Evaluation of Stream Virtual Machine on Raw Processor

Jinwoo Suh, Richard Lethin†, Stephen P. Crago, Janice O. McMahon, and Dong-In Kang

University of Southern California
Information Sciences Institute
3811 N. Fairfax Dr. Suite 200
Arlington, VA, 22203
{jsuh,crago,jmcmahon,dkang}@isi.edu

†Reservoir Labs, Inc.
632 Broadway Suite 803
New York, New York, 10012
lethin@reservoir.com

Abstract

Stream processing exploits the properties of stream applications such as parallelism and throughput-oriented nature of the applications. One of the most recent approaches is community-supported Morphware Stable Interface (MSI) [11] used as a stable abstraction between High-Level Compilers (HLC) and Low-Level architecture-specific Compilers (LLC). We focus on one part of the MSI, the Stream Virtual Machine (SVM) [4][7][11]. We implemented a High-Level Compiler that produces SVM output renderings and SVM implementation. The SVM is implemented with the Raw Compiler as the LLC and an accompanying library. We also implemented stream applications such as matrix multiplication, FIR bank, and Ground Moving Target Indicator (GMTI) using the implemented compilers. These applications are optimized and the results are analyzed. The results show that the SVM framework is generally suitable for streaming applications on Raw processor.

The authors gratefully acknowledge the extraordinary support of the MIT Raw team for the use of their compilers, simulators, Raw chip, and their generous help.
The authors acknowledge Morphware Forum [2] for the Streaming Virtual Machine standard upon which this implementation was performed.
This effort was sponsored by Defense Advanced Research Projects Agency (DARPA) partially under agreement number F30602-01-C-0171 through the Air Force Research Laboratory, USAF and under grant number NBCH1050022 through the Dept. of Interior National Business Center (NBC). The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsement, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), Air Force Research Laboratory, or the U.S. Government.
The authors appreciate proofreading by Susan Reckitt.

1. Introduction

In this paper, we describe the implementation of a streaming model of execution using a proposed standard streaming API, and evaluate the performance results for the implementation running on the Raw research microprocessor [8][16]. In our streaming execution model, finite-duration tasks running on programmable hardware computational cores read operands from local on-chip memories, and write results back to local memories.

We expect the access patterns to local memory to be regular, e.g., sequential, which assists with getting performance from the hardware cores. Data movement between the on-chip local memories and off-chip memories is via explicit initiation of transfers. If there are multiple cores on the chip, it is also possible to set up direct links to transfer sequences of data directly from core to core.

Based on the hypothesis that this execution model is general and reflects the means by which contemporary processors are programmed to achieve high performance when executing programs with substantial amounts of data parallelism[1] as are found in signal processing, a standard way of describing computations in this model was developed.

This is the Stream Virtual Machine (SVM) framework [7][11]. The SVM framework consists of two parts, a set of C-idiomatic API for expressing computations in this execution model, along with a standard way for writing machine models (in an XML syntax) that describe hardware targets. One use of the SVM framework is as an intermediary between two compilers; a High Level Compiler (HLC) that is responsible for analyzing the input code and performing coarse grain transformations of the

[1] The basis for this hypothesis is that all contemporary processors are implemented in CMOS and that they face the same physical factors in any organization (e.g., communication costs dominating computation costs) that tries to push performance.

application, such as extracting parallelism and coarse grain load balancing, and a Low Level Compiler (LLC) that maps the tasks of the SVM to the hardware computational cores, performing standard compiler transformations such as instruction selection, register allocation, and instruction scheduling.

The output of the High Level Compiler is the application code after it has been mapped into the streaming execution model as expressed in the SVM form. Like all compilers, HLC uses an abstract machine model as a basis for the feasibility constraints and cost functions that drive the transformations and optimization. A HLC has been developed for SVM called R-Stream, and machine models and LLCs have been developed for Raw [8] [16], MONARCH [15][17], TRIPS [1][14], and Smart memories [9][10]. These research projects are being conducted within the DARPA Polymorphous Computer Architecture (PCA) research program [4] and with the coordination of the Morphware Forum [11].

In this paper, we experimented with the R-Stream 2.1 HLC and a low level compiler for Raw architecture (See Figure 1). We implemented the SVM LLC by using Raw's C compiler and implementing the rest of the SVM API as library. We, then, implemented several stream applications such as matrix multiplication, FIR filter banks, and Ground Moving Target Indicator (GMTI) [2]. These applications are optimized and analyzed.

