Computer Sciences Department

Signature Matching in Network Processing using SIMD/GPU Architectures

Neelam Goyal
Justin Ormont
Randy Smith
Karthikeyan Sankaralingam
Cristian Estan

Technical Report #1628

January 2008

Signature Matching in Network Processing using SIMD/GPU Architectures

Neelam Goyal Justin Ormont Randy Smith Karthikeyan Sankaralingam
Cristian Estan

Department of Computer Sciences, University of Wisconsin–Madison
{neelam,ormont,rsmith,karu,estan}@cs.wisc.edu

Abstract

Deep packet inspection is becoming prevalent for modern network processing systems. They inspect packet payloads for a variety of reasons, including intrusion detection, traffic policing, and load balancing. The focus of this paper is deep packet inspection in intrusion detection/prevention systems (IPSes). The performance critical operation in these systems is signature matching*: matching payloads against signatures of vulnerabilities. Increasing network speeds of today's networks and the transition from simple string-based signatures to complex regular expressions has rapidly increased the performance requirement of signature matching. To meet these requirements, solutions range from hardware-centric ASIC/FPGA implementations to software implementations using high-performance microprocessors.*

In this paper, we propose a programmable SIMD architecture design for IPSes and develop a prototype implementation on an Nvidia G80 GPU. We first present a detailed architectural and microarchitectural analysis of signature matching. Our analysis shows that signature matching is well suited for SIMD processing because of regular control flow and parallelism available at the packet level. We examine the conventional approach of using deterministic finite automata (DFAs) and a new approach called extended finite automata (XFAs) which require far less memory than DFAs, but require scratch memory and small amounts of computation in each state. We then describe a SIMD design to implement DFAs and XFAs. Using a SIMD architecture provides flexibility, programmability, and design productivity which ASICs lack, while being area and power efficient which superscalar processors lack. Finally, we develop a prototype implementation using the G80 GPU as an example SIMD implementation. This system out-performs a Pentium4 by up to 9X and shows SIMD systems are a promising candidate for signature matching.

1. Introduction

In the last two decades, the devices handling Internet traffic have been extended with a variety of features that give them stronger control over the network traffic. As a result of this trend, many forms of deep packet inspection are supported by today's routers and switches. Unlike earlier forms of processing based on packet header fields, deep packet inspection requires the reading of the much larger packet payload. Coupled with increases in network speeds this leads to tremendous increases in the amount of processing required. Fortunately parallelism is plentiful as many network packets can be processed independently. Several solutions exist for handling the processing needs of modern network devices ranging from ASIC hardware to high performance microprocessors. Table 1 outlines these tradeoffs which we discuss in detail in Section 2.3. While ASICs provide high area/energy efficiency and performance, they have poor flexibility and high system design cost since hardware and software must be carefully co-designed. At the other end of the spectrum, general purpose microprocessor based systems are very flexible and have low design costs, but cannot match the performance of specialized hardware. An ideal platform should provide high performance, programmability and flexibility, while leveraging off of available hardware or architectures, without requiring specialized design.

Network processors like the Intel IXP, IBM PowerNP, and platforms like Raza's Thread Processors, specialize a superscalar architecture with networking-specific functionality to increase efficiency without sacrificing programmability or design cost. They all utilize extensive multi-threading to exploit the data-level parallelism found at the granularity of packets to hide memory latencies. This threading abstraction means *independent* control-flow and temporary data management for each thread. This independent control flow means repeated execution of the same set of instructions (on separate processors, or multiple times on the same processor for each thread) even if the same computation must be performed on many packets.

Examining various forms of deep packet inspection we see that often the same processing is applied to each packet, making the Single Instruction Multiple Data (SIMD) paradigm ideal for this type of processing. A single instruction can simultaneously perform operations for many packets provid-

ing large savings in power and area overheads. As shown in Table 1 SIMD processing can achieve area/performance efficiencies much higher than superscalar processors. Their rate of performance improvement is high owing to design simplicity. They are abundantly available as GPUs and hence have low design costs. With new processors such as Intel's Larrabee and AMD's Fusion integrating the CPU and the SIMD engine on the same chip, we expect it to become even cheaper to add SIMD processing to a system. With high performance microprocessors moving towards SIMD-like organizations, mapping signature matching to a SIMD-like organization will be essential to get high performance from future general purpose processors. Today's choices for IPSes are ASICs (or network processors) for high performance at a high cost, or general purpose processors for low performance at low cost. Using SIMD architectures and specifically GPUs introduces a new design point - IPSes with potentially high performance at low cost. However, the design simplicity of SIMD platforms is offset by programmability challenges. *The key questions in applying SIMD processing in deep packet inspections are: (1) Can they be easily programmed? and (2) Can we obtain high performance from these systems and realize the performance potential?*

To help us understand the viability of SIMD GPUs as a platform for the high-throughput processing needs of modern network devices, in this paper we focus on a specific form of deep packet inspection: *signature matching*, the most processing-intensive operation of intrusion prevention/detection systems (IPSes). These systems have a large number of signatures describing various types of attacks (e.g. buffer overflows). Their goal is to protect the computers within their network by filtering out packets that attempt to perpetrate one of the attacks described by the signatures. The core operation in signature matching is determining whether any of the regular expressions from the signature database match the payload of a given packet. Other applications like virus scanning and XML processing are similar in the computation they perform. In this paper, we show that SIMD processing is well suited to signature matching and show a prototype implementation on a GPU. The main contributions of this paper are:

- A fundamental analysis and characterization of the signature matching problem for IPSes in terms of the control-flow, memory accesses, and available concurrency.
- Demonstration of the SIMD paradigm for signature matching.
- A proof-of-concept SIMD implementation. We develop a fully functional prototype of a signature matching module on a Nvidia G80 GPU. This prototype out-performs a Pentium4 based software system by up 6X to 9X.

The remainder of this paper is organized as follows. Section 2 describes the signature matching problem, solutions, and a detailed analysis. Section 3 shows a SIMD design for signature matching and describes our prototype GPU implementation. Section 4 discusses performance results, Section 5 discusses related work, and Section 6 concludes.

2. Signature Matching

Stand-alone IPSes operate at speeds ranging from tens of megabits per second to gigabits per second. Intrusion prevention functionality is also present in routers and switches ranging from low-cost low-speed models to expensive multi-gigabit products. Signature matching may be performed by a central processor akin to the CPUs of workstations, by a processor on an interface card or on a special purpose blade, or by an ASIC. Besides signature matching, intrusion prevention requires other operations such as flow reassembly and various forms of preprocessing. There are great differences among these network devices in terms of the operations they perform and the size and complexity of the signature databases they can support. But it is widely accepted that, for the entire range of solutions, signature matching is the most processing-intensive among the operations required for intrusion prevention. As the number and complexity of signatures are steadily and rapidly increasing, we expect signature matching to become even more of a performance bottleneck in the future. For example Cabrera et. al. [6] reported in 2004 that signature matching accounts for approximately 60% of the processing time of the open source IPS Snort. Since then, the number of signatures increased from 1,458 to 5,464.

