The Stream Virtual Machine

Francois Labonte
Stanford University
flabonte@stanford.edu

Peter Mattson
Reservoir Labs
mattson@reservoir.com

Ian Buck
Stanford University
ianbuck@stanford.edu

Christos Kozyrakis
Stanford University
christos@ee.stanford.edu

Mark Horowitz
Stanford University
horowitz@ee.stanford.edu

Abstract

Stream programming is currently being pushed as a way to expose concurrency and separate communication from computation. Since there are many stream languages and potential stream execution engines, this paper proposes an abstract machine model that captures the essential characteristics of stream architectures, the Stream Virtual Machine (SVM). The goal of the SVM is to improve interoperability, allow developpment of common compilation tools and reason about stream program performance. The SVM contains control processors, slave kernel processors, and slave DMA units. Is is presented along with the compilation process that takes a stream program down to the SVM and finally down to machine binary. To extract the parameters for our SVM model, we use micro-kernels to characterize two graphics processors and a stream engine, Imagine. The results are encouraging; the model estimates the performance of the target machines with high accuracy.

1. Introduction

In the past 30 years, we have experienced tremendous growth in computer performance, while keeping the sequential execution model presented to the programmer basically the same. This execution model is exemplified by the imperative programming languages developed during the same time period, C and C++. These languages abstracted out the implementation differences of various sequential computers (pipeline and cache hierarchy structure) but captured the fundamentally common features (sequential control flow and unified memory system). Nevertheless, our ability to continue to scale the performance of this execution model has recently come under fire due to excessive power consumption, wire delay issues [1], and limited extractable ILP in sequential applications. The difficulties with the sequential model have encouraged researchers to explore other execution models that match the intrinsic constraints of the underlying VLSI technology and the parallelism in emerging applications [7]. One result of this exploration is an increasing interest in "stream" computations.

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

The stream abstraction suits data intensive applications with regular communication patterns. Not all applications fit this model but it is a natural fit for the DSP [6], multimedia[16] and scientific[5] computing domains. Many research groups have developed architectures for stream applications. The stream architecture space spans from configurable or statically scheduled tiled processors (Raw [20], TRIPS [19]), to SIMD stream coprocessors with a large lo-

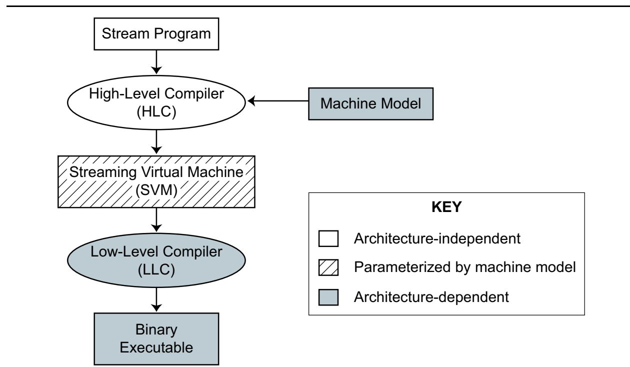

Figure 1. The two-level compilation approach for stream applications.

cal memory for stream buffering (Imagine [18]), and to commodity graphics processors [4]. Similar diversity exists with streaming programming languages. They vary from synchronous data-flow languages with infinite linear streams (StreamIt[21], Simulink[14]), to languages with support for multi-dimensional streams and stencils (Brook [3]), and to array languages (Matlab [13]). The fragmentation in stream architectures and languages creates an interoperability problem that hinders the wide adoption of stream computing. To run any stream program on any stream architecture, one must develop a separate compiler for every language and architecture pair.

The two-level compilation approach in Figure 1 can mitigate the engineering complexity of developing a new stream language or architecture. The high level compiler (HLC) is written once for each stream language and is responsible for parallelism detection, load balancing, coarse-grained scheduling of stream computations, and memory management for streaming data. The HLC inputs a stream program and targets an abstract architecture model. The abstract model is parameterized to describe basic topology and performance features of specific stream architectures. The low level compiler (LLC) is written once for each stream architecture and is responsible for instruction memory management and scheduling within each kernel. It inputs the abstract stream code and generates binary code. Apart from allowing for interoperability, this compilation model allows the language and HLC developers to focus on fundamental stream optimizations rather than over-specializing the compiler for the idiosyncratic features of any particular architecture.