The rest of the paper is as follows: Section 2 describes the SVM framework and the architecture of Raw processor. Section 3 describes the High Level Compiler and how we use the Raw C compiler and a library of calls to be an LLC for the SVM form. Section 4 describes the applications implemented. Section 5 shows the implemented results and analysis results are presented. Section 6 concludes the paper.

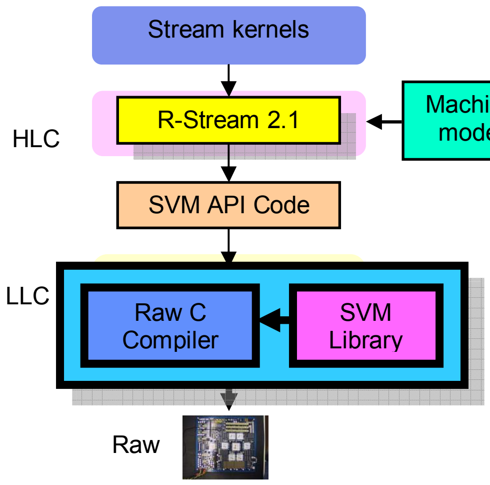

Figure 1. HLC and LLC implementation for Raw

2. Stream Virtual Machine and Raw Processor

2.1. Stream Virtual Machine [7][11]

The Stream Virtual Machine (SVM) framework [7][11] is a draft standard API within a set of standards intended to allow expressions of multiple levels of mapping of signal-knowledge processing applications to a class of programmable processors called Polymorphous Computer Architectures (PCAs). The API framework is designed to allow the integration of tools and design flows that allow the development of such applications, mappings, and systems in a way that achieves performance (FLOPS/W) that is competitive with ASIC, but which is based on programmable hardware.

The main idea of the SVM framework is developing a stream virtual machine that can be used universally for multiple languages and multiple architectures. The SVM framework exposes parallelisms and communications in applications using SVM APIs such that a tool chain can optimize application performance by utilizing the exposed information.

The HLC generates SVM API based code. The mapping of the application to the architecture is also expressed in SVM API. There are several main concepts in the SVM. One of them is kernel. The SVM API provides the kernel abstraction as a unit of computation. Kernels consist of computation expressed with a restricted subset of C, operate on a set of input and output streams, may contain local state, and generally encapsulate clean, data-parallel code. Kernels can be asynchronously monitored and controlled from a control thread running on a different processor [7].

Another concept in SVM is stream. A stream is a collection of sequential records of data. The streams can be in memory or communication channel (FIFO). The streams can be sent to a kernel that is a collection of computation. When a stream is input to a kernel, the records in the stream are consumed by the kernel sequentially and the kernel computes output that is sent to output stream sequentially. A block of data that can be accessed randomly (in contrast to sequentially) is called block. The data in the block can be accessed in any order. There are several APIs that control the kernels such as start, stop, and pause kernel operations. There are SVM APIs describing dependencies between kernels.

HLC uses machine model to generate tailored code for target architecture. The machine model describes the target machine abstractly using universal descriptions. Currently the machine model contains one primary master that is used initially to execute a serial code and to start other components. When a parallelized code is executed, secondary masters[2]are called by the primary master.

[2] The use of secondary masters is outside the SVM specification. At the August 2006 Morphware meeting, we agreed not to extend the SVM specification to allow secondary masters.

The secondary masters are responsible for preparation of input data and calls stream processors. The stream processors are responsible for execution of the computations. The machine model also describes other resources such as DMA, local memory, global memory, and networks.

Finally, the SVM code is compiled using LLC to generate binary code for a target platform. LLC is responsible for software pipelining, detailed routing of data items, management of instruction memory, and interfacing between stream processors and control processors [11].
With the SVM approach, when a new language or a new architecture is introduced, it can leverage the work done before. For example, if a new architecture is introduced, it can write an LLC for the architecture and use existing HLC with no or little modifications.

2.2. Raw Processor [8] [16]

Microprocessor manufacturing technologies have been advanced rapidly, and as a result, more and more transistors are populated in a processor. One of the resulting technical challenges from such advances is power dissipation problem. One of the solutions for the power problem is using multi-cores (multi-tile) with a little lower clock speed in a processor. By using multi tiles in a processor, the overall power dissipation problem is eased.