While performance and cost are the primary concerns for the builders of devices implementing intrusion prevention, flexibility is also an important consideration. All devices support updates to the signature database, but often more flexibility is required: the ability to change the various operations performed. This can help fix bugs uncovered after the device was deployed and more importantly counteract radically new attacks or evasion techniques. In such cases updating the signature database may not be enough and updates to preprocessing operations or other parts of the system may be required.

2.1 Implementation Tradeoffs

The architects of network devices performing signature matching have to balance a large number of often contradictory goals when designing the signature matching module: meeting performance targets, minimizing the unit cost, staying within the power budget and form factor imposed by the overall architecture of the device, minimizing time to market and maximizing the flexibility to update functionality after the device is deployed. The platforms typically considered are: ASICs, FPGAs, network processors, and general purpose processors. Table 1 shows a comparison of different metrics for these platforms.

ASICs are typically used in the highest speed devices because they are the only platform able to meet the stringent performance requirements. At slightly lower speeds FPGAs

Parameters	ASIC	FPGA	SIMD	Network processors	General purpose microprocessors
Physical constraints					
Cost	Highest	Medium	Low	Medium-Low	Low
Power Efficiency	Highest	Low-Medium	High	Medium	Lowest
Area Efficiency	Highest	Worst	High	Medium	Low
System design					
Flexibility	Worst	Medium	?	Medium	Best
Design time	Highest	Medium	?	Low	Lowest
Performance					
Peak performance	Highest	Medium	Medium	Medium	Lowest
Application Performance	Highest	Medium	?	Medium	Lowest

Table 1. Implementation alternatives for signature matching. Question marks indicate investigation in this paper.

become a viable alternative if flexibility is valued much, but they consume significantly more power than ASICs. Network processors and multicore processors targeted to network applications offer lower throughput, but they have cost and power advantages. General purpose processors are the platform with the lowest performance, but best in terms of cost, flexibility and time to market. Note that the price of a general purpose processor may be higher than that of a network processor or an ASIC, but all networking devices that have intrusion prevention functionality already contain a general purpose processor, so if the existing processor can be used for signature matching, the added cost is zero.

SIMD architectures may prove to be a platform well-suited for signature matching. Their simple structure allows them to be more efficient than network processors and general purpose processors, and they have 30 to 40-times higher peak performance. Their programability makes them more flexible than ASICs. The fact that SIMD processors are already mass-produced for GPUs makes them cheaper and allows them to be built with the most recent fabrication processes. Further investigation is needed for some of the metrics as shown by the question marks in the Table. It is not clear whether this architecture is flexible enough to support the types of operations required in signature matching, nor is it clear that it will maintain its good performance if the way to map these operations to the architecture is too cumbersome. This paper gives quantitative answers to these questions. We show that GPUs can provide significantly higher performance at a cost similar to general purpose processors.

2.2 Application description

Due to their ability to make finer distinctions than simple string matching, regular expressions are today the primary method for defining IPS signatures. Since each signature describes a family of attacks against a specific application, individual signatures are associated with the port number the application runs on. The core operation in signature matching is determining whether a given packet matches any of the regular expressions associated with the destination port of the packet. Matching the expressions individually is too slow, so current implementations, such as Cisco IPS [1], use a combined representation for the entire set of signatures in the form of deterministic finite automata (DFAs). Besides the DFA-based method which maps very naturally to a SIMD architecture we also evaluate a recently proposed method based on extended finite automata (XFAs) which has been shown to have significantly better performance than DFAs on general purpose processors. The per byte processing is more complex and less uniform than for DFAs, thus this is a more challenging workload for a SIMD architecture.

2.2.1 Using DFAs to match a set of regular expressions

DFAs consist of a number of states each of which has a transition table with 256 pointers to other states. During matching a "current state" variable is repeatedly updated and it moves between the states of the DFA. Some of these states are marked as "accepting states" and indicate that the signature matches. DFAs are are not a compact way of representing regular expressions, but they have two major advantages: there is a fast matching procedure, and it is possible to compose the DFAs corresponding to multiple signatures into a combined DFA that recognizes all signatures in a single pass. The basic processing steps of a DFA are very simple. The DFA reads a byte from the input, uses it to index into a transition table pointed to by the current state variable, and the value it reads out is the next state. If this is an accepting state the DFA raises an alert which is handled by other parts of the system. These steps are repeated until the end of the input is reached.

Unfortunately, with the types of regular expressions used by IPS signatures, when DFAs representing individual signatures are combined the composite DFA is much larger than the sum of the sizes of the DFAs for individual signatures and often exceeds available memory. One approach [23] to reducing the memory footprint combines signatures into multiple DFAs rather than one. This induces a space-time trade-off between memory and inspection time: the larger the memory budget, the fewer DFAs are required (and thus the faster the matching). In this paper we use multiple DFAs and we set a memory budget of 64 MB per signature set

which leads to the use of between one and seven DFAs to represent the full signature set.

2.2.2 Using XFAs to match a set of regular expressions

The core reason for the state space explosion of DFAs is that the only way for them to differentiate between two input strings is to have them lead to distinct states. XFAs [22] are similar to DFAs, but they also use a small scratch memory in which they store bits and counters that indicate the progress of the matching operation. Individual states have small programs associated with them that update the scratch memory. When the XFA reaches an accepting state it checks the scratch memory and it raises an alert only if certain conditions are met. The use of the scratch memory allows XFAs to eliminate all major causes of state space explosion. For each of the three signature sets we considered, a single XFA using no more than 3MB of memory can match the entire signature set. The per byte processing of an XFA is more complex than that of a DFA, but when compared against multiple DFAs representing the entire signature set, XFAs are typically still faster.

The basic scratch memory operations performed by XFAs are setting or resetting a bit in a bitmap and setting, incrementing or testing a counter. When processing a byte the XFA may perform multiple such operations on separate bits and counters. To improve performance, XFAs also use a special type of counter: the offset counter. For some types of signatures the counter would need to be incremented for each input byte (for example the signature may need to see whether the number of non-newline characters after an SMTP keyword is above a threshold required to trigger a buffer overflow). Instead of using a traditional counter that would need to be explicitly incremented on each byte, XFAs use offset counters that work as follows: after the keyword is recognized, the packet offset at which the alert should trigger is stored in a sorted list; if a newline is seen before that offset, the program associated with that state disarms the offset counter, otherwise the alert is triggered when that offset is reached.

