... 10
    3.10.1 Strip-mining ......................................... 11
    3.10.2 Software Pipelining ...................................... 11
    3.10.3 Double Buffering ....................................... 11
    3.10.4 SRF Allocation ......................................... 11
    3.10.5 Stream Operation Ordering .................................. 11

4 Code Generation .................................. 11
  4.1 Intercluster Communication ...................................... 11
  4.2 Loop Unrolling and Loop Software Pipelining .......................... 11
  4.3 Communication Scheduling ...................................... 12
    4.3.1 Register Allocation ...................................... 12

5 Open Issues ................................... 12

A Frontend Intermediate Representation 12

B Global Phase Intermediate Representation 12

April 15, 2002
List of Figures

1  Compilation Process .................................................. 5
2  Example of StreamSelfProduct from StreamMD .......................... 6

List of Tables

1  Pass Summary — Optimization importance is on a scale of 1–9 where 9 is crucial. Difficulty is on a scale of 0–9 where 0 stands for already implemented, 1 for can be implemented immediately, 2 for I think I know how to do it, and up to 9 which is probably impossible. ........... 6
1 Introduction

This document describes the SSS compilation system. The input is a program written in Brook [Buc02] and the compilation process involves three main phases: parsing of the Brook program, global analyses and optimizations resulting in a Stream Virtual Machine (SVM) [Lab02] program, and code generation to the target architecture.

The current target is the SSS as described in [DKE+02].

1.1 Frontend Parsing

The parsing of the Brook code is performed by meta-compilation [Che02]. The output of this compilation phase is the program represented in an IR along with a data-flow graph. This phase also identifies the kernel functions, stream operations, and scalar operations.

1.2 Global Passes

This phase deals with analysis and optimizations common to all stream architectures, as defined by the SVM. These include explicit insertion of synchronization points, data partitioning, task partitioning, conditional conversion, and stream scheduling (strip-mining, software pipelining, double-buffering, and SRF allocation) [KMD+01].

1.3 Code Generation

Code generation for the SSS is more involved than that of current architectures. In fact, this step compiles the SVM code into the ISA of one of the target machines, which include the SSS, a cluster computer, and workstations (both single and multiprocessor). Specifically for the SSS, this stage includes mapping the kernels onto the clusters: split and join kernels as necessary, insert inter-cluster communication, convert remaining control-flow to conditional streams/predication and loops, and VLIW scheduling.

The entire compilation process is depicted in 1.

The rest of this document will expand on each of the stages, identify specific passes, and rank their importance. For each pass, we will either propose an algorithm or try to assign it a difficulty level. The information is summarized in Table 1 below.

2 Frontend

The frontend converts Brook code into an intermediate representation, which includes the parse-tree, and the data-flow graph. The data-flow graph includes a stream-flow graph and the scalar part of the code.

The frontend is implemented with a metacompiler . . . [Che02].

3 Global Phase

The global compilation phase accepts the parse-tree and flow graph as inputs, and produces SVM code and an IR as output. The SVM code can be used for performance analysis required for optimization, and along with the IR is used as the input of the code generation phase.

This section will detail the global compilation passes and the information required.
Global Phase 5

Frontend Metacompiler

Global Compilation Phase

Expand Stream Operators

Reduction Optimization

Persistent Storage Management

Conditional Conversion

Split/Merge Kernels

Split Records

Data Partitioning

Task Partitioning

Synchronization

Stream Virtual Machine

Stream Scheduling

Stream Virtual Machine

Intercluster Communication

Loop Optimization

Communication Scheduling

Register Allocation

SSS Code
### 3.1 Expand Brook Stream Operators

Most of the Brook stream operators have a straightforward representation in hardware. The most notable exceptions are **StreamProduct** and **StreamSelfProduct** which produce an \( O(n^2) \) number of stream pair elements. These operators are used to express certain aspects of the algorithm in a simple way, but will incur a high performance penalty if run directly on the stream processor.

An example of this can be found in the molecular interaction calculation of StreamMD (Figure 2). Here, **StreamSelfProduct** is used to produce all unique molecule interaction pairs. But the compiler should understand and optimize this chain of pair-generation, value calculation, and reduction (details to be added in future versions of the document).

```plaintext
stream waterMolecule (* waterMoleculePair) [2] pospair;
stream molclField (* molclFieldPair) [2] forcepair;

pospair = pos1.selfproduct();
forcepair = force.selfproduct();
MolclInteractions(in pospair, out forcepair);
Reduce(forcepair);
```

Figure 2: Example of StreamSelfProduct from StreamMD

Kernels must be generated for any other operators that are not directly supported by the hardware.

