Appears in the 2009 IEEE International Symposium on Performance Analysis of Systems and Software

Evaluating GPUs for Network Packet Signature Matching

Randy Smith Neelam Goyal Justin Ormont Karthikeyan Sankaralingam Cristian Estan
University of Wisconsin–Madison
{smithr,goyal,ormont,karu,estan}@cs.wisc.edu

Abstract

Modern network devices employ deep packet inspection to enable sophisticated services such as intrusion detection, traffic shaping, and load balancing. At the heart of such services is a signature matching engine that must match packet payloads to multiple signatures at line rates. However, the recent transition to complex regular-expression based signatures coupled with ever-increasing network speeds has rapidly increased the performance requirements of signature matching. Solutions to meet these requirements range from hardware-centric ASIC/FPGA implementations to software implementations using high-performance microprocessors.

In this paper, we propose a programmable signature matching system prototyped on an Nvidia G80 GPU. We first present a detailed architectural and microarchitectural analysis, showing that signature matching is well suited for SIMD processing because of regular control flow and parallelism available at the packet level. Next, we examine two approaches for matching signatures: standard deterministic finite automata (DFAs) and extended finite automata (XFAs), which use far less memory than DFAs but require specialized auxiliary memory and small amounts of computation in most states. We implement a fully functional prototype on the SIMD-based G80 GPU. This system out-performs a Pentium4 by up to 9X and a Niagara-based 32-threaded system by up to 2.3X and shows that GPUs are a promising candidate for signature matching.

1. Introduction

Network devices are increasingly employing deep packet inspection to enable sophisticated services such as intrusion detection, traffic shaping, and quality of service. Signature matching is at the heart of deep packet inspection and involves matching pre-supplied signatures to network payloads at line rates. While there are other necessary facets such as flow reassembly, stateful analysis, and other preprocessing, it is widely observed that signature matching is the most processing-intensive component and the bottleneck to increased performance. For example, Cabrera *et al.* [7] reported in 2004 that signature matching alone accounts for approximately 60% of the processing time in the Snort Intrusion Prevention System (IPS). Since that time, the number of signatures has more than tripled.

Table 1 shows a qualitative comparison of different architectures used and proposed for signature matching. On one hand, ASICs provide high area/energy efficiency and performance, but they have poor flexibility and high system design cost. At slightly lower speeds FPGAs become a viable alternative if flexibility is valued, but they consume significantly more power than ASICs. At the other end of the spectrum, general purpose microprocessor-based systems are very flexible and have low non-recurring and design cost but cannot match the performance of specialized hardware and have high per-unit costs. They rank lowest in performance, but best in cost, flexibility and time to market.

Between these extreme points are network processors such as the Intel IXP, IBM PowerNP, and Cisco QuantumFlow. These specialize a superscalar architecture with networking-specific functionality to increase efficiency without sacrificing programmability or design cost, and minimize non-recurring expenses. They all utilize extensive multi-threading to exploit the data-level parallelism found at packet granularity to hide memory latencies. This threading abstraction provides *independent* control-flow and temporary data management for each thread. But, this independent control flow yields inefficiencies, since the same set of instructions are repeatedly and independently executed on many packets simultaneously. Custom solutions have also been proposed for regular-expression matching (which can be used for signature matching) and XML processing both in industry [29] and academia [5].

Opportunity: In light of these alternatives, for deep packet inspection we observe that often the same processing is applied to each packet, suggesting that the Single Instruction Multiple Data (SIMD) paradigm is ideal for this type of workload. A single instruction can simultaneously perform operations for many packets providing large savings in power and area overheads. SIMD processing can achieve area/performance efficiencies much higher than superscalar processors (Table 1, Section 2.2). Further, they are abundantly available as GPUs and hence have low design costs. Their simple structure yields performance superior to network- and general-purpose processors, yet they retain flexibility absent in ASICs.

Challenges: GPUs provide a highly simplistic memory model that assumes mostly regular access to memory; applications with recursive data structures or irregular memory accesses may perform poorly. However, the data access patterns associated with packet inspection primarily involve (recursive) graph traversal and are a challenge to SIMD architectures. Thus, to

Parameters	ASIC	FPGA	GPU/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.

assess the suitability of GPUs for signature matching, we seek to answer the following two key questions:

- *Do GPU/SIMD architectures have the flexibility needed to support typical signature matching operations?*
- *Does the process of mapping these operations to SIMD architectures negate potential performance gains?*

Contributions: In this paper, we examine the viability of SIMD-based architectures for signature matching. Overall, we show that GPUs (and the SIMD paradigm in general) can support signature matching and provide higher performance at a cost similar to general purpose processors. In support of this result, our work makes the following additional contributions:

- we present a characterization of signature matching for IPSes in terms of the control-flow, memory accesses, and available concurrency;
- we provide a fully-functional prototype implementation of signature matching on an Nvidia G80 GPU;
- we conduct a performance comparison to multithreaded out-of-order and in-order architectures, including a Niagara system and an Intel Core2 system.