2.3 Application analysis

Figure 1 outlines the basic code executed to perform string matching. Each packet is extracted and passed on to a string matching routine which implements a DFA or XFA to detect intrusions. We show a single routine that can be used for both DFAs and XFAs. For a DFA, lines marked "`no effect for DFA`" are not executed. In this section, we analyze the basic properties of this code and its effect on the microarchitecture by running it on an Intel Core2 processor and examining performance counters. For the quantitative results presented in this section, we used representative traces and signatures whose details we describe in Section 4.

The four main components of the signature matching problem are: (1) a state machine that describes the set of patterns to be detected, (2) auxiliary data that must be maintained for each packet as it is processed, (3) a packet buffer that contain the packet data, and (4) an interpreter that reads packet input, walks through the state machine, and updates the auxiliary data. The interpreter code is compiled to generate a sequential instruction stream or a wide-SIMD instruction stream where a packet is processed on each processing element of the SIMD architecture.

2.3.1 Memory

Analysis: The four main data structures are the packet buffer, the state machine data structure, instructions and temporary data associated with each packet as it is processed, and the offset list (last two not present for DFAs). The packet buffer is typically several megabytes and depending on the implementation, some form of DMA or IO access is used to copy contents in a batched fashion from the network link into the processor's memory space. Accesses to the packet buffer are regular and in multi-threaded implementations the packet buffer may be logically partitioned, with different threads simultaneously operating on different packets. The state machine data structure can be several gigabytes in size for DFAs depending on the number of signatures that are to be matched. To avoid the state space explosion of combining signatures, XFAs augment each state with local variables and a few simple *instructions*. For XFAs the state machine data structure is typically a few megabytes and in our measurements always less than a 3MB. Accesses to this state machine data structure are irregular and uncorrelated as they are driven by packet data input. The instructions are local to a state, and the temporaries and the offset list are by local to a packet and less than a kilobyte in size. Accesses to these structures is also typically irregular. Since the state machine is the largest data structure, it contributes most to memory access latencies.

Quantitative results: To study the memory access behavior to this structure we instrumented our implementation to measure how many times every state is visited while processing a network stream for a given set of signatures. Every state contains a transition table (indexed by a character and containing a pointer to the next state) and other auxiliary data, amounting to a total of slightly more than 1024 bytes of memory per state. Since the DFAs have a large number of states and hence excessive memory requirements, we used multiple DFAs in these experiments and set their data size to 64 MB.

Figure 2 shows the frequency at which each state is visited sorted from high to low. The right hand side figure shows DFAs and the left hand side figure shows XFAs. We can clearly see a 90/10 rule here: for most of the signature sets, less than 10% of the states contribute to more than 90% of the visits. This is why IPSes built with high performance microprocessors perform well even for large XFAs and DFAs — although their overall memory requirement is large, the working set size consists of only hundreds of

```
APPLY(state_machine_t *M, unsigned char* buf, int len) {
 state* curState = M.start
 execInstrs ( curState->instrs)
 // Level-2 control flow
 for i = 0 to len do
    curState = curState->nextState(buf [i])
    // Level-3 control flow
    execInstrs ( curState->instrs);       // checks accepting state for DFA
                                          // executes instructions for XFA
    // Check the offset list
    while (offsetList->head.offset==i ) do // no effect for DFA
      execInstr (offsetList->head->instr)  // no effect for DFA
      offsetList->head = offsetList->head.next // no effect for DFA
}

state_machine_t *M = read_signatures();
trace = read_input_trace(trace);
// Level-1 control flow
for each packet in trace;
    char *buf = packet.bytes;
    APPLY(M, unsigned char* buf, packet.length)
```

Figure 1. DFA and XFA processing

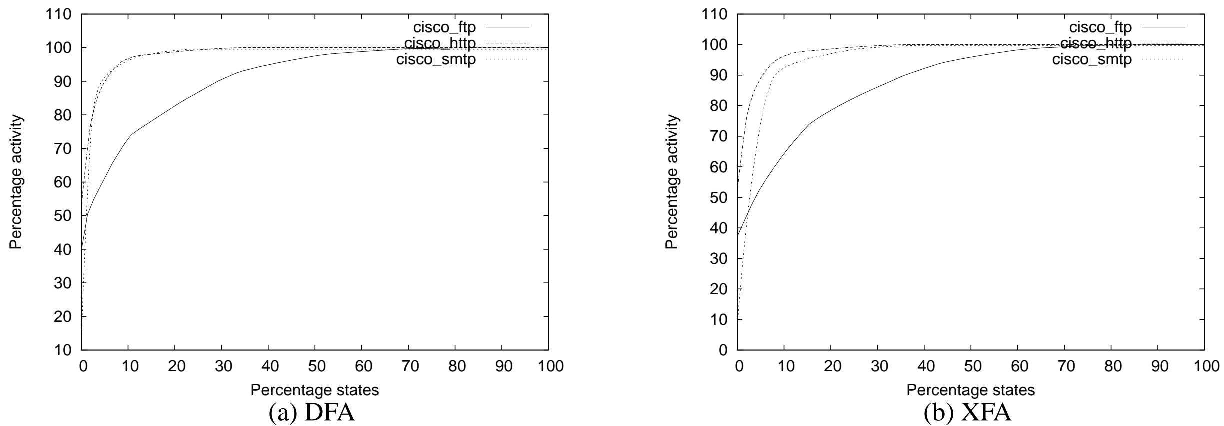

Figure 2. Frequency distribution of states visited

states. The FTP protocol alone exhibits anomalous behavior with a large fraction of states being visited with high probability.

2.3.2 Control-flow

Analysis: The string matching application has control-flow at three granularities. First, at the coarse-grain the `apply` function is repeatedly called for each packet in the trace. Second, this function loops over every character in a given packet. For a DFA, the loop body is a single large basic block. For XFAs, a third level of control-flow exists: for every state the interpreter must loop over every instruction for that state, and then loop over the entries in the offset list. While processing the instructions (the `exec_instr` function) many conditional branches arise.