---

1 for SSS hardware
Algorithms, Feasibility, and Importance

The algorithms for expansion are easy to develop and necessary for correct execution. For the more complex and optimized versions of the $O(n^2)$ operators algorithms must be developed, although manual translation is possible. The manual translation is far from trivial and a completely general and optimal algorithm is not feasible in my mind.

Required SVM Properties

- Description of the available stream addressing modes (stride, index, bit-reversed, ...).

### 3.2 Reduction Identification and Optimizations

Reductions must be identified and then the appropriate kernels generated. Several optimizations can then be applied:

- Kernel Combining – if the order of generated values is known (or does not matter) the reduction can be combined with the value calculation.

- Sorting vs. Memory – when the order does matter the compiler must choose whether to enforce it by sorting the values (based on the reduction element location), or to use an indexed store to memory. When using a store, it is necessary to first combine the reduction with the value calculation and then split this kernel at every memory operation.

- Atomic Add&Store – if the hardware supports atomic fetch&op instructions per stream element, they might be used to reduce the amount of synchronization and message passing.

Algorithms, Feasibility, and Importance

Required SVM Properties

- stream addressing modes.

- availability of atomic Add&Store.

### 3.3 Persistent Storage Management

Persistent storage is necessary for some kernels such as reductions and sorting. Since Brook does not allow persistent writable storage, the compiler is responsible for generating the code to manage and synchronize it. This must be done in conjunction with the read-only storage semantics of Brook and certain `const` parameters of kernels.

Algorithms, Feasibility, and Importance

In some cases this pass is necessary (reductions, sorting) whereas in others it is an optimization (kernel arguments). It is clear how to manage this storage manually, but a completely automated process may be difficult.

Required SVM Properties

- Type of persistent storage (stack, scratch-pad, heap memory)

- Primitives for storage manipulation?
3.4 Conditional Conversion

Convert conditionals within kernels to conform to the target architectures conditional support (branching, conditional streams [KDR+00], predication, ...). This is a required phase for the SSS since it does not support all forms of branching. On other targets this is an optimization phase, since conditional stream may improve performance in any case. For example, in a modern out-of-order processor branch mispredictions can account for over 20% of the execution time of low prediction accuracy programs.

Algorithms, Feasibility, and Importance

This is a necessary pass for the SSS and a useful optimization for any target. Some aspects are well understood (converting simple if-then-else to predication or selects) whereas others require research (general conversion to conditional-streams).

Required SVM Properties

- List of conditional primitives supported.
- Associated penalty for using the primitives.
- Possible performance evaluation for primitive selection.

3.5 Split/Merge Kernels

Identify kernels that require splitting due to memory operations or conditionals (these may be inserted by other compilation passes). Identify kernels that might be merged to improve performance, such as a sequence of kernels with limited input and output data, but significant temporary data. Another important reason to split kernels is to allow for register allocation, but this might only be done in the code generation phase or as a second iteration after register allocation fails.

Algorithms, Feasibility, and Importance

This seems feasible and we can build on algorithms developed for StreaMIT.

Required SVM Properties

- Estimate of available local storage.
- Accurate model of local storage?

3.6 Split Records

Enumerate the record fields used by each kernel. Based on the information in the stream-flow graph and the hardware parameters, decide whether to split records to reduce SRF usage and memory BW.

Algorithms, Feasibility, and Importance

A first-cut algorithm is very feasible.

Required SVM Properties

- Max number of kernel streams (stream buffers).
- Estimate of available local storage.
- SRF BW.
- Memory BW.
- Execution BW.

April 15, 2002
3.7 Data Partitioning and Replication

Data partitioning is necessary since the amount of memory on each node is limited. But good data partitioning is crucial for achieving high performance.

Partitioning falls into two broad categories: static communication patterns, and dynamic communication patterns.

3.7.1 Static Communication Patterns

Tools and methods such as METIS [ref] already exist, and we must make sure Brook provides enough information to interface with them. Stencils are a good start, but more information is required for unstructured meshes and more general graphs. It is important to note that all the known techniques are heuristic in nature, and mostly based on flawed assumptions of the machine model. They still provide a useful abstraction and good quality partitions.

3.7.2 Dynamic Communication Patterns

Dynamic patterns are harder both from the partitioning algorithm side, and from the Brook side. Initially we should try to interface to existing libraries or rely on user directives.

Algorithms, Feasibility, and Importance

A study is under way to determine what primitives and directives should be supported in Brook.