We focus on signature matching but note that the overall approach and conclusions generalize to other applications such as virus scanning and XML processing, which are also dominated by similar regular expression matching. We give further details about signature matching and conduct a detailed analysis in Section 2. Section 3 shows a SIMD design for signature matching and describes our prototype GPU implementation, and Section 4 discusses performance results. Section 5 gives the related work and Section 6 concludes.

2. Signature Matching Analysis

In this section we first present the techniques and data structures employed in signature matching in detail. We then perform a detailed analysis of the memory usage, control flow, and concurrency properties with respect to the requirements and properties of SIMD architectures.

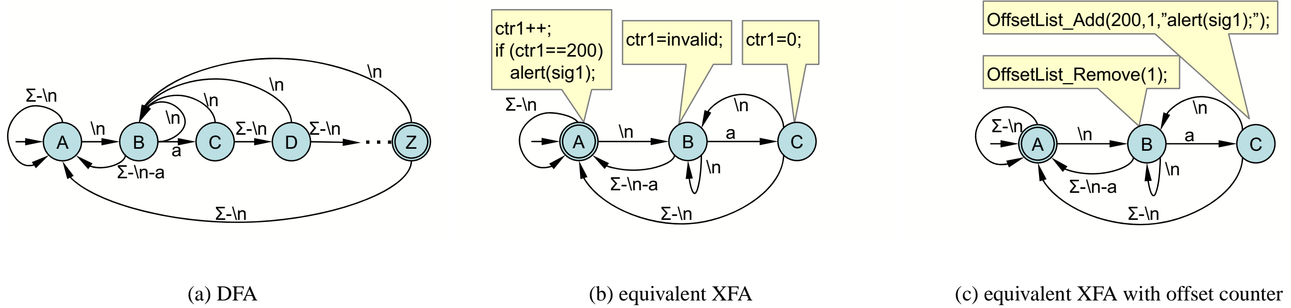
2.1. Application description

The core operation in signature matching is determining whether a given packet payload matches any of the signatures corresponding to the packet's protocol (HTTP, SMTP, etc). Currently, regular expressions are the *de facto* standard for *expressing* signatures due to their ability to succinctly capture entire classes of vulnerabilities rather than just specific exploits [6], as is the problem with strings. On the flip side, finite automata–or state machines–are used for *matching* signatures to input. In this work, we evaluate two such matching mechanisms, standard Deterministic Finite Automata (DFAs), which recognize exactly the class of regular expressions, and the recent Extended Finite Automata (XFAs) [25, 26], which significantly reduce the memory requirements of DFAs.

2.1.1. DFAs for Matching Regular Expressions A DFA is an augmented directed graph $(S, \Sigma, \delta, s, F)$ with states (nodes) S, a designated start state $s \in S$, a set of accepting states $F \subseteq S$, and an alphabet Σ whose symbols are attached to edges. For each state $t \in S$ and each $\sigma \in \Sigma$, there is exactly one edge from t to $t' \in S$ labeled with σ. Altogether, these sets of edges constitute the *transition table* δ. During matching the DFA is traversed, beginning at the start state, by following corresponding labeled transitions from state to state as each byte in the input is read. If an accepting state is reached, the DFA additionally signals acceptance at that state before following the next transition. Figure 1a shows the DFA corresponding to the regular expression `/.*\na[^\n]{200}/`, which is simplified but characteristic of buffer overflow signatures.

This simple model has two advantages. First, matching is straightforward and fast, requiring just a single table lookup or pointer dereference per input byte. Second, DFAs are composable, meaning that a set of DFAs (one per signature) can be composed into a single composite DFA so that in principle it is possible to match all signatures in a single pass over the input. Unfortunately, DFAs often do not interact well when combined, yielding a composite DFA whose size may be exponential in the input and often exceeds available memory resources. To reduce the memory footprint, Yu *et al.* [32] proposed the creation of multiple composite DFAs. In this paper, we employ this technique with a memory budget of 64 MB per signature set, which produces between one and seven composite DFAs per signature set.

2.1.2. XFAs for Matching Regular Expressions With DFAs, many distinct states must be created in order to si-



(a) DFA (b) equivalent XFA (c) equivalent XFA with offset counter

Figure 1. DFA and equivalent XFAs that recognize the signature `/.*\na[^\n]{200}/`. The XFA in (c) uses offset counters to reduce the overall number of instructions executed as compared to the XFA in (b).

multaneously track the matching progress of distinct, complex signatures. In the DFA model, the subsequent blow-up in memory usage occurs because there is no difference between the computation state and explicit DFA states. The XFA model [25, 26] addresses this problem by separating the computation state from explicit states in a manner that retains the advantages of the DFA model while simultaneously avoiding a memory blow-up. Specifically, XFAs extend DFAs with an auxiliary "scratch" memory used to store variables such as bits and counters which are used with states to track matching progress (computation state) more compactly than explicit states alone can do. Small programs attached to states update variables in scratch memory when reached. Acceptance occurs when an accepting state is reached and scratch memory variables have specific, accepting values. Figure 1b shows an XFA with a counter that is equivalent to the DFA in Figure 1a but requires just 3 instead of 203 states.

