Stream Virtual Machine and Two-Level Compilation Model for Streaming Architectures and Languages

Peter Mattson Richard Lethin
Vassily Litvinov
Reservoir Labs, Inc.
mattson,lethin,vass@reservoir.com

François Labonté Ian Buck
Christos Kozyrakis Mark Horowitz
Stanford University
flabonte,ianbuck@stanford.edu
christos,horowitz@ee.stanford.edu

ABSTRACT

This paper summarizes and includes some text from the prior work The Stream Virtual Machine, *by François Labonté, Ian Buck, Peter Mattson, Christos Kozyrakis, and Mark Horowitz, presented at PACT 2004.*

The stream computing paradigm separates the application's computational *kernels* from communication *streams*, matching the structure and performance constraints of modern multiprocessors and data intensive applications with regular communication patterns. The multitude of stream architectures and languages create interoperability problems and cause duplication of compiler and tool development efforts. To address this, the Morphware Forum [8] is developing a two-level complication process, whereby a language-specific *high-level* compiler interfaces with an architecture-specific *low-level* compiler using an architecture model and Streaming Virtual Machine [7] code. We describe this interface and highlight the challenges of this approach.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features – *frameworks.* D.3.4 [**Programming Languages**]: Processors – *compilers.*

General Terms

Languages, Compilers, Standardization.

Keywords

Streaming Virtual Machine, SVM, HLC, LLC, streaming, streaming architectures, streaming languages, stream, kernel.

1. INTRODUCTION

This paper summarizes and includes some text from the prior work The Stream Virtual Machine, *by François Labonté, Ian Buck, Peter Mattson, Christos Kozyrakis, and Mark Horowitz, presented at PACT 2004.*

The traditional model of sequential program execution is blind to parallelism and memory hierarchies. The stream computing paradigm has emerged to account for these shortcomings, which are more and more noticeable when programming many modern processors. A *stream program* separates out computational *kernels* that perform computations on individual data elements from communication *streams* that move data between kernels and from/to the main memory.

The advantages of this decomposition are multi-fold. First, separation of communication (the gathers and scatters of data to and from global memory) from the actual computation allows communication to be scheduled ahead of the corresponding computation, thereby hiding the cost of the large memory latency that is unavoidable in modern machines. Stream programs explicitly identify which variables are only names for values in a communication stream and don't need to be written back to memory, and which variables hold persistent application state. This information reduces the global memory bandwidth, another critical resource in a modern machine. In addition, the stream formulation allows the compiler to expose data-level parallelism between stream elements in a kernel and thread-level parallelism across kernels. Finally, the streaming abstraction matches well the structure and performance constraints of modern (multi)processors. Thus, it is easier to communicate performance bottlenecks back to the programmer. For example, since communication is explicitly visible to the programmer, it is easy to point out where in the application the memory bandwidth becomes a bottleneck, such that the programmer gains insight into what is limiting the performance of the application.

The stream abstraction suits data intensive applications with regular communication patterns. Not all applications fit this model but it is a natural fit for the DSP [4], multimedia [8], and scientific [3] computing domains. Many research groups have developed architectures for stream applications. The stream architecture space spans from configurable or statically scheduled tiled processors (Raw [12], TRIPS [11]), to SIMD stream coprocessors with a large local memory for stream buffering (Imagine [10]), and to commodity graphics processors [2]. Similar diversity exists with streaming programming languages. They vary from synchronous data-flow languages with infinite linear streams (StreamIt [14], Simulink [6]), to languages with support for multi-dimensional streams and stencils (Brook [1]), and to array languages (Matlab [5], ZPL [12]), with each language offering its own set of advantages and trade-offs to the application programmer. The fragmentation in stream architectures and languages creates an interoperability problem that hinders the wide adoption of stream computing. To run any stream program on any stream architecture, one must develop a separate compiler for every language and architecture pair.

