A STREAM VIRTUAL MACHINE

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Francois Labonte

June 2008

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Mark Horowitz Principal Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Bill Dally

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Christos Kozyrakis

Approved for the University Committee on Graduate Studies.

Abstract

Uniprocessor performance scaling has ended because of power and complexity issues. In order to continue to scale computer performance, future machines will be parallel collections of processors. Thus the key problem in computing becomes how to program these parallel machines, since unless the user applications leverage the parallel hardware, application performance will not scale.

While creating a general multi-threaded parallel application is known to be difficult, many if not most performance critical applications have regular data parallelism, which allows one to use parallel processors while maintaining a single thread of control as a programming abstraction. In this "Stream" programming abstraction, there is a single control thread like before, but the operations are issued from this control thread operate on blocks of data at a time rather than single values. This idea of launching block operations is actually an old programming model; vector machines used a version of it. Adding a compiler managed buffer memory to stage data transfers and chain operations we get a stream programming model.

Many interesting stream machines have been built, but each created their own software system. These software systems all have to solve the same hard problems of data partitioning and computation scheduling. We created a Stream Virtual Machine (SVM) as a virtual machine layer to present a single abstraction to a common system software framework. The SVM exposes key components of the underlying machines such as the number and sizes of the stream memories, stream processors and Direct Memory Access (DMA) engines. Each one of these components is characterized by performance parameters that allow the system software to make informed decisions for efficient mapping and scheduling of computations and data transfers. The SVM

defines an API that a stream compiler will use as an output and that each machine implements.

We evaluate the SVM and show it is an efficient abstraction for a number of stream machines, including non-conventional stream engines like Graphics Processors Units (GPUs) and Chip Multi Processors (CMPs). On GPUs we've found that performance parameters extracted through simple tests give good predictions of the run-time of applications. To evaluate the performance overhead and scalability of the SVM, we have implemented it on the Smart Memories chip multiprocessor. Data parallel applications were written for streaming, then converted to SVM code using a stream compiler, which were then compiled down to machine binary using our SVM library implementation for Smart Memories. The applications have shown good performance scaling as we increase the number of streaming processors and memories. The overhead of the SVM API versus native calls is small, less than 0.5 percent in most cases.

Acknowledgement

I am very grateful to have had the opportunity to work with my advisor, Mark Horowitz. He gave me a lot of responsibilities, was patient and most importantly steered me to concentrate my efforts where they would bear the most fruits.

I've had the chance to be a student in Bill Dally's classes and be apart of his research efforts, notably the Merrimac group, where my thesis project got started. He's always been very available to me and was a great source of inspiration for which I am grateful.

Chrisos Kozyrakis joined Stanford after I did but he made some great contributions to my research and helped me get to know other students with common interest by organizing paper discussions groups and computer architecture group lunches.

My experience in graduate school was shaped by the other graduate students I had the chance to work with. Mark Horowitz's VLSI group's discussions were interesting enough to make me look forward to our group lunch on Wednesdays. I've been fortunate to be a part of two research groups and be the bridge between them, the Merrimac group provided me with the inspiration and the motivation for Streaming while the Smart Memories group challenged me to harness streaming and make it work. I have fond memories of lively discussions with many faculty and students from the Merrimac group including Mendel Rosenblum, Pat Hanrahan, Mattan Erez, Tim Knight, and many others. Smart Memories was also a great group to be apart of, facing a daunting implementation schedule with little man-power, I am proud to have been part of the effort with great people such as Alex Solomatnikov, Amin Firoozshahian, Ofer Shacham and others. My office in Gates 320 was also a great location for support and discussion alike, and my numerous officemate that followed

me on my way from the desk next to the door to the one next to the window are great friends. Finally I'd like to thank my family and friends around campus and around the world for their moral support in this endeavour.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

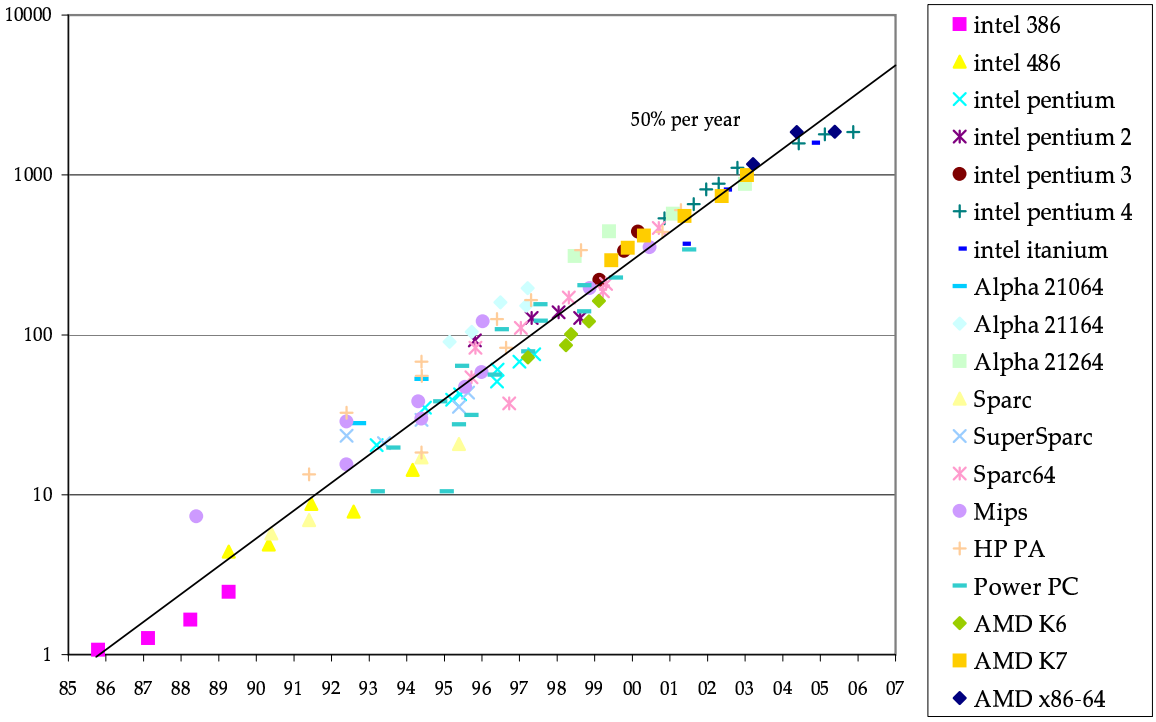

Figure 1.1: Spec Int2000 performance by year of introduction

For the last 20 years, microprocessors have improved their performance at an

average rate of 50% a year as seen in Figure 1.1. Processor designers have leveraged process scaling to scale clock frequency and create more complex architectures that extracted more parallelisms from the instruction stream to improve performance. However since 2004, the rate of improvement of single thread applications has dropped because power constraints now limit the architectural improvements that can be deployed [14].

Currently the only way to continue to scale computer performance at the historical rate is to use the extra transistors provided by process scaling to build additional processors on the die. While the advantage of these multiprocessor chips is obvious for computers running server type applications which effectively need to execute many different programs (one for each user) at the same time, the advantages for desktop applications is less clear. Here the number of programs running simultaneously is small so to improve the performance we have to deal with the challenge of having a single application use multiple processors. Historically this has been a difficult task.

Rather than trying to create a universal parallel programming method, it is often easier to try to solve the problem for a restricted application domain first. Looking at the applications that are currently driving desktop performance requirements the hardest, one notices that most of these applications deal with large amounts of data. In the consumer sector, multimedia workloads are driving the purchase of new computers and cell phones. Embedded systems are also seeing an increase in their workload with more sensor data, higher data rates and more complex processing requirements. In addition, many of these applications, not only deal with large data sets, they also have large amounts of data parallelism - they apply the same computation to a large number of data elements. To more easily describe this type of computation, many people have proposed using a "Stream" programming model [37][30] for these type of applications. Stream programs extend normal programs by allowing the programmer to gather data into a "stream" and then allow the programmer to apply functions to streams as well as integer and floating point numbers. Examples of stream like programming models include StreamIT from MIT [34], StreamC from Stanford [1], Simulink from MathWorks [24], synchronous dataflow language from UCB [18], etc.

This dissertation attempts to leverage both these trends, the growth of parallel processor chips and the growing compute demands of data parallel applications, by using the unique properties of steaming applications to greatly simplify mapping these applications to parallel processors. There are many characteristics of streaming applications that make mapping easier. Most important, the dataflow graph for most stream programs can be statically analyzed. This means that the compiler can explicitly manage the memory hierarchy, prefetching the data that the application needs so it never needs to stall waiting for memory. This prefetching allows these applications to overlap computation and communication. It can further optimize communication by knowing which data values really need to be written back to memory, and which were simply temporary values and can be destroyed.

Given the large variety of both stream programming languages and parallel execution models, this thesis proposes a Stream Virtual Machine (SVM), an abstract machine that many systems could use to make porting stream applications easier, and then evaluates the effectiveness of this model. To provide the needed context for this discussion, the next chapter presents an introduction to previous stream processors and the languages and programming systems that are used to program these architectures. This diversity in languages and processors has led to the same concepts being invented many times. The SVM provides a stream software abstraction that allows many of the software tools to be shared between machines.

1.1 Research Contributions

The original research contributions of this thesis to the field of computer architecture and stream processing are:

1. A stream machine abstraction that allows one to reason about how to map stream program on different stream processors. This is the first attempt to make stream programs and transformations portable across different implementations.

2. An API that allows for low-level mapping of stream applications onto stream machines which allows for performance evaluation of stream applications on

hardware.

3. Definition and methods of gathering of simple performance parameters that help predict run-time of stream kernels, information that can be used by a stream compiler.

4. Implementation and evaluation of the stream programming model on a chip multiprocessor by using only general chip multiprocessor components.

1.2 Thesis Roadmap

Chapter 3 describes our stream software abstraction. It is a general way to program many architectures with the stream programming model. It is both an abstraction that generalizes streaming optimizations and a streaming application programming interface (API). The SVM models the hardware at a higher level of abstraction. It has a number of parameters that try to capture the essential features of the hardware so that higher level software tools can generate code optimized for each stream processor.

Recent graphics processors are programmable and are now used for general purpose computation as a stream-like processor. In order to evaluate if our stream software abstraction is a reasonable virtual machine, Chapter 4 looks at mapping the SVM to modern graphics processors. This will help us evaluate if the SVM model is a flexible enough target to allow a compiler to make make good decisions on mapping and scheduling the computation. The results were promising, and as graphics processors become more generally programmable [10] this model will match even better.

Chapter 5 then looks at mapping the SVM on today's homogeneous chip multiprocessors. It looks at whether this mapping is possible, and what hardware is needed in these machines to make this mapping efficient. To investigate possible trade-offs, we use the Smart Memory design platform. Smart Memories is a chip multiprocessor designed to support different execution models including streaming. Since Smart Memories was designed to have a very flexible memory system, we use it to explore some implementations of SVM API calls needed to support streaming. This experience demonstrated the benefits of having a flexible memory system, since it allowed

us to create features that were not anticipated in the original design. Finally we will conclude by describing the benefits of stream processing on modern CMPs, and the benefits of using a SVM to decouple high-level stream decisions from the detailed hardware implementation.

Chapter 2

Stream Processing

In order to understand the appeal of stream programming, we first need to understand the basic "math" that underlies computer performance. As we will see in Section 2.1, the performance really depends on two factors, one is the rate that the machine can perform the average operation, and the other is the average number of operations that the machine can find to execute in parallel. To scale performance, successive processor generations have tried to improve both factors, increasing clock frequency and mechanisms for increased parallelism at a great cost in complexity and increased power dissipation. Interestingly, this processor review will show that data parallelism is already being exploited in a small way. High-end uniprocessors used data parallelism to improve their performance by converting it into instruction level parallelism (ILP) that they can exploit.

Unfortunately power, complexity, and long cache miss wait times have made it hard to continue to scale the performance of a single processor, and the industry has moved to building multiple processors on each chip to continue to improve chip performance. The traditional way to write programs for single cores does not lead to improved performance on multi-processor systems and a new approach to write parallel programs is needed for these machines. Fortunately for many data parallel applications there is a nice programming model, called Streaming, that maps well to these new parallel machines, providing a simple way to both extract parallelism for the multiple processors, and hide the latency of the memory fetches. Section

2.2 describes this programming abstraction, and how it can be leveraged by parallel machines.

Many machines have been built to exploit streaming applications, and Section 2.4 describes a few of these machines. Even though many of the hard problems that need to be solved to map stream applications to processors do not depend on the hardware details, each of these systems created their own software system, which both causes researchers to have to duplicate effort, and prevents the initially primitive software system from evolving into mature systems. After looking over the machines, it seems possible to create a common Stream Virtual Machine which can represent any of these systems. The next chapter describes this virtual machine in more detail.

2.1 Processor Performance

$$CPUtime = \frac{IC \times CPI}{Clockrate} \tag{2.1}$$

Equation 2.1 is the classical computer system performance equation, which relates the execution time of a program to the product of the Instruction Count (IC) and the average Cycles per Instruction (CPI) divided by the clock rate [7]. The instruction count depends on the instruction set architecture and the compiler technology used. The instruction cost is unlikely to dramatically improve since the RISC versus CISC debate has shown that complex instruction that decrease the instruction count actually increase the cycles per instruction and are not helpful. In fact modern CISC architectures now decompose complex instruction into simple ones. Performance improvements have come from an increase in the clock rate and a reduction in the CPI which are described next.

2.1.1 Clock Rate

Increasing the clock frequency without hurting the average cycles per instruction has been one of the main ways to improve performance. As Figure 2.1 shows we were on a fast clock scaling curve which was the result of both process scaling and deeper pipelines, but this trend has now slowed because of excessive power dissipation and

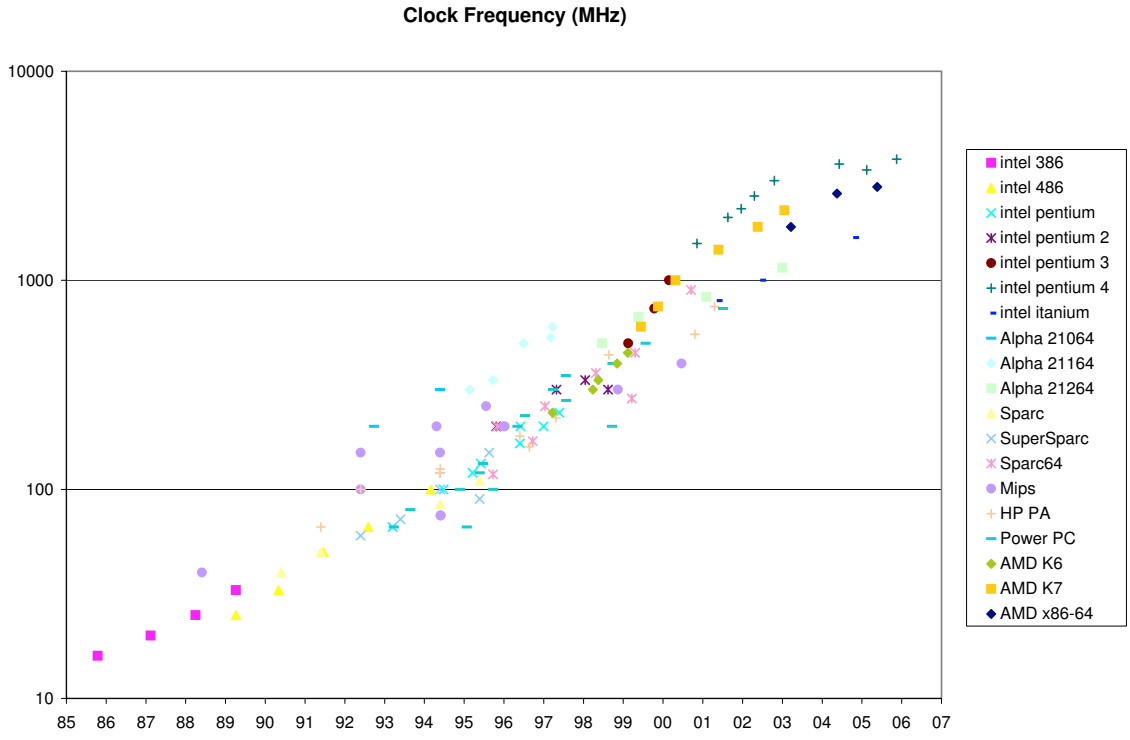

Figure 2.1: Evolution of Processor Clock Rate

the limits of pipelining. Building machines with very deep pipelines is inefficient from a power perspective [38], and since all machines are now power limited, both pipeline depth and clock frequency have recently decreased. While clock rate will continue to increase as integrated circuit technology scaling provides faster gates, this will be at a much slower pace than before.

2.1.2 Parallel Issue

To decrease the average cycles per instruction, the application needs to have instructions that run in parallel (they don't depend on each other) and the hardware needs to have parallel functional units that can exploit this parallelism. Extracting parallelism from the application consists of finding operations that have no mutual dependencies and can be executed independently on available resources. This type of parallelism

is called instruction level parallelism. Super-scalar architectures extract this parallelism through a combination of hardware that detects independent instructions and compilers which attempt to schedule them adjacently. VLIW architectures rely entirely on compilers to explicitly schedule instructions that can run in parallel, in the same issue slot, in order to simplify the hardware. To exploit this parallelism requires building hardware that can fetch, schedule, execute, and retire multiple instructions each cycle. Since instruction stalls are not uncommon, to achieve good performance requires allowing the machine to execute instructions out of program order, which further increases machine complexity. Even with this enhancement, the amount of parallelism that can be exploited is limited and few processors issue more than 4 instructions per cycle[36][31].

Data level parallelism is not universal but is prevalent in classes of applications in domains that deal with a lot of data: multimedia, digital signal processing and scientific applications. While the amount of instruction-level parallelism is low, data parallelism can be very large, as large as the dataset sometimes. In fact applications that appear to have large ILP often are converting data parallelism to ILP by a process called loop unrolling.

Loop unrolling done by a compiler or dynamically in out-of-order processor allows multiple iterations of a loop to be executed at the same time. If the loop was data parallel, then these instructions will be independent and can be exploited as ILP in a superscalar processor. The addition of a short vector unit to a processor allows for further exploitation of data parallelism at the cost of aligning the memory and moving the data to the short vector register file[35].

While it is possible to do this conversion, there are better methods of exploiting this parallelism. The basic problem is that in the current programming model, memory fetches are hard to disambiguate, and since control dependencies are hard to predict, a lot of hardware and power is spent on speculating to speed up loops that get executed very often. True data parallel sections are often easy to analyze leading to much more efficient hardware implementations.

2.1.3 Parallel Processors

Since exploiting ILP is power inefficient, CPU architect have moved to build explicitly paralel systems. The size and complexity of the processors becomes a design option as well as whether to use homogeneous processors or heterogeneous processor designs. Simple CPUs with a core and a caches are quite small compared to current complex uniprocessors. For the same die area as a complex core, tens to a hundred of the simple cores can be built. The next design problems come from determining how to build the memory system for so many cores, since many complex issues need to be resolved, like what memory consistency model is needed, what interconnection network will be used.

The answer on how many different cores, of which type, and how to build the memory system will depend on the programming model. More complex cores makes sense when the amount of parallelism is limited, but when it's easy to generate parallel computations, simple cores are the way to go. Applications with large amount of data parallelism can leverage numerous simple cores but they need a programming model that makes it easy. It is those applications that are a good fit is the stream programming model.

2.2 Stream Programming

The problem with parallel machines is that reasoning about multiple threads of control is difficult because of the need to communicate and synchronize properly. People have a hard time getting the communication correct and debugging these problems is very hard. The stream programming model allows the user to think about a single thread of control in charge of delegating operations on large amounts of data in parallel.

In the stream programming paradigm streams are groups of identical data types that are consumed and produced by kernels of computations. The data type of a stream is not limited to a simple data type but can be an aggregation of data like a C structure such that each stream element has the same size. Kernels are functions with variables and flow control, but can only access data in the stream, they can't

contain references to main memory. A very strict model would require sequential access to streams but a more relaxed model allows a certain window of visibility or full indexed access to small streams [15]. Thus kernels are free of arbitrary memory accesses with their attached high latency costs and the system can prefetch all the streams required before a kernel gets to be executed. Applications with sufficient data parallelism can amortize the costs of setting up kernels and completely hide the cost of memory transfers.

2.2.1 Data Parallelism

An application exposes data parallelism when multiple data elements of the same type can be produced independently. A finite impulse response (FIR) filter is a good example where each output element is the result of a computation of a limited number of input elements but each output element doesn't depend on other output elements' computations. The parallelism is easy to extract because in essence there is a single thread of control with the same processing being applied to each data element

Among applications that have a lot data parallelism, we are interested in an important subset, those that have a static analyzable dataflow. This static dataflow can be analyzed by a compiler to reason about data movements and computations allowing data movements to be scheduled in advance of the computation. This means the data is prefetched before it is needed, overlapping the fetch time with computation, thus the application can work at the maximal rate without having memory latency wait times.