This paper makes the following contributions towards supporting the two-level stream compilation model. First, we introduce the Stream Virtual Machine (SVM), an abstract model for stream computations. The SVM consists of an architecture model and an API. The SVM architecture model is a set of parameters that makes the SVM match a given architecture. It captures the common characteristics of stream processors and abstracts out any implementation differences. The SVM API provides separation of control and data-intensive code, explicit communication via streams, and explicit memory management for streaming data. Second, we demonstrate how to describe advanced graphics processors (GPUs) using the SVM architecture model. While GPUs were not designed specifically for streaming, SVM can closely capture their behavior for stream applications. Finally, we verify the validity and usefulness of the abstract model by comparing performance estimates for stream applications described at the SVM level to actual execution times on available stream processors.

Section 2 presents the underlying architecture model for the SVM. Section 3 summarizes the application programmer interface (API) for SVM code. In section 4, we demonstrate how to derive the SVM architecture model parameters for two GPUs using a series of micro-benchmarks. Section 5 presents the experimental validation of the SVM model. Finally, Section 6 discusses related work and Section 7 concludes the paper.

2. The SVM Architecture Model

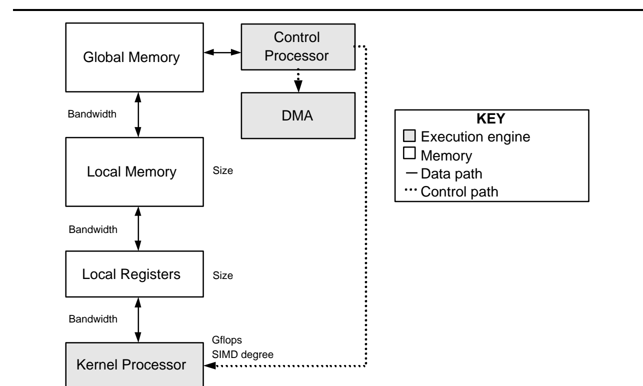

Figure 2. SVM Architectural Model

A simple example architecture model of the SVM shown in figure 2 contains three different types of execution engines, each with its own thread of control. The **Control Processor** is the master processor and controls the operation of the entire machine. The control processor essentially issues all the operations that the machine executes. The **Kernel** and **DMA** engines simply allow the control processor to execute higher level operations. The kernel processor accelerates the computation of each of the stream kernels, while

the DMA engine is used to manage the data movement that each kernel requires.

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. It also explicitly provides the dependence between all the stream operations to each of execution engines. Thus the execution engines are able to reorder computation for more efficient execution. Local memory provides the needed space to buffer data between different execution engines.

For the SVM to work as a good parametrized architectural model, one must be able to abstract a wide class of stream machine into this model, while maintaining the ability to estimate the performance of the real machine. While clearly modeling the computational performance of the engines is important, it is also important to model the size and bandwidths of the memory in the machine. A SVM has three types of components here listed with their important parameters:

Processors can have multiple master 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 operating frequency, mix of functional units, number of registers 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.Stream processors take advantage of high bandwidth local RAM memories or FIFOs that link stream processors together to reduce demands on global memory bandwidth through re-use and producer-consumer locality.

Network Links 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.

3. The SVM API

A SVM can "execute" a C program that uses the SVM API to specify:

1. How computation is partitioned and assigned to stream processors.
2. How data is partitioned and assigned to locations in local memories.
3. How data is moved among local memories or between local memories and the global memory.
4. How computations on stream processors and/or data movements are synchronized.