Another advantage of the multi-tiles in a processor is short signal travel distance. One of the main contributing factors for rapidly increasing performance of the processors was rapidly increasing clock speed and chip area. As the clock speed increases and chip area increases, it takes more and more cycles for a signal to travel from one side to the other side. Thus, by using multi-tiles, each tile occupies a fraction of the chip space. So, it is easier to make a faster processor since the signals need to travel only a short distance.

One example is the Raw chip implemented at Massachusetts Institute of Technology (MIT) [8] [16]. The current Raw implementation contains 16 tiles on a chip connected by a low latency two-dimensional mesh network. The Raw prototype board has been tested up to 300 MHz and is expected to operate at higher frequency as the processor itself was successfully tested at higher frequency. Peak performance using the 300 MHz board is 4.8 GOPS.

Each tile has a MIPS-based RISC processor with floating-point units and a total of 128 KB of SRAM, which includes switch instruction memory, tile (processor) instruction memory, and data memory. Raw uses general parallelism, which includes streaming, ILP, and data parallelism.

The Raw has four networks: two static networks and two dynamic networks. Communication on the static networks is performed by a switch processor in each tile [13]. The switch processor is located between the computation processor and the network. It provides throughput to the tile processor of one word per cycle with a latency of three cycles between nearest neighbor tiles.

One additional cycle of latency is added for each hop in the mesh through the static networks. When the dynamic network is used, data is sent to another tile in a packet. A packet contains header and data. If the data is smaller than a packet, dummy data is added to make a packet. If the data is larger than the packet, multiple packets are sent. The memory ports are located at the 16 peripheral ports of the chip. All tiles can access memory either through the dynamic network or through the static network.

In Raw, network ports are mapped to registers so that accessing registers corresponds to communication. This feature is useful to reduce the number of instructions significantly when the code is optimized as it eliminates load/store operations. This is exploited in our implementation of matrix multiplication and FIR banks as shown in Section 5.

3. High Level Compiler (HLC) and Low Level Compiler (LLC)

3.1. HLC

In our experiments, we used the R-Stream 2.1 high-level compiler [12]. R-Stream 2.1 is an experimental research compiler written with the objective of performing high-level mapping transformations from a common expression of an application to multiple PCA architectures, using the SVM as the means for expressing the mapped application.

R-Stream 2.1 accepts application written in a C dialect that simplifies the extraction of program semantics, particularly with respect to abstraction of arrays. This C dialect is nicknamed “Gumdrop.” The distinctive feature of Gumdrop is the ability to indicate that references to arrays are abstract. The separation of the extensional semantics of arrays from the intentional semantics of arrays provides the HLC greater flexibility in mapping, especially in the presence of distributed local memories in separate address spaces, which is a common feature of all of the PCA architectures (and in general seems to be a feature of other high performance embedded architectures, such as IBM’s Cell [5] or Clearspeed’s [3]).

R-Stream 2.1 was designed to use the Gumdrop abstract array extensions after experience with the Brook language in our R-Stream 1.0 compiler. The R-Stream designers concluded that the principal benefit of the stream abstraction in Brook was the separation of the intentional and extensional semantics of storage elements, but that exposing the stream abstraction to the programmer caused the programmer to over-specify the mapping. This is because the stream abstraction is inherently one dimensional. The use of this abstraction in the SVM reflects the view that abstraction is a useful way to express an efficient physical execution model, where some inner or near inner loop is “streamed.” However, for the programmer, it is overly constraining: attempts to express multi-dimensional algorithms (such as operations over three dimensional radar cubes in GMTI) in Brook are baroque code expressions with early physical bindings that are far from efficient execution. To accept them, the HLC would be forced to "undo" much of what programmers were writing in order to get good mappings. Writing loops over abstract arrays is much more natural for programmers.

Abstract arrays represent both a limit to the R-Stream 2.1 compiler and an inherent limitation of the C language. R-Stream 2.1 does not have passes to raise code that is written in terms of C references to abstract arrays. Mainly, this is driven by the fact that this is a very hard problem to do in general and that it is easy to write code in C where it is impossible to raise arrays. Other limitations of the 2.1 mapper surface in the Gumdrop dialect, such as the requirement that loops to be mapped have explicit indication of the iteration variable (as "do loops"), that loops have simple stride expressions and fixed extents, and that array sizes be static and known at compile time.

R-Stream 2.1 performs the following mapping steps: loop unrolling, loop interchange, partial fusion, tiling, and high-level software pipelining. The loop interchange, fusion, and tiling are based on greedy algorithms that attempt to find fusions of stages in an algorithm like GMTI. Such fusion of stages allows producer-consumer locality to be exploited, saving chip IO bandwidth. This fusion transformation is interesting because the stages in an application like GMTI can be separated by various "corner turns" so the fusion cannot generally be along an inner dimension as is most common; the fusion has to comprise two dimensions. Furthermore, this fusion is constrained by the fixed local memory capacity of the distributed memory of the processors.