Most data intensive applications from the digital signal processing and multimedia domains fall into this category because of the regular nature of the computations. Exploiting data parallelism to improve performance is an old idea, in fact, vectors were an early streaming model where the data parallelism was expressed at the instruction level making a single instruction perform the same operation on vectors of data. Vector instructions are then chained to compute output vectors in parallel. The data parallelism of individual instruction is explicit which allows for parallel execution as well as prefetching of future operands. Streaming extends this model to an kernel, a

group of instructions and adds a memory hierarchy in front of the registers to stage even more data elements. Streaming also adds the chaining of kernels through the stream memory, to decompose steps in the productions of output. [1]

In synchronous dataflow, instead of expressing the computation as parallel instructions, the process of generating a single data parallel result is decomposed into separate tasks with static data movements. This does a good job of exploiting data locality and spreading the workload across multiple processing elements although not very equally. Another downside is that strict syncronous dataflow can only deal with applications that have a predictable amount of computation which rules out applications with a data dependent amount of computation that cannot be determined at compile time. More recent extensions to the synchronous dataflow to allow dynamic rates lead to a stream astraction.

2.2.2 Stream Program Abstraction

Many signal processing applications are great candidates for streaming, starting with any kind of filtering, fast Fourier transform (FFT) as well as convolutions in multiple dimensions. Compression and decompression of video data and the generation of three-dimensional graphics are both complete applications whose natural data parallelism has made them successful in the stream programming model.

In the scientific domain we will look at how an application like molecular dynamic requires some rethinking of the application to fit the streaming model. Molecular dynamics is a physical simulation of particles where forces between particles are calculated at each time step and moved as a result. Forces are only evaluated for particles within a cutoff region to limit the calculation.

In the classical model for each molecule we loop through each other molecule, compute the distance and if it is within the threshold range we compute the force resulting from that molecule. Finally all the forces are summed for this molecule and it's position is updated. In order to fit the streaming model we will try to optimize the memory accesses. Figure 2.2 shows how each iteration proceeds.

[1] which is again not new, and was pioneered by sychronous dataflow [18]

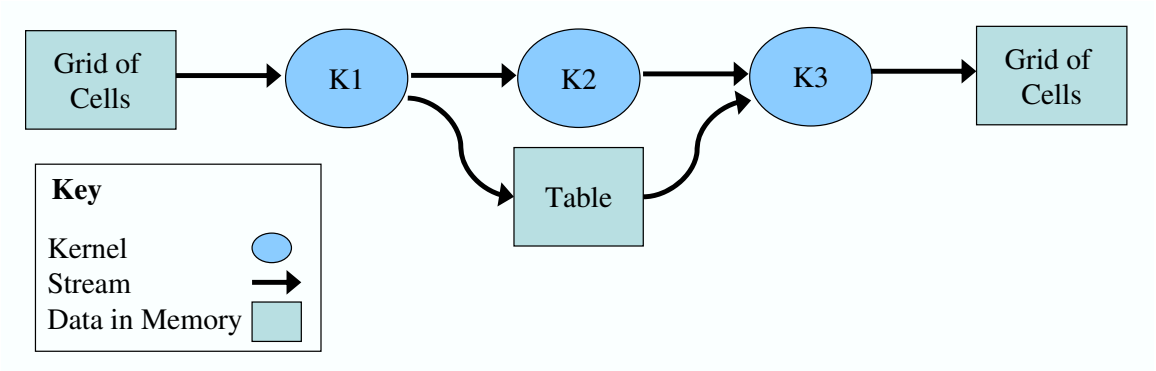

Figure 2.2: Molecular Dynamics Stream Diagram

To avoid fetching data elements of the molecule that are not useful to compute the position but used to compute the effective force, we will first compute in K1 the distance of the molecule and if it is within range save the distance and generate the memory index for the extra data elements required to compute the force. In kernel K2, the force computation that only depends on the distance will take place while the memory access fetches the required data elements for the final force computation in kernel K3.

The stream implementation has allowed us to minimize memory transfers by fetching extra molecule data only for the ones that required a force computation. The latency of the memory access has been hidden by further computation in kernel K2.

2.3 Stream Machines

With a programming model that exposes data parallelism with statically analyzable data flows, it is possible to build a multiprocessor to efficiently execute these applications. A stream processor has a high number of compute unit and a memory hierarchy that can be explicitly controlled.

As much as conventional processors are desired for their fast clock rates and low memory latencies, stream processors strike a balance between their bandwidth and parallel compute rate. The goal is not to waste the expensive part of the system,

memory bandwidth, whereas the incremental cost of adding ALUs is small. For most applications, the goal is to maximize the use of the memory bandwidth, although there will always be certain applications for which we are limited by the compute rate. This leads to an important metric for stream processors, the compute to memory bandwidth ratio which is a measure of how many arithmetic operations are available for each word of bandwidth to data stored outside of the chip.

What sets stream architectures apart from conventional processors is how they handle data at the level of streams, doing bulk operations that gather and stage data operands before they are operated on. With enough data parallelism the prefetching of data can be done ahead of when it's needed, hiding the memory latency as long as there is enough memory bandwidth to cover the computation time of a previous block. This technique, called double-buffering, allows the overlap of computation and communication which is central to stream architectures.

Locality is exploited in stream processors by keeping temporary data inside the chip and close to where it is needed. Unlike caches on conventional processors, the data is explicitly moved from main memory to on-chip memory. Modern prefetchers on caches which predict strided accesses can do a good job as long as the data being accessed is contiguous or at a regular stride distance [5]. But they fail when the data needed is indexed and the full data structure doesn't fit in the cache.

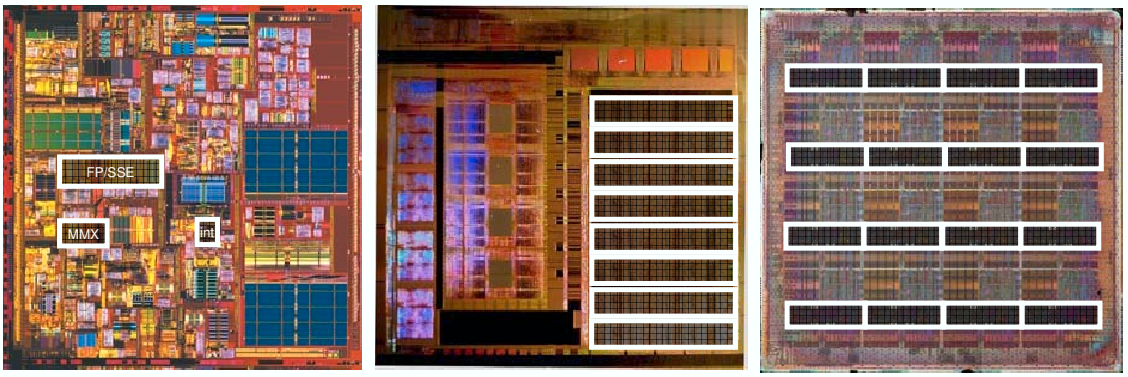

Figure 2.3: ALU areas of Pentium 4 Northwood, Imagine and RAW

Figure 2.3 shows die photos of a a conventional processor, the Pentium 4 Northwood, and two stream processors, Imagine and RAW. The area of the die of the ALUs is highlighted to show the respective proportion of area devoted to them. The Pentium 4 Northwood has 512KB of on-die cache seen on the right, most of the other area of the core is dedicated to speculation hardware. The Pentium 4 has three areas of ALUs, the regular integer, MMX which provides integer short vectors and the floating point and SSE floating point short vectors. The Imagine processor has 8 clusters that contain 6 ALUs each, just left of it is a 128KB stream memory. RAW's 16 tile each contain an integer and floating point ALU in the middle, with some caches and routing network logic.

2.3.1 Imagine

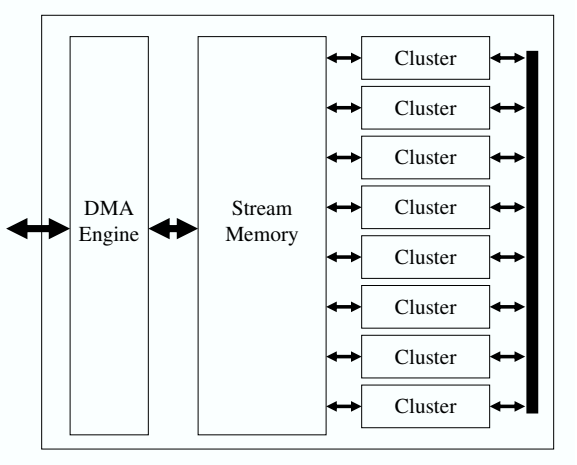

Figure 2.4: The Imagine Stream Processor

Imagine, shown in Figure 2.4, is a stream processor that was designed at Stanford University in Bill Dally's research group. In order to achieve a high level of utilization of compute units, Imagine has eight identical compute clusters which exploit data parallelism by executing the same instruction in a SIMD fashion. This saves on instruction bandwidth and complexity for storing and issuing them. Inside each

cluster a mix of six ALUs allows exploitation instruction-level parallelism within each data paralel execution function.

The memory hierarchy of Imagine is split into three levels, registers, stream memory and main memory. Table 2.1 shows the hierarchy size per cluster and then total for the eight cluster Imagine prototype. The stream memory provides buffering and staging of stream data used as kernel operands.

Memory	Per Cluster	Total
32-bit register	272	2176
Stream memory	16kB	128kB

Table 2.1: Imagine memory hierarchy

Imagine has a stream load/store unit that allows for concurrent transfers in between the stream and main memory as well as processing of stream elements by the clusters. The memory system of Imagine is designed to optimize high bandwidth at all levels between the different levels of the memory hierarchy. Memory transfers between the stream memory and main memory are aggregated for sequential access to the main memory to benefit from DRAM improved sequential access. The stream memory itself is designed as a much wider SRAM than the cluster's access size with some buffers such that sequential accesses to the stream memory can be done less often which frees the stream memory for transfers with the main memory system or other stream accesses.

StreamC and KernelC are the two languages that were created along with the Imagine processor in order to program it. StreamC allows the composition of streams and kernels and to string them together to constitute a full application. KernelC allowed the programming of kernels with special stream accesses and ways to use the inter-cluster communication network.

Although dedicated purpose ASICs have a higher density of ALUs, for a programmable processor, Imagine has a lot of ALUs to exploit DLP across it's clusters and both ILP and DLP through loop unrolling within them. To keep those ALUs supplied with data, the memory system is simple and independently capable of moving data at a high bandwidth during computation.

2.3.2 RAW

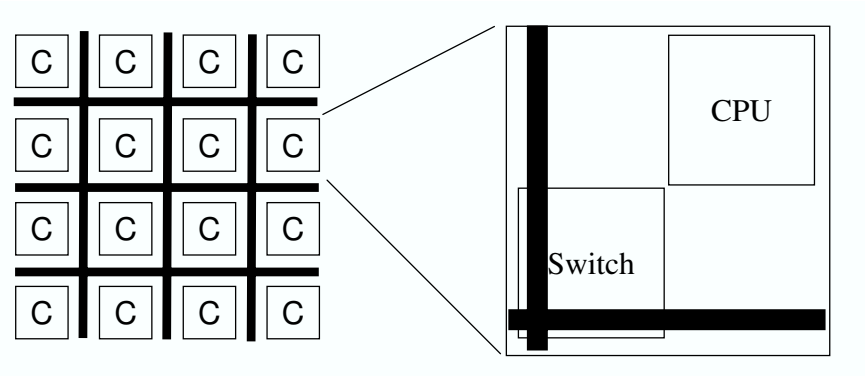

Figure 2.5: RAW Processor

A different approach to streaming was taken for the RAW chip microprocessor designed at MIT in the research group of Anant Agarwal and Saman Amanrasinghe. As seen in Figure 2.5, RAW has two special communication networks that enable the processors to communicate directly through their register files. The first network is statically routed which gives it a shorter latency while the second one dynamically routed has lower throughput and greater latency. What makes RAW a stream machine is it's high density of ALUs and high bandwidth dataflow networks to keep them supplied with data.

Since each processor has it's own program counter, it can execute different instructions and as such processors can be different steps in an execution pipeline where the data is passed in the communication networks. RAW is a dataflow architecture and is very well fitted to implement any synchronous dataflow programming model.

StreamIt [34] is the stream programming system that was designed to program the raw processor which follows very much the synchronous dataflow model. Figure 2.6 shows a graphical depiction of a streamIt application. Each node is a kernel named a filter and each link is a stream. Filters can peek and pop or push on a stream depending if it's an input or output stream.

Composed of a multitude of simple processors with a powerful communication network, RAW exploits data and instruction level parallelism across it's many cores. The

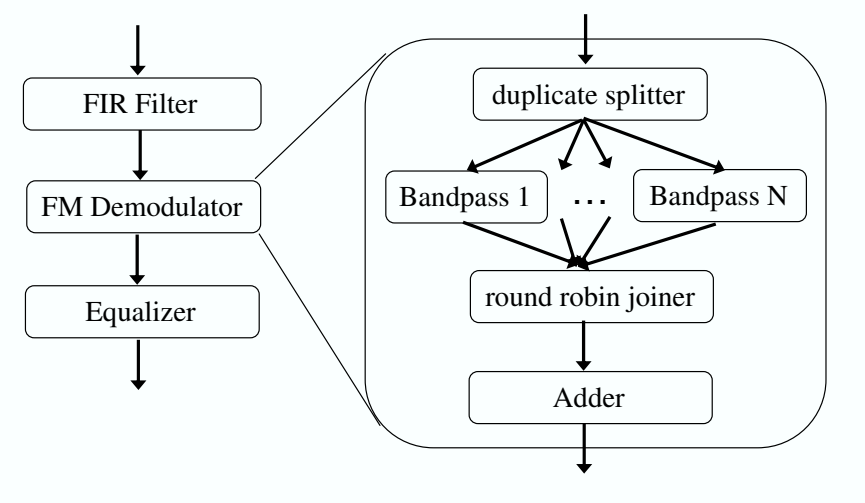

Figure 2.6: FM radio application block diagram in StreamIt

bandwidth of the network connecting all the processors augments the data load/store capacity of the processors through register to register communication. RAW is a very different machine than Imagine but they both combine high number of ALUs and high bandwidth networks to exploit properties of streams.

2.3.3 Other Stream Processors

Since 2001, graphics processors (GPUs) have started to expose a programming interface and there has been more and more efforts to use them for for general purpose computing. One of the first efforts, Cg[23] allowed programming kernels in a C-like language but had to contend with limitations of the architecture such as no branching and no scattered writes as the output of kernels. The Brook for GPU programming system[4] was geared towards full application development with more control between kernels and generated Cg for kernels. CUDA is the newest programming system[3] for Nvidia GPUs, it combines full application development with C kernels with an API to use specialized instructions. Scattered writes are permissible and it exposes parallel thread which can share a small amount of state and do branching with small performance loss as long as there are a certain number of threads taking a branch.

The Cell processor[27] first introduced as the processor of a game console, the Play Station 3, was also meant for general stream programming and many efforts have produced systems for it. Sequoia[11] is a system that allows for efficient partition of datasets and computation across multiple hierarchies of memory like the cell processor in a single or multiple core configuration.

2.4 Common Features in Stream Machines

Stream machines are characterized by high ALU density and high bandwidth to link their memory hierarchies but they do differ in their implementation details. An abstraction that would describe a stream machine in terms of it's essential stream components could be used to create a general framework to help develop the software tools needed to compile stream applications.

Stream processors are being designed and introduced at an increasing rate to improve application performance. Despite the commonalities in their designs, applications are not portable on the different architectures and they don't share a programming language or system. Currently every new architecture brings it's own language and compiler re-inventing the wheel every time for all the known stream optimizations.

What is needed is a unified stream programming model that can allow an application to be compiled onto different architectures and the sharing of a framework of compilers and languages. One approach to solving this problem is presented in the next chapter.

Chapter 3

Stream Virtual Machine

The lack of a common high-level stream language to program stream architecture is an impediment to shared research and growth of the stream programming paradigm. Compilers are notoriously hard to design and debug and take a long time to develop to a mature usable state. If every new stream architecture has to reinvent the entire compiler wheel with their programming system, improvements will be slow. What is needed is a common infrastructure that can enable the programming of different stream processors using a choice of different stream programming languages.

This infrastructure needs to be light weight in it's final implementation, with low performance overhead, so that the programmability will not kill performance. In order to allow improvements to the compiler and underlying architecture, the model needs to be adaptable and capture the most basic operations characteristic to computations with streams.

To address this issue, this chapter introduces the Stream Virtual Machine which is both an an abstraction model used to describe a stream processor in terms of it's stream components, and an Application Programming Interface (API) that describes how a stream computation can take place on these components. A general stream compiler can then take a stream program in a high level language and an abstract SVM model of a system and generate SVM API code for this machine. The stream processor then only needs to implement the SVM API as a set of library calls, and compile the SVM code as a regular compiler would do, without worrying about stream

transformations, since this code is essentially uniprocessor code, the parallelization and data blocking has already been handled by the stream compiler.

3.1 SVM Design

Our first task in creating the Stream Virtual Machine was to decide which level of abstraction to use for the Stream Virtual Machine. In the spectrum of abstraction levels in programming systems where algorithms are at the top and assembly is at the bottom, the SVM goal of being a performance target argues to situate it close to the hardware. Unfortunately, the need for portability on different processors argues in the opposite direction, to high levels of abstraction. To resolve this conflict we took advantage of the fact that all new machines generally have a native C compiler. In some sense because our machine targets are high performance processors, the lowest common denominator to them is the C programming language. We leverage the fact that C is only a little higher than assembly. By generating C code for different processors and having their C compiler generate the final assembly we hope the SVM would incur small performance overhead without sacrificing portability.

As a result, our infrastructure splits up the compilation process of a stream program into two parts as seen in Figure 3.1. The Stream Virtual Machine API is an intermediate version of the code, namely C with SVM API calls that is produced by a High Level Compiler (HLC). This compiler is a source to source compiler which takes a high-level stream language program and a SVM machine model description, and produces SVM API code specifically for the intended machine. The machine's Low Level Compiler (LLC) is a simple C compiler, and with it's own implementation of the SVM library finally produces the assembly code for the application.

The challenges of the SVM lie in defining an architectural and execution model of stream programs that captures the essential characteristics common to stream processors that are required to produce efficient code for them. The SVM API also has to be simple and lightweight so that it doesn't require much more work to implement the calls of the general SVM API than the processor's native calls. A model that accomplishes these goals is described in the next section.

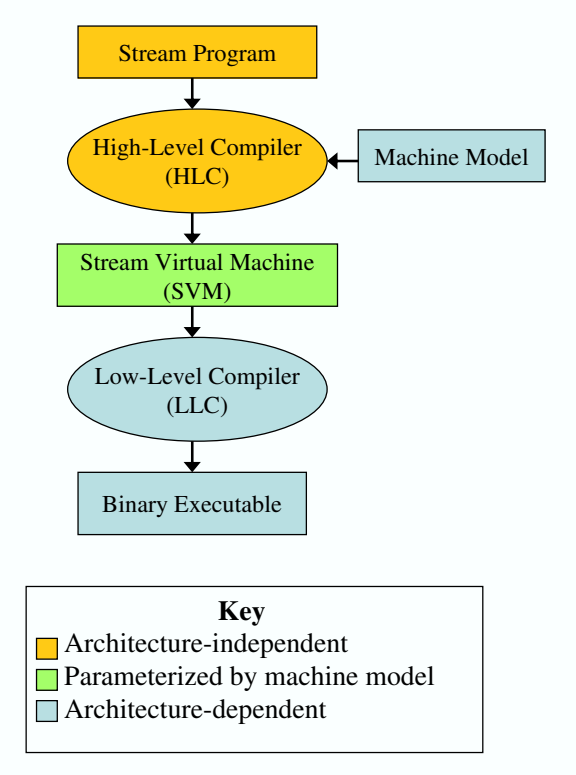

Figure 3.1: Compilation Process

3.2 SVM Architectural Model

The first issue is the minimum set of resources that the SVM should have. Most virtual machines have generally described a single processor with some resources such as registers, stack and memory. In our case the stream paradigm requires a concurrency of operations to enable the overlap of computation and memory operations. Thus the SVM will have more than one thread of execution that will interact. The two stream machines that were described in Section 2.3, were attached processors - they assumed some control process was launching commands for it to execute, and there were two types of execution that ran concurrently: computation and data movement. This leads to a model with three threads of control.