The SVM API has several usability objectives (in descending order of importqance):

1. Support efficient translation of API calls for varied streaming architectures.
2. Express mapping using standard C to enable low-level C compiler to parse, analyze, and translate the API calls with minimum modification.
3. Require only local analysis of single processor code by LLC to translate calls.
4. Allow simple construction of functional simulator through direct implementation of calls for easy verification or performance estimation.

In essence, an application expressed for a SVM using C and the SVM API consists of one main() function for each master processor in the target SVM. Each main function is executed as the *control thread* for that processor. Control threads synchronize using a thread virtual machine API that provides standard synchronization constructs (e.g., barrier, etc.). The control thread running on a master processor 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 dependences.

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. Both conventional and stream programming languages can be compiled to the SVM API. 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.

3.1. SVM API Constructs

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.

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,
    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 scatter and gather (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:

1 Some SVM calls simplifi ed/modifi ed here and in example for explanatory purposes.

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

Kernel dependences: The SVM API uses explicit dependences 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.

3.2. SVM API Example

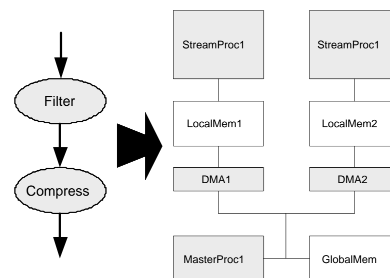

Figure 3. Simple example application and target SVM

Consider mapping an application to a simple SVM as shown in Figure 3. The application filters then compresses

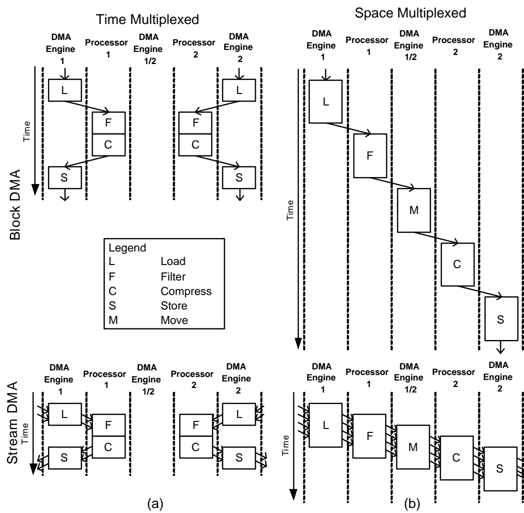

Figure 4. Possible mappings of example

the image by a constant ratio. The simple SVM consists of one master processor and two stream processors as shown in Figure 3. 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 (Figure 4a), or space-multiplexed such that one processor executes each kernel on all the data (Figure 4b). Data transfers could use blocks or streams for DMA. The SVM API can be used to specify any of the four mappings shown, as well as others not shown. Table 1 shows commented SVM API code for the time-multiplexed blocked and space-multiplexed streaming mappings.

3.3. 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. A simple RISC processor is smaller than a stream processor, but the high ALU:overhead ratio of a stream processor offers much higher efficiency (ops per unit area or power) for code with high data-level- or instruction-level- parallelism. 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 a stream processor before the previous kernel finishes. Explicitly encoding the dependences 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 API supports two kinds of DMA: block DMA and streaming DMA. Block DMA moves each large block of data as an atomic operation. Streaming DMA moves data between FIFO queues. Other kernels may be producing and consuming data to and from those queues simultaneously since single consumer/producer queues do not require heavy weight synchronization. Streaming DMA can reduce the latency of a series of kernels, especially when space-multiplexing is used as shown in Figure 4. Most contemporary DMA engines do not directly support streaming DMA, but it can be implemented with block transfers as double- (or N-degree) buffering coordinated by either the master or stream processor.

The SVM API was developed for use with a larger class of architectures than the SVM model presented in this paper. For instance, the stream construct can be mapped directly to an inter-processor FIFO in the Raw architecture [20].

4. SVM Characterization

In the two-level compilation model, the HLC generates SVM API 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 2).