This type of regular control is well suited for a SIMD architecture, with each processing element handling a packet. Minimizing divergence in control flow is crucial to exploiting the benefits of the SIMD design. Since control is predictable and regular at the first level, it can be exploited by distributing packet data to each processing element. A single *interpreter* instruction, then processes data for each packet exploiting predictability at the second level. When packet sizes vary, control-flow at this level will diverge, as some packets will complete processing before others. As described in detail in Section 3 we sort packets to minimize divergence of control flow at this level. Thus, on a SIMD de-

```
for (i=0; i < state->num_instrs_; i++)
    switch(state->instrs_[i]->type) {
        case OFFSET_SET: ...
        case OFFSET_RESET: ..
        ....
        case BIT_CLEAR:
        case BIT_SET:
        case BIT_TEST:
    }
```

Figure 3. Control flow in executing an XFA's instructions

Protocol	Measured on Core2				Simulated SIMD	
	Prediction accuracy		ILP		% Divergence	Average divergence
	DFA	XFA	DFA	XFA		
ftp	97.5	97.6	1.52	1.52	2.3%	2
http	99.2	99.3	1.83	1.80	0.3%	2
smtp	98.3	98.2	1.45	1.46	61%	2.21

Table 2. Control flow behavior and available concurrency

sign, DFAs have almost no divergence in control flow, since they have only these two levels of control flow.

The third level of control flow which exists only for XFAs, will inevitably diverge. Figure 3 outlines the code for executing instructions in XFAs focusing on the points where control flow could diverge for different packets. Depending on the state that a packet is in, its instruction could be different and hence it could take different branches in the control flow graph. The key question is for real signatures and real network traffic what is common case behavior and how often does divergence occur.

Quantitative results: To understand the effects of control flow we measured branch prediction rates on a Core2 system and instrumented our interpreter to measure the divergence. Table 2 shows the branch prediction accuracy in the second and third columns for different protocols for both DFAs and XFAs, when processing real traces. Prediction accuracies are measured using performance counters in the processor. We see that the prediction accuracies are really high demonstrating that control flow patterns are repeated.

Columns six and seven show divergences in control flow for the protocols. We calculate divergence percentage for groups of 32 packets, because 32 is the SIMD width of our prototype implementation. We define divergence as follows: If the ith character of packet in any PE takes a conditional branch different from any other PEs then that SIMD instruction is said to have diverged and a divergence value of 1. Divergence percentage is the percentage of such instructions compared to the total number of SIMD instructions executed. The last column has the average divergence among the instructions that diverge. If 2 packets take different directions then the divergence is 2. First we see that the percentage of SIMD instructions that diverge is small for ftp and http, but more than 50% for SMTP. However, when there is a divergence, the average divergence is only 2 implying that on average 30 PEs still execute the same instructions. As a result, SIMD with low overhead branching support can perform well even for behavior like SMTP.

While traditional SIMD designs did not have support for such types of control flow within a SIMD instruction, such support can be added at a performance penalty for such branches. The Nvidia G80 SIMD architecture supports such type of control flow with a penalty of 31 cycles in addition to the extra cycles to execute the divergent control path, according to our measurements.

2.3.3 Concurrency

Signature matching is heavily data-parallel and contains abundant parallelism since each packet can be processed independently. Within a packet's processing, the level of concurrency is limited, since the per-byte transitions are serially dependent on each other. For XFAs, the interpreter that executes the XFA's instructions also has only limited amount of instruction-level parallelism. Columns four and five in Table 2 show the measured ILP using performance counters on a Core2 processor. We also examined the branch prediction accuracies and the cache hit rates and found them to be greater than 95%. Hence, for this application ILP is simply limited by inherent dependencies and not by control-flow or memory latencies. Investing in extracting further ILP is likely to provide diminishing returns, and only reduce area and power inefficiencies for this application.

2.3.4 Summary and IPS Requirements

To summarize, signature matching requires memories that can support efficient regular accesses, some fast accesses to a small memory, and capabilities to hide the latency of

irregular accesses. The control-flow is largely predictable and a classical SIMD architecture can efficiently support DFA processing. However XFAs can exhibit data-dependent branching — the management of the local variables like the counters and bits is dependent on the packet data. Such data-dependent branching cannot be efficiently supported with a classical SIMD organization using predication. However, such branching is infrequent and hence some form of escape mechanism that temporarily deviates from SIMD processing will suffice. And finally, at the packet level there is abundant data-level parallelism. Based on these characteristics, we now describe a SIMD design for signature matching.

3. Architecture and Implementation

In this section we first describe a SIMD architecture for signature matching that can support both DFA and XFA processing by deviating from classical SIMD execution in the uncommon case. Modern GPUs implement several primitives of this architecture and we build a prototype software implementation on an Nvidia GPU as proof-of-concept.

3.1 SIMD architecture

Figure 4a shows a high level processor organization with the mapping of the different components to the architecture for a 32-wide SIMD machine. A sorting engine reads packets off the network interface and creates groups of 32 packets which are largely similar in size. Finding 32 packets of exactly same size can introduce large latencies, whereas having unequal sized packed, introduces wasted work, as every packet will take as many cycles to process as the largest packet in the group. In our analysis, we found that examining 2048 packets was sufficient to create groups with reasonably equal sized packets.

3.1.1 Memory system

The memory hierarchy includes three types of memories: (1) Local memory: a software managed small fast memory that supports irregular accesses, (2) Regular storage: a software managed large fast memory for regular accesses, and (3) Irregular storage: a hardware managed large memory with long latencies for irregular accesses. The local memory is local to each processing element and is used to store the packet-specific auxiliary data. The regular storage is used to store the packet buffer accesses to which are regular. The irregular storage is used to store the state machine data structure.

The state machine data structure can be several megabytes in size and accesses to this irregular storage structure can take several cycles. Caching or multi-threading can hide this latency. As shown in our application analysis a small percentage of the states contribute to a large fraction of the memory accesses in the state machine. Hardware managed caching will effectively capture this working set size and our analysis on the Intel Core2 processor shows that a 32KB level-1 cache can sustain a 98% hit rate.

Alternatively, multi-threading can hide memory access latency as well, by switching to a new thread when waiting on memory. The thread management unit switches to a new SIMD thread whenever a memory request is generated. When the memory request of all of the processing elements of a SIMD thread return, it can be re-scheduled for execution. A large number of physical registers is required to support such multi-threading. Since the working set size of this application is quite small, hardware managed caching instead of such multi-threading can also hide memory latencies.