Figure 3.2 shows the components of the SVM architectural model. There is a control processor that initializes everything and will orchestrate the stream program

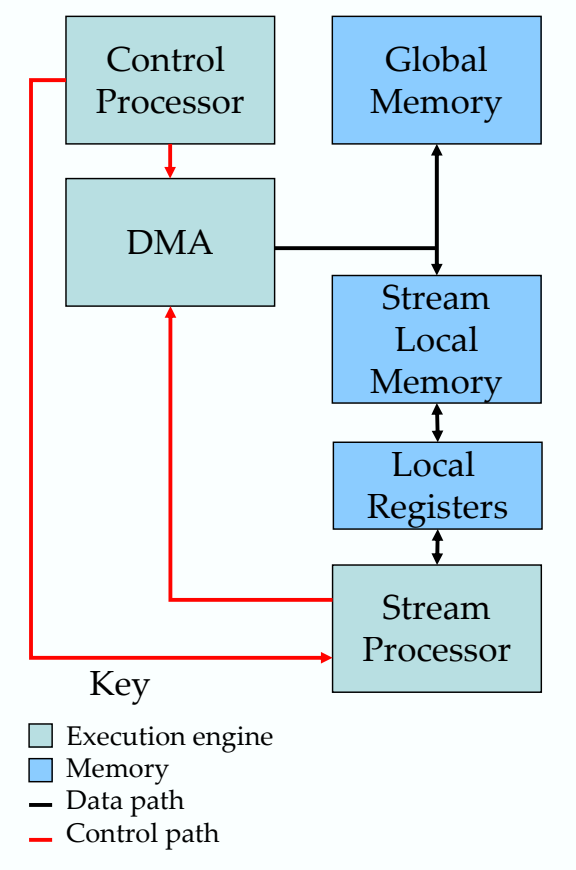

Figure 3.2: SVM Machine Model

execution. The control processor is often a conventional processor as it does have access to all of main memory and can initiate operations on the other two "processors", the stream and the Direct Memory Access (DMA) processors. The stream processor does all the data intensive computation and can also initiate operations on the DMA processor. The DMA engine provides memory transfers between main memory and the local stream memory.

By giving each of the processors its own thread of control, the SVM allows the control processor to run ahead of the actual data execution essentially prefetching future operations for the data and memory execution units. These computations

and memory transfers operations have some dependencies between them. Some dependencies are straightforward producer-consumer dependencies but others relate to resources constraints like the allocation of the local memories.

To deal with this autonomy, the SVM API allows the specification of the dependencies between computation kernels and DMA transfers, as they are known at the compile time of the stream program. The control processor can dispach operations to other processors with their dependencies information such that the execution engines are able to reorder computations for more efficient execution.

The stream local memory lies outside of the general memory map of the low-level C-compiler and is allocated by the stream compiler as stream input and output data for computation kernels on the stream processors.

3.2.1 SVM Parameters

For the SVM to work well as an architectural model, one must be able to abstract a wide class of stream machine into this model. A high-level stream compiler should not have to worry about specific details of an architecture but still be able to generate efficient code for this architecture.

The set of parameters to describe a SVM needs to be small in order to make things easier for a high-level stream compiler to reason about the system and make informed decisions. The high-level stream compiler's task is to parallelize the application and schedule the memory transfers so that the data is local when needed and the overall system bandwidth is minimized. The result of this optimization is a stream computation graph with all of it's intricate details of memory transfers and computations [1]. Blocking the data and allocating the stream memory requires knowledge of the memory sizes. Scheduling computation kernels and memory transfers requires approximates of the time they will take to complete. Thus the following parameters are used to characterize an SVM model:

Processors can be controlled by other processors (this is the case for stream processors and DMA engines). DMA engines can only run special kernels which will be described in the next section, otherwise processors can run user-defined code.

These processors capable of running user-defined code are then characterized by such factors as their register size, operating frequency, mix of functional units and SIMD level.

Memories 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.

Network Links 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.

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.

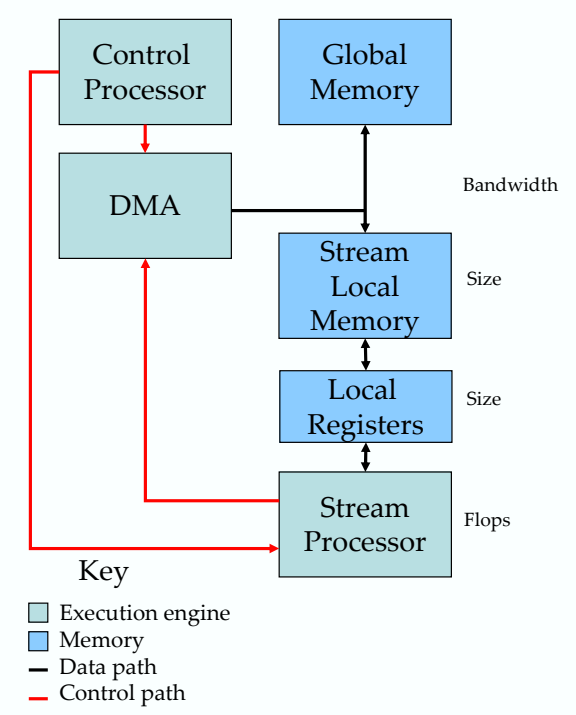

Figure 3.3: SVM Parameters

Figure 3.3 shows the SVM machine model with it's essential parameters. More stream processors with their memories and DMA processors can be added. In a stream machine with multiple stream processors, there can now be many different possibilities of how an application will be mapped on the machine. For example one kernel can be mapped in a data-parallel fashion across all the kernel processors or each processor can expect a different kernel to form a pipeline, or for some applications, a combination of the two approaches.

3.3 SVM API

In this section we will first introduce the general computation model used by the SVM. Then we will present the specific calls that enable the implementation of the execution model. Finally we will talk about the communication and synchronization model required by the SVM API.

3.3.1 SVM Execution Model

The computational model of the SVM is centered around the control processor. It initializes all the other processors and is the synchronization point for some critical sections in the program where all processors need to wait for everyone else to be done.

Looking at a graph of a stream computation with streams connecting kernels, both the transfers and computation kernels will be split across multiple stream processors. Furthermore at the level of a single stream processor and DMA engine, because the size of the stream data probably exceeds the small stream local memory, the transfers and computations will need to be blocked. In order to overlap the computation and data movements, the next data block will be fetched while the current one is being processed by the stream processor, effectively double-buffering the computation.

It is possible to have different kernels running on adjacent stream processors where the first one produces a stream to be consumed by the second one. Operations have to be blocked and double-buffered as well to fit in local stream memories and overlap each other.

3.3.2 API Details

The SVM API is used to specify how the computation and data are partitioned, moved and synchronized in the stream system. For example it specifies where computation is scheduled and to which specific stream processors is assigned. The data behind such computations is partitioned on the appropriate local memories. The API also specifies how the data is moved among local memories or between local memories and the global memory. Finally the API specifies the synchronization in between computations and data movements.

There are many usability objectives that the SVM API fulfills, the most important being that it supports efficient translation of API calls for various stream architectures. The use of standard C enables low level compilers(LLC) to parse, analyze and translate the API calls with minimum modifications. The C compiler requires only local analysis of single thread code by LLC to translate calls. The use of C also allows for the simple construction of a functional simulator through direct implementation of the calls for easy verification or performance estimation.

In essence, an application expressed for a SVM using C and the SVM API consists of a main() function for the control thread. This control thread can invoke special functions on specific stream processors called *Kernels*. Kernels operate on data located in the local memories of the stream processors as specified by *Blocks* and *Streams*. Pre-defined kernels executed by DMA engines provide data movement. Kernels synchronize with the control thread using special functions and with each other using explicitly specified dependencies.

The SVM API is intended to express a specific mapping of an application derived from portable code, not as a programming tool (though it could be used as such). It is akin to a high-level assembly language. Constructs in some stream programming languages resemble those of the SVM API because such languages aim for a programming model that resembles the execution model, but the constructs do not necessarily have a one-to-one correspondence with SVM API constructs. For instance, the conceptual **k**ernels in an application written with a stream programming language are usually not the same as the SVM API **K**ernels in a mapping of that application. The former are based on a logical decomposition, the later are based on a hardware-optimized

mapping.

An example of this would be in a filter application, a high-level programming language might have a FIR filter kernel followed by a decimation kernel to downsample filtered data to a lower rate. The high-level compiler will merge these two conceptual kernels into a single one to reduce the amount of computation and data movement. Finally the high-level compiler will split the merged kernel into many iterations to block the data transfers such that they fit the local memory. Each one of these iterations of the merged kernel will be an SVM **K**ernel call.

3.3.3 SVM API Constructs

This section introduces the SVM API constructs and explains how they meet the objectives outlined above. The SVM API uses strict C syntax, but follows an object-oriented paradigm that couples a struct type "class" with function "methods" that perform related operations. All "methods" take a pointer to a struct "object" as the first argument. Each "class" has an initialization "method" that serves as a constructor. The terms class, method, and object are used henceforth without qualification. The SVM specification document [25] contains a detailed description of all the calls.

Block and Stream: The SVM API uses *Block* and *Stream* objects to assign data to specific hardware locations in the stream processors local memories and to 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 and Streams are initialized with the following methods [1]:

```
void svm_blockInit(svm_Block* b,
    mm_Mem ramLocation, size_t address,
    size_t capacity, size_t elementSize);

void svm_streamInitRAM(svm_Stream* s,
    mm_Mem ramLocation, size_t address,
```

[1] Some SVM calls simplified/modified here and in example for explanatory purposes.

```
    size_t capacity, size_t elementSize);
```

Blocks implement random-access read and write methods; streams implement blocking peek, pop, and push methods. Both blocks and streams have layout constraints that enable meaningful aliasing of blocks and/or streams within memory (e.g., one block can refer to a region of another block).

Kernels: The SVM API uses Kernel objects to map functions, usually corresponding to some portion of a computation intensive loop, to stream processors. Each specific function is represent by a Kernel "subclass" that "inherits" from the Kernel class by enclosing the struct used for Kernel class inside its own struct and calling Kernel methods either directly or from within its own methods. A typical Kernel subclass is initialized using a method of the form:

```
void svm_kernelSubclassInit(
        kernelSubclass* k,
mm_Proc procLocation,
        kernel specific arguments);
```

DMA Kernels: The SVM API uses special pre-defined DMA kernel subclasses to describe DMA transfers. DMA kernels are executed by DMA engines rather than stream processors. DMA kernels include move (equivalent to memcpy), strided record copies (read a records, advance b records, repeat), and indexed scatter and gather (read records from within a block given a block or stream of indices). Each kind of DMA kernel has variations to handle block to block and, stream to stream, block to stream, and stream to block transfers. For example, a move DMA kernel for a stream to stream transfer is initialized using the following method:

```
void svm_moveS2SInit(svm_MoveS2S* k,
   mm_Proc dmaLocation, svm_Stream* srcStr,
   svm_Stream* destStr, size_t length);
```

Kernel dependencies: The SVM API uses explicit dependencies to provide flexible synchronization between kernels, including DMA kernels, independent of the

control thread. Prior to execution, a kernel may be marked as dependent on another kernel using the following method:

```
void svm kernelAddDependence(svm Kernel* k,
svm Kernel* dependsOnKernel);
```

Kernel control: The SVM API uses kernel methods to provide synchronization between kernels and the control thread. The simplest and most common form of synchronization is for the control thread to execute a kernel using the *svm_KernelRun* method and wait for it to finish executing using the *svm_kernelWait* method. *svm_kernelRun* does not immediately execute a kernel, it enqueues it for execution on a specific processor. When that processor finishes executing a kernel it selects an enqueued kernel as the next kernel to execute only if all kernels it depends on have finished executing.

More rarely, the control thread may asynchronously pause or end a kernel's execution using the *svm_kernelPause* or *svm_kernelEnd* methods. In some cases a kernel may pause itself using *svm_kernelPause*; *svm_kernelWait* allows a control thread to wait for a kernel to finish executing or pause itself. If a kernel is paused, the control thread can interact with it by altering the fields of the kernel object then calling *svm_kernelRun* again to resume execution.

3.3.4 SVM API Example

Consider mapping an application to a simple SVM as shown in Figure 3.4. The application filters then compresses the image by a constant ratio. The simple SVM consists of one master processor and two stream processors as shown in Figure 3.4. The application may be mapped in several ways, depending on capabilities of the hardware. For explanatory purposes, assume the two optimal Kernels consist of the filtering and compression stages. The Kernels may then be time-multiplexed such that each processor executes both kernels on half the data, or space-multiplexed such that one processor executes each kernel on all the data. Data transfers could use blocks or streams for DMA. Figure 3.5 illustrates these possibilities. The SVM API can be used to specify any of the four mappings shown, as well as others not shown.

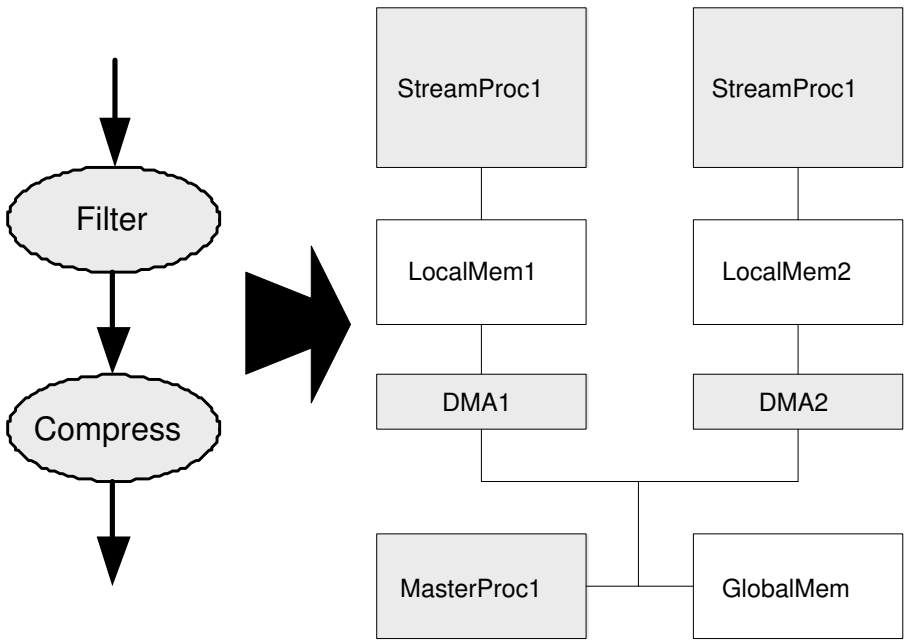

Figure 3.4: Simple example application and target SVM

3.3.5 SVM API Implementation

The SVM API efficiently maps applications to varied streaming architectures. Its constructs exploit the nature of streaming architectures at a level of abstraction that allows straightforward translation to a specific architecture while still leaving some room for architectural innovation.

The control thread/kernel division exploits the heterogeneity of stream architectures that contain simple RISC master processors and stream processors but as we will see later, also works for homogeneous processor systems. The SVM API is designed to capture computation intensive loops in Kernels and assign them to stream processors.

The SVM API is designed to enable a stream processor to execute kernels in rapid succession with minimum intervening overhead. Separating initialization of a kernel from execution enables the control thread to transmit the kernel arguments to

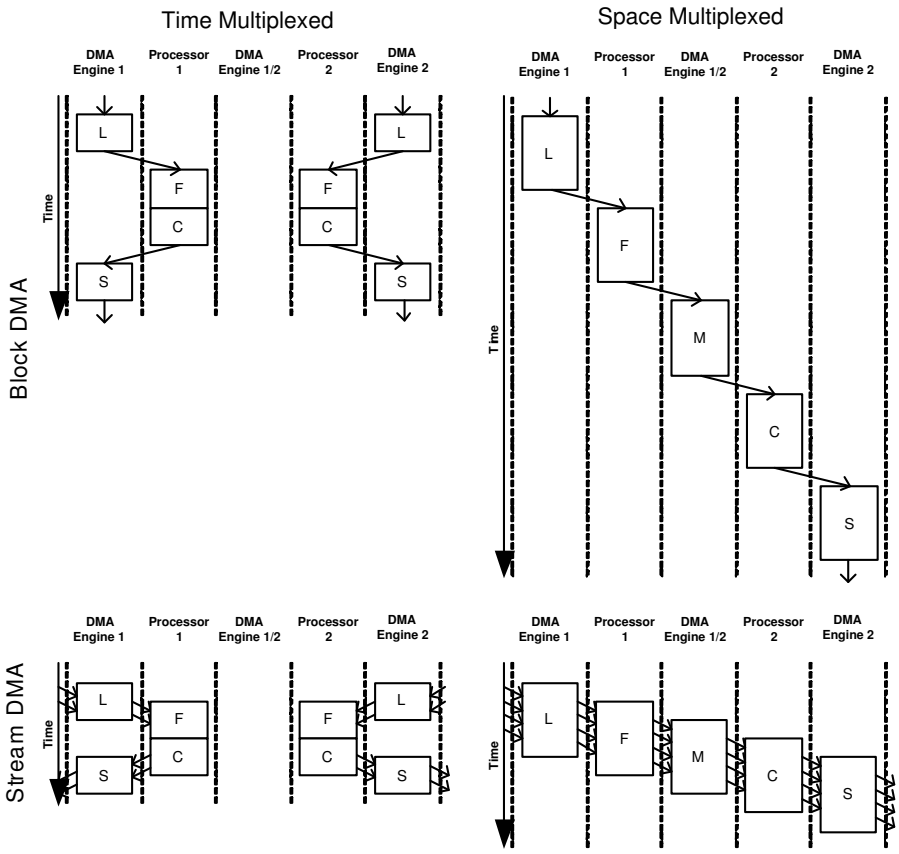

Figure 3.5: Possible mappings of example

a stream processor before the previous kernel finishes. Explicitly encoding the dependencies between kernels allows the stream processor and DMA engine to synchronize directly using an architecture-specific method without the master processor acting as "middle man". The DMA engine and stream processor may communicate through the local memory or use a hardware scoreboard (as in the Imagine architecture). Regardless of implementation, such direct communication allows a DMA load to be followed more immediately by a kernel execution, for instance.

The SVM was adopted as a standard within the Darpa Polymorphous Computing Architecture (PCA) project. This project included many funded computer architecture projects: Stanford Smart Memories [32], MIT Raw [33], UT Austin Trips [29]

and USC Monarch [9]. Given that these were all quite different parallel architectures, the SVM had to go through many iterations of revising details of the API call to satisfy all the architectures that they could implement the SVM efficiently. At the same time there was a lot of pressure to keep the SVM model and it's parameters and the API small and simple in order to make it easy to implement and to generate code for by a high-level stream compiler.

Before going through the process of compiling a stream program down to the SVM API and running it on a machine, a lot of questions still remain about the design of the SVM. One of the first question was wether the SVM machine model with it's simple parameters can allow good estimates of runtime for the high-level compiler to make good decisions, which we will address in the next chapter by looking at how well the SVM models real Graphics Processor Units (GPUs).

Chapter 4

SVM study on Graphics Processors

In the two-level compilation model, the HLC generates SVM code without knowing the detailed features of the specific stream processor targeted. To produce high performance code, the HLC uses the values of the SVM parameters that describe the topology and performance features of the targeted processor at the abstract level of the SVM (see Section 3.2).

SVM Parameter	Imagine	Merrimac
Local Memory Capacity	256 kB	1 MB
Global to Local Memory Bandwidth	2.3 GB/s	38 GB/s
Local to Register Memory Bandwidth	19.2 GB/s	256 GB/s
Register File Bandwidth	326.4 GB/s	1.5 TB/s
Peak GFLOPS	24	128
SIMD Degree	8	16

Table 4.1: SVM parameters for two stream processors

For stream processors that implement an execution model similar to that of the SVM, the values for the SVM parameters are fairly obvious by looking at their block diagram. Table 4.1 presents the SVM parameters for the Imagine [16] and Merrimac [6] stream processors. Nevertheless, for processors that use alternate organizations, the SVM parameters that characterize their behavior with stream applications are less apparent and can require an experimental approach. In addition we are interested in whether the SVM model will have sufficient fidelity to be a good representation for

the overall system. To test out the SVM in this situation, the next section explores modeling graphics processors (GPUs) as a SVM.

4.1 GPUs for Streaming Computations

GPUs are custom processors for high-bandwidth 3D graphics for personal computers. Their basic task is to transform a set of triangles that describe a scene into a rasterized image under the control of an API like OpenGL or Direct3D. During the past decade, the computation capabilities of GPUs has exploded to hundreds of GFLOPS as the demand for more realistic 3-D images expanded. Modern GPUs are also equipped with a large, high-bandwidth frame buffer (hundreds of MBytes) and a dedicated, high-speed interface to the main memory controller of the computer (GBytes/sec).