For stream processors that implement an execution model similar to that of the SVM like the Imagine [8] and Merrimac [5] stream processors, the values for the SVM parameters are fairly obvious by looking at their block diagram. Nevertheless, for processors that use alternate organizations, the SVM parameters that characterize their behavior with stream applications are less apparent and require an experimental approach. In this section, we use graphics processors (GPUs) as an example of how to characterize non-conventional processors at the SVM level.

Time-multiplexed, block DMA	Space-multiplexed, streaming DMA

Time-multiplexed, block DMA

```
// Declare blocks for input, intermediate, and output data
svm_Block inputB1,    inputB2;
svm_Block unfilterB1, unfilterB2;
svm_Block filterB1,   filterB2;
svm_Block compressB1, compressB2;
svm_Block outputB1,   outputB2;
// Declare kernels to load, filter, compress, and store data
svm_MoveB2B load1,        load2;
Filter      filter1,      filter2;
Compressor  compressor1,  compressor2;
svm_MoveB2B store1,       store2;
// Initialize blocks
svm_blockInit(&inputB1,    GMEM,  gin,    IMGSZ/2, PIXSZ);
svm_blockInit(&inputB2,    MEM,   gin,    IMGSZ/2, PIXSZ);
svm_blockInit(&unfilterB1, LMEM1, LADDR1, IMGSZ/2, PIXSZ);
svm_blockInit(&unfilterB2, LMEM2, LADDR1, IMGSZ/2, PIXSZ);
svm_blockInit(&filterB1,   LMEM1, LADDR2, IMGSZ/2, PIXSZ);
svm_blockInit(&filterB2,   LMEM2, LADDR2, IMGSZ/2, PIXSZ);
svm_blockInit(&compressB1, LMEM1, LADDR1, IMGSZ/8, PIXSZ);
svm_blockInit(&compressB2, LMEM2, LADDR1, IMGSZ/8, PIXSZ);
svm_blockInit(&outputB1,   GMEM,  gout,   IMGSZ/8, PIXSZ);
svm_blockInit(&outputB2,   GMEM,  gout,   IMGSZ/8, PIXSZ);
// Load input data into local memories
svm_moveB2BInit(&load1, DMA_ENGINE1, inputB1 unfilterB1);
svm_moveB2BInit(&load2, DMA_ENGINE2, inputB2, unfilterB2);
svm_kernelRun(&load1);
svm_kernelRun(&load2);
// Filter data in local memories after it has been loaded
filterInit(&filter1, PROC1, unfilterB1, filterB1);
filterInit(&filter2, PROC2, unfilterB2, filterB2);
svm_kernelAddDependence(&filter1, &load1);
svm_kernelAddDependence(&filter2, &load2);
svm_kernelRun(&filter1);
svm_kernelRun(&filter2);
// Compress data in local memories after it is filtered
compressorInit(&compressor1, PROC1, filterB1, compressB1);
compressorInit(&compressor2, PROC2, filterB2, compressB2);
svm_kernelAddDependence(&compressor1, &filter1);
svm_kernelAddDependence(&compressor2, &filter2);
svm_kernelRun(&compressor1);
svm_kernelRun(&compressor2);
// Store compressed data to global memory
svm_moveB2BInit(&store1, DMA_ENGINE1, compressB1, outputB1);
svm_moveB2BInit(&store2, DMA_ENGINE2, compressB2, outputB2);
svm_kernelAddDependence(&store1, &compressor1);
svm_kernelAddDependence(&store2, &compressor2);
svm_kernelRun(&store1);
svm_kernelRun(&store2);
// Wait for data to be stored
svm_kernelWaitMultiple(&store1, &store2);
```