3.1.2 Control flow

Control-flow is largely predictable and a classical SIMD architecture can efficiently support DFA processing. But, to support XFAs data-dependent branching is required. First, the number of XFA instructions and the type of instruction that executes for each byte in a packet are dependent on the state. As a result, the processing elements can require different instruction streams. We use a hybrid predication and branching strategy to accomplish this control flow. Predication is used to construct a frequent subset of instructions which every processing element executes. Branch instructions are used to implement control-flow to infrequently accessed regions of the program. The branch instructions execute at every PE, and the target address of the branches across all PEs is compared. If they differ, the PEs enter a serialized mode. They are grouped based on their branch target, and PEs with the same branch target execute together. In a group of 32 for example, if one PE deviates from the others, the first 31 execute together, and then the one PE. This serialized execution continues until a control-flow merge point — an unconditional branch to the merge point that is inserted by the compiler. If there is no difference across all the PEs, SIMD execution resumes. Compared to predication, such branch instructions take longer to execute because, the addresses must be compared across all PEs, even when there is no divergence, whereas predication based control is local to each PE.

3.2 Execution flow

The overall execution flow proceeds as follows. Packets from the network interface are sorted and written to a packet buffer and a separate packet length array records the length of each packet. The interpreter running on the SIMD array processes these packets in lock-step fashion across the PEs. Each PE requests a memory access to lookup the transition tables of its state to proceed to the next state. When waiting for memory, the threading unit activates a new SIMD thread that processes a new set of 32 packets. For DFAs, execution proceeds in this fashion until all bytes in that group of packets is processed. Any intrusions detected are passed back to the IPS system using a bit-vector. We assume the SIMD array is controlled by a master processor that takes actions when intrusions are detected. For XFAs, data depen-

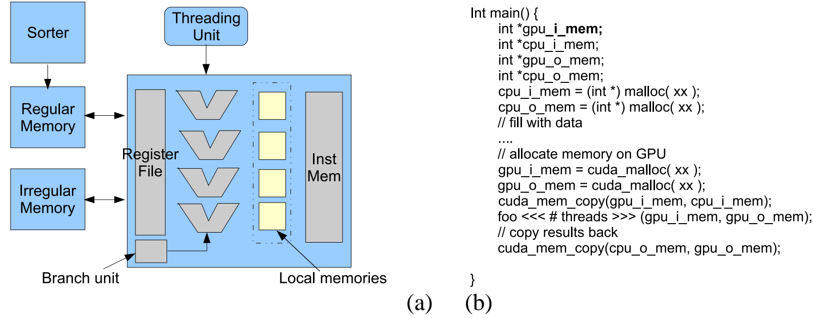

Figure 4. Architecture overview and code example for GPU execution.

dent branching can occasionally deviate from this regular flow of control.

3.3 Prototype GPU implementation

GPU architectures have traditionally included several of the components in our high level architecture — they are SIMD, multi-threaded to hide memory latency, and support fast regular memory access in the form of texture memories. However, the key to supporting XFAs is data-dependent branching when necessary. DirectX 10 compliant GPUs include support for predication and data dependent branching [10]. The Nvidia G80 GPU is implemented as a SIMD array with extensions for data dependent branching [13]. To evaluate how well the SIMD paradigm is suited for IPSes we build a prototype IPS using this GPU. We briefly describe the organization of the G80 processor and our software implementation.

3.3.1 G80 Organization

For the scope of this paper we focus only on the programmable components of the G80 chip. It is organized as a 16-core SIMD machine, with each SIMD core being logically 32-wide. The Compute Unified Device Architecture (CUDA) defines the processor architecture and a set of software drivers interface to the GPU [9]. The processor has three types of memories: a small local memory of 8KB per core that can be accessed in 1 cycle, cacheable texture memory of 128MB with a 8KB cache, and uncached memory (or global memory) of 768 MB that can take 400 to 600 cycles to access. The clock frequency of the SIMD core is 1.35GHz. It includes a threading unit that schedules threads to hide memory latencies.

The processor uses a streaming programming model and a kernel of instructions is the level of abstraction visible to the processor. The kernel consists of a set of 32-wide SIMD instructions and the threading unit (based on programmer specifications) creates a number of "threads" using the kernel's code. They are not like processor threads, but are instead simply logical copies of the kernel. The different threads execute asynchronously and need not be in lock-step or have same types of control flow, and get a partition of the local memory each. The threads are completely invisible to each other and the processor does not guarantee any synchronization through the memories[1].

3.3.2 Software implementation

We implemented a fully functional signature matching system on the G80 GPU. The programming model for the CUDA architecture is shown in Figure 4b. Any code that must execute on the GPU is defined as a C or C++ function with a special directive and this function is referred to as a kernel[2]. The main code executes on the CPU and the GPU acts as a co-processor. Special function calls copy data to/from the GPU memory and CPU memory. The `foo <<< # threads >>> (arg0, arg1, arg2)`, which gets translated into a set of system calls to the GPU device driver, triggers the execution of the kernel `foo` on the GPU. In the version of the system drivers we use, this is a blocking call.

The interpreter was written using this streaming model and the applications partitioned into two main functions: (1) `xfa_build` which builds the state machine data structure. This kernel is completely sequential and executes as a single thread utilizing one processing element in the entire chip. Because DFAs and XFAs are recursive data structure, they cannot simply be copied from CPU memory to GPU memory (pointers will be different in each address space). Each state is copied and the transitions are rebuilt on the GPU. To construct multiple DFAs or XFAs simultaneously this kernel could be parallelized. (2) The `xfa_trace_apply` which processes the packets.

In our implementation, we pass a large trace of 65536 packets to the GPU and initiate processing. Once the processing of all packets is complete, a bit-vector is passed back to the CPU indicating which packets were detected as attacks. This batch processing is an artifact of our prototype implementation and the PCI-Express based interfaces that modern PC-based system use. We envision real systems to utilize one of the two following techniques. (1) Upcoming processors like Intel's Larrabee chip and AMD Fusion are

[1] The processor includes 16KB of "shared memory" for localized synchronization of all threads within a core. We do not use any of the share memory in our implementation and require no synchronization.

[2] It may contain calls to other functions, but they must and will be inlined.

likely to have more closely integrated interfaces connecting a CPU and GPU or GPU-like processor. Hence CPU and GPU are likely to have a shared memory space. (2) Provide a shared memory in standalone IPS systems that uses such a SIMD architecture.

3.3.3 Optimizations

We started with a strawman implementation that maintained the state machine and the packet buffer in global memory and accessed the packet buffer one byte at a time. Our performance analysis showed that using texture memory and accessing larger words can provide significant performance improvements. We modified our implementation to fetch 8 bytes at a time from the packet buffer, which resulted in approximately 2X improvement compared to byte accesses. The auxiliary data is maintained in local memory at each processing element.

During performance analysis of this prototype implementation, we realized the number of registers in a kernel is a critical resource. It dictates how many of the processing elements in a SIMD core can be used. Each SIMD core has a total of 256 physical registers per PE, which can be split among the different threads. For a kernel with 32 registers, at most 8 threads can execute concurrently in a single core. This level of concurrency may not be sufficient to hide memory latencies. Reducing the number of registers used a the expense of spills to local memory was found to be beneficial for this application. We forced the compiler to never use more than 8 registers per kernel.