Harnessing the computational power of GPUs for stream applications is enabled by the fact that GPUs are becoming more programmable. Previous generation GPUs allowed application programmers to write code for two parts of the graphics pipeline, the vertex engine that typically operates on geometric vertices [20] and the fragment engine that typically performs shading and blending operations on the output pixels. Several researchers have demonstrated impressive application performance by programming fragment engine in assembly or low level languages like Cg [23]. Current GPUs have unified their shading architectures so that the same hardware is used both for vertex and fragments[22]. New programming systems like CUDA make it easy to write parallel code to run on the GPU and the control from the host CPU [3] [26].

The basic components of a personal computer with a programmable GPU fit within the SVM architecture model as shown in Figure 4.1. The main CPU is the SVM master processor, the memory controller is the SVM DMA engine, the main memory is the SVM global memory, the frame buffer as the SVM local memory, and the fragment engine with its registers as the SVM stream processor with its local register file. Hence, it is possible to target GPUs as a stream processor with the two-level compilation model. Using a compiler like the one for Cg as the LLC, the two-level compilation model has the potential to allow general, high level, stream applications to be easily targeted to GPU hardware.

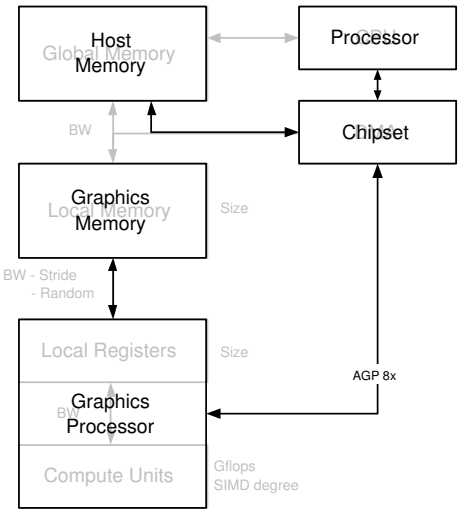

Figure 4.1: Graphics Processors mapped to SVM

To explore the feasibility of this approach, the next section will look at how we can extract SVM performance parameters from simple experiments and then validate if the SVM model can accurately predict performance from the simple parameters, versus the actual run-time of small applications.

4.1.1 Characterization Methodology

GPU	Tech (um)	Trans (M)	Mem Bw (GB/s)	Clock (MHz)	Texture Mem MB
Nvidia GeForceFX 5900 Ultra	0.13	130	27.2	450	256
ATI Radeon 9800 Pro	0.15	107	21.8	380	256

Table 4.2: GPU published specs

In this study, we investigate the SVM model parameters for the best graphics processors available in 2003, the ATI Radeon 9800 Pro and the Nvidia GeForceFX 5900 Ultra. The public characteristics of the two GPUs are listed in Table 4.2.

Certain SVM parameters such as the frame buffer capacity are easy to obtain from datasheets. On the other hand, sustained GFLOPS and bandwidth parameters are difficult to calculate from advertised peak rates due to the irregular nature of the GPU organization. Fragment engines use packed (SIMD) arithmetic to achieve high computational throughput. However, that generation of GPU's instruction set did not include any control flow instructions such as branches[1], which complicates computations with conditional statements. Memory accesses to program data must use texture load and store instructions into the two dimensional frame buffer. Furthermore, load accesses are first filtered by a cache optimized for the spacial locality of texture accesses where an individual access interpolates a texture sample by looking into neighboring values in a two dimensional space. Finally, fragment programs undergo recompilation when loaded on the GPU. The vendor-provided loaders perform dynamic reallocation of register and expand or transform assembly instructions into native operations of the graphics engine.[2]

To accurately estimate the SVM model parameters for the two GPUs, we use a series of micro-benchmarks. Each micro-benchmark targets a specific performance feature of the GPU, such as its sustained performance in the presence of conditionals or the sustained bandwidth from the local memory to the local register file. We wrote the benchmarks in OpenGL ARB fragment program assembly [2] or compiled them from Cg. Special care was necessary to ensure that the loader does not optimize away the resource constraint we are trying to measure in each case. The test platforms were both high end workstations, containing a 3GHz Pentium 4 with a 800MHz front-side bus, 2GB of 400MHz DDR DRAM and an AGP 8x bus, running Windows XP and latest release drivers from the graphics card vendors.

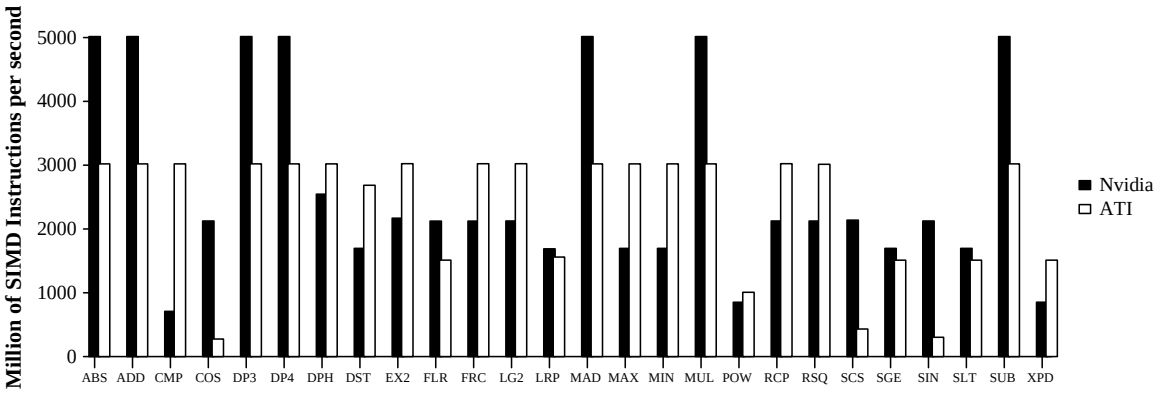

Figure 4.2: Instructions per second

4.1.2 Micro-Kernel Analysis

Compute Unit

Figure 4.2 shows the number of SIMD instructions per second that can be executed for each of different compute operations[3]. Many of these instructions operate on all four components of the data. This measurement was made using only one live register, because we will see later that the number of live registers can have an impact on the peak instruction throughput.

The ATI hardware is consistent at 3G instructions per second (12 Gflops) for most operations, with only a few exceptions. These exceptions are assumed to be caused by functions that are implemented in multiple instructions (up to 10 for trigonometric functions). The 3G instruction rate matches the published specs for the ATI part: 380MHz with eight fragment pipelines gives theoretical maximum performance of 3.04 G Instructions/s.

The Nvidia hardware has more variation in performance. It implements most combinations of multiplies and additions at 5G Inst/s (20 Gflops) while all other instructions are performed at less than 2.5G Inst/s. Given the GeForceFX clock

[1]The opposite constraint is true for the vertex engine.
[2]More recent GPUs have relaxed these constraints in a unified shader model
[3]Instruction description available in [2]

is supposed to be at 450MHz with four fragment pipelines, this suggests that each pipeline has three Multiply-Add units for a theoretical throughput of 5.4G Inst. All other instructions seem to benefit from only one functional unit per pipeline, also trigonometric functions benefit from some acceleration that bring their throughput up to 2G Inst/s.

Reductions

Because there can be no shared variables within a kernel between elements, there is no way to program the GPU to do reductions in a single pass. So reductions have to be implemented as multiple passes where we read in a number of elements and reduce these elements together to produce a smaller stream, and so forth till we produce a single value. While the multiple passes do add some overhead for small streams, for large streams the time is dominated by the first round of reduction, and from our experiments a reduction rate of 0.3 G Inst/s is seen, about a tenth of the potential of the machines.

Conditionals

In streaming applications, conditionals occur when the amount of work differs on a per element basis in the stream. Because stream architectures are data parallel, there are two ways to deal with this: predication and conditional streams. Short branches are good candidates for predication because the overhead of predicated instructions is low compared to the one of splitting a kernel.

Conditional streams are more useful for long branches where operations on the different element are substantially different. In this case, the elements of a streams are classified into two separate streams depending on whether or not they are taking the branch. Then two different kernels are applied to the two streams [17]. This method suffers from the overhead of the split into two kernels and recombination into a single stream at the end (if the join is necessary), but it is faster if a large amount of work must be done for a small number of elements.

On graphics processors, predication can be implemented by using a KILL instruction which stops any processing for the stream element depending on the test value. Like many predication schemes, killing a fragment does not reduce the run-time of the fragment program, since all operations are still performed, just the results are not used.

The ATI hardware presents a way to implement conditional streams, since it does a visibility check (Z-buffer) before operating on a fragment. One could then have the sorting kernel set this value and have the hardware cull the fragment automatically, assuming the Z-buffer check was very fast. Unfortunately the interaction between the rasterizer, which produces fragments in blocks, the visibility test and the fragment program pipeline makes this scheme less useful for general computation. A simple experiment shows the problem. First create a simple checkerboard pattern of visibility over a area. When the checker-board squares are less than 4 on a side, the run-time of the program is the same as if all fragments were visible. Since in graphics visibility is often correlated, the hardware checks to see if any of the data in the 4x4 block being scheduled needs work. If it does, the entire block is scheduled.

Figure 4.3 shows a graph of both hardware conditional methods for a long branch (60 Instructions). The graph plots the rate of useful SIMD instructions vs the proportion of a branch taken. If the machine handled conditionals perfectly it would be constant at 3G Inst/s. For conditional streams, the best case is where the fragments taking the branch are adjacent, as for the random distribution, fragments taking the branch were randomly distributed. The additional cost of a pass is about 17ms, the difference between conditional stream and predication when the branch is always taken.

There is little advantage for hardware conditional streams unless the branch is expected to be taken less than 10% or the application has a strong correlation between taking the branch and their location (like visibility in graphics).

Local Register File

In both graphics processors the architectural size of the register file is 32 registers (of four floats each). This limitation is enforced by the graphics card driver. We will now

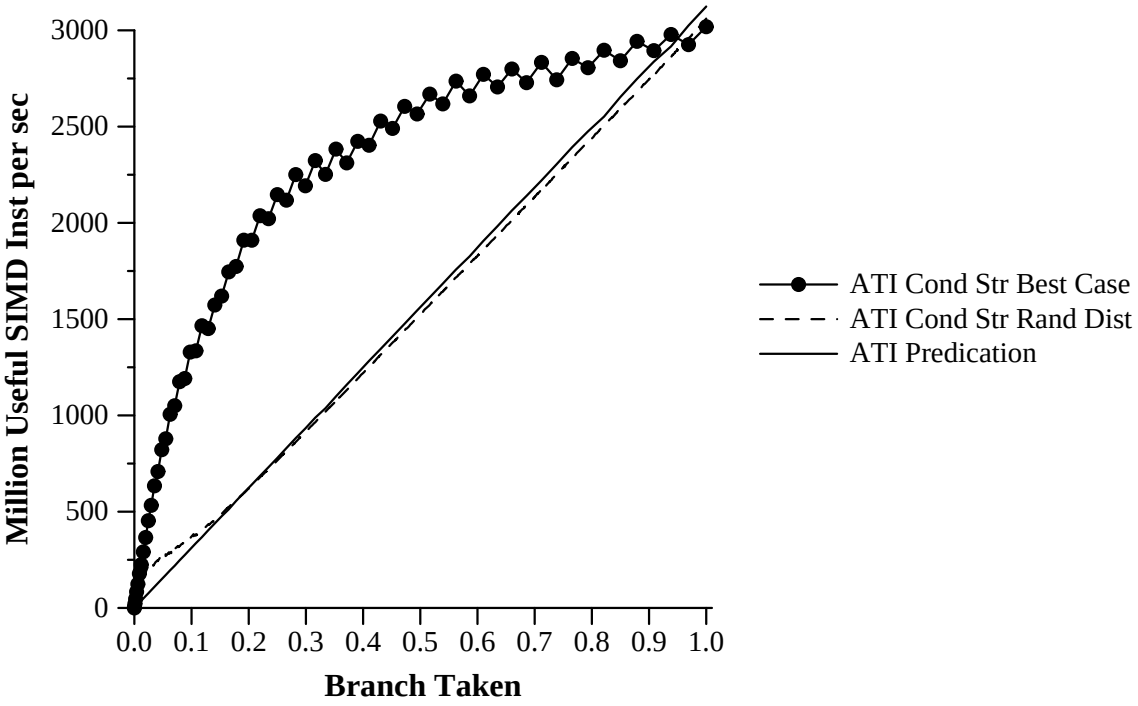

Figure 4.3: Predication vs Conditional

look at how the bandwidth between local registers and the compute units affects the machine performance.

In a machine with a fully connected register file, we would expect the instructions per second to remain constant independently of the number of live register. Figure 4.4 shows that ATI's register organization matches our expectation with performance unchanged as the number of live registers is increased (only MUL is shown for simplicity) The data for Nvidia clearly indicates a hierarchical register organization, since performance drops dramatically as the number of live registers increases.

On Nvidia, most instructions' performance falls off sharply once more than three live registers are used, just like MUL instruction in Figure 4.4. Dot product (DP3) instructions which take two vectors and produce a scalar value degrade in performance much slower than MUL instruction. The less dramatic cutoff in performance for dot products leads us to believe that this performance limitation is a register write bandwidth issue. For the ATI chip, the register bandwidth is roughly 179 GB/s.

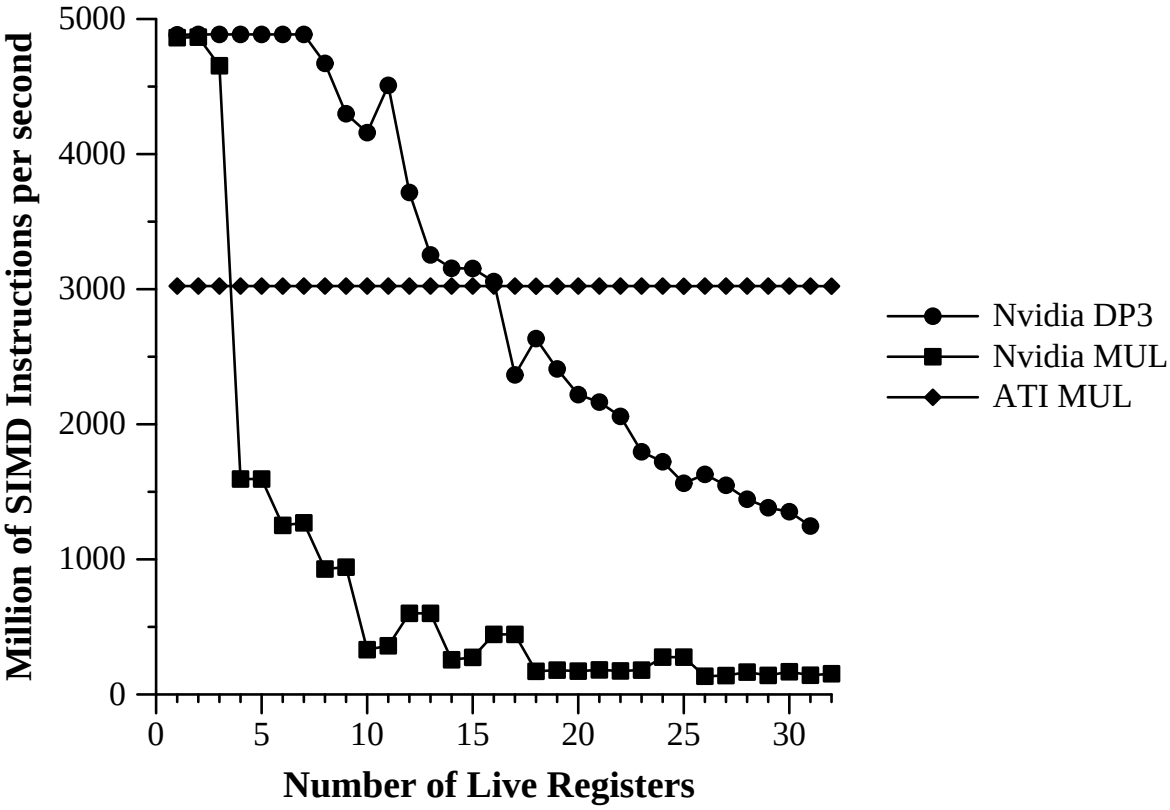

Figure 4.4: Impact of Live Registers on FLOPS

Nvidia's local register bandwidth is a function of the number of live registers

Local Memory

The local stream memory on the GPU is the graphics memory which is nominally 256MB on our graphics card. Some of this space is used by the frame buffers, and vertex data. The actual size of streams that can be present is 176MB out of 256MB for both architectures. Individual streams (textures) can be at most 4k by 4k (of 4 floats) on Nvidia and 2k by 2k on ATI. Textures in one dimension are also limited to 2k for ATI and 4k for Nvidia, making them useful only for small streams.

GPU's memory systems have a cache to capture spacial locality in texture accesses where an individual access interpolates a texture sample by looking at neighboring

values. GPUs use the texture cache as a bandwidth amplifier like many DLP processors. Unfortunately, if we truly are streaming data out of the local memory, and only reading it once, this cache will not help our memory performance.

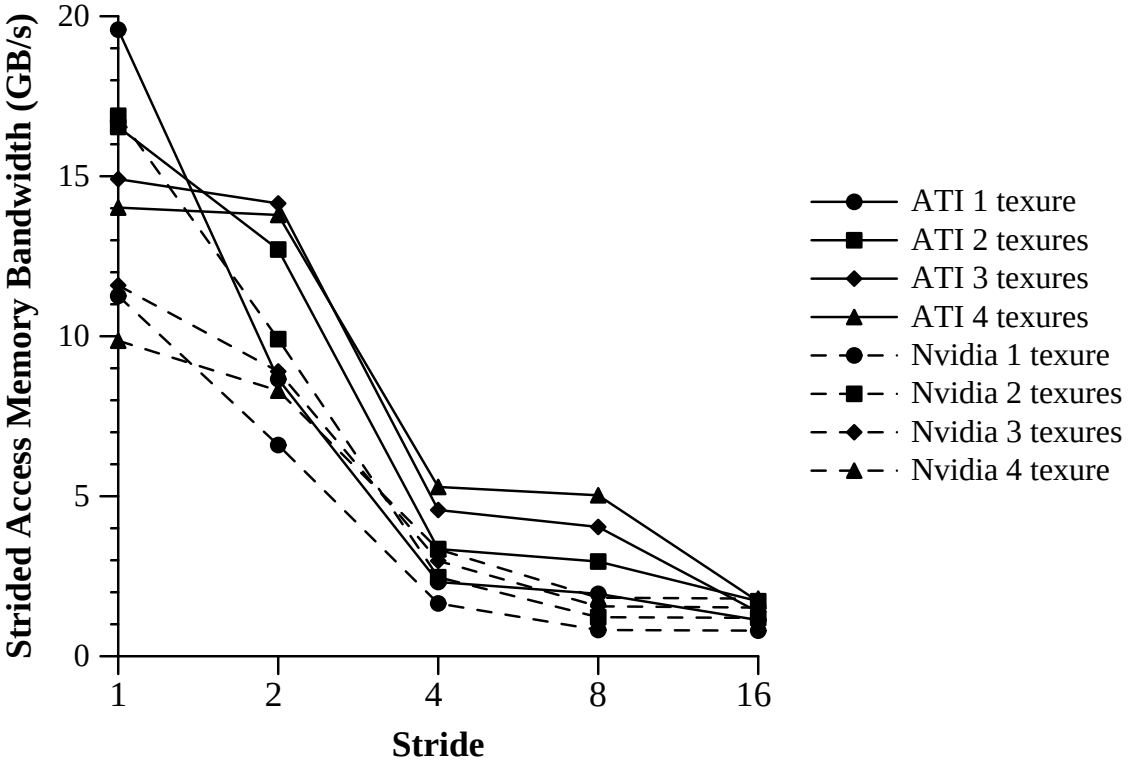

Figure 4.5: Strided Memory Bandwidth

Figure 4.5 shows the effective memory bandwidth when varying the access stride and the number of textures accessed in a kernel (all accessed with the same stride). ATI's memory access seem optimized for accessing a single texture at unit stride while Nvidia is optimized for two textures at unit stride. ATI with 19.6GB/s for single texture, unit stride reaches close to its theoretical bandwidth of 21.8 GB/s. Nvidia reaches its maximal bandwidth with two textures at unit stride, 16.9 GB/s which is farther from it's own theoretical bandwidth of 27.2 GB/s.