Space-multiplexed, streaming DMA

```
// Declare streams for input, intermediate,
// and output data
svm_Stream inputS;
svm_Stream unfilterS;
svm_Stream filterS;
svm_Stream uncompS;
svm_Stream compressS;
svm_Stream outputS;

// Declare kernels to load, compress, move, filter,
// and store data
svm_MoveS2S load;
Filter      filter;
svm_MoveS2S move;
Compressor  compressor;
svm_MoveS2S store;

// Initialize streams. Streams do not need to hold all
// data at once just some buffer.
svm_streamInit(&inputS,    GMEM,  gin,    IMGSZ,   PIXSZ);
svm_streamInit(&unfilterS, LMEM1, LADDR1, BUFSZ,   PIXSZ);
svm_streamInit(&filterS,   LMEM1, LADDR2, BUFSZ,   PIXSZ);
svm_streamInit(&uncompS,   LMEM2, LADDR1, BUFSZ,   PIXSZ);
svm_streamInit(&compressS, LEM1,  LADDR2, BUFSZ/4, PIXSZ);
svm_streamInit(&outputS,   GMEM,  gout,   IMGSZ/4, PIXSZ);

// Load input data into local memories
svm_moveS2SInit(&load, DMA_ENGINE1, inputS, unfilterS);
svm_kernelRun(&load);

// Filter data in local memories after it has been loaded
filterInit(&filter, PROC1, unfilterS, filterS);
svm_kernelRun(&filter);

// Store compressed data to global memory
svm_moveS2SInit(&move, DMA_ENGINE1, filterS, uncompS);
svm_kernelRun(&move);

// Compress data in local memories after it is filtered
compressorInit(&compressor, PROC2, uncompS, compressS);
svm_kernelRun(&compressor);

// Store compressed data to global memory
svm_moveS2SInit(&store, DMA_ENGINE2, compressS, outputS);
svm_kernelRun(&store);

// Wait for data to be stored
svm_kernelWait(&store);
```

Table 1. SVM API code for two possible mappings of example

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 tens 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 generally programmable. Current generation GPUs allow application programmers to write code for two parts of the graphics pipeline, the vertex engine that typically operates on geometric vertices [11] 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 engines in assembly or low level languages like Cg [12].

The basic components of a personal computer with a programmable GPU fit within the SVM architecture model as shown in Figure 5. The main CPU is the control processor, the memory controller is the DMA engine, the main memory is the global memory, the frame buffer as the local memory, and the fragment engine with its registers as the stream processor with its local register file. Hence, it is reasonable

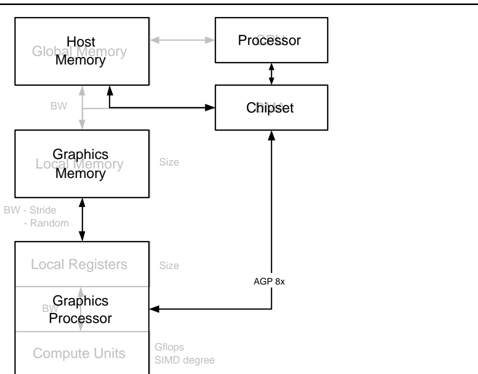

Figure 5. Graphics Processors mapped to SVM

to target GPUs as stream processors 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.

4.2. Characterization Methodology

In this study, we investigate the SVM model parameters for the best graphics processors currently available, the ATI Radeon 9800 Pro and the Nvidia GeForceFX 5900 Ultra.

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, their instruction set does not include any control flow instructions such as branches [2], 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 spatial 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.

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.

4.3. Micro-Kernel Analysis

4.3.1. Compute Unit Figure 6 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 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.

4.3.2. 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 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 7