We found the data-dependent branching support in the G80 to be sufficiently powerful for this application. We developed a micro-benchmark that controlled the level of branching in a SIMD group. As more and more PEs a SIMD group diverge, the additional cycles required was approximately linear - indicating the overheads in addition to serialization were small. While such serialization overheads can be untenable for some applications, since the average divergence we see is quite small (between 1 and 2), this overhead can be tolerated.

To summarize, we have outlined a SIMD like organization that can support DFAs and XFAs efficiently. As proof of concept, we implemented a prototype IPS signature matching system on the Nvidia G80 GPU. In the next section we evaluate the performance of our implementation.

4. Results

We now evaluate the performance of signature matching using SIMD processing. We use our GPU implementation for performance evaluation and compare the performance to a software implementation running on a Pentium4 system. We first describe the machine configurations and datasets. We analyze both DFAs and XFAs using standard network traces and signature sets. We show the performance potential of the technique, performance on representative network traces and signature sets, and analysis of the implementation - focusing on the benefits of threading and memory management optimizations.

4.1 Datasets and system configuration

We examine three protocols FTP, SMTP, and HTTP, and use a set of signatures from Cisco Systems [1]. We also examined the Snort signature sets and found the results to be qualitatively similar and hence present only the Cisco results in this paper. We converted the signatures to DFAs and XFAs. Since the DFAs required large amounts of memory we partitioned the DFAs into multiple DFAs to meet a memory budget of 64 MB for the state machine. Table 3 describes the details of the three signature sets. We implemented the XFA and DFA interpreters in C++ for our baseline microprocessor system. We compare performance to a Pentium4 system and measure speedup in terms of total execution time. We use a Nvidia G8800 GTX card plugged into a Pentium4 system running at 3GHz, which is our baseline. We first process the trace using the CPU and measure execution cycles. The packet buffer is then passed on to the GPU to measure execution cycles.

4.2 Performance potential

Figure 5 compares the ideal performance of the G80 to our baseline. We measure this ideal performance by constructing a simple DFA with just one state that remains in this state irrespective of input. On the x-axis we show the number of threads and the y-axis shows ns/byte. With only one thread, we are comparing the performance of one processing element to the Pentium4 processor and it is about 4 orders of magnitude worse. Each byte in the packet requires two memory accesses amounting to 800 cycles of memory delay, resulting in this performance difference. As we increase the number of threads, the multi-threading capability of the G80 is effective at hiding memory latency and with 8192 threads performance levels off. The G80 out-performs the Pentium4 running the same DFA, by about 12X. The speedup on this simple DFA places an upper-bound for what the G80 implementation can achieve on real traces. Comparing the raw peak performances the G80 is capable of approximately 36 times better performance. This lost performance potential is primarily due to memory throughput and latency.

The dotted line shows performance when the memory system is idealized for the G80. We create this idealized memory configuration modifying the interpreter to remove the accesses to global memory. First, we repeatedly process the same byte instead of fetching every byte of the packet from memory. Second we replace the transition lookup using global memory with a dummy load from local memory. The DFA transitions for this experiment are independent on input, hence this modified code quite accurately estimates the effect of perfect memory. While multi-threading is effective, the performance of this idealized memory configuration shows a large amount of memory latency is still not hidden.

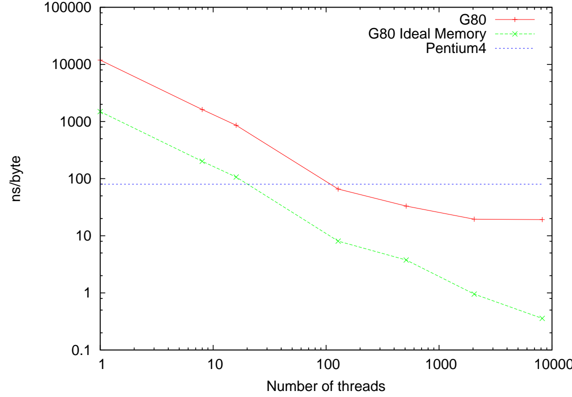

Figure 5. Performance of G80 implementation on best-case DFA.

Protocol	# Sigs	# States (for XFAs)	Speedup					
			Implementation		80% Cache hits		Perfect caching	
			DFA	XFA	DFA	XFA	DFA	XFA
ftp	31	323	8.6	7.2	10.5	8.5	11.1	8.9
http	52	1759	9.2	5.6	11.1	6.0	11.7	6.3
smtp	96	2673	8.1	7.8	9.9	9.3	10.4	9.7

Table 3. Description of signatures and performance comparison to Pentium4 on representative traces. Columns 6-8 are based on estimates from our memory access latency model.

Initially, we see the idealized memory system reflecting difference in latencies. After 2048 threads, bandwidth becomes a constraint for the real system and hence performance levels off, whereas the ideal system scales further as it also has infinite bandwidth. Comparing these two systems we estimate that effective memory access latency is 16 cycles - we substract the difference in execution cycles and divide by the total number of memory loads.

4.3 Overall Performance

We now present performance results for full traces and full signature sets. Table 3 shows the speedup compared to our Pentium4 baseline for the three protocols. For DFAs the three protocols show similar speedups averaging 8.6X. This is to be expected because DFAs have regular control flow and all these DFAs have a large number of states. The XFAs on the other hand show more divergent behavior. Overall XFAs perform about 6.7X better on the G80 compared to our baseline system. HTTP shows the least performance improvement because it executes more XFA instructions on average than the other protocols.

The three main components of execution time are: 1) transferring data to the GPU, 2) computation, and 3) transferring data back. The second component dominates the execution time, and we expect double-buffering can hide the transfer delays. The version of the drivers we use does not provide support for such double buffering - but such support is soon expected. For the HTTP protocol, for example, the break down of the times for processing 16K packets is: 34ms(transfer), 212ms(compute), and 0.18ms (transfer-back). Since transfer time is much smaller than computation time, double-buffering will be effective. The state machine data structure must be created during initialization. By far, this takes the largest amount of time as it executes on a single processing element and must recursively build the state machine data structure and take 15 minutes for the HTTP XFAs. Database update time will be dictated by this.

Overall the G80 implementation performs significantly better than the Pentium4 implementation. However the speedup is still far below 36X difference in peak performance between the two systems. We perform a study of optimizations that can improve performance.