The effective bandwidth for both architectures drops significantly at a stride of 4, ATI maintains the same bandwidth for a stride of 8, while Nvidia drops further and

remains stable at a stride of 16. We guess that a cache line for ATI is a square of size 4 by 4 (vectors of four floats) while it is a 8 by 8 square for Nvidia. If ATI prefetches more than two cache lines on a miss, then the bandwidth would fall as it does to a half from a stride of 8 to a stride of 16.

Unit stride is the default access mode for most stream computation where multiple streams can be accessed at the same time as input. Assuming kernels with two input streams the effective unit stride bandwidth for both architectures is 16 GB/s.

Another important bandwidth parameter is when the local memory is being accessed randomly with each kernel loop generating an index. Because we have multiple parallel instances generating a random address, there is possibility of both reuse in the cache, as well as bank conflicts in the memory system.

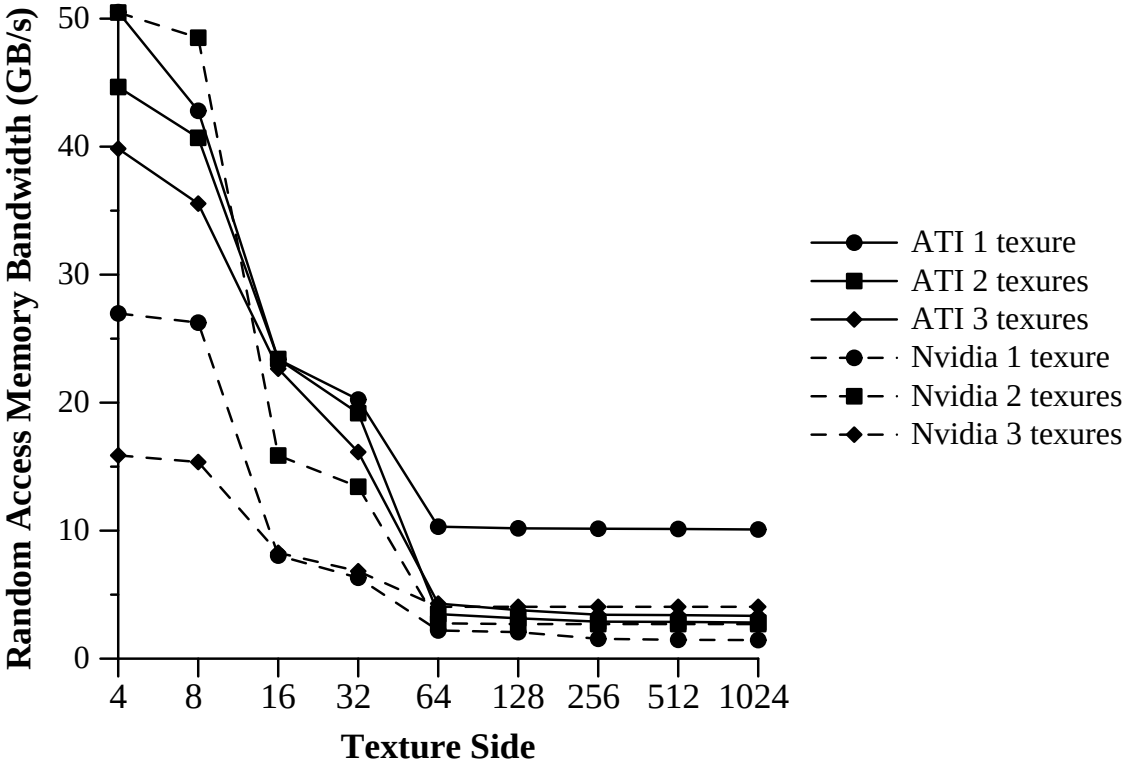

Figure 4.6: Random Memory Bandwidth

Figure 4.6 shows the random access memory bandwidth. The experiment is set up to read a texture normally (single stride) and use the texture data as an address

to sample possibly multiple other textures. The data in the starting texture was initialized to random values. The size of the texture being randomly accessed is shown on the X axis, and is the size of one side of a square texture.

Both ATI and Nvidia hardware behave as expected increasing the effective memory bandwidth when the accessed texture is small. ATI and its cache amplifies its bandwidth to better than the external bandwidth (21.8GB/s) with textures of lower than 8 by 8 (1kB). The drop in bandwidth from 8 on a side to 16 leads us to believe that the texture cache size is between 1kB and 4kB. More than 1 texture get into each other's way and lower the effective bandwidth. Maybe the most surprising part of the graph is that the ATI chip achieves half the theoretical bandwidth on random accesses to a large texture, where the cache is not providing much benefit.

The Nvidia hardware is again optimized for two textures, and achieves more than 3x bandwidth gain for small textures. The same drop-off in effective bandwidth from a texture side of 8 to 16 suggests that Nvidia texture cache size is also between 1kB and 4kB.

This data indicates that these machine can achieve close to their peak memory performance if the accesses are customized for each machine, and that random accesses to small data structures will be quite effective. The latter is important if we want to implement small lookup tables for use in some of our kernels.

Global Memory

In the worst-case, the Global Memory will be placed in the processor's memory and be visible to both the host CPU and the stream co-processor. It would store streams that either don't fit the local memory and/or need to be manipulated by the host processor.

Today the host memory is connected to a memory controller chipset which has private links to both the processor and the graphic processor. The current generation graphics link is called AGP 8x which has a peak bandwidth of 2GB/s.

On GPUs, most transfers between the host memory and the graphics memory are to transfer textures to the graphics processor, and not to transfer data from the GPU back to host memory. In addition, sometimes the textures for an application can

exceed the storage available in the graphics memory. As a result, the graphics driver usually makes itself a copy of the texture in the host's memory in case that texture is evicted from the graphics memory and needs to be re-transfered later. Since we need to send all data to the graphics processor as textures, this copying by the driver will slow down the effective transfer rate.

Architecture	Global Mem to Local Mem	Local Mem to Global Mem
Nvidia	350 MB/s	181 MB/s
ATI	920 MB/s	125 MB/s

Table 4.3: Global to Local Memory Bandwidths

Table 4.3 shows the bandwidth for data transfer between main memory and the graphic memory in both directions. ATI does better in global to local memory while Nvidia does better in local to global memory. Overall, local to global memory bandwidth is much lower than global to local memory bandwidth, although we cannot see any reason why other than the drivers are not optimized for it as it is not in the critical path of graphics applications.

4.1.3 Analysis

SVM Parameter	ATI	Nvidia
Local Memory Capacity	176 MB	176 MB
Global to Local Memory Bandwidth	0.92 GB/s	0.35 GB/s
Local to Global Memory Bandwidth	0.13 GB/s	0.18 GB/s
Local to Register Memory Bandwidth	16 GB/s	16 GB/s
Register File Bandwidth	179 GB/s	F(#reg)
Peak GFLOPS	12 Gflops	F(#reg)
SIMD Degree	4x4	?

Table 4.4: SVM parameters for two stream processors

Our use of micro-kernels has enabled us to extract key machine model parameters for the use of GPUs as stream processors in table 4.4. The complex nature of GPUs

requires us to consider expanding our performance parameters to better capture the capabilities of the machine in terms of types of memory access and dependence on such things as number of live registers. Given these complexities, we were encouraged the SVM model still performed well as the next section will show

4.2 SVM Validation

The first optimizing high level compiler is presently in development (Reservoir Labs' R-Stream compiler), but not yet available. Hence, we compare hand-written SVM code running on a simulator that estimates run-times based on machine model parameters to low-level code running on respective architectures.

We are leveraging work done on porting stream application to GPUs from [4] where part of the Brook streaming language was compiled down to GPUs. This system was used to generate low-level GPU code and to serve as framework to hand-write the SVM code (kernels and control code).

The SVM simulator is an implementation of the SVM API that runs SVM code correctly and estimates the run-time of an application. Run-time estimates are based on the bandwidth requirements of the different levels of memory hierarchy and the computation run-times of kernels. Sometimes, computation and DMA transfers overlap up to a synchronization point, like when a kernel A is running while kernel B's input data is loaded, kernel B has to wait for A to complete and it's data to be loaded. The SVM simulator takes into account the greater run-time of the dependencies.

The SVM simulator evaluates kernel run-times using a linear model, each kernel having a startup and tear-down cost independent of the number of stream elements to be consumed, and a incremental cost with the number of stream elements to be consumed. So in addition to the SVM API code, the SVM simulator requires the linear cost function of each kernel. The kernel schedule from a low-level compiler would be ideal, but for the GPUs they were estimated looking at the fragment program instructions requirements in terms of memory, local registers and arithmetic instructions.

The three architectures we will be looking at are Imagine [16], a dedicated stream

architecture and the GPUs from both ATI and Nvidia. The SVM code for both GPUs differ only in their kernel costs functions and their bandwidth for different types of accesses. The SVM code for Imagine differs from the GPU SVM code in that it has a smaller local stream memory which forces some applications to be further strip-mined when they do not completely fit the local stream memory. Also Imagine has some support for reductions which has to be implemented as multiple passes on the GPU. The Imagine hardware evaluation was done using its native programming system of StreamC, KernelC [16] run on the cycle accurate simulator.

4.2.1 Validation Results

Three kernels were chosen and evaluated for different input data sets sizes to compare how the SVM simulator, calibrated with the machine model parameters extracted through micro-kernels, compares to the actual run-times: Image Segmentation, 2D FFT and Matrix Vector-Multiply.

Matrix Vector Multiply contains a reduction in one dimension which on the GPU have to be implemented in multiple passes which favor dimensions which are a multiple of 4, the reduction factor. Figure 4.7 shows the comparisons of run times for the architectures and their SVMs. The ATI hardware incurs a greater cost when executing reductions by redirecting an output stream as the input stream of the next kernel. This is reflected in the high cost even for low dimensions. It is also very sensitive to the number of reduction passes necessary visible in the sawtooth behavior.

This kernel is bandwidth limited for all architectures, with a high initial cost for reductions on the ATI hardware. The SVM simulator tracks fairly well the performance behavior although it does not capture the sensitivity to the dimensions on the ATI hardware.

FFT is an example of a kernel which shows almost identical performance behavior on all 3 architectures as shown in Figure 4.8. At small dimensions, the runtime is dominated by a one time startup cost that we didn't model very well from our experiments, specially for the Nvidia part. Although the SVM simulator is not very accurate for the small dimensions, it tracks well the performance of larger data-sets.

Figure 4.9 compares the run-time of the image segmentation kernel on the actual GPUs and their estimated run-time by the SVM simulator. Unfortunately a native Imagine implementation was not completed in time. This kernel is compute limited and scales linearly with the number of pixels to be processed. On both GPUs the kernel costs functions based on the arithmetic ops, local register used and stream memory access pattern are quite close to the actual ones.

We have learned from this experiment that it is possible to model and predict the run-time of stream kernels using simple performance parameters. This first generation of programmable GPUs has required a few of these parameters to include other factors such as memory access pattern for memory bandwidth and register usage for computation rate on the Nvidia one. The memory bandwidth requirement still hold for current GPUs because of the nature of their memory system that is optimized for the common case of accessing big continuous blocks of memory, but performance degradation due to register size has been solved. Being able to predict the run-time of kernels will allow a stream compiler to make better decisions on dividing computation and communication.

4.3 Conclusion

Although from 2003 to 2007, graphics processors have evolved quite a bit and have become even more friendly to general computation [10][22], they remain specialized processors with impressive performance mostly for their graphics purpose. The SVM has proved to be an accurate model in predicting kernel performance from performance parameters extracted using simple micro-kernel experiments. With these parameters, it means that a high-level stream compiler can make good decision about blocking partitioning and splitting data and kernels.

Now that we've established the SVM as a simple parametrized model that provides enough information to estimate kernel run-times, we need to talk about implementation of the SVM API. In the next chapter we present the implementation of the SVM on a multiprocessor system, Smart Memories.

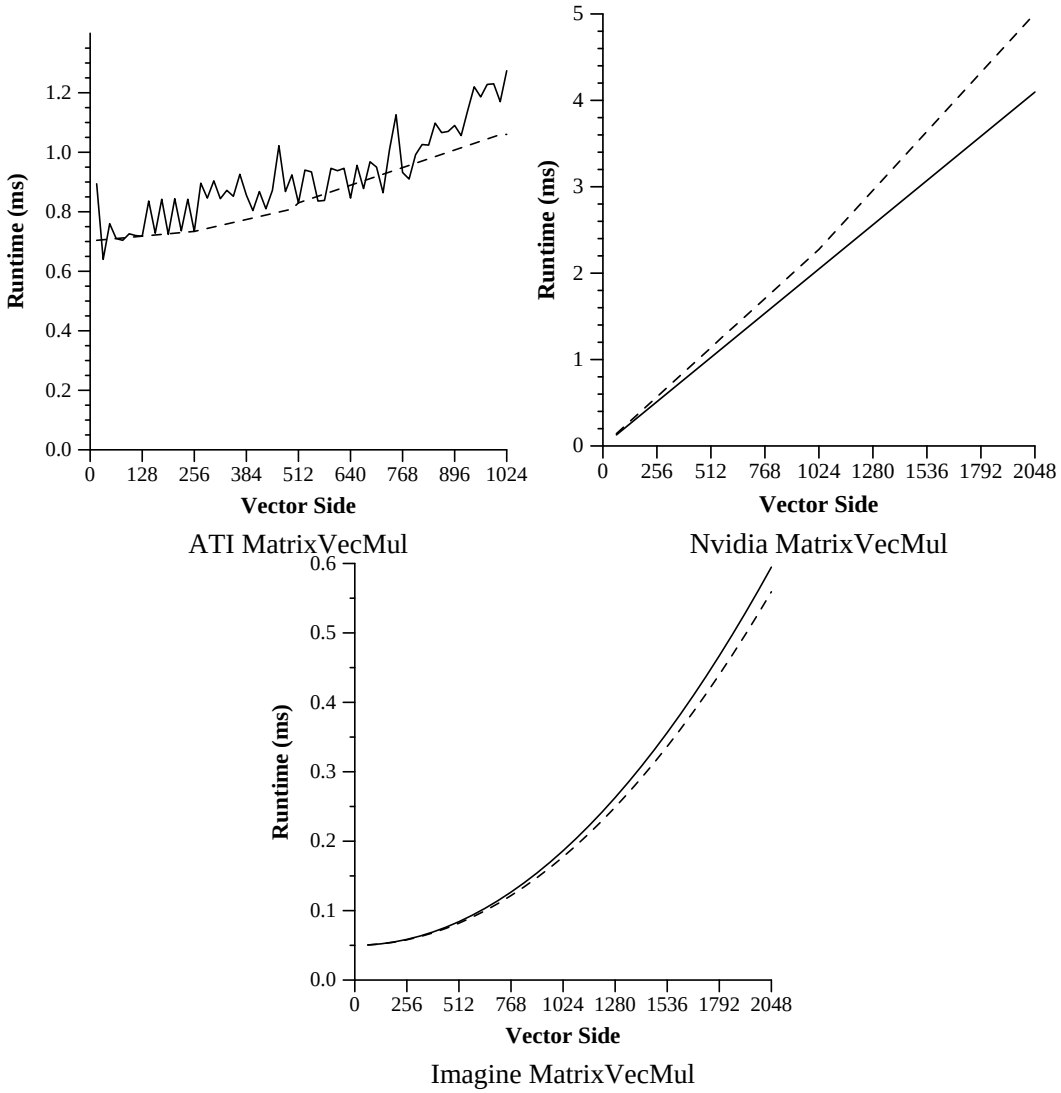

Figure 4.7: Matrix Vector Multiply run-times for different architectures (solid) vs their SVM (dashed)

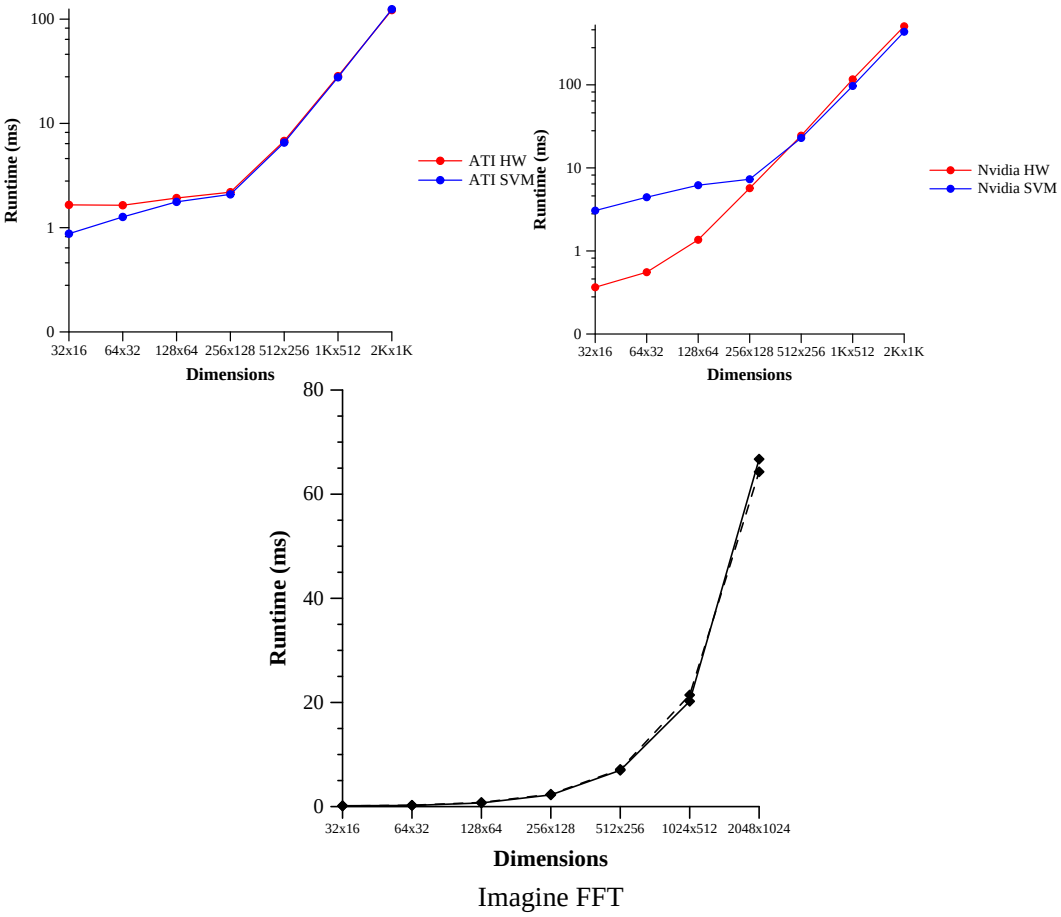

Figure 4.8: 2D FFT run-times for different architectures (solid) vs their SVM (dashed)

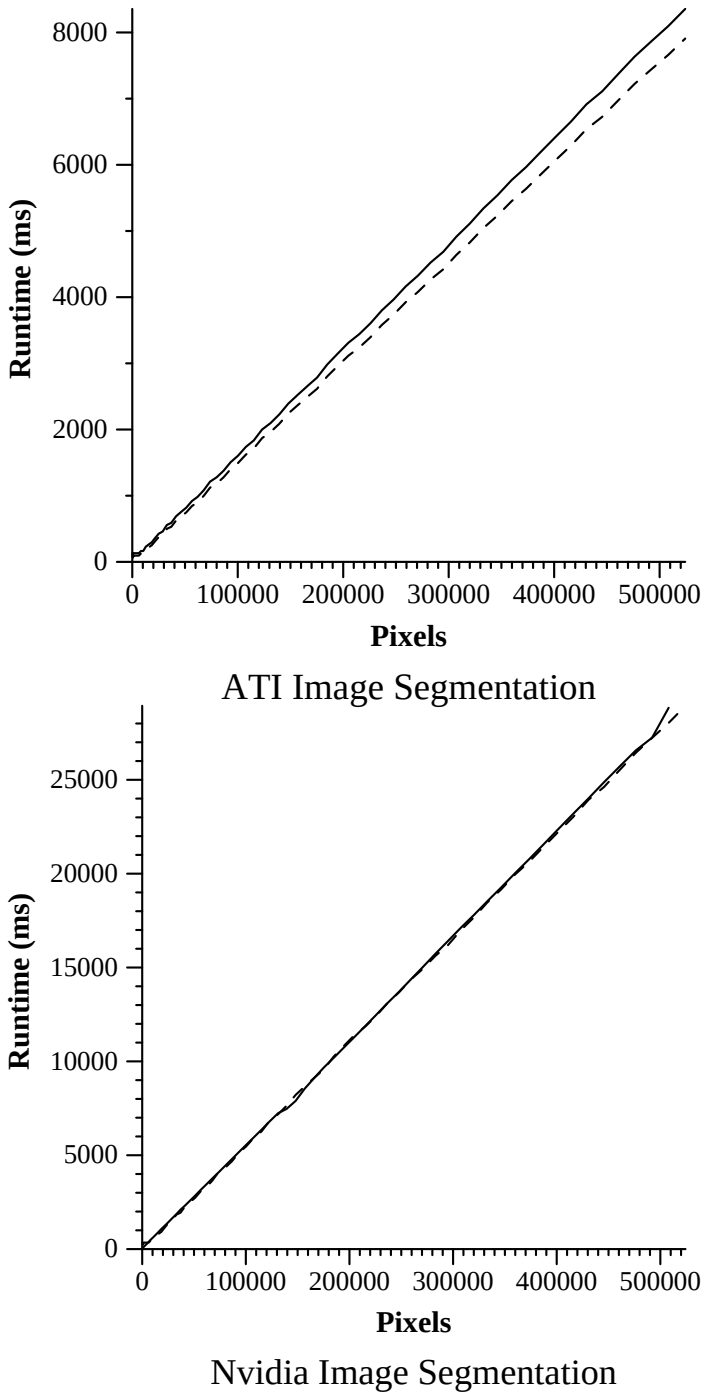

Figure 4.9: Image Segmentation run-times for different architectures (solid) vs their SVM (dashed)

Chapter 5

Streaming on Smart Memories

This chapter explores implementing a streaming application on a homogeneous chip multiprocessor, exploring both what hardware is needed to efficiently implement the SVM API, and what advantages are gained by customizing the memory system for streaming applications. To enable us to be more concrete in this discussion, we will use the Smart Memory (SM) multiprocessor as the evaluation platform. It is a system with a very flexible memory system which allows us to create new memory configurations easily. The first section of this chapter will give an overview of the Smart Memory architecture, focusing on the flexible memory system and DMA engine, since we will use both in the SVM implementation.