The XFA model retains the composability inherent to DFAs, but per-byte processing is more complex since programs attached to states must be executed whenever such states are reached. This cost is offset by the reduced memory footprint so that in practice, XFAs are typically smaller *and* faster than groups of DFAs. For each signature set in our evaluation, we needed only one composite XFA which required less than 3 MB of memory total (compared to up to 7 composite DFAs using 64 MB total).

XFAs do introduce challenges for SIMD architectures, though. Most variable updates involve simple operations such as setting, clearing, and testing bits, or setting, incrementing, and testing counters. However, these operations decrease the regularity of the application and introduce the need to access and manipulate additional memory in a manner that is distinct from the access patterns for traversing states. In addition, some XFAs use so-called *offset counters*, which are functionally equivalent to standard counters but store the counter implicitly in an *offset list*, also stored in scratch memory. Figure 1c shows the counter in Figure 1b expressed as an offset counter. Such counters eliminate explicit increment instructions but require additional work at each byte read (see Figure 2 below). They reduce the overall number of executed instructions at the expense of decreasing the regularity even further. Altogether, XFAs can impose a significant performance penalty on SIMD architectures.

2.2. Application analysis

Figure 2 outlines the basic code executed to perform signature matching. Each packet is extracted and passed to a 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 module are (1) a state machine for the set of patterns to be detected, (2) auxiliary data maintained for each packet as it is processed, (3) a packet buffer that contains 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.

2.2.1. Memory

Analysis: The 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.

In the state machine structure, each state contains a 256-entry transition table (indexed by byte value and containing a pointer to the next state) and other housekeeping data, summing to slightly more than 1024 bytes per state. This data structure can be several gigabytes in size for DFAs depending on the number of signatures that are to be matched. For XFAs, the state machine data structure is typically a few megabytes and in our measurements always less than 3MB. Accesses to this