4.4 Enhancements

Hardware caching - Texture memory: As shown in our analysis of performance potential, while multi-threading is able to hide the memory latency to some extent, a significant portion of performance loss is due to memory latencies to the uncached memory space. Accesses to this uncached memory belong to one of two large data structures — the packet buffer and state machine data structure. The G80 includes a large hardware managed texture cache that can exploit locality. To simplify hardware, this cache can only be read from the GPU, and its contents is effectively pre-loaded by the CPU[3].

The packet buffer and the state machine data structure are effectively read-only data structures from the `xfa_apply`

[3] In reality the CPU marks a region of memory as texture cacheable and at run-time the cache is populated by the GPU. Explicit writes to this region of memory from the GPU are forbidden.

kernel that processes the packets. The packet buffer accesses show no re-use but high spatial locality and essentially benefits from pre-fetching. We developed a microbenchmark to isolate the benefits of texture caching and noticed that on the G80, regular accesses to texture memory were typically twice as fast as regular accesses to global memory. The delays to access the state machine dominate (64-bits of packet data generates 8 such accesses) and mapping the packet buffer to texture memory did not provide substantial improvements.

The transition table is an ideal candidate for texture memory and hardware caching should effectively capture the working set of the "hot" states as best as possible. However, the state machine is a recursive data structure that includes pointers in the transition table. Hence it cannot be created on the CPU side and simply copied over to the GPU since the CPU address space and GPU address spaces are different. In fact, the lack of support for writing to texture memory curtails its use for any recursive data structure. *Thus, we propose exposing the GPU address structure to the CPU.* With our implementation we can estimate the benefits of such caching but cannot accurately measure it. We use a simple model of memory accesses to estimate the benefits. From our initial performance potential experiment we estimate that the average memory access latency is 16 cycles. If the entire working set size fits in the cache, memory access latencies will be the latency to the texture cache. Columns 4-7 in Table 3 show the benefits of such caching with a hit rate of 80% and 100%, for a latency of 4 cycles.

Software managed memory: Software managed caching can also help hide memory access latencies. The signature sets and traces can be profiled to determine the set of hot states and these states can be saved in fast software managed memory with only misses being sent to the global uncached memory. Several processors include such software managed memories – Imagine [17] has a stream register file, and the IBM Cell processor has 256 KB of software managed storage at each core [16]. The G80 includes such software managed memory called "shared memory" which is 16KB of storage space shared between all the processing elements of a core and can be accessed in 4 cycles. This is further organized into 16 banks of 1KB each and is specialized for regular accesses that each go to a different bank.

We considered the use of this memory for caching the hot states, but our estimates show the space is too small to be useful. First with 16KB, only 16 states can be cached. Furthermore, unless each PE accesses a different state, bank conflicts dominate and reduce the benefits of this memory. The caching results in Table 3, column 4-7 can be achieved with a software managed cache as well. No one has demonstrated the use of such software managed memories for signature matching. *Our analysis shows, software managed memories can indeed perform well.*

4.5 Limitations

The limitations of our prototype implementation are the following. First we process packets in batches, because of the interface limations between GPU and CPU. Second, we are performing only signature matching, which is one component (compute intensive) of an IPS. In future work we will examine other functions. Finally, this system is as prone to worst case attacks as other IPS software systems. A head-to-head performance comparison against network processors should provide interesting efficiency and cost/performance comparisons.

4.6 Discussion

From our analysis of the application we identify two performance problems that can be alleviated with small changes to the SIMD engine. The first problem is that our key data structure – the state machine – requires a high volume reads from off-chip memory that pushes the limits of the available memory bandwidth. While our measurements show that there is much locality in how it is accessed, it is not possible to store it in the large texture memory for which hardware caching is implemented. The reason is that the G80 implementation disables writes to textures to simplify the cache design and avoid synchronization issues, while the CPU cannot build the data structure because it does not have access to the address structure used by the GPU. A second problem is that the processing elements of a core need to start working on packets at the same time especially for XFAs because of the initialization required. To achieve the best throughput the system must batch together packets of similar sizes increasing the latency for processing individual packets.

- **Hardware caching for recursive data structures:** Hardware caching recursive data structures like our state machine data structure can provide significant performance improvement. This can be achieved by simply allowing a special mode where a single thread executes and writes to the texture cache to build this data structure. Alternatively, the GPU address space can be exposed to the CPU in some fashion to build such data structures on the CPU and copy them over.
- **Larger software managed memories:** Software-managed on-chip memories larger than the current 16KB per core could also be used to explicitly manage locality in the state machine data structure.
- **Resettable local memories:** The local memory state must be cleared before a packet is processed. A hardware extension that clears it automatically could reduce the instruction count for XFAs. More importantly, it could eliminate the requirement that all processing elements start working on their packets at the same time since single-instruction resets would lead to little divergence.

The goal of this paper was to investigate the suitability of SIMD processing for signature matching. The two key

questions are whether high performance can be sustained and the programmability effort involved.

Performance: Overall the performance improvements of our implementation compared to a Pentium4 system are substantial, about 8X. However, the improvements are far below the difference in peak performance - about 36 fold. The Pentium4 sustains 50% of its peak with an IPC of approximately 1.5. Simply based on a comparison of peak performance and achieved performance, we estimate the G80 sustains about 10% of its peak performance. While substantial, applying our suggested improvements could improve performance further.

Design and Programmability: In our experience the effort in programming this SIMD design were definitely higher than programming a general purpose microprocessor. However, we did not have to perform explicit parallelization of this problem because of its nature and there is no synchronization required between the threads. We found the compiler and the optimizations it applied to be quite sophisticated. The challenge was mainly debugging, given the esoteric development environment that does not provide access to the machine registers directly.

5. Related Work

The work most closely related to ours can be grouped in terms of application analysis and implementation of signature matching, analysis and extensions of SIMD architectures, and applications on GPUs.

This paper is the first to present a detailed analysis of signature matching for network processing. FPGA and ASIC implementations for high speed intrusion detection have been recently explored [4, 7, 14, 8, 20]. Tuck et al. describe design techniques for IPS/IDSes for ASIC and software implementations targeted at general purpose programmable processors. Brodie et al. describe a pipelined implementation of regular-expression pattern matching that can map to an ASIC or FPGA [5]. Alicherry et al. examine a novel Ternary CAM based approach [2]. Tan and Sherwood describe a specialized hardware implementation that performs several pattern matches in parallel where each pattern is a simple state machine [21]. Software based techniques to improve IPSes and algorithmic improvements are unrelated to our work.