With this background on Smart Memory, Section 5.2 describes some of the configurations that were considered for stream execution. This section describes both different hardware configurations and different ways of slicing the computation over the processors. It will also show how certain resource ratios affect the optimal method of slicing an application over the processors.

Section 5.3 then goes into detail on how the API calls were implemented focusing mostly on synchronization and how the SVM execution model was used. This implementation provides estimates for the costs/overheads of these operations. These numbers are then used in Section 5.4 to evaluate the overhead of the SVM on this machine, and the benefits of customizing the memory system for streaming. Maybe not surprising, but most of the benefits come from the application partitioning and

prefetching the data, so a cache coherent chip multiprocessor running a streaming application has only a small overhead to our customized SVM implementation.

5.1 Smart Memories

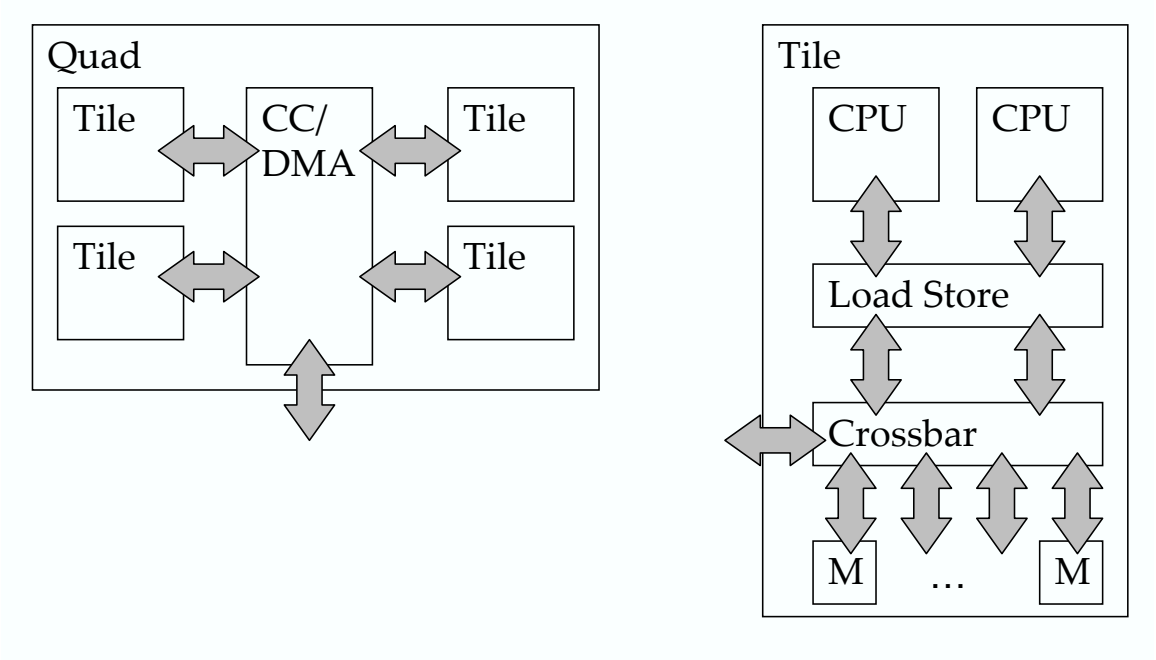

Figure 5.1: Smart Memories Quad and Tile

Smart Memories is a modular architecture that keeps local communication paths short. The smallest unit is the tile which contains two processors and some configurable memory mats interconnected by a crossbar. The number of processors and memory mats in a tile is configurable but we'll concentrate here on the first implementation of Smart Memories which has this specific configuration: two processors and sixteen memory mats.

The processors of Smart Memories are Tensilica LX synthesizeable cores, configurable processors that allow for customized instructions to be added[12]. Up to three instructions can be issued at a time with VLIW-like instructions when possible, while

not giving up the short instructions encoding when it is not[8]. Given the modular architecture of Smart Memory, the processors could even be easily replaced by a future generation Tensilica processor as long as the interfaces remained the same. In fact during the lifetime of the Smart Memories project, a single issue processor was first used and then upgraded to the VLIW capable processor.

The next step up from the tile in the architectural hierarchy is the quad which contains four tiles and a cache controller and direct memory access (DMA) engine to communicate with the outside network. Quads are then linked together by an intercommunication network using low swing wires[13]. Figure 5.1 shows the organization of the quad and tile. Also attached to the network are memory controllers that interface large banks of memory with the network.

5.1.1 Flexible Memory System

Smart Memory is unusual compared to most multiprocessors in that it has a flexible memory system. This flexibility is enabled by different parts of the system, from special memory instructions in the Tensilica processor, the routing of the memory access on the tile, to the handling of the memory transaction by the cache controller on the quad and finally the memory controller attached to the network.

Most machines have a defined protocol for the memory system that dictates the routing and handling of memory transaction based on the operation and the address. For a system like the Raw[33] chip multiprocessor, there is one level of caches for instructions and data but no coherency between other caches of the system and processor communicate using a separate network writing and reading to FIFOs. Classical chip multiprocessors use some form of cache coherency using a directory or another protocol. The Imagine processor[16] has stream memory system that gets accessed only by the DMA engine and the stream processor. We explore here how the different parts of the Smart Memory architecture contribute to the flexible memory system.

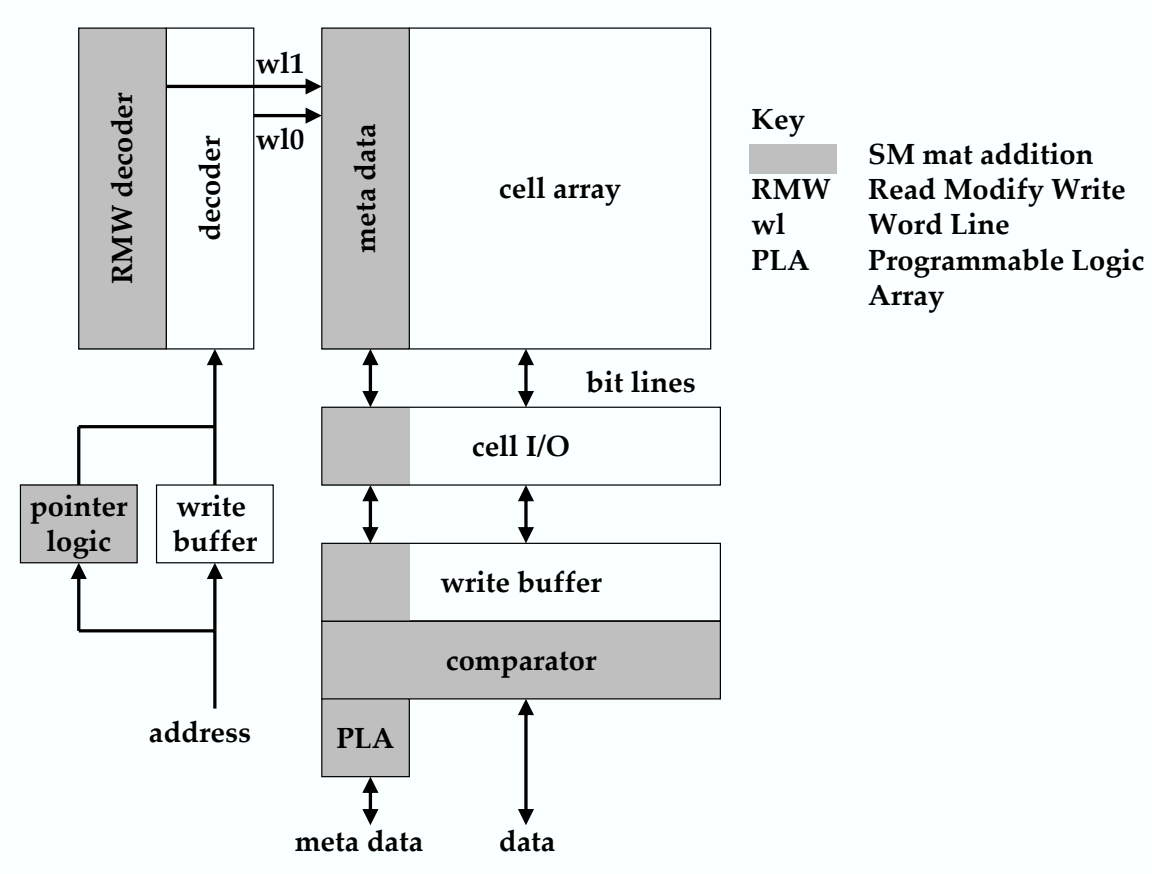

Figure 5.2: Memory Mat

Memory Mats

As the basic configurable memory unit, a memory mat consists of 4kB of SRAM with some peripheral logic to allow it to fill different roles. A memory mat can be configured to be used as a basic scratch memory, a FIFO or as part of a cache containing tags or data for the cache. This programmability has some overheads in terms of area and power over a specialized memory unit but if it is implemented using custom circuits these overheads have been found to be in the order of 15 percent for area and 20 percent for power [21]. For every word (32 bits) some meta-data bits are kept that are used for different purposes depending on the use of the memory mat. As an example, a memory mat containing tag data would use meta-data to keep track

of a cache line as valid and to implement the replacement policy.

Figure 5.2 shows the components of a memory mat on Smart Memory with the additional parts that enable the memory mat to be more than a simple SRAM in gray. The pointer logic is used in the fifo mode to keep track of the head and tail pointers for a number of FIFOs implemented on the memory mat.

DMA Engine

A Direct Memory Access engine is integrated to the cache controller to offload from processors the bulk transfers of data. It is particularly useful in gathering and scattering data in between local memory on a tile and the outside.

The DMA engine supports different types of operations which are differentiated by the way the addresses are generated and the mapping between the contiguous data on the tile and its origin or destination on the outside. Besides the obvious transfer of a single block of data, the DMA engine supports two addressing modes: strides and indexed accesses.

Strided accesses specify a record size and stride in between two successive records that are to be gathered or scattered. A good example for their use is to get a column of elements from a matrix that has been stored in a row major way. Strides are also useful for decimating filters, allowing one to express the skipping of every other elements in a simple and compact way that minimize usage of on-chip memory and address generation from the processing element.

For any other distribution of records, it is possible to use indices in a tile's memory as the location of each of the records to be transferred. A common non-trivial memory access patterns is the butterfly combination while computing the discrete Fourier transform using the Cooley-Turkey FFT algorithm. Address indices can be computed in advance by a processor on a memory mat used as scratch memory. In the case of an unstructured mesh of elements each element can contain a list of other elements to which it is connected to which can be used as indices to fetched connected elements.

DMA transfers are configured and initiated by writing to memory-mapped registers, with a final write to a control register that starts the DMA transfer. As part of the configuration of a DMA transfer, one can tell the DMA controller the memory

address to write to and the data to be written when the operation completes. In this way, it is possible to notify a processor of the completion of the DMA transfer. The write can either allow a requesting processor to trigger an interrupt call on itself, or it can the free a lock that a processor will eventually wait on. Thus without having to poll the DMA controller and generate extra memory traffic, it is possible to be efficiently alerted of the completion of a DMA transfer.

There could be multiple DMA transfers simultaneously from within the same quad, using named DMA channels. The serialization point of the quad, the network interface can only send a single message per cycle, so in effect having multiple DMA channels will cause simultaneous transfers to be interleaved.

There could be advantages to interleave such transactions in order to give the interconnection network a more uniformed traffic pattern. Though in our case most traffic goes straight to the memory controllers and cache lines are inter-leaved between different memory controllers giving most access patterns a uniformed distribution unless they are the worst case stride and always hit the same memory controller.

5.1.2 Smart Memories Use Cases

To understand how the flexible memory mats, tile crossbar and cache controller can work in a real configuration we will look at the most common configuration, system-wide coherent L1 caches and how to implement fast synchronization operations.

Cache Coherent Configuration

Instruction and data caches are separate and connect to the processor through two different processor ports. The caches can be shared among the two processors in each tile but since the caches can only handle a single access per cycle it means that one of the processors will be stalled when both processors have a cache access.

In order to implement a cache using memory mats, some will be configured as mats containing tags and some will contain data. The parameters to choose from in designing our caches will cache line size, associativity and cache size. In the simplest configuration we will have a single tag mat and a single data mat, where the cache

line size is a single 32 bits word, and is a direct mapped cache. To increase the cache line size, we can either add another data mat in parallel or shift the address bits to the tag mat so that data words now map to one tag. To add more associativity to our single-word line size direct mapped cache we will add another tag and data mat in parallel such that accesses will be tested in parallel.

The processor load/store unit and the memory mats have to be configured to know what to do on a cached memory access. The tag mats are accessed in parallel with a compare operation with the address value. The tag mats are programmed to use some of the address to lookup their RAM, if the tag stored at that address matches and the meta-data bit containing valid is high, the tag mat asserts its output that goes out to other memory mats. The data mats were are also accessed in parallel at the same time as the tag mats. If the output of the associated tag mat goes high indicating a hit, the data mat will return the data value to the load/store unit. No result indicates a miss, at which point the load/store unit initiates a miss to the cache controller.

Memory Configuration	Cache Data Mats	Cache Tag Mats	Total Memory Mats
two 8kB one-way I-cache	2	1	6
shared 64kB two-way D-cache	8	2	10
Total			16

Table 5.1: Smart Memories sample cache configuration

The first major choice is the size of the cache line for both the instruction and data cache. Since the memory map of the system separates the instruction and data segments it is possible to have different cache line sizes for both. The instruction port of our processor is 64 bits wide (eight bytes) so it is our natural minimal cache line size. So at a minimum the instruction cache will use two memory mats to contain the data and another mat to contain the tags. If we wish to have more associativity for the same cache line size we need to add another three mats, two more for data and another one for tags. The data cache can have a cache line size of a four bytes because it is the size of data port as well. The larger cache line size amortizes the cost of the tag mat across more data mats but reduces the possible associativity of

the cache. Table 5.1 shows the memory mat usage for a sample cache configuration of two 8kB direct mapped instruction caches and a shared 64kB two-way data cache.

Synchronization Operations

Something that was not exploited in the configuration of memory mats as caches was the configurability of the processors. The Tensilica LX allows us to add special operations, and in our case we will add special types of loads and stores instructions to support fast synchronization methods.

We've seen that the meta-data bits in the tag mats are used to save such information such as if the cache line is valid. In the data-mats or any other mat configured as a regular SRAM they could be used for other purposes. By using one of the meta-data bits as a lock on the current word we can implement a system that provides single word locking. This does require that every word in the rest of the memory system keeps that meta-data bit around including when a word is written back from the cache.

We will create three new memory operations to implement fast single word locks. Safe-load will stall the process if the safe meta-data bit is not set; if it is set, it will unset it and return with the value. Safe-store stalls when the safe meta-data bit is set; if is unset, it sets and it and writes it's value to that location. Finally, always-safe-store never stalls, always sets the safe meta-data bit and writes it's word at the location.

Some of the implementation details require the memory controller to keep track of each processor stalled on a memory operation. Successful safe write and loads are propagated to the appropriate memory controller too wake a processor waiting on that memory access. The process of waking is fast, just a re-issue of the instruction. It is also possible to keep track of the local quad processors which are blocked on a such access and when the access is to a local tile, wake the appropriate quad processor.

With these instructions it is possible to implement all kinds of fine grain locks that are fast and efficient. We will see in Section 5.3 that we will build upon them to implement synchronization of the SVM library.

5.2 SVM on Smart Memories

The Smart Memories memory mats can be configured in many different ways by configuring them to perform different memory operations. At a higher level the memory mats will be configured to act as caches, FIFOs and static SRAM.

To implement the Stream Virtual Machine on Smart Memories, we will have to decide how to use the memory resources and create a SVM model to be used by the high-level compiler to generate a SVM source file that will be recompiled using our SVM library.

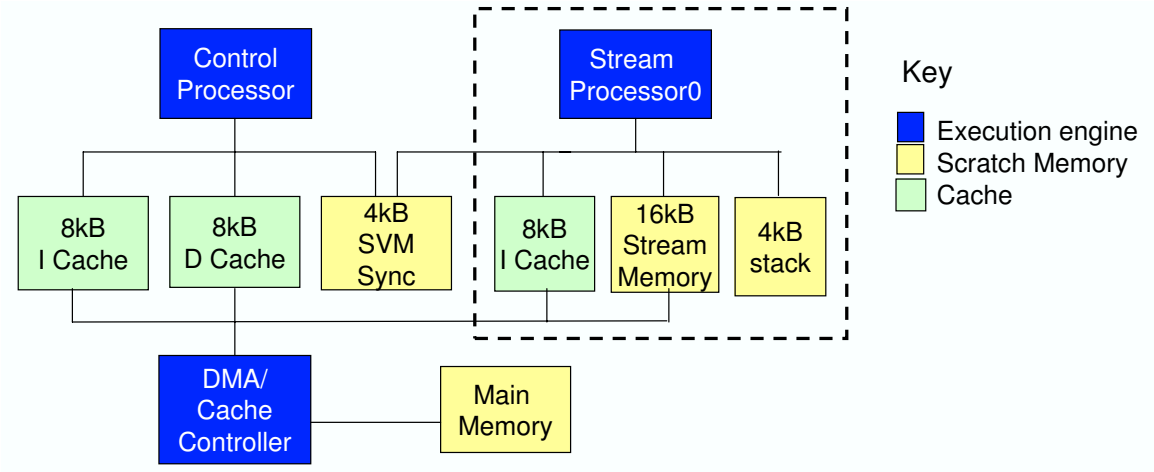

Figure 5.3: Smart Memories SVM Mapping

Figure 5.3 shows the configuration of a Smart Memories tile as the simplest stream system possible - a single stream processor. The control processor has some amount of instruction and data cache to help with bookkeeping and a small amount a scratch SRAM to help communicate and synchronize with the stream processor and DMA engine. As seen in the previous section, the instruction cache needs a width of 64 bits since that is the instruction port size for the VLIW processor. So at a minimum the cache line has to be 64 bits, using two data mats, with another memory mat containing the tags as a one-way cache.

The stream processor requires some amount of instruction memory which could be SRAM but that would require pre-loading kernel codes ahead of time, and some

smart partitioning for codes that don't fit the SRAM. Using an instruction cache eliminates both these issues at a minimal performance hit. The stream memory's size is the main parameter passed to the high-level compiler generating SVM API code. The DMA engine will bring data in and out of the stream memory and the stream processor will operate on the given data. Finally, to avoid running out of registers for temporary variables in the stream processor, a separate scratch SRAM is allocated and given to the low-level compiler as the stack location.

Table 5.2 shows the allocation of the sixteen memory mats on a tile for this particular configuration. The unit of the stream processor which is separated in the dashed box can be replicated many times, two per tile as it uses eight memory mats, half of a tile's memory resources. Because of the quantized nature of the allocation of memory mats as tags or data ressources and the limited number of mats that are avaiable on our prototype, we end up with caches that have a significand area overhead for the tags. Dedicated caches on processors have cache lines that are 128 bytes wide whereas our cache line size are 4 or 8 bytes wide. Our overhead for tags is in the area of 50% to 100% whereas a conventional cache would have an overhead of 5% to 10%.