Furthermore, decomposition of an application into kernels and streams hinders portability. An application developed with a specific hardware configuration in mind may be based on a decomposition that is too coarse or too fine-grain for a different architecture or even for a different configuration of the same architecture. For example, a kernel may prove to be too large to fit in a processing unit's local memory, or it may be too small, missing an opportunity to run several kernels on the same processing unit and incurring unnecessary communication overhead.

2. TWO-LEVEL COMPILATION MODEL

To address compiler implementation challenges, the Morphware Forum [8] is developing a *two-level compilation* approach, whereby the compilation is split between a *high-level* and a *low-level* compiler. The two compilers communicate via code conforming to the *Streaming Virtual Machine* (SVM) interface [7]. The high-level compiler is specific to the particular programming language and allows the application to be written in a way that is not specific to the particular hardware. The low-level compiler, on the other hand, performs architecture-specific optimizations and produces executable code. This scheme is intended to factor out the compilation phases that are common across stream architectures, reducing the overall compiler development effort.

To address portability challenges, we support writing applications in high-level languages and performing decomposition into streams and kernels automatically in the compiler, rather than encoding it in source programs. The compiler then performs *mapping* of the streams and kernels onto the target architecture's processing units, memories, and communication channels. One advantage of this approach is a potential for a significant reduction in application development time.

One interesting question that arises in this scheme is: which of the two compilers performs such mapping. On the one hand, mapping requires the knowledge of the target architecture. On the other hand, it also requires analysis of the source program and typically some loop and data layout transformations. Furthermore, the same mapping approaches are likely to be useful for multiple target architectures.

The answer we adopt is to implement mapping in the high-level compiler. To keep it architecture-independent, however, the implementation needs to be parameterized by certain characteristics of the target hardware. These parameters are provided by an *architecture model*, which presents the hardware as an abstraction suitable for the compiler.

3. ARCHITECTURE MODEL

The architecture model represents the target hardware by abstracting it into the following components:

Processors, which represent general-purpose processors, stream processors, and direct-memory access (DMA) engines. DMA engines can only run special data transfer kernels. The other processors capable of running user-defined code are further characterized by such factors as their operating frequency, mix of functional units, number of registers and SIMD level. Each stream processor and DMA engine has one or more *master* processors that control its operation.

Memories, which come in three different flavors: FIFOs, RAMs and caches. All types are characterized by their size in bytes. RAMs are also defined by their coherence with regards to other memories in the system and the bandwidth for different types of accesses, namely sequential and random access. Stream processors take advantage of high bandwidth local RAM memories or FIFOs that link stream processors together to reduce demands on global memory bandwidth through re-use and producer-consumer locality.

Network Links, which connect one or many senders (processors, memories or network links) to one or many receivers. Each network link is characterized with a bandwidth and latency.

4. SVM INTERFACE

The SVM interface is used to describe the mapping of the input program onto the target hardware, produced by the high-level compiler and accepted by the low-level compiler. It is a procedural interface based on the C language. The SVM functions can be invoked from the program's main thread of control to assign kernels to processors and data to memories, to indicate data and control dependencies, and to start, pause, or resume the execution of kernels.

SVM functions operate on (conceptual) objects of the following data types:

Block and *stream* objects assign data to specific hardware locations in the stream processors local memories and refer to locations in the global memory for DMA transfers. A block is simply an array assigned to a location in a memory. A stream is a FIFO queue implemented as a circular buffer assigned to a location in a memory. Blocks implement random-access read and write methods; streams implement blocking peek, pop, and push methods.

Kernel objects map the application's kernels to stream processors, and can be initialized, started, paused/resumed, and waited upon. In addition, SVM offers a function to add an explicit control dependence between a pair of kernels, thus allowing them to synchronize independently of the main control thread.

DMA kernel objects describe DMA transfers and are executed by DMA engines. A DMA kernel can express move (equivalent to memcpy), strided scatter and gather (read n records, advance m records, repeat), and indexed scatter and gather (read records from within a block given a block or stream of indices). Each kind has further variations to handle block to block, stream to stream, block to stream, and stream to block transfers.