R-Stream 2.1 also performs high-level software pipelining, performing a fixed static scheduling of the tiles (as "kernels in SVM terminology) across processors and in time. This is performed with a high-level modulo scheduling. Thus, in the GMTI example, several "cubes" corresponding to different samples can be in flight simultaneously. R-Stream 2.1 arranges for double-buffering of local memory and generates bulk memory read and write operations (stream copy in SVM terminology) which correspond to DMA operations.

While R-Stream 2.1 can map across multiple PCA chips, this often is due to simplifications in the machine model for each of the chips. It is possible to model Raw as a single unified processor comprised of multiple Raw tiles acting synchronously, or as a chip multiprocessor with independent tiles. In our experiments, we model Raw as having independent tiles.

While the transformations that R-Stream 2.1 can perform are relatively powerful and illustrative, the implementation of the transformations is somewhat rigid. The sequence of transformations to be performed is fixed, and structured closely to correspond to the needs of the GMTI implementation we have. Limitations in the R-Stream 2.1 implementation prevent the compiler from performing certain simple "cleanup" optimizations before output is emitted. The reliance on fixed iteration extents, simple loops, and known array extents is deeply baked in to the mapper. The newer version of the compiler, R-Stream 3.0, is being built to address these limitations [13].

3.2. LLC

In our implementation of LLC, instead of building standalone compiler, we leveraged the available gcc compiler tailored for Raw by using library approach to SVM construction. In this approach, we build a library for SVM APIs that is used for an SVM code (See Figure 1). Although this approach does not provide full optimizations that may be provided by full SVM compiler, it enabled us to access the SVM framework in a short time period with minimal cost. In our current implementation, all SVM APIs in the specification are implemented.

The library provides several functions to support the SVM APIs. One of them is maintaining kernel data structure. When a kernel is started, the kernel data structure is passed to the secondary master. However, since the kernel start API is non-blocking API, it may return even before the kernel started. If the caller returns from a function in which the API is called, the memory space for kernel data structure can be freed and when the secondary master is ready to execute the kernel, the kernel data structure is not available. To solve this problem, our library maintains a copy of the kernel data structure in library memory space.

Another function that library provides is handling of data buffer for streams through dynamic networks. The dynamic network guarantees the order of data between two tiles. However, if two sender tiles send data to a destination tile, the order of the data in the receiver tile is not known at compile time. Thus, the library needs to identify the source of data whenever a packet of data arrives. Also, if a data being received belongs to a stream that is to be received later, then the data needs to be stored in a buffer. The library keeps buffers for storing such data for proper operation of the dynamic network.

4. Signal Processing Applications

In this paper, we implemented two kernels and one compact application: matrix multiplication, FIR bank, and GMTI that are described in this section.

4.1. Matrix multiplication

Matrix multiplication, C = AB, calculates an output matrix C from A and B, where A, B, and C are matrices. We implemented systolic matrix multiplication and mapped as shown in Fig. 2. A matrix data travels from left to right, and B matrix data travels from top to bottom. Each tile in the path of A and B computes matrix multiplication using incoming data from tiles on left and up sides. The result data travels to right side destination tiles. The matrix sizes for A and B are 3 by 128, 128 by 256, respectively.

4.2. FIR bank

FIR bank is one of the kernel benchmark suite [2] specified by Massachusetts Institute of Technology Lincoln Laboratory. The FIR bank implements a set of M FIR filters and each FIR filter m, $m \in \{0, 1, \ldots, M\text{-}1\}$, has a set of impulse response coefficients $wm[k]$, $k \in \{0, \ldots K\text{-}1\}$. It is mathematically specified as:

$$y_m[i] = \sum_{k=0}^{K-1} x_m[i-k]w_m[k], \text{ for } i = 0,1,\ldots,N-1 \cdot$$

The filters are distributed over tiles such that each tile has M/T filters, where T is the number of tiles. Each tile computes its own filters. Therefore, there is no communications among tiles. The FIR is implemented in frequency domain which is more efficient than time domain when data size is large. Therefore it requires FFT, complex product, and IFFT operations. In this implementation, a few optimizations in application level were performed: i) the bit-reversal operations are eliminated by bit-reversing filter coefficients and ii) using radix-4 FFT and radix-4 IFFT.