Memory Configuration	Cache Data Mats	Cache Tag Mats	Total Memory Mats
8kB one-way I-cache	2	1	3
8kB two-way D-cache	2	2	4
4kB Sync SRAM			1
8kB one-way I-cache	2	1	3
16kB Stream SRAM			4
4kB Stack SRAM			1
Total			16

Table 5.2: Smart Memories mat usage in stream system

5.3 SVM API

The SVM execution model has different threads of control that can run out of sync as the operations they are waiting on complete potentially out-of-order. Next we'll look

at how we solved some of these issues on Smart Memories, including issues of synchronization and waiting, as well as how to deal with a very simple non-autonomous DMA engine.

5.3.1 Sync Operations

The first issue we'll address is low overhead synchronization. This will enable some tasks to wait on others to complete before they can start. For example, a kernel must wait for the DMA fetching it's data to complete before it can start. To allow kernels to work on small blocks of data, this synchronization must be efficient.

A first naive solution might have a waiting thread of control poll a memory location to check if the other task has completed, but this solution is inefficient both in terms of power dissipation and memory bandwidth usage. Using interrupts to wake a process from a waiting state also won't work since each interrupt service routine is in the order of 50 instructions. For small tasks this overhead is too high.

Instead we use the capabilities of the Smart Memories flexible memory to solve this problem efficiently. We use the safe memory instructions described in Section 5.1.2. By having different resources issue safe writes to a specific memory location for each task, any waiting processor that has a blocked safe-load will be awaken, giving almost instantaneous synchronization.

5.3.2 Simple DMA Controller

As we've seen in the previous section on the design of Smart Memories that the DMA engine was deliberately kept simple with no capability for queuing request, only the capability to launch as many transfers as there are DMA channels, monitor their progress and notify other tasks when the DMA completes by issuing a final memory write of programmable type, address and data.

In our execution model of having the stream processor control all of it's DMA transfers within a double-buffered loop, most likely there will be more than a single transfer to be launched at each instance. Also in Smart Memories, four processors share a DMA engine with many DMA channels but most likely less than one per

processor. So the queuing issue is further complicated by issues of synchronization and possible concerns for arbitration.

With the large number of simple processors systems like Smart Memory provide, we can dedicate a processor to orchestrate DMA transfers, monitor completion of DMA transfers and start the ones that are waiting. In order to handle having multiple stream processors initiate transfers at the same time with little overhead, we will use a memory mat as a FIFO. By writing a single pointer to a data structure that contains a DMA transfer request, many DMA requests can happen at the same time without any interruption from the requesting stream processors.

The processor managing the DMA engine can then launch any DMA transfer by copying the necessary configuration register to start the DMA transfer on a free DMA channel. Another memory mat can also be used as a FIFO in which DMA channels write their ID upon completion of the transfer in order to have CPU clean up after the transfer and start the next transfer.

In this simplified model, the CPU managing the DMA engines has to do a lot of polling on both FIFOs for new and completed DMA transfers which generates a lot of memory traffic. While this traffic is local to the tile, the processor is still wasteful of energy. Again we can use the Smart Memories synchronization memory operations to enable fast and energy efficient memory synchronization.

By using a memory mat as scratch memory, a single word can be used to synchronize the DMA service process to incoming work. Rather than polling the queue, the DMA manager does a safe-load of the word, which means that it will block until someone else does a safe-write to that same word. When work is generated, either by the stream processors or the control processor, they issue an always-safe write to the synchronization word. We use an always-safe-write, whose behavior will be that if the safe bit is already set in the word the write doesn't complete but is reported as successful and thus doesn't block the issuing processor, since it is possible that other work has already arrived. Since we are using this location to "wake up" the processor, there is no need to record how much work is waiting. On "wake up", the DMA manager services both FIFOs for incoming DMA requests and completed DMA transfers, then issue a safe read to the synchronization word. This read puts the processor to

sleep until more work arrives. There is a race condition between writing to the FIFO and the "wake up" location since the FIFO write needs to complete first. We handle this race by placing both the FIFO and the "wake up" location on the same tile since all operations are ordered in a tile.

As DMA requests will come from multiple stream processors to use the limited resource of the DMA engine, we could conceive of a need for arbitration. In the case of applications running in a time-multiplexed mode and all stream processors execute in parallel with the same expected execution time, the policy desired will be one of equality. Whereas in a space-multiplexed execution model, with some stream processors having more load than others and are expected to set the critical path of execution, we would like higher priority for their transfers.

In the end, we found that arbitration gave little benefits because the policy desired generally occurred at the steady state. For a policy of equality between kernels that have identical execution time, they will get staggered in time, starting as soon as the last transfer they were waiting for completes, then issue their next commands when they complete their kernel.

For space-multiplexed kernels, at steady-state all non-critical path kernels will complete and will wait for the critical path kernel too be done to push or pull data to/from it. Given that the computation is not bandwidth limited the DMA engine will also be idle, then upon the critical path kernel completion, it will issue it's DMA commands which will be executed first in order, as if they had higher priority.

The overheads of implementing an arbitration on the DMA manager can be evaluated when looking at which one of the FIFO service would suffer. Since most of the work would come when there is a lot of contention, the main risk would be to delay servicing a completed DMA transfer and issuing the next one. As in the reasoning behind having only two DMA channels to keep the network interface busy during the switching of a transfer, as long as DMA transfers would be longer than the total time to service them, the performance cost would be minimal.

DMA Manager outside Quad

Dedicating a processor per Quad to manage the DMA engine of the Quad sacrifices one eight of computing power to system management issues which is an expensive cost. There is nothing that prevents a processor from managing DMA engines in other Quads, the only cost would be extra latency for each inter-quad transfers in between the Quad with the Stream Processor and DMA Engine to the Quad of the DMA manager. Since this latency needs to be hidden by the operation of the other channel, this increases the requirement for long DMA transfers

With only one DMA manager controlling multiple DMA engines access, it cannot let all requests wait in a single FIFO: a request for an idle DMA engine could be stuck waiting behind requests for a busy DMA engine. A separate FIFO for each different DMA engine is needed. Each memory mat on Smart Memories can be configured as a single FIFO of 1024 words or two FIFOs of 512 words. The FIFO for completions doesn't need to be split since completed requests have to wait for the DMA manager to perform maintenance on them, so there is no separate contention for completed transfers.

In a large system, having more DMA transfers to manage causes more contention for the DMA manager so the number of DMA managers will have to be scaled to prevent it from becoming a bottleneck for the application.

5.4 Performance Study

Ground Moving Target Indicator (GMTI)[28] is a test application meant to represent future workloads of avionics. As part of the Polymorphous Computing Architecture (PCA) program, MIT-LL has developed a version of the GMTI application meant to stress future computing architecture on such a workload. The front-end of the application consisting mostly of signal processing is called the Integrated Radar Tracker (IRT).

Most of the processing is composed of classical signal processing algorithms such as fast Fourier transform and finite impulse filters. But there are also a lot of matrix

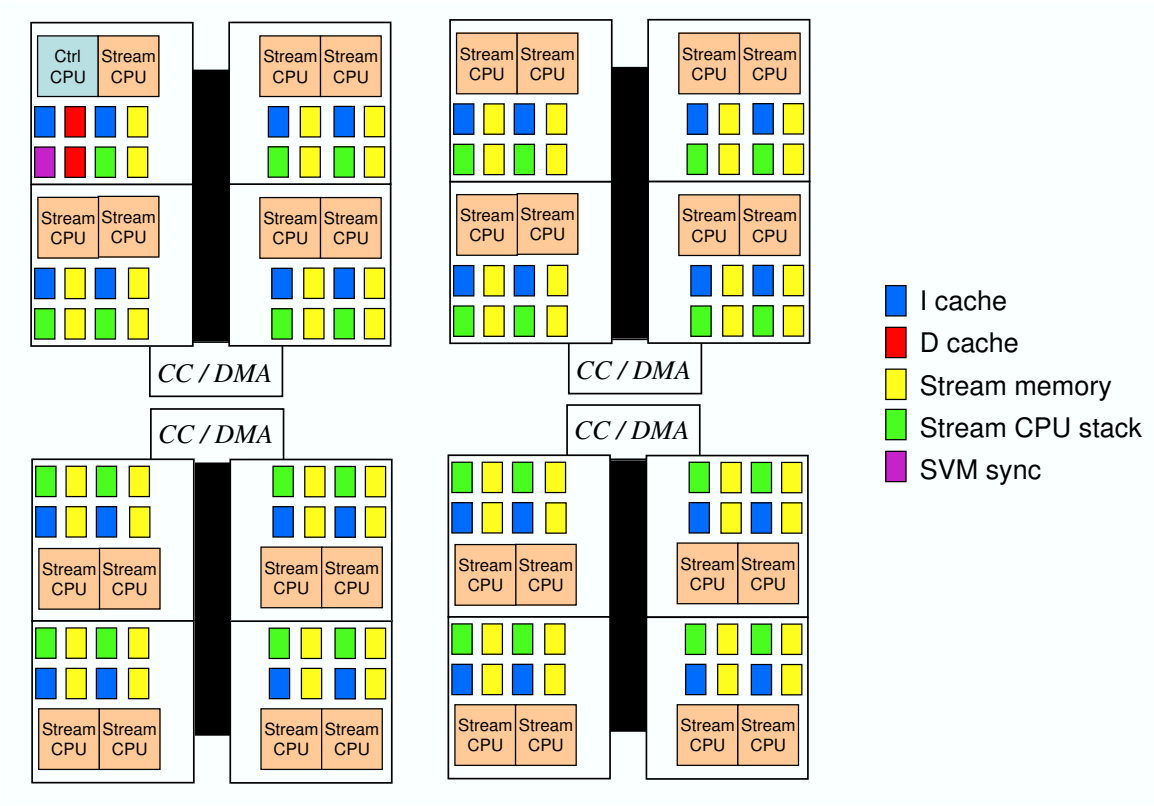

Figure 5.4: Full Smart Memories system configured for streaming

manipulation and solvers. The input data is 16-bit integers but the computation is all done in floating point in order to accommodate the expanding dynamic range of the intermediate results.

5.4.1 GMTI Kernel description

1. The **time delay equalization (TDE)** stage compensates for differences in the transfer fucntion between channel.

2. The **adaptive beam forming** stage transforms the filtered data into the beam-space domain to allow detection of target signals coming from a particular set of directions of interest while filtering out spatially-localized interference.

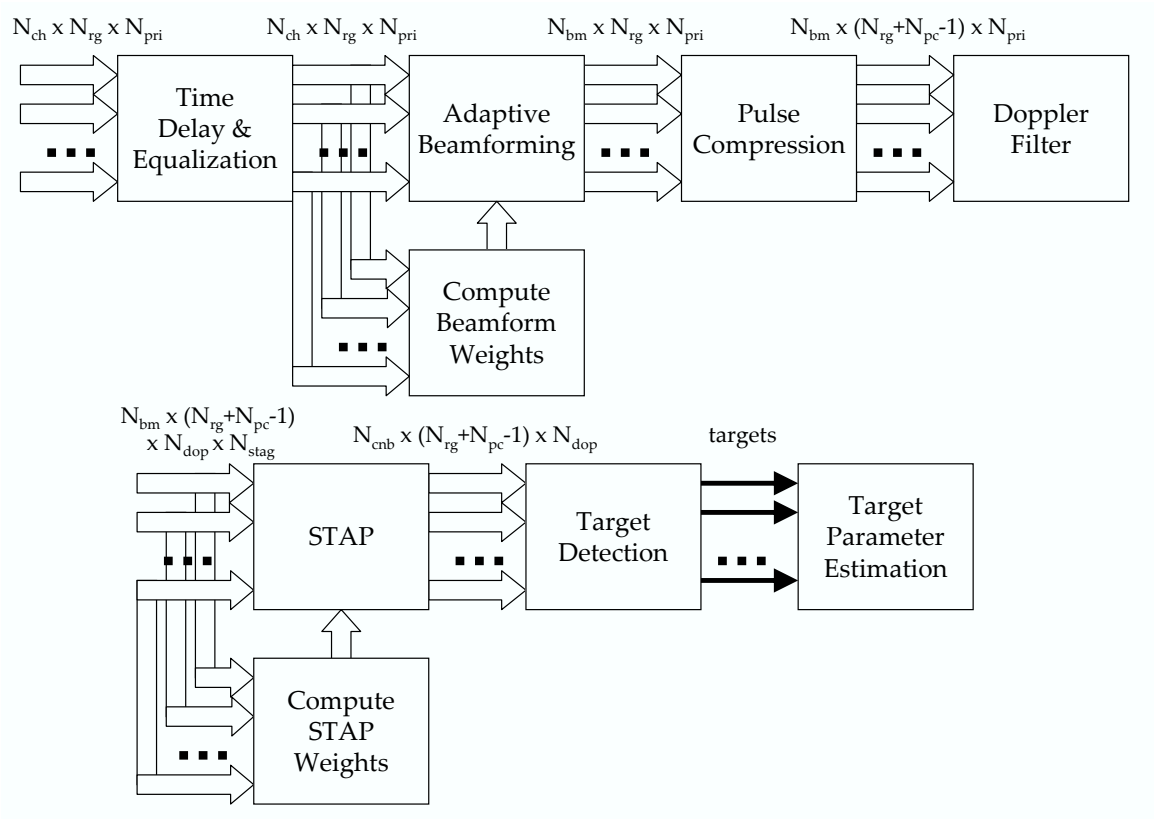

Figure 5.5: GMTI application

3. The **pulse compression** stage filters the data to concentrate the signal energy of a relatively long transmitted radar pulse into a relatively short pulse response.

4. The **doppler filtering** stage processes the data so that the radial velocity of targets relative to the platform can be determined.

5. The **space time adaptive processing** stage is a second beamforming stage which removes further interference and ground clutter interference.

6. The **target detectio** stage uses constant false-alarm rate (CFAR) detection to compare a radar signal response to its surrounding signal responses to determine whether a target is present and uses target grouping to eliminate multiple target reports that are actually just one target.

7. The **target parameter estimation** stage estimates target positions in order to pass them to the tracking algorithms.

5.4.2 Application Performance

The GMTI application was written for the R-stream high-level compiler[19] in such a way that the application could also be compiled by a regular C compiler to produce a regular single thread application in order to compare application run-times between streaming and a single-thread version. Three different data set sizes were also selected to represent different types of workload based on the size of the input radar data which are summarized in Table 5.3. N_a

Category	Small	Medium	Large
FFT size	64	512	4096
TDE filter taps	12	32	36
N_{ch} channels	6	8	9
N_{rg} range gates	36	450	2691
N_{pri} pulse repetition interval	15	48	31
N_{bm} beam	4	4	7
N_{pc} filter taps pulse compression	12	32	167
N_{dop} Doppler bins	14	47	30
N_{stag} pri staggers	2	2	2
N_{cnb} clutter null beams	4	2	5
Parallelism for TDE and Adaptive Beam Forming $N_{ch}xN_{rg}xN_{pri}$	3.2k	173k	751k
Parallelism for Pulse Compression $N_{bm}xN_{rg}xN_{pri}$	2.1k	86k	584k
Parallelism for Doppler Filter $N_{bm}x(N_{rg}+N_{pc}-1)xN_{pri}$	2.8k	86k	584k
Parallelism for STAP $N_{bm}x(N_{rg}+N_{pc}-1)xN_{dop}xN_{stag}$	5.3k	181k	1.2M
Parallelism for Target Detection $N_{cnb}x(N_{rg}+N_{pc}-1)xN_{dop}$	2.6k	45k	429k

Table 5.3: GMTI data set

The GTMI application was compiled using the R-stream compiler for the different

data set sizes for different number of stream processors to evaluate the scaling of the streaming model for different data set sizes. The streaming execution model used was the one supported by the R-stream compiler, which is time multiplexing the kernels. Graph 5.6 shows the speedups of running the different dataset sizes on different number of stream processors. The scaling is very good for the GMTI application but it is important to remember that if we are comparing the scaling for equal computing resources, the streaming model in Smart Memories always uses two additional processors, the control processor and the DMA engine manager.

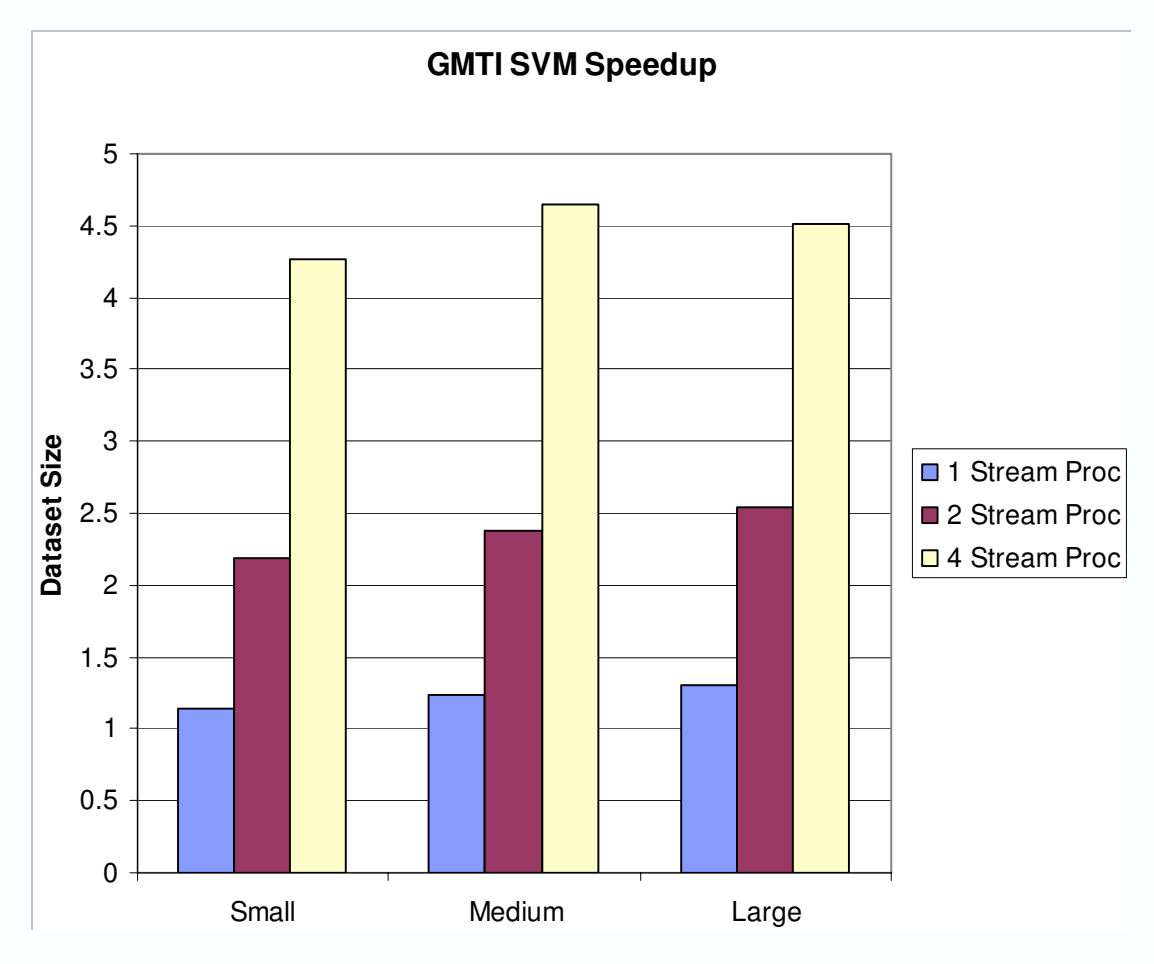

Figure 5.6: GMTI speedups vs single thread case

There is good concurrency of data transfers and computation in the GMTI application, since in fact most data transfers are small compared to the computation times. Graph 5.7 shows for a two stream processor configuration the overlay of computation kernels and DMA transfers on the two DMA channels.