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

trace_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
}
```

Figure 2. **DFA and XFA processing pseudo code**

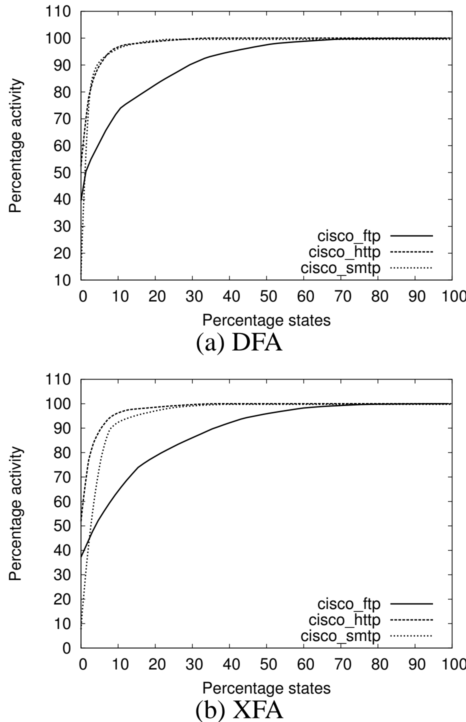


Figure 3. **Frequency distribution of states visited**

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 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 we instrumented our implementation to measure how many times every state is visited while processing a network stream. Figure 3 shows the frequency at which each state is visited,

Protocol	Measured on Core2				SIMD Divergence	
	Prediction accuracy		ILP			
	DFA	XFA	DFA	XFA	%	Ave.
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 parallelism**

sorted from high to low. For both DFAs and XFAs, for most sets, less than 10% of the states contribute to more than 90% of the visits. Hence, this working set size consisting of only hundreds of states can fit in the cache of a high performance microprocessor. Note the distinct behavior of the FTP distribution. Unlike other protocol signatures, most FTP signatures look for valid commands with arguments that are too long. As a result, a significantly larger number of distinct states are traversed (corresponding to valid command sequences) to confirm the presence or absence of a match.

2.2.2. Control-flow

Analysis: Signature matching 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 each 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 at the lower two levels is well suited for a SIMD architecture. In the first level, each processing element handles a packet. At the second level, a single *interpreter* instruction then processes data for each packet exploiting predictability. 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, DFAs have almost no divergence, since they have only these two levels of control flow.

The third level of control flow, which exists only for XFAs, will inevitably diverge. Within each state, a large case statement in the interpreter decides what functionality must be performed based on the XFA instruction. Depending on the XFA 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 then, 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 (using performance counters) on a Core2 system and instrumented our interpreter to measure the divergence. Table 2 shows the branch prediction accuracy in the 2nd and 3rd columns for different protocols for both DFAs and XFAs. The high prediction accuracies demonstrate that control flow patterns are repeated.

Our analysis is based on the notion of divergence, which we define as follows: if the ith character of a packet in any processing element (PE) takes a conditional branch different from any other PEs, then that SIMD instruction is said to have diverged and has a divergence value of 1. Divergence percentage is the percentage of such instructions compared to the total number of SIMD instructions executed. Columns 6 and 7 show divergences in the control flow for the protocols. We calculate divergence percentage for groups of 32 packets, because 32 is the SIMD width of our prototype implementation. The last column has the average divergence among the instructions that diverge. If two 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. Packet size distributions for SMTP packets have more variability and provide less group-of-32 equal-sized packets, which leads to the increased divergence. 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.

Traditional SIMD designs do not have support for such types of control flow within a SIMD instruction, but such support can be added at some performance cost. For example, the Nvidia G80 architecture supports such types of control flow at a penalty of 31 cycles. Further, Fung *et al.* evaluate several techniques such as dynamic warp formation [14].

2.2.3. Concurrency Signature matching is heavily data-parallel since each packet can be processed independently. Within a packet's processing, though, the level of concurrency is limited since per-byte transitions are serially dependent on each other. For XFAs, the interpreter that executes XFA instructions also has only limited amount of instruction-level parallelism. Columns 4 and 5 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.

2.2.4. 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 since the management of the local variables (counters and bits) is dependent on the packet data. Such data-dependent branching cannot be efficiently supported with a classical SIMD organization using predication. Fortunately, such branching is infrequent and hence some form of escape mechanism that temporarily deviates from SIMD processing will suffice.

Our characterization shows that GPUs are a viable target for signature matching. GPU architectures have recently incorporated several features including wide-SIMD, extensive multi-threading to hide memory latency, and support for fast regular memory access in the form of texture memories. Further, support for data-dependent branching (as needed for XFA workloads) is emerging. DirectX 10-compliant GPUs include support for predication and data dependent branching [12], and the Nvidia G80 GPU, for example, is implemented as a SIMD array with extensions for data-dependent branching [19].

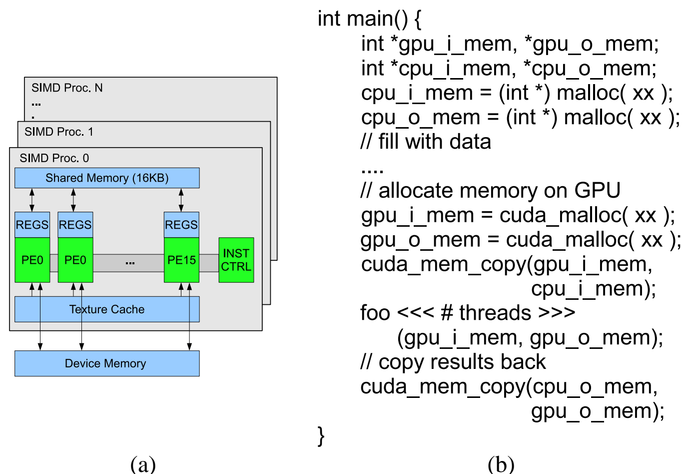


Figure 4. High-level GPU architecture and code example

3. Architecture and Prototype Implementation

To evaluate how well the SIMD paradigm is suited for signature matching, we built a prototype using the Nvidida G80 GPU. In this section, we briefly describe the organization of the G80 processor and our software implementation.

3.1. Prototype GPU implementation