The parameters for the results reported in this paper are: K=12, N=32, and the length of the input data is 1024 complex data.

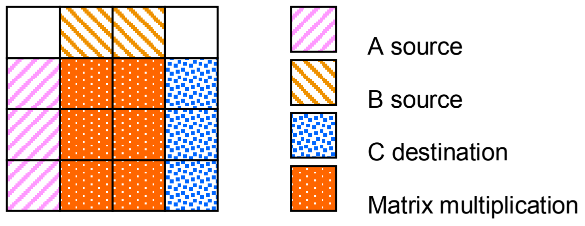

Figure 2. Matrix multiplication

4.3. Ground Moving Target Indicator (GMTI) [2]

GMTI is a compact application that represents airborne radar applications used to locate a target on the ground [2]. It consists of several stages: Time Delay and Equalization (TDE), adaptive beamform, pulse compression, Doppler filter, space-time adaptive processing, target detection, and target parameter estimation. TDE and pulse compression stages are mainly convolution in frequency domain. Thus, they consist of FFT, multiplication, and IFFT operations. Other stages mainly include matrix operations and FFT operations.

5. Implementation Results

To assess the SVM framework for Raw, we used matrix multiplication, FIR bank, and GMTI. The matrix multiplication and FIR bank were hand-coded using SVM APIs so that HLC is bypassed that allowed us to isolate the issues related to HLC. The code was then compiled using LLC that consists of SVM library and Raw C compiler. The code was executed on Raw hand-held board. GMTI is written in C-dialect code ("Gumdrop" code) that provides hints to HLC, and then compiled with HLC. The output code from HLC expressed in SVM APIs is compiled with LLC, and then executed on Raw processor.

The performance results for matrix multiplication are shown in Figure 3. The figure shows the number of cycles used for computation of each multiplication-addition pair, i.e., the total number of execution cycles is divided by the number of multiplication-addition pair in the matrix multiplication. On Raw, multiplication and addition each needs one cycle to execute. Thus, the lower bound of the number of cycles for a multiplication-addition pair is two. Figure 3 shows the number of cycles for a multiplication-addition pair as a function of the number of words per communication. The number of words per communication is the message size in words when a tile sends a message to a neighbor tile or a tile in the rightmost column for result data sending.

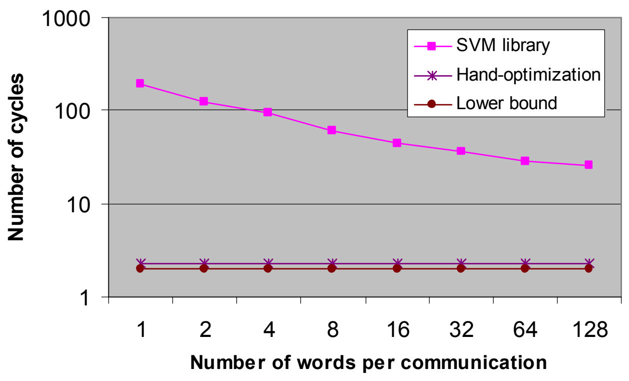

Figure 3. Matrix multiplication results

The curve in Figure 3 named "SVM Library" shows the performance when a full implementation of SVM API is used. The curve shows that the initial cost of the communication using library approach can be amortized over a long sequence of data. This is a good property since stream processing tends to have a long sequence of data. Note that the performance of the library may be optimized depending on the situation.

In matrix multiplication, we optimized it in several ways. One way is utilizing available multiple networks among tiles so that a network between two tiles handles only one stream. Then, the overhead for managing multiple streams in a network can be eliminated. Another optimization was using hand-assembly code for critical section of the code. This allows us to use minimal number of instructions and optimal instruction scheduling.

The last optimization is using network ports as operands as described in Section 2. The curve named

"Hand-optimization" in Figure 3 shows the performance when these optimizations were applied. The most optimized results show that it takes only about 10% more overhead than the theoretical lower bound, and the main reason for the overhead was due to the loop outside of the deepest loop and software pipeline overhead.

FIR bank was specified for two data sets in Polymorphous Computing Architecture (PCA) kernel benchmark specification [2]. The first set is a large data set: number of filters = 64, number of input data = 4096, and the number of filter coefficients = 128. The second data set is a small size: number of filters = 32, number of input data = 1024, and the number of filter coefficients = 12.