5. CHALLENGES

The two-level complication approach presents the following challenges. First, the framework for defining architecture models used by the high-level compiler needs to be general enough to cover a variety of stream architectures. It needs to be accurate enough to make performance estimates based on the models adequate. And yet it needs to be concise enough to be usable in the compiler (passing the entire chip schematic, as an extreme example, would clearly not work).

Likewise, the SVM interface should be general enough to abstract the details of target architectures, but allow efficient implementation on each architecture. Furthermore, the procedural nature of the interface requires precise definition of the behavior of each function, not just its type signature. When a behavioral detail is unspecified, the two sides of the interface (the high-level and the low-level compilers) are likely to make inconsistent assumptions about it. Finally, it is currently unclear how easily the low-level compiler will be able to derive performance-critical high-level information, such as connection topology, from a procedural specification.

6. CONCLUSIONS

We have presented a two-level approach to compiling applications for streaming architectures, which factors out the compilation phases that are common across architectures into high-level compilers and collects architecture-specific phases in low-level compilers. The high-level compiler is presented with an abstract architecture model of the target architecture to allow it to perform mapping of the application onto the target hardware.

Streaming Virtual Machine is a procedural interface between the high-level and the low-level compilers, providing data types and functions to assign kernels to processors and data to memories and to specify the operating logic of the program, including its data and control dependencies. This approach presents several challenges, such as covering a variety of architectures and mappings without sacrificing run-time efficiency and conveying high-level program information within a procedural interface.

The two-level compilation approach is currently under development by the members of the Morphware Forum, targeting C as the input language and Smart Memories, TRIPS, Raw, and Monarch as the target architectures.

7. ACKNOWLEDGMENTS

The primary authors of the full SVM Specification [7] are Peter Mattson, William Thies, Lance Hammond, and Mike Vahey, with additional contributions by University of California Information Sciences Institute, University of Texas at Austin, and IBM Austin Research Laboratory.

8. REFERENCES

[1] I. Buck. Brook specification v0.2. October 2003.

[2] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream computing on graphics hardware. In *Proceedings of SIGGRAPH*, 2004.

[3] W. J. Dally, P. Hanrahan, M. Erez, T. J. Knight, F. Labonte, J.-H. Ahn, N. Jayasena, U. J. Kapasi, A. Das, J. Gummaraju, and I. Buck. Merrimac: Supercomputing with streams. In *Proceedings 2003 SuperComputing*, nov 2003.

[4] M. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, C. Leger, A. A. Lamb, J. Wong, H. Hoffman, D. Z. Maze, and S. Amarasinghe. A stream compiler for communication-exposed architectures. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA USA, Oct. 2002.

[5] The Mathworks, Inc. *Using Matlab*, 6 edition, 2002.

[6] The Mathworks, Inc. *Simulink Reference*, 5 edition, 2003.

[7] P. Mattson, W. Thies, L. Hammond, and M. Vahey. Streaming virtual machine specification 1.0. Technical report, 2004. http://www.morphware.org.

[8] Morphware Forum. http://www.morphware.org

[9] J. D. Owens, U. J. Kapasi, P. Mattson, B. Towles, B. Serebrin, S. Rixner, and W. J. Dally. Media processing applications on the Imagine stream processor. In *Proceedings of the IEEE International Conference on Computer Design*, pages 295–302, sep 2002.

[10] S. Rixner, W. J. Dally, U. J. Kapasi, B. Khailany, A. Lopez-Lagunas, P. R. Mattson, and J. D. Owens. A bandwidth-efficient architecture for media processing. In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pages 3–13. IEEE Computer Society Press, 1998.

[11] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ilp, tlp, and dlp with the polymorphous trips architecture. In *Proceedings of the 30th annual international symposium on Computer architecture*, pages 422–433. ACM Press, 2003.

[12] L. Snyder. *Programming Guide to ZPL*. MIT Press, Cambridge, MA, USA, 1999.

[13] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. The raw microprocessor: a computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22:25–35, March 2002.

[14] W. Thies, M. Karczmarek, and S. Amarasinghe. Streamit: A language for streaming applications. In *International Conference on Compiler Construction*, Grenoble, France, Apr. 2002.