G80 Organization: For the scope of this paper, we focus only on the programmable components of the G80 chip. As depicted in Figure 4a, the G80 is organized as a 16-core SIMD machine, with each SIMD core being logically 32-wide. The Compute Unified Device Architecture (CUDA) [20] defines the GPU architecture, and a set of software drivers interface to it. The processor has four types of memories: a small local memory of 8KB per core that can be accessed in 1 cycle, 16KB of "shared memory" for localized synchronization of all threads within a core, cacheable texture memory of 128MB with an 8KB cache, and uncached memory (or global memory) of 768 MB that can take 400 to 600 cycles to access. A threading unit exists to schedule threads and hide memory latencies. The clock frequency of the SIMD core is 1.35GHz.

The processor uses a streaming programming model, with a *kernel of instructions* 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.

These are distinct from typical processor threads in that they are 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 each gets a partition of the local memory.

The programming model for the CUDA architecture is shown in Figure 4b. Any code that must execute on the GPU is referred to as a *kernel* and is defined as a C or C++ function augmented with special directives. 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. For example, 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.

Software implementation: We implemented a fully functional signature matching system on the G80 GPU. We developed a DFA/XFA signature matcher using the above streaming model that has two main components (or kernels): `fa_build` and `trace_apply`. `fa_build` builds the state machine structure for DFAs or XFAs on the GPU and executes prior to matching. This kernel is completely sequential and executes as a single thread utilizing one processing element on the chip. DFAs and XFAs are recursive data structures and cannot simply be copied from CPU memory to GPU memory (pointers will be different in each address space). Thus, each state is copied and transitions are rebuilt.

The other kernel, `trace_apply`, processes packets and performs matching. In our implementation, we pass a large trace of 65,536 packets to the GPU and initiate processing. A sorting engine creates groups of 32 packets which are largely similar in size. Every SIMD processor is assigned one such group of packets. Finding 32 packets with identical size can introduce large latencies, whereas having unequal sized packets introduces wasted work, as every packet will take as many cycles to process as the largest packet in the group. In our implementation we found that a window of 2048 packets provides groups with little variance in the packet size. Once the processing of all packets is complete, a bit-vector is passed back to the CPU indicating which signatures matched which packets. 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 would utilize some form of shared memory to address this.

3.2. Optimizations

Long-word access to memory. We started with a strawman implementation that maintained the state machine and 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 2$\times$ improvement compared to single-byte accesses.

Register pressure. The number of registers in a kernel is a critical resource, since the physical register file of 256 registers is shared between threads. For example, for a kernel with 32 registers, at most 8 threads can execute concurrently in a single core. To increase concurrency, we limited the number of registers used to eight, thus creating spills to local memory as necessary.

Branching. We found the data-dependent branching support in the G80 to be adequate, since the average divergence we see is quite small (between 1 and 2). We developed a microbenchmark that measured the level of branching in a SIMD group. As more and more PEs of a SIMD group diverge, the additional cycles required was approximately linear - indicating that the overheads in addition to serialization were small.

4. Results

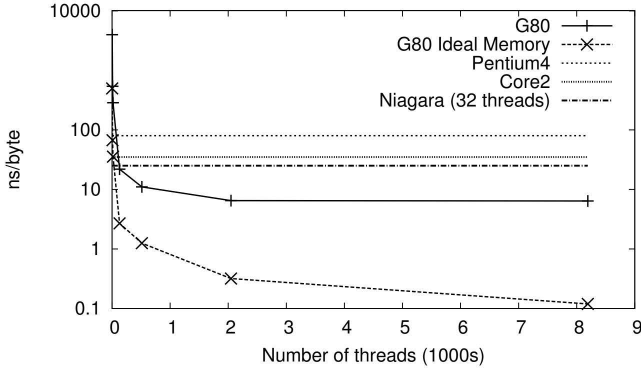
We now report the performance evaluation of our prototype GPU-based signature matching system. Our prototype system is a Nvidia G8800 GTX card plugged into a Pentium4 system running at 3GHz. Our baseline for comparison is a software implementation running on this Pentium4 system. We also examine performance on an Intel Core2 system (Clovertown Intel Xeon E5345 2.33GHz)[1]. Optimizing for MMX/SSE can improve the performance of these implementations which we will investigate in future work. In addition, we developed a pthreads-based multithreaded implementation and measure its performance on a Sun Niagara (SunFire T2000). We first describe the data-sets and machine configurations. We then show performance potential and performance on representative network traces and signature sets. We conclude the section with a discussion of optimizations.

4.1. Datasets, system configuration, and Metrics

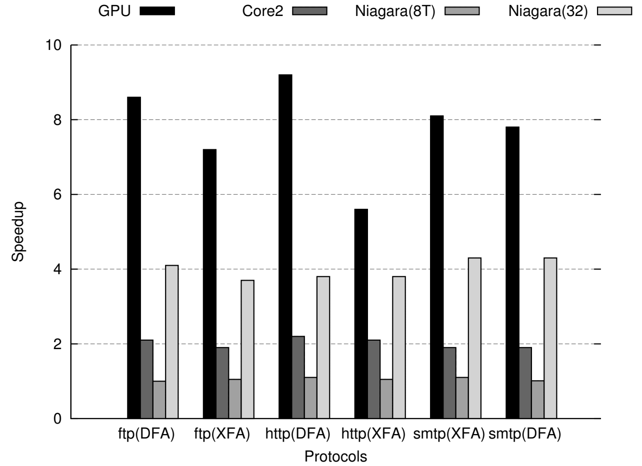
Our study uses signatures for three common protocols – FTP, SMTP, and HTTP – taken from both Cisco Systems [9] and Snort [22]. We found the Snort-based measurements to be qualitatively similar to those for Cisco signatures and hence show only the results for the Cisco sets. We first converted individual signatures to distinct DFAs and XFAs. We then created composite DFAs [32] with memory limited to 64 MB per signature set. Table 3 describes the details of the three signature sets. When summed across all protocols and features, total signatures can number in the thousands, but during runtime only those signatures that correspond to a packet's protocol must be matched. There are no fundamental limitations to our system to simply run all protocols and scale up. For input data our experiments follow standard practice. We use a 10GB network trace collected on the edge of a large academic network that is replayed through our prototypes.