It is implemented manually using SVM APIs. We performed several optimizations including all three optimizations applied for matrix multiplication. An additional optimization applied for FIR bank is using broadcast capability of switch processor that reduces the workload for tile processor. In the broadcasting scheme, when a switch processor receives data from a source, it duplicates the data and sends one to the computing processor and sends the other to another destination switch processor which performs the same operation.

The FIR bank are optimized in several ways in algorithmic level: using radix-4 FFT, elimination of bit-reversal, using overlap-save method, minimization of address calculations using offsets, and preventing register spilling by restricting the number of registers used. The implementation results are shown in Figure 4 and Figure 5. In Figure 4, UB denotes upper bound considering only floating point operations. Since there are 16 tiles each of which can compute one floating point operation, the UB is 16 floating point operations per cycle.

IUB denotes the upper bound when load/store operations are considered as well as the number of floating point operations. Since the load/store operations were inevitable in our FIR bank implementation that uses Radix-4 FFT, the IUB is a "practical" upper bound. The performance in Figure 4 is obtained when input and output data are in cache so that time to access the external memory is not considered. Our results show that the hand-optimized results are very close to the "practical" upper bound with only about 10% overhead.

The result denoted as "compiler optimization" is obtained using LLC with algorithmic optimizations only. It shows about three times of the difference between hand-optimized performance which are mainly due to additional instructions and non-optimal instruction schedules.

Figure 5 shows the effect of accessing data from memory. It takes about 16% additional cycles when data is accessed from memory. The additional cycles are not significant since in FFT computation, memory access is not frequently performed due to the fact that once data is loaded from memory to cache, then the data is used many times before final result is stored in memory.

Figure 6 and Figure 7 show the results of GMTI implementation. Figure 6 shows the execution schedule of each processor including a primary master, secondary masters, and stream processors. Note that tile 0 is mapped to the primary master, one secondary master, and one stream processor in time-shared mode. Other tiles are mapped to one secondary master and one stream processor. The execution schedule shows parallelization of HLC.

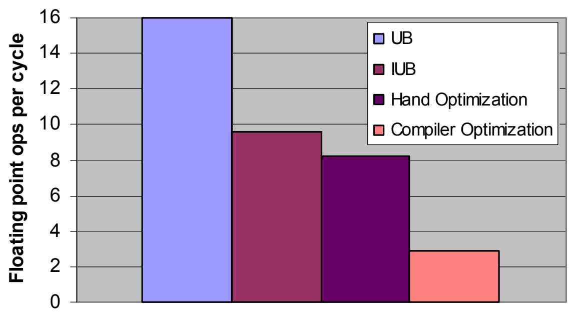

Figure 4. Throughput results for FIR bank (data from cache)