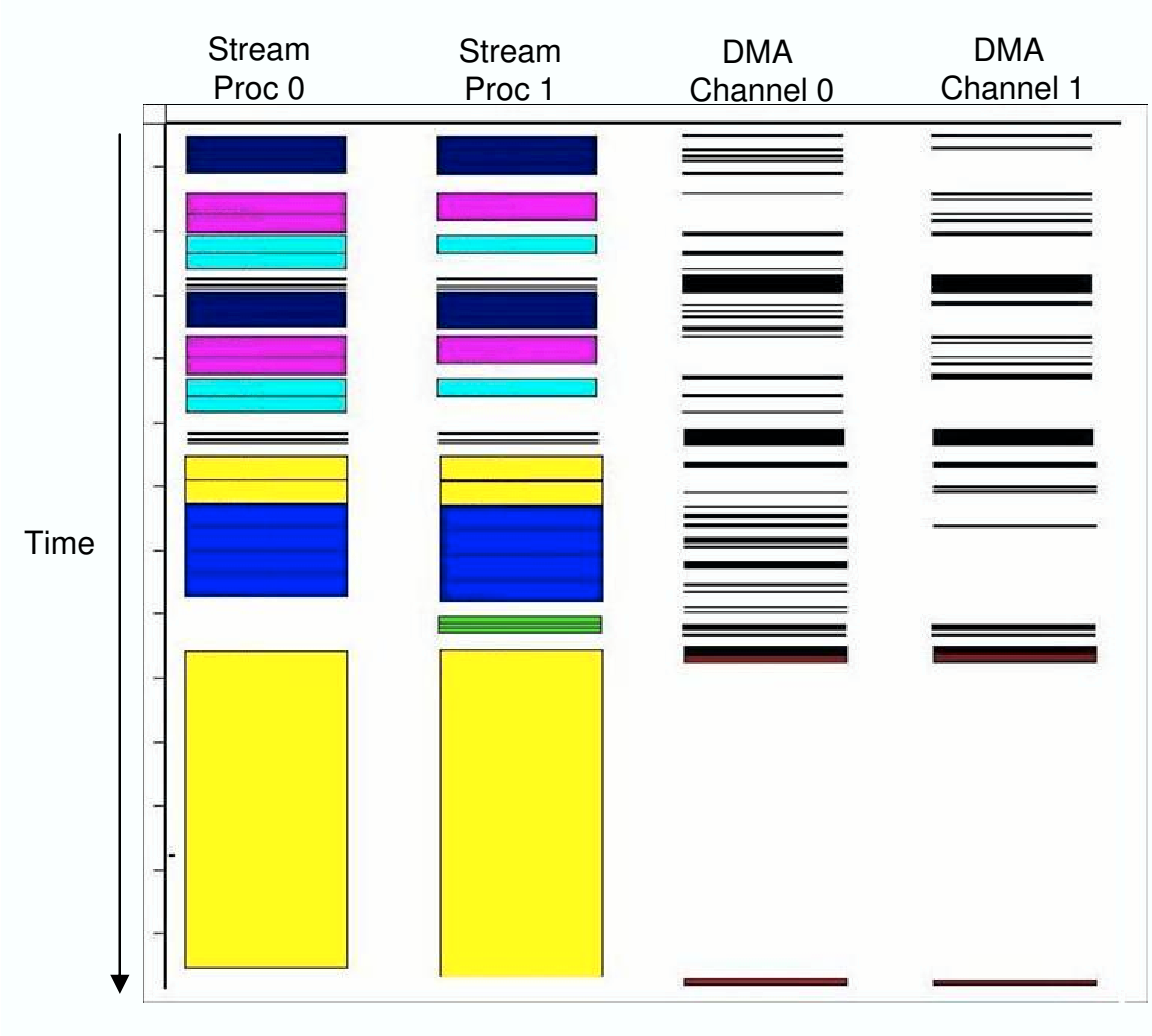

Figure 5.7: GMTI application run-time resource usage

5.4.3 Overheads of Virtualization

The decision to use a high-level compiler to generate SVM API code instead of using direct primitive of the underlying architecture allows the high-level compiler to be relieved of knowing of the specifics of each stream architecture but the cost comes in terms of performance. We can evaluate an upper bound on the performance loss due to using the SVM API by evaluating the number of instructions that run code for the SVM, managing data structures that are intrinsic to the SVM and are not essential to manipulating the underlying hardware constructs. Although this code is not always in the critical path, in the worst case it always is and evaluating the proportion of this code with regards to the overall number of cycles of the application runtime we have an upper bound for the performance penalty of the SVM API.

Figure 5.8 shows a graph of the ratio of SVM API instructions over the run-time in cycles of the application GMTI application for different dataset sizes and number of stream processors. As expected, the impact is greater on smaller dataset sizes running on the largest number of stream processors since thereis the least application work on each processor. As the dataset size increases the double-buffered super kernels are getting amortized across more iterations and the performance penalty limit is less.

The overhead of the SVM API is quite reasonable, and the upper bound on the GMTI application for any combination of the larger dataset is below 0.3%.

5.4.4 Streaming Advantage

A lot of the advantages of streaming come from blocking the application to sizes that fit the small local memories and chaining operations to preserve the locality of the blocking. These same properties are beneficial and will be exploited on cache systems as well. In many ways comparing an unoptimized application using caches to a stream version is unfair to the cache systems.

In order to get a fairer view of the advantage of stream hardware, we can use the SVM API code generated by the Stream compiler and run it on a system with caches. All that is needed is to change the kernels that load and store data because we will no longer prefetch data using a DMA into a dedicated memory space. Instead

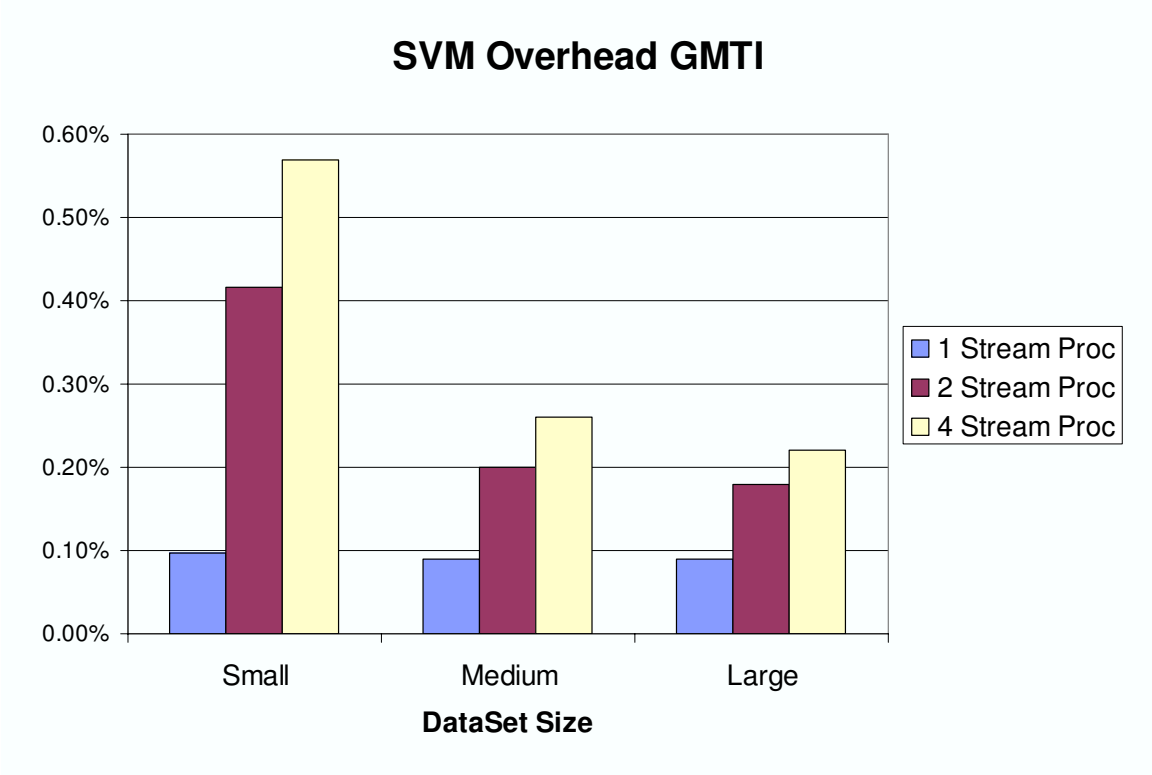

Figure 5.8: SVM Overheads for GMTI application

we modify the code by replacing the calls to streamLoad and StreamStore with the proper memory address calculation.

In order for the cache system to benefit from the stream blocking and locality their cache size will be made the same size as the stream memory. For a given size stream memories are simpler and smaller in die area than caches. This makes the cache system a little more resource expensive.

Figure 5.9 shows the result of this run-time comparison for the GMTI application. The Stream configuration of Smart Memories is faster but only by an average of five percent. The SVM configuration benefits from prefetching of data through the DMA engine but on an architecture with hardware predictive prefetching this advantage would diminish. As the computation is more and more dominated by the stream portion of the application (larger datasets) the improvement increases and it decreases

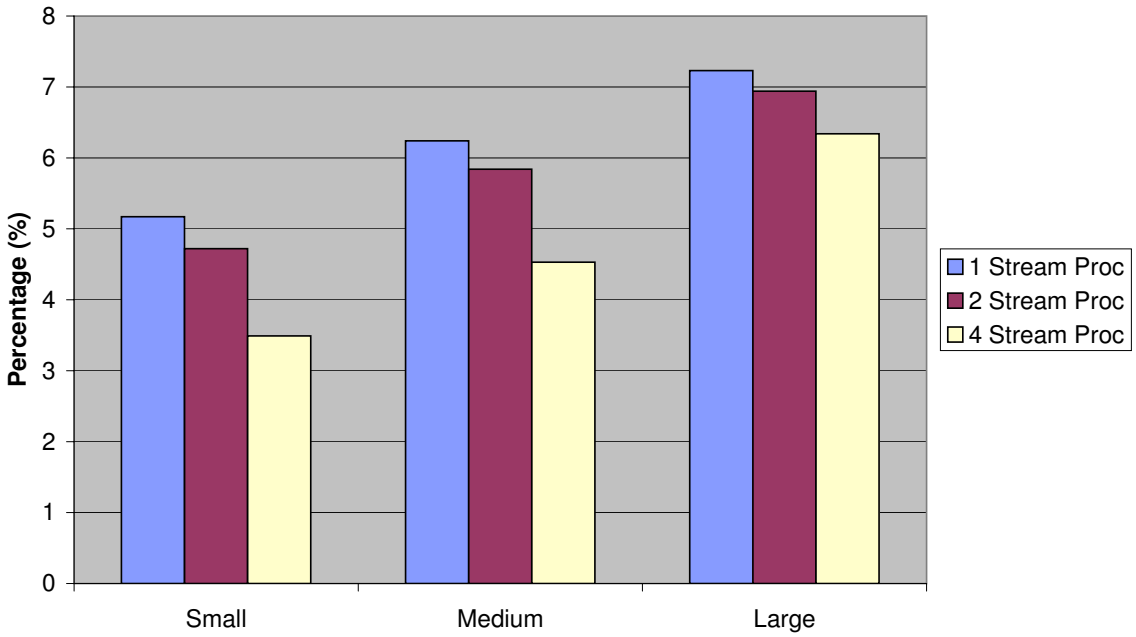

Figure 5.9: Performance advantage of GMTI SVM code in the stream configuration vs a cache configuration

with more stream processors.

An important caveat is that we didn't explore rebalancing the ratio of computation and memory bandwidth for both the stream and conventional processor. For a dedicated stream processor, memory bandwidth ressources are more expensive than additional ALUs on the die, so it is in the interest of the processor to be memory bandwidth limited for most cases by adding ALUs. In our test system, adding ALUs to our processor in the form of a short vector unit would have required modifying the load/store system for any real improvement in performance and unfortunately this option was not explored. The performance of computation limited kernels could have been improved on a stream processor of equivalent die area by adding ALUs to compensate for the added cache overhead of conventional processors.

Chapter 6

Conclusion

New programming models are needed to effectively use future parallel systems. Stream programming is one such model. By retaining a "single thread of control" this computational model is easier to understand and reason about than normal threaded applications. By having the kernels that this thread controls operate on large "streams" of data, it also exposes sufficient parallelism to leverage modern machines. While this general compute model has been widely adapted, hardware implementations are all slightly (or not just slightly) different, and many stream languages have been proposed. This leads to each group working on their own software system, which leads to difficulties in sharing infrastructure, and slows down the overall streaming software development.

To address this issue, we introduced the Stream Virtual Machine (SVM), an abstraction layer that allows one to share some of the compiler infrastructure. This layer can model most stream hardware, and should allow Steam software systems to compile into SVM code, and then need a simpler SVM to native code translation. By defining the abstraction level as a C-language API, it is easy for different underlying architectures to implement the SVM and for a stream compiler to generate SVM code. However, for this model to be useful it must be to possible to accurately model the underlying performance and resource constraints of the native hardware.

The SVM machine model's simple parameters of execution rate and bandwidths has shown to be a good predictor of the run-time of simple kernels as was shown in

our study on graphics processors. By extracting the performance parameters through very simple experiment we were able to predict the application run-time of simple kernels very accurately which would help a high level stream compiler being fed such performance parameters make good decisions.

The SVM also clarifies some of the important implementation issues for hardware to support streaming efficiently. The base SVM model has three threads of control – one for the control processors which issues kernel operations, one for the DMA engine, which fetches the needed data, and one for the kernel execution. By mapping the SVM onto the Smart Memory chip multi-processor, we showed that the entire interface could be implemented using just a few specialized memory structures that were available using Smart Memory's flexible memory system. Using this system we were able to show that while using local memory (stream register files) saves hardware overhead, for many applications it provides only a modest performance boost over caches.

The Stream Virtual Machine is a good abstraction layer to allow stream programs to be portable across different architectures without much performance loss over a native implementation. The fact that there is only a small performance advantage of stream memories over caches makes it an interesting programming target even for common chip multiprocessors which lack a good application base of parallel programs.

Bibliography

[1] Peter Mattson Abhishek Das, William J. Dally. Compiling for stream processing. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, September 2006.

[2] Bob Beretta and al. *OpenGL ARB fragment program*. OpenGL ARB, sep 2002. http://oss.sgi.com/projects/ogl-sample/registry/ARB/fragment_program.txt.

[3] Ian Buck. Gpu computing: Programming a massively parallel processor. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, page 17, Washington, DC, USA, 2007. IEEE Computer Society.

[4] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: Stream computing on graphics hardware. In *Proceedings of SIGGRAPH*, 2004.

[5] T.-F. Chen and J.-L. Baer. A performance study of software and hardware data prefetching schemes. In *ISCA '94: Proceedings of the 21st annual international symposium on Computer architecture*, pages 223–232, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[6] William J. Dally, Patrick Hanrahan, Mattan Erez, Timothy J. Knight, Francois Labonte, Jung-Ho Ahn, Nuwan Jayasena, Ujval J. Kapasi, Abhishek Das, Jayanth Gummaraju, and Ian Buck. Merrimac: Supercomputing with streams. In *Proceedings 2003 SuperComputing*, nov 2003.

[7] John L. Hennessy David A. Patterson. *Computer Architecture a Quantiative Approach*. 1996.

[8] James Kim Dhanendra Jani, Gulbin Ezer. Long words and wide ports: Reinventing the configurable processor. In *Proceedings of the 16th Hot Chips Conference*, August 2004.

[9] Jeff Draper, Jacqueline Chame, Mary Hall, Craig Steele, Tim Barrett, Jeff LaCoss, John Granacki, Jaewook Shin, Chun Chen, Chang Woo Kang, Ihn Kim, and Gokhan Daglikoca. The architecture of the diva processing-in-memory chip. In *ICS '02: Proceedings of the 16th international conference on Supercomputing*, pages 14–25, New York, NY, USA, 2002. ACM.

[10] Stuart Oberman Erik Lindholm. Nvidia geforce 8800 gpu. In *Proceedings of the 19th Hot Chips Conference*, August 2007.

[11] Kayvon Fatahalian, Timothy J. Knight, Mike Houston, Mattan Erez, Daniel Reiter Horn, Larkhoon Leem, Ji Young Park, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006.

[12] Ricardo Gonzalez. Configurable/extensible processors change system design. In *Proceedings of the 11th Hot Chips Conference*, August 1999.

[13] Ron Ho. *On-Chip Wires: Scaling and Efficiency*. PhD thesis, Stanford University, 2004.

[14] Mark Horowitz and William Dally. How scaling will change processor architecture. In *Solid-State Circuits Conference, 2004. Digest of Technical Papers*, pages 132–133. IEEE International, 2004.

[15] Nuwan Jayasena, Mattan Erez, Jung Ho Ahn, and William J. Dally. Stream register files with indexed access. In *Proceedings of the Tenth International Symposium on High Performance Computer Architecture*, Madrid, Spain, February 2004.

[16] Ujval Kapasi, William J. Dally, Scott Rixner, John D. Owens, and Brucek Khailany. The Imagine stream processor. In *Proceedings 2002 IEEE International Conference on Computer Design*, pages 282–288, sep 2002.

[17] Ujval J. Kapasi, William J. Dally, Scott Rixner, Peter R. Mattson, John D. Owens, and Brucek Khailany. Efficient conditional operations for data-parallel architectures. In *Proceedings of the 33th Annual International Symposium on Microarchitecture*, page 159. IEEE Computer Society, 2000.

[18] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, January 1987.

[19] Allen Leung, Benoit Meister, Eric Schweitz, Peter Szilagyi, David Wohlford, and Richard Lethin. R-stream: High level optimization for pca. Technical report, Reservoir Labs, 2006.

[20] Erik Lindholm, Mark J. Kligard, and Henry Moreton. A user-programmable vertex engine. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 149–158. ACM Press, 2001.

[21] Ken Mai. *Design and Analysis of Reconfigurable Memories*. PhD thesis, Stanford University, 2005.

[22] Michael Mantor. Radeon r600, a 2nd generation unified shader architecture. In *Proceedings of the 19th Hot Chips Conference*, August 2007.

[23] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: a system for programming graphics hardware in a c-like language. *ACM Transactions on Graphics (TOG)*, 22(3):896–907, 2003.

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

[25] Peter Mattson, William Thies, Lance Hammond, and Mike Vahey. Streaming virtual machine specification 1.2. Technical report, 2007. http://www.morphware.org.

[26] John Nickolls. Nvidia gpu parallel computing architecture. In *Proceedings of the 19th Hot Chips Conference*, August 2007.

[27] D. C. Pham, T. Aipperspach, D. Boerstler, M. Bolliger, R. Chaudhry, D. Cox, P. Harvey, P. M. Harvey, H. P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Pham, J. Pille, S. Posluszny, M. Riley, D. L. Stasiak, M. Suzuoki, O. Takahashi, J. Warnock, S. Weitzel, D. Wendel, and K. Yazawa. Overview of the architecture, circuit design, and physical implementation of a first-generation cell processor. *Solid-State Circuits, IEEE Journal of*, 41(1):179–196, 2006.

[28] Albert Reuther. Preliminary design review: Gmti narrowband processing for the basic pca integrated radar-tracker application. Technical report, MIT Lincoln Laboratory, 2003.

[29] K. Sankaralingam, R. Nagarajan, P. Gratz, R. Desikan, D. Gulati, H. Hanson, C. Kim, H. Liu, N. Ranganathan, S. Sethumadhavan, S. Sharif, P. Shivakumar, W. Yoder, S.W. Keckler R. McDonald, and D.C. Burger. The distributed microarchitecture of the trips prototype processor. In *39th International Symposium on Microarchitecture (MICRO)*, 2006.

[30] Ehud Shapiro. The family of concurrent logic programming languages. *ACM Comput. Surv.*, 21(3):413–510, 1989.

[31] M. D. Smith, M. Johnson, and M. A. Horowitz. Limits on multiple instruction issue. In *ASPLOS-III: Proceedings of the third international conference on Architectural support for programming languages and operating systems*, pages 290–302, New York, NY, USA, 1989. ACM.

[32] Alex Solomatnikov, Amin Firoozshahian, Francois Labonte, Mark Horowitz, Christos Kozyrakis, Kunle Olukotun, and Ken Mai. Smart memories: A configurable processor architecture for high productivity parallel programming. In *Proceedings of the 30th GOMACTech Conference*, April 2005.

[33] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, Jae-Wook 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.

[34] William Thies, Michal Karczmarek, and Saman Amarasinghe. Streamit: A language for streaming applications. In *International Conference on Compiler Construction*, Grenoble, France, April 2002.

[35] Marc Tremblay, J. Michael O'Connor, Venkatesh Narayanan, and Liang He. Vis speeds new media processing. *IEEE Micro*, 16(4):10–20, 1996.

[36] David W. Wall. Limits of instruction-level parallelism. In *ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 176–188, New York, NY, USA, 1991. ACM.

[37] R.S.V. Whiting, P.G.; Pascoe. A history of data-flow languages. *Annals of the History of Computing, IEEE*, 16(4):38–59, Winter 1994.

[38] Victor Zyuban and Philip Strenski. Unified methodology for resolving power-performance tradeoffs at the microarchitectural and circuit levels. In *ISLPED '02: Proceedings of the 2002 international symposium on Low power electronics and design*, pages 166–171, New York, NY, USA, 2002. ACM.