2 The opposite constraint is true for the vertex engine.

3 Instruction description available in [2]

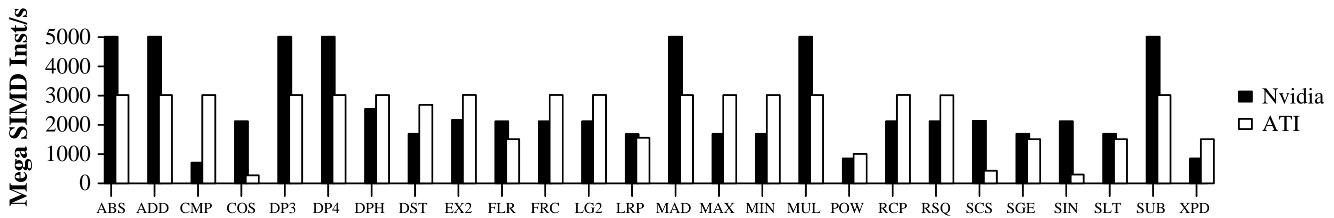

Figure 6. Instructions per second

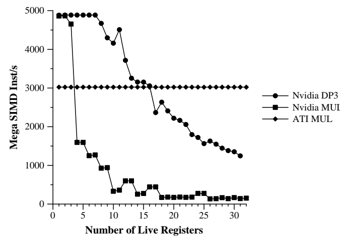

Figure 7. Impact of Live Registers on FLOPS

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 7. 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 bandwidth issue. For the ATI chip, the register bandwidth is roughly 179 GB/s. Nvidia's local register bandwidth is a function of the number of live registers

4.3.3. Local Memory The local stream memory on the GPU is the graphics memory which is nominally 256MB on current 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 DSP processors. Unfortunately, if we truly are streaming data out of the local memory, and only reading it once, this cache will not improve memory bandwidth.

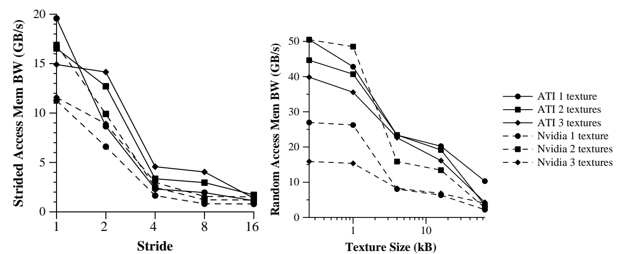

Figure 8. Memory Bandwidth

Figure 8 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 seems to be optimized for accessing a single texture at unit stride (19.6 GB/s) while Nvidia is optimized for two textures at unit stride (16.9 GB/s).

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 like for a SVM block. 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 8 also 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.

Both ATI and Nvidia hardware behave as expected increasing the effective memory bandwidth when the accessed texture is small. The drop in bandwidth for both architectures from 1kB to 4kB leads us to believe that the texture cache size is around that value. The Nvidia hardware is

again optimized for two textures, while ATI achieves peak performance with one.

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.

4.3.4. 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.

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.

4.4. SVM Parameters

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

Table 2. 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 2. The complex nature of GPUs requires us to consider expanding our performance parameters slightly to capture the dependence of the Nvidia machine's performance to the number of live registers.

5. SVM Validation

Our goal is to show that the SVM is a reasonable intermediate format that can be targeted by a HLC to give high performance for a given stream architecture. We have characterized a non-stream specific architectures like GPUs for our SVM machine model and this section uses this model to estimate their performance on a few applications.

5.1. Methodology

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 test architectures are Imagine [8], 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 [8] run on its cycle accurate simulator.

5.2. Validation Results

Three applications 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. In all cases, the SVM provides a good estimate of the model machine performance.

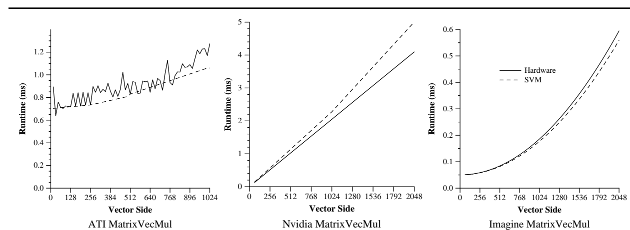

Figure 9. Matrix Vector Multiply run-times

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 9 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 application 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 an application which shows almost identical performance behavior on all 3 architectures as shown in Figure 10. Although the SVM simulator is not very accurate for the small dimensions it tracks well the performance of larger data-sets