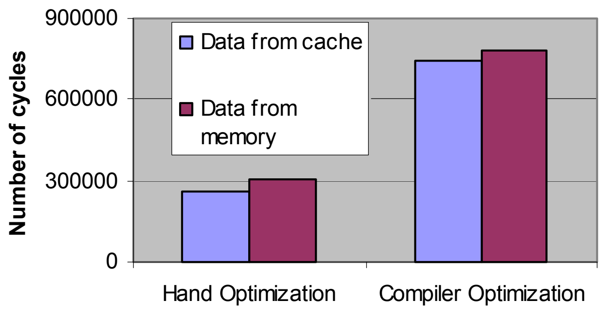

Figure 5. Latency results for FIR bank

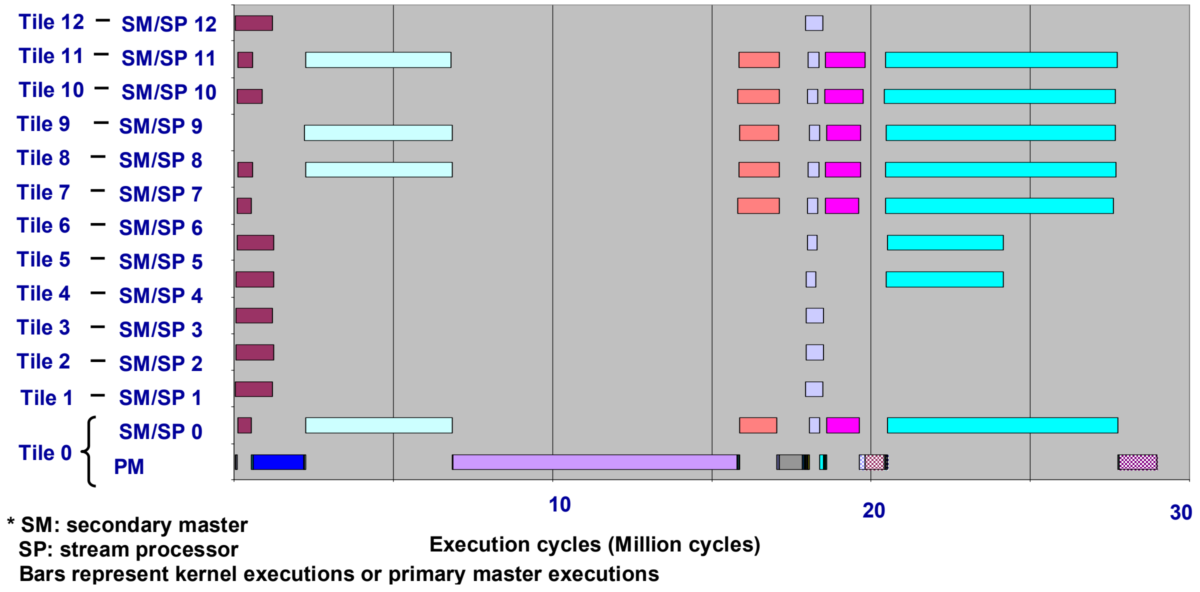

Figure 6. GMTI execution schedule

The application is parallelized up to 12 processors. Note that HLC actually can parallelize up to the maximum number of tiles on Raw processor, i.e., 16 parallelization. However, there was a problem in execution of four tiles that prevented execution of 16-tile version code and we are investigating the problem. Fig. 6 shows that some portions of the application are not parallelized due to serial nature of the portions of the application. However, the available slots may be used by using software task pipeline technique. The utilization (number of floating point operations/(number of floating point unit * number of execution cycles)) is about 0.5%. Some of the reasons for the low utilization are the empty schedules in Fig. 6, many load/store operations, and some redundant operations.

Since GMTI is a large application, analyzing the performance using our rigorous steps including optimizations is not feasible. Therefore, we chose one stage, Time Delay and Equalization (TDE), in the GMTI and performed optimizations and compared the result that is shown in Figure 7. In Figure 7, x-axis shows several steps in TDE. In each step, the first bar marked as "R-Stream" shows the performance of using HLC and LLC with only algorithmic optimizations used in FIR banks. The next bar marked as "direct copy" shows the performance when data is moved without using SVM calls. Then, hand-assembly performance shows when critical sections of the code are optimized using hand-assembly.

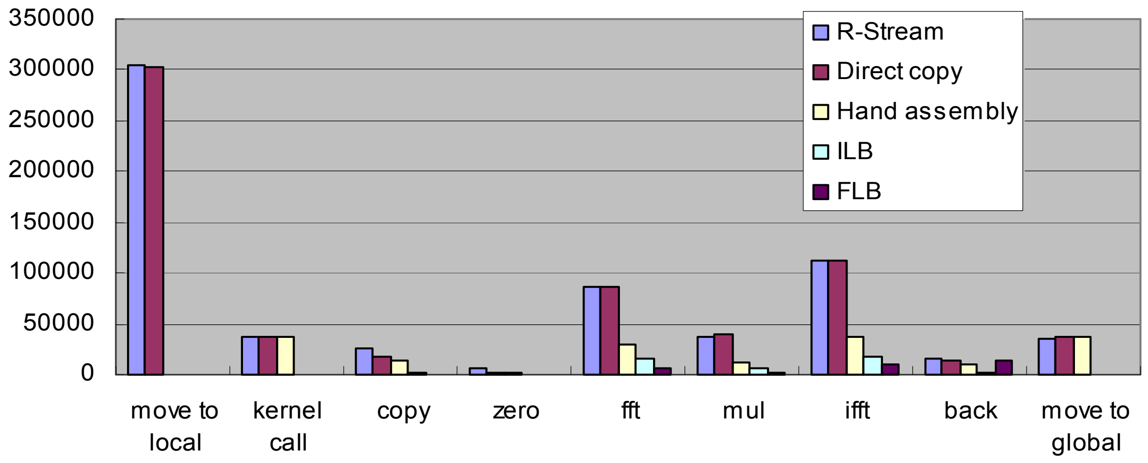

Figure 7. Breakdown of the cycles for the first stage (TDE) of GMTI

The ILB denotes the "practical" lower bound that include load and store operations as well as floating point operations. Note that the hand-assembled code performance is close to the ILB.