We implemented the XFA and DFA matching routines in C++ for our baseline Pentium4 system. Our multi-threaded im-

1 Clovertown's inter-core communication is structured so that its parallel scalability is quite poor; we report only single thread performance for this platform.



(a) Best-case DFA (logscale on y axis)



(b) Benchmark signatures

Figure 5. Performance comparison to baseline Pentium4 system

Protocol	# Sigs	# States (XFAs)	Speedup					
			G80		80% hits		Perfect	
			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. Columns 6-9 are estimates.

plementation minimizes synchronization overheads and uses a single packet-pointer queue which contains pointers to the packet buffer. Reads to this packet queue are protected with a fine-grained lock, and all other accesses are contention-free. We also implemented a lock-free version that uses more memory and indexes the packet buffer with the thread number, but this provided less than 2% improvement and shows our lock implementation is quite efficient.

We measure speedup in terms of total execution time. For our multi-threaded experiments, we measure execution cycles only for packet processing and exclude the time taken to set up the packet-pointer buffer. We ran single-threaded, 8-way and 32-way multithreaded experiments on our Niagara machine.

4.2. Performance Potential

Figure 5a compares the ideal performance of the G80 to our baseline (Pentium4). We measure this ideal performance by constructing a simple DFA with just one state that remains in this state irrespective of input. The x-axis gives the number of threads and the y-axis shows execution time measured in nanoseconds per byte. With only one thread, we are comparing the performance of one processing element to the Pentium4 processor and it is about four 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, with performance leveling-off at 6.4ns/byte using 8192 threads. Thus, the G80 out-performs the Pentium4 running the same DFA (80 ns/byte) by about 12×. The speedup on this simple DFA places an upper-bound for what the G80 implementation can achieve on real traces. The Core2, which runs at a slightly lower frequency than the Pentium4 but can extract much more ILP, is about twice as fast as the Pentium4. With 32 threads, the Niagara is about 3.2× better than the Pentium4.

Considering peak GOPS, the G80 is capable of approximately 36× better performance than the Pentium4. This lost performance potential is primarily due to memory throughput and latency. The dashed line shows performance when the memory system is idealized for the G80. We create this idealized memory configuration by replacing the accesses to global memory with dummy loads to local memory. Comparing to the real system, we see that after 2048 threads bandwidth becomes a constraint for the real system and performance levels off, whereas the ideal system scales further as it also has infinite bandwidth. Comparing these two systems, we estimate that the effective memory access latency for the real system is 16 cycles, computed by subtracting the difference in execution cycles and dividing by the total number of memory loads.

4.3. Overall Performance

We now present performance results for full traces and signature sets. Table 3 shows the speedup compared to the Pentium4 baseline for the three protocols. For DFAs, the three protocols exhibit similar speedups, averaging 8.6×. This is to be expected because DFAs have regular control flow and all these DFAs have a large number of states. XFAs on the other hand have more divergent behavior, performing about 6.7× better on the G80 overall. HTTP shows the least performance improvement because it executes more XFA instructions on average than the other protocols. The Core2 out-performs our Pentium4 baseline by 2× because it can extract more ILP in the matching routine.

The Niagara platform serves as an evaluation point that is similar to modern network processors, as it can execute 32 threads on a single chip. Figure 5b compares performance of

the G80 and Niagara implementations to our baseline. The GPU bars reflect values identical to those in Table 3. The Niagara can effectively use thread-level parallelism across different packets; with 32 threads it is about 4× better than Pentium4 baseline. However, the G80 still outperforms Niagara by 2.3×. We observe that across the different protocols and traces, relative performance is quite similar.

Note that the GPU execution includes separate time to transfer data to the GPU, execution time of the kernel, and time to transfer data back. With the HTTP protocol, for example, the breakdown of the times for processing 16K packets is as follows: 34ms(transfer), 212ms(compute), and 0.18ms (transfer-back). In general, transfer time is quite small. Double-buffering, which is expected to arrive in next-generation GPUs, can effectively hide this delay. Also, 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. Signature database update time will be dictated by this.

Overall the G80 implementation performs significantly better than the Pentium4 implementation, achieving on average 8.6× and 6.7× speedups for DFAs and XFAs, respectively. However, the speedup is still far below the ideal 36× difference in peak performance between the two systems. Further, based on a comparison of peak performance and achieved performance, we estimate the G80 sustains only 10% of its peak. Below, we discuss potential optimizations to move closer to the ideal.

4.4. Discussion

Hardware caching - Texture memory. 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 are effectively pre-loaded by the CPU.[2] 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. As shown in our performance analysis, memory latencies are not completely hidden by multithreading and caching can help hide this delay.