Required SVM Properties

- Network characteristics.
- SRF size.
- SRF BW.
- Execution BW.
- Memory BW.

3.8 Task Partitioning

Rather than relying solely on data parallelism, some applications may exhibit task parallelism that can be utilized by space-sharing the resources among tasks.

Algorithms, Feasibility, and Importance

Look at RAW compilation system.

Required SVM Properties

- Network characteristics.
- SRF size.
- SRF BW.
- Execution BW.
- Memory BW.
3.9 Synchronization

Identify all implicit (some memory operations) and explicit (reductions) synchronization points and insert appropriate synchronization primitives. An optimization phase should follow that attempts the following:

- Push down barrier-waits as much as possible to allow work to hide the barrier latency.
- Localize synchronization when possible (pair-wise barriers).
- Replace some barriers with atomic Fetch&Op type primitives.

Algorithms, Feasibility, and Importance

This is a Necessary pass. Basic and conservative synchronization is very feasible and a good start. The first two optimizations should not be difficult to achieve.

Required SVM Properties

- List of available synchronization primitives.
- List of associated costs.

3.10 Stream Scheduling

This set of optimizations and analyses has been explored in Imagine, and described in [KMD+01]. A more detailed explanation will be provided in future revisions. For now, I will just mention that completely automatic strip-mining may not be achievable from the current Brook level, and that hints or a different structure of the kernels is necessary. Also, for optimal performance, the length of streams must be known but this is not always possible. In these cases, the compiler must either profile the application or assume very long streams and deal with the consequences (double buffering or extra SRF space).

Algorithms, Feasibility, and Importance

Some form of stream scheduling is necessary, and optimization is crucial. Most of these passes have been implemented to a degree in Imagine and the algorithms are known. The exceptions that require further research are mentioned above (strip-mining and variable-length streams).

Required SVM Properties

- SRF size.
- SRF BW.
- Memory BW.
- Execution BW.
- Persistent storage information.
- Local storage estimate (register files).
3.10.1 Strip-mining
Required SVM Properties

3.10.2 Software Pipelining
Required SVM Properties

3.10.3 Double Buffering
Required SVM Properties

3.10.4 SRF Allocation
Required SVM Properties

3.10.5 Stream Operation Ordering
Required SVM Properties

4 Code Generation

4.1 Intercluster Communication
In the SSS and Imagine the stream processor is composed of several clusters, where clusters can share data either through explicit use of an inter-cluster interconnection network or by communicating through memory. Clustering is not exposed at the Brook or SVM level the code generation phase must extract the neighbor-usage information from the SVM language or the IR and insert appropriate communication instructions or memory operations. In case memory operations are inserted the kernel must be split and sent back to iterate over the global phase.

Algorithm, Feasibility, and Importance
Since Brook provides a clear syntax for accessing neighbors, it should be straightforward to translate into communication primitives. However, some communication may not be clearly expressed, and a good automated SIMD compiler does not exist.

Required SVM or IR Properties
- Neighbor communications.
- Kernel code.

4.2 Loop Unrolling and Loop Software Pipelining
These are two common and important loop optimization techniques. They increase parallelism and allow for a better schedule, by either replicating the loop body several times or overlapping several loop iterations in one scheduling block.

Algorithm, Feasibility, and Importance
These optimizations are part of Imagine’s kernel compiler and the algorithms are well known [MKOR01].

Required SVM Properties or IR Properties
- Kernel code.
4.3 Communication Scheduling

This step is the main code generation phase where assembly instructions are printed and scheduled. The scheduling is both for instructions and the communications between the local register files.

Algorithms, Feasibility, and Importance

The Imagine compiler does communication scheduling [MDR⁺00], and since the SSS architecture is similar in many ways the algorithms can be imported to the SSS compiler.

Required SVM Properties or IR Properties

- Kernel code.

4.3.1 Register Allocation

Register allocation can be done after scheduling or combined with it. In any case some kernels may fail register allocation and must be split and sent back to the global compilation phase.

Algorithms, Feasibility, and Importance

Imagine uses a simple register allocator that tends to fail on large kernels. Further research on how to improve the register allocator and combine it with the scheduler is required.

Required SVM Properties or IR Properties

- Kernel code.

5 Open Issues

This entire document is currently an open issue. As certain passes become concrete, more specific open issues will emerge.

A Frontend Intermediate Representation

Insert brief IR specification for the result of the frontend stage.

B Global Phase Intermediate Representation

Insert brief IR specification for the global stage.

References


April 15, 2002