The bar marked as "FLB" shows the lower bound when only floating point operations are considered. The results show that the hand-optimized code obtains very close performance to the "practical" lower bound that is expected to be obtained if HLC and LLC incorporate the optimization techniques we applied. The performance of the R-Stream is also encouraging as the difference between the R-Stream and hand-optimization code is only about three times for major computation parts even when the R-Stream is in prototype stage.

The results also reveal where the current tool chain needs improvement. One of the improvements needed is better capability of parallelization. Although current HLC parallelizes up to the maximum number of processors, in some stage in GMTI, it fails to parallelize the code due to memory constrains. We expect HLC can parallelize these sections with better code analysis capability. Another area of improvement is data movement as shown in Figure 7, and we are working to improve these.

6. Conclusion

The authors have presented implementations of SVM framework for Raw processor. We implemented R-Stream as an HLC. HLC performs parallelization of the code and was able to generate up to the maximum number of processors in several stages in GMTI. We build SVM library to be used for LLC in conjunction with the C compiler for Raw. Using the tool chain, we implemented several signal processing kernels and a compact application. The implemented tool chain enables full path from high level stream languages to the processor and quick assessment of the SVM approach.

The implementation results show that the current tool chain in SVM framework provides a reasonably good performance in several key sections of the code. We applied several manual optimizations to understand the performance issues in SVM framework and were able to obtain the performance very close to the theoretical peak performance of the kernels. We expect similar performance improvement can be obtained using optimizations when the HLC and LLC are mature enough.

7. References

[1] D. Burger, S.W. Keckler, K.S. McKinley, et al., "Scaling to the End of Silicon with EDGE Architectures," IEEE Computer, 37 (7), pp. 44-55, July, 2004.

[2] W. Coate and J. Lebak, "Morphing Scenarios For The GMTI Portion Of The PCA Integrated Radar Tracker," Massachusetts Institute of Technology Lincoln Laboratory, 2004.

[3] Clearspeed, http://www.clearspeed.com/aceleration /technology, 2006

[4] Defense Advanced Research Projects Agency, http://www.darpa.mil/ipto/Programs/pca/index.htm, 2006.

[5] M. Gschwind, "Chip Multiprocessing and the Cell Broadband Engine," Computing Frontiers, May 2006.

[6] R.J. Haney, J.M. Lebak, M.A. Alexander, H. Chan, P.A. Jackson, and E.L. Wong, "Polymorphous Computing Architecture (PCA) Kernel Benchmark Measurements on the MIT Raw Microprocessor," ESC-TR-2006-063, Massachusetts Institute of Technology Lincoln Laboratory, 2006.

[7] P. Mattson, W. Thies, L. Hammond, M.V. Raytheon, "Streaming Virtual Machine Specification," Version 1.0.1, http://www.morphware.org , March 2005.

[8] J. Miller and A. Agarwal, "Software-based Instruction Caching for Embedded Processors," Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XII), San Jose, CA, October 2006.

[9] F.Labonte, P. Mattson, I. Buck, C. Kozyrakis and M. Horowitz, "The Stream Virtual Machine," PACT, September 2004.

[10] K. Mai, R. Ho, E. Alon, D. Liu, Y. Kim, D. Patil, and M. Horowitz. Architecture and Circuit Techniques for a Reconfigurable Memory Block. ISSCC, February 2004.

[11] Morphware forum, http://www.morphware.org, 2005.

[12] Reservoir Labs., "R-Stream - Streaming Compiler," http://www.reservoir.com/r-stream.php, 2006.

[13] E. Schweitz, R. Lethin, A. Leung, B. Mester, "R-Stream, a Parametric High-Level Compiler," High Performance Embedded Computing Workshop, 2006.

[14] A. Smith, J. Burrill, J. Gibson, B. Maher, N. Nethercote, B. Yoder, D.C. Burger, K.S. McKinley," Compiling for EDGE Architectures," 2006 International Conference on Code Generation and Optimization (CGO), March, 2006.

[15] J. Suh and J. O. McMahon, "Implementations of FIR for MONARCH Processor," 10th High Performance Embedded Computing Workshop, Boston, MA, Sept. 2006.

[16] M. B. Taylor, et. al, "Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams," Proceedings of International Symposium on Computer Architecture, München, Germany, June 2004.

[17] M. Vahey, et. al, "MONARCH: A First Generation Polymorphic Computing Processor," 10th High Performance Embedded Computing Workshop, Boston, MA, Sept. 2006.