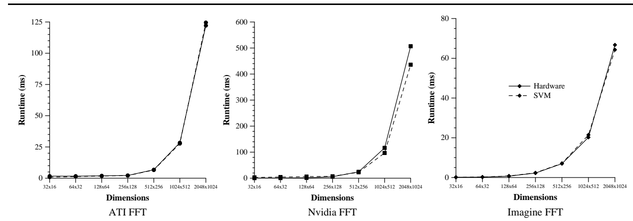

Figure 10. 2D FFT run-times

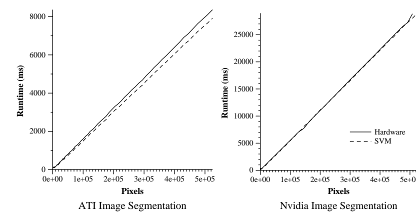

Figure 11. Image Segmentation run-times

Figure 11 compares Image Segmentation both GPUs to their SVM. This application 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.

6. Related Work

Stream programming and languages inherit a lot from the body of work on synchronous data-flow programming environments [10]. Generally stream programming languages strove to express data-level parallelism and the separation of computation and communication such that a compiler can reason about the program without losing too much expressiveness.

More recent work on stream programming languages like StreamIt [21] constrain the program to synchronous data-flow to enable powerful compiler optimizations. The Brook streaming language [3] augments C with concepts of streams and kernels. Kernels are fully data-parallel without carried state, while streams can only be manipulated through defined stream operators.

Data-parallel architectures have made a come-back recently due to the diminishing returns of ILP gains in a single thread. Vector processors whether for embedded [9] or scientific purposes share a lot of common benefits with dedicated stream architectures for media [8] or scientific [5] applications. Some other architectures like Raw [20] exploit

streams by mapping an application spatially across homogeneous simple processors.

This is not the first time that GPUs have been used for general purpose computation whether by porting complete a single application [17] or by creating a general purpose framework to program GPUs[4]. This works differs in that it uses GPU as an example architecture targeted for the stream programming model through an intermediate representation, the SVM, without loss of performance.

7. Conclusion

We have presented the Stream Virtual Machine (SVM) as an intermediate form that can represent diverse architectures targeted by a stream programming model. The SVM architectural model defines the essential characteristics of a stream machine. These parameters specify the size and bandwidths of the memories, and the computation rates of the processors. We extract these parameters by using microkernels run on targeted machines with no public architectural details.

The initial results are quite promising. With these limited parameters, the SVM model is able to capture the performance trend quite accurately and is able in most cases to match the absolute performance. Our next step is to create a compilation system that uses this model as an intermediate form.

8. Acknowledgments

We would like to acknowledge the Morphware Forum for its pioneering and continuing effort in creating a standard for The Streaming Virtual Machine. The Morphware Forum is a collaboration between several institutions as part of the DARPA Polymorphic Computing Architectures program. The primary authors of the full SVM Specification [15] are: Peter Mattson, William Thies, Lance Hammond, and Mike Vahey. Additional contributions were made by the following organizations: University of California Information Sciences Institute, University of Texas at Austin, and IBM Austin Research Laboratory.

References

[1] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate versus ipc: the end of the road for conventional microarchitectures. In *Proceedings of the 27th annual international symposium on Computer architecture*, pages 248–259. ACM Press, 2000.

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

[3] I. Buck. Brook specifi cation v0.2. October 2003.

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

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

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

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

[8] U. Kapasi, W. J. Dally, S. Rixner, J. D. Owens, and B. Khailany. The Imagine stream processor. In *Proceedings 2002 IEEE International Conference on Computer Design*, pages 282–288, sep 2002.

[9] C. Kozyrakis. Scalable vector media-processors for embedded systems. Technical report, 2002.

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

[11] E. Lindholm, M. J. Kligard, and H. 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.

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

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

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

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

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

[17] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics*, 21(3):703–712, July 2002. ISSN 0730-0301 (Proceedings of ACM SIGGRAPH 2002).

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

[19] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ilp,

tlp, and dlp with the polymorphous trips architecture. In *Proceedings of the 30th annual international symposium on Computer architecture*, pages 422–433. ACM Press, 2003.

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

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