A review of SIMD implementations is provided in [12]. Nichols et al. describe compiler and programming language aspects of coarse grained SPMD processing which is similar in spirit to the multi-threaded SIMD implementation of GPUs. Erez et al. describe irregular computation on SIMD architecture focusing on irregular memory access paterns and software transformations [11]. They examine scientific workloads and focus on locality and parallelization optimizations. For scientific applications they conclude control-flow flexibility provides at best 30% performance improvements. In our workloads, the parallelization is straightforward and the locality can be easily determined through profiling. Our results show that, the benefits of control-flow flexibilit can be quite dramatic for signature matching. Bader et al. [3] examine irregular applications and a mathematical model for the Cell processor which is a multi-core 4-wide SIMD architecture. Pajuelo et al. propose microarchitecture techniques to exploit SIMD parallelism using speculation for vector-like regions in scalar code [18].

Jacob and Brodley [15] demonstrate a version of the open source intrusion detection/prevention system Snort that uses the NVidia 6800 GT graphics card performing simple string matching. Our evaluation on a more flexible newer-generation GPU with more recent, larger databases of more complex signatures and using more efficient matching algorithms gives us much more promising results with respect to the potential of GPUs to support higher throughput. Also, since we focus more on mapping the signature matching operation to the GPU as opposed to building a fully functional system that can be deployed, we are able to better understand how various decisions by the architects of the GPU affect its ability to support high-speed signature matching. Several high performance applications have been mapped to GPUs and Pharr and Fernando provide a good overview [19].

6. Conclusion

In this paper, we examined the feasibility of using a SIMD architecture for performing signature matching, the most processing-intensive operation for network intrusion prevention systems. This paper is the first to perform a detailed application analysis examining the basic memory, control flow, and concurrency properties of signature matching. We examine two techniques for signature matching: using multiple deterministic finite automata (DFAs) and using extended finite automata (XFAs) to represent the set of regular expressions to be matched against the packet payload. The first requires simple processing for each input byte while the second has been showed to achieve better performance on general purpose processors, but requires more complex and less uniform per byte processing.

This paper is the first to identify that SIMD processing maps well to this application. We outlined a SIMD design that includes extensions for irregular control flow that can support the non-uniform behavior exhibited by XFAs. To evaluate this design we implemented signature matching on an Nvidia G80 GPU which shows 6X to 9X better performance than a Pentium4 system. We also propose simple extensions to the memory system that can improve performance for pattern matching without slowing down other applications or increasing the complexity of the chip. Our proof-of-concept implementation shows that network devices can offload signature matching to a SIMD engine to achieve cost-effective performance improvements. Since regular expression matching is central to a number of other

applications such as network traffic policing, XML processing, and virus scanning we believe that solutions similar to ours can help such applications benefit from the performance potential of SIMD engines.

References

[1] Cisco intrusion prevention system. http://www.cisco.com/en/US/products/sw/secursw/ps2113/index.html.

[2] M. Alicherry, M. Muthuprasanna, and V. Kumar. High Speed Pattern Matching for Network IDS/IPS. In *ICNP '06. Proceedings of the 2006 14th IEEE International Conference on Network Protocols*, pages 187–196, November 2006.

[3] D. Bader, V. Agarwal, and K. Madduri. On the Design and Analysis of Irregular Algorithms on the Cell Processor: A Case Study of List Ranking. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007*, pages 26–30, March 2007.

[4] Z. Baker and V. Prasanna. Automatic Synthesis of Efficient Intrusion Detection Systems on FPGAs. *IEEE Transactions on Dependable and Secure Computing*, 3:289–300, October 2006.

[5] B. C. Brodie, D. E. Taylor, and R. K. Cytron. A scalable architecture for high-throughput regular-expression pattern matching. In *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 191–202, Washington, DC, USA, 2006. IEEE Computer Society.

[6] J. B. Cabrera, J. Gosar, W. Lee, and R. K. Mehra. On the statistical distribution of processing times in network intrusion detection. In *43rd IEEE Conference on Decision and Control*, Dec. 2004.

[7] Y. H. Cho and W. H. Mangione-Smith. Deep Packet Filter with Dedicated Logic and Read Only Memories. In *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'04)*, pages 125–134, 2004.

[8] C. Clark and D. Schimmel. Scalable Parallel Pattern Matching on High Speed Networks,. In *Proc. 12th Ann. IEEE Symp. Field Programmable Custom Computing Machines (FCCM '04)*, pages 249–257, 2004.

[9] Nvidia Compute Unified Device Architecture. Online http://developer.nvidia.com/.

[10] Shader Model 4 (DirectX HLSL). Online http://msdn2.microsoft.com/en-us/library/bb509657.aspx.

[11] M. Erez, J. H. Ahn, J. Gummaraju, M. Rosenblum, and W. J. Dally. Executing irregular scientific applications on stream architectures. In *ICS '07: Proceedings of the 21st annual international conference on Supercomputing*, pages 93–104, New York, NY, USA, 2007. ACM Press.

[12] G. C. Fox. What have we learnt from using real parallel machines to solve real problems? In *Proceedings of the third conference on Hypercube concurrent computers and applications*, pages 897–955, 1988.

[13] Nvidia G80 Specs. Online http://www.nvidia.com/page/8800_features.html.

[14] B. Hutchings, R. Franklin, , and D. Carver. Assisting Network Intrusion Detection with Reconfigurable Hardware. In *10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'02)*, pages 111–120, 2002.

[15] N. Jacob and C. Brodley. Offloading IDS computation to the GPU. In *ACSAC*, Dec. 2006.

[16] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D.Shippy. Introduction to the cell multiprocessor. *IBM Journal of Research and Development*, 49(4):589–604, July 2005.

[17] B. Khailany, W. J. Dally, S. Rixner, U. J. Kapasi, P. Mattson, J. Namkoong, J. D. Owens, B. Towles, and A. Chang. Imagine: Media processing with streams. *IEEE Micro*, 21(2):35–46, March/April 2001.

[18] A. Pajuelo, A. González, and M. Valero. Speculative dynamic vectorization. In *ISCA '02: Proceedings of the 29th annual international symposium on Computer architecture*, pages 271–280, Washington, DC, USA, 2002. IEEE Computer Society.

[19] M. Pharr and R. Fernando. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley Professional.

[20] H. Song and J. W. Lockwood. Efficient packet classification for network intrusion detection using fpga. In *FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pages 238–245, New York, NY, USA, 2005. ACM Press.

[21] L. Tan and T. Sherwood. A high throughput string matching architecture for intrusion detection and prevention. In *32st International Symposium on Computer Architecture (ISCA 2005)*, pages 112–122, 2005.

[22] XFAs: Fast and compact signature matching. Under submission. Details omitted for anonymity.

[23] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *ANCS '06: Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*, pages 93–102, 2006.