Two large data structures — the packet buffer and state machine data structure — contribute the most to memory space usage and can potentially benefit. However, accesses to the packet buffer are far fewer than to the state machine data. For every 64 bits of packet data (*e.g.* 2 memory reads), 8 state machine accesses are generated. Thus, state machine accesses dominate and mapping the packet buffer to texture memory did not provide substantial improvements.

On the other hand, the state machine data is an ideal candidate for caching, since the working set is quite small and hardware can capture these "hot" states. However, we cannot map this to the texture memory because it is a recursive data structure. 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. The GPU cannot directly write to the texture memory either.

Caching recursive data structures such as a state machine can yield significant performance improvements. 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.

We can estimate the benefits of such caching using a simple model of memory accesses. From our performance potential experiment we estimate that the average memory access latency is 16 cycles. Thus we can derive the execution time of non-memory instructions for each trace. We can then add in memory access time for different cache scenarios. If the entire working set size fits in the cache, memory access latencies will be the latency to the texture cache. Columns 6-9 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 [18] has a stream register file, and the IBM Cell processor has 256 KB of software managed storage at each core [17]. 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 that can be accessed in four 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. 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. Thus, *large storage is required to be effective.* The caching results in Table 3, Columns 6-9 suggest a large software managed cache can perform well for this application.

Resettable local memories. Local memory state must be cleared before each packet is processed. A hardware extension that clears it automatically could reduce the instruction count for XFAs.

4.5. Limitations and Future Work

The limitations of our prototype are as follows. First, we process packets in batches because of the interface limitations between GPU and CPU. Based on our measurements, we conservatively estimate that a buffer of 2048 packets completely minimizes overheads of data-dependent branching. Second, we

2 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.

are performing only signature matching, which is just one component (albeit compute-intensive) of an IPS. In particular, we do not perform reassembly, normalization, or other common pre-processing tasks. Third, our use of sorting to better examine the potential for SIMD architectures will need further study.

In future work we will examine other functions and explore a full system design, but this architecture does not have any fundamental limitation in realizing a full IPS. For example, stream reassembly can be performed by maintaining per-flow state in memory and loading it up just prior to packet processing. Further, a head-to-head performance comparison against network processors should provide interesting efficiency and cost/performance comparisons.

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.

To the best of our knowledge, our work is the first to present a detailed analysis of signature matching for network processing. Many FPGA and ASIC implementations for high-speed intrusion detection have been recently explored [3, 8, 15, 11, 27]. Tuck *et al.* [31] describe pattern matching optimizations for ASIC as well as software implementations targeted at general purpose programmable processors. Brodie *et al.* [5] describe a pipelined implementation of regular-expression pattern matching that can map to an ASIC or FPGA. Alicherry *et al.* [1] examine a novel Ternary CAM based approach. Tan and Sherwood [28] describe a specialized hardware implementation that performs several pattern matches in parallel where each pattern is a simple state machine. LSI Logic's Tarari 8000 [29] series of content processors boards support efficient regular expression processing. Performance of over 1 million RegEx rules processed simultaneously at deterministic speeds has been reported with their proprietary NFA processing on a specialized regular expression processor [30]. Software-based techniques to improve IPSes and algorithmic improvements are unrelated to our work.

Erez *et al.* [13] describe irregular computation on SIMD architectures, focusing on irregular memory access patterns and software transformations. 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 flexibility can be quite dramatic for signature matching. Bader *et al.* [2] examine irregular applications and a mathematical model for the Cell processor, a multicore 4-wide SIMD architecture.

Jacob and Brodley [16] demonstrate a version of the open source IPS Snort that uses the NVidia 6800 GT graphics card to perform *simple string matching*. This work bears some superficial resemblance to ours, but there are a number of fundamental distinctions. First, Snort uses string matching to pre-filter traffic, but for many packets it must also perform the more complex regular expression matching for all signatures not ruled out by the pre-filter. This makes Snort vulnerable to attacks in which packets are crafted to pass the pre-filter and cause expensive regular expression processing for many signatures. Our own measurements show that such attacks can slow Snort down by more than 100× on a general purpose processor. The magnitude of these attacks is only increased if the string matching pre-filter is run on a fast GPU and regular expression matching is run on a slower general-purpose CPU. Second, our use of more powerful signature matching algorithms that can evaluate multiple complex signatures in a single pass is more representative of current trends in signature matching system design [4, 6, 9, 25, 26, 32]. Finally, our evaluation on a more flexible newer-generation GPU is better suited for judging the suitability of present and future GPUs for such workloads. Our results are much more promising with respect to the potential of GPUs to support higher throughput.

Lastly, for bio-informatics workloads, Cmatch and MuMerGPU [23], provide techniques for exact and approximate string matching, but these do not generalize to regular expressions. Seamans and Alexander [24, 10] discuss ways to map a special type of regular expressions used for antivirus scanning from the open source ClamAV toolkit to a GPU. But, these do not generalize to IPS signatures. Our technique and XFAs can be used to support ClamAV type signatures, but less efficiently. Pharr and Fernando [21] provide a good overview of several high performance applications mapped to GPUs.

6. Conclusion

In this paper, we examined the feasibility of using SIMD architectures 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. Our study examined both DFAs and XFAs for matching signatures to payloads. DFAs require simple processing for each input byte with high memory requirements, whereas XFAs reduce memory and achieve better performance but require more complex and less uniform per-byte processing, which can impact SIMD performance.

To the best of our knowledge, this work is the first to quantify SIMD processing for this application. We implemented signature matching on an Nvidia G80 GPU and observed 6× to 9× better performance than on a Pentium4 system. Our proof-of-concept implementation shows that network devices can offload signature matching to a SIMD engine to achieve cost-effective performance improvements. More generally, regular expression matching is central to many other applications such as network traffic policing, XML processing, and virus scanning; analyses similar to ours may also help such applications benefit from the performance potential of SIMD engines.

References

[1] 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.

[2] 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.

[3] 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.

[4] M. Becchi and P. Crowley. An improved algorithm to accelerate regular expression evaluation. In *ANCS*, December 2007.

[5] B. C. Brodie, D. E. Taylor, and R. K. Cytron. A scalable architecture for high-throughput regular-expression pattern matching. In *ISCA 2006*, pages 191–202, 2006.

[6] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards automatic generation of vulnerability-based signatures. In *IEEE Symposium on Security and Privacy*, Oakland, California, May 2006.

[7] 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.

[8] 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.

[9] Cisco intusion prevention system (Cisco IPS). `http://www.cisco.com/en/US/products/ps6634/products_ios_protocol_group_home.html`.

[10] Clamav: Clam antivirus. Available at `http://www.clamav.net/doc/latest/signatures.pdf`.

[11] 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.

[12] Shader Model 4 (DirectX High Level Shading Language) Available at `http://msdn.microsoft.com/en-us/library/bb509635(VS.85).aspx`.

[13] M. Erez, J. H. Ahn, J. Gummaraju, M. Rosenblum, and W. J. Dally. Executing irregular scientific applications on stream architectures. In *ICS 2007*, pages 93–104.

[14] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic warp formation and scheduling for efficient gpu control flow. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 407–420, 2007.

[15] 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.

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

[17] 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.

[18] 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.

[19] NVIDIA corporation. Technical Brief: NVIDIA GeForce 8800 GPU Architecture Overview, 2006.

[20] NVIDIA corporation. NVIDIA CUDA Programming Guide, 2007. Available at `http://developer.nvidia.com`.

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

[22] M. Roesch. Snort - lightweight intrusion detection for networks. In *13th Systems Administration Conference*. USENIX, 1999.

[23] M. C. Schatz, C. Trapnell, A. L. Delcher, and A. Varshney. High-throughput sequence alignment using graphics processing units. *BMC Bioinformatics 8:474*.

[24] E. Seamans and T. Alexander. *Chapter 35: Fast Virus Signature Matching on the GPU, Gems 3. Editor: Hubert Nguyen.* Addison-Wesley Professional.

[25] R. Smith, C. Estan, and S. Jha. XFA: Faster signature matching with extended automata. In *IEEE Symposium on Security and Privacy*, May 2008. Oakland.

[26] R. Smith, C. Estan, S. Jha, and S. Kong. Deflating the Big Bang: fast and scalable deep packet inspection with extended finite automata. In *SIGCOMM*, Aug. 2008.

[27] 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.

[28] L. Tan and T. Sherwood. A high throughput string matching architecture for intrusion detection and prevention. In *ISCA 2005*, pages 112–122, 2005.

[29] Tarari 8000. Online http://www.lsi.com/DistributionSystem/AssetDocument/documentation/networking/tarari_content_processors/LSI_PB_2pg_GP8000up.pdf.

[30] Tarari T10 Technology. Online http://www.lsi.com/DistributionSystem/AssetDocument/LSI-PB_2pg_T10_Silicon0923.pdf.

[31] N. Tuck, T. Sherwood, B. Calder, and G. Varghese. Deterministic Memory-Efficient String Matching Algorithms for Intrusion Detection. In *INFOCOM 2004*, pages 2628–2639, March 2004.

[32] 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 2006*, pages 93–102, 2006.