Offloading IDS Computation to the GPU

Nigel Jacob
Tufts University
Computer Science Department
161 College Ave, Medford, MA 02155
nigel.jacob@tufts.edu

Carla Brodley
Tufts University
Computer Science Department
161 College Ave, Medford, MA 02155
carla.brodley@tufts.edu

Abstract

Signature-matching Intrusion Detection Systems can experience significant decreases in performance when the load on the IDS-host increases. We propose a solution that off-loads some of the computation performed by the IDS to the Graphics Processing Unit (GPU). Modern GPUs are programmable, stream-processors capable of high-performance computing that in recent years have been used in non-graphical computing tasks. The major operation in a signature-matching IDS is matching values seen operation to known black-listed values, as such, our solution implements the string-matching on the GPU. The results show that as the CPU load on the IDS host system increases, PixelSnort's performance is significantly more robust and is able to outperform conventional Snort by up to 40%.

1 Introduction

A signature-matching intrusion detection system (IDS) is susceptible to attacks that flood the IDS with packets (possibly containing attack signatures). When under such an attack, an IDS can fail in two ways that can be described as 'open—closed'. Firstly (open), it can fail by packets being dropped by the host operating system as it attempts to throttle packet processing. Secondly (closed) it can fail through either the host and/or the IDS crashing. We propose a solution to both these scenarios by using the graphics processing unit (GPU) resident in video-cards as an auxiliary processor for the purposes of off-loading some portion of the packet processing to the GPU. Wallach, et al [2] describe a class of algorithm complexity attacks which would likely be effective against a GPU-based solution such as we have developed, but we have not explored this attack-class in this paper.

Current-generation GPUs are designed to act as high-performance stream-processors. Their performance is derived from parallelism at both the data and instruction-level. Additionally, current-generation GPUs are architected to take advantage of the independence of the data-elements in traditional graphics scenarios. This data-independence enables a memory-model that is simple and fast. Although these characteristics allow for high levels of performance, they pose challenges when conventional (i.e., non-graphical) applications are ported to the GPU. The streaming architecture can be a difficult programming environment for developers accustomed to working with traditional CPUs. In particular, the simplified memory model is restrictive when it comes to implementing well-known data-structures such as linked-lists and trees. Thus, although the GPU provides a new opportunity for optimizing the runtime performance of conventional algorithms, these algorithms must first be redesigned into a form that is appropriate for the GPU's parallel, stream-architecture.

In this paper we use the GPU to our advantage by demonstrating that offloading IDS computation to the GPU allows the IDS to function effectively at significantly higher packet-processing and CPU-load. To this end, we ported the Snort Intrusion Detection System [1], an opensource signature-matching IDS, to the NVidia 6800 GT graphics card. In our experimental work we simulated an attack on both our GPU-ported version of Snort (called PixelSnort) and standard Snort. The results show that as the CPU load on the IDS host system increases, and IDS performance decreases, PixelSnort's performance is significantly more robust and is able to outperform conventional Snort by up to 40%.

The idea of using ASICs for IDS is not new; Tiger[39], Puma[37], Lynx[38] have each employed ASICs for offloading and have demonstrated good performance under various loads. The contribution of this paper is of a proof-of-concept that the GPU can also be used for offloading and present a new avenue of exploration for researchers.

We proceed in the remainder of the paper as follows. In Section 2 we survey the related work in IDS performance issues and existing approaches to mitigate these problems. In Section 3, we introduce PixelSnort as an implementation of Snort that makes use of the GPU for string-matching. Section 3 also provides an overview of issues related to developing code for the GPU both in terms of the relevant API's and architectural considerations. Section 4 describes our experiments and shows that PixelSnort is able to outperform Snort for certain workloads. Finally, in Section 5 we draw conclusions and discuss future directions for this work.

2 IDS Performance Issues

Signature-matching intrusion detection systems have two types of performance limitations: 1) CPU-bound limitations that arise due to string-matching and 2) I/O-bound limitations caused by the overhead of reading packets from the network interface card. This classification scheme is borne-out by runtime profiling results for Snort. For example, Ranum's [18] runtime profile of Snort yielded the following breakdown of Snort's wall-clock time : 33% reading packets off the network interface, 33% pattern matching and 33% in performing some action based on the patterns in the packets (such as alerting the sysadmin that an attack is occurring). In practice, actual runtime breakdown varies with the network traffic and Snort's configuration. For example, under high network load and a Snort configuration that is more likely to cause more packets to be matched by the rules, the relative time spent by Snort doing pattern matching will be higher. In [16], Jalote, et al measured Snort's string-matching as taking up to 60% of total runtime.

At the time of this writing, the NVidia 6800's native datatype is 24-bit floating point, hence it's raw performance is measured in floating-point-operations-per-second (flops). This accuracy limit will likely be overcome in future versions with wider floating point implementations.

2.1 CPU-Bound Limitations

In the case of CPU-bound limitations, performance issues arise due to other applications executing on the IDS host which causes the load on the system to increase. Once this occurs, all applications on the host will experience a loss in runtime performance as each application receives fewer time-slices per unit time. In an IDS system, load has a direct impact on performance. The central operation of a signature-matching IDS is a comparison of incoming network packets to a set of signatures (or fingerprints) of known attacks. These signatures consist of a set of bytes that occur within the context of an attack. For example, in the case of a virus, the signature could be a few bytes from the viruses shell-code which may be known to be invariant across instantiations of the virus [36]. The comparison of packets to signatures is a string-matching operation which is CPU-intensive.

Snort's string-matching algorithm has changed over the lifetime of the project. Snort v1.* used the Boyer-Moore [3] algorithm which displayed good general performance, but did not scale well with increasing numbers of incoming packets. Recently, Snort has moved to a more sophisticated scheme which uses a packet pre-processor and the Aho-Corasick algorithm [4] to achieve scalable performance. The GPU-solution presented in this paper does not use the Aho-Corasick algorithm but rather makes use of a parallel string-matching algorithm that is more readily adapted to the GPU's architecture (see Section 3.2.2 for details).

Research into improving the runtime of the packet comparison falls into three categories: improvements in the string-matching algorithm portion of an IDS [28],[29], utilizing packet-header characteristics to optimize comparison [27] and using hardware to improve IDS performance [16]. The first two approaches are complementary and have been incorporated into most software-based IDSs, while the third approach has been widely deployed in router technology [16].

2.2 I/O-Bound Limitations

Reading packets from the network interface card (NIC) is a major operation of an IDS. Packet reading can become a bottleneck when the number of packets overwhelms the IDS host's internal packet buffers. Our solution does not address I/O-bound-related issues, rather, it focusses on lessening the CPU-load of the IDS host.

3 PixelSnort

Our solution to the problems of IDS performance is to off-load packet processing to the GPU. Initially, we foresaw two potential benefits from this approach. Firstly, by off-loading possibly one-third of the processing that the IDS performs, the CPU would be left to handle other tasks. Secondly, the GPU's performance characteristics are well known so there is the possibility of accelerating the packet processing. For example, the NVidia 6800 GT (which was used in our research) is capable of performing 54 GFlops [7] as compared to the Intel Pentium 4 Xeon 2800 MHz which is rated at 5.6 GFlops [8]. In this section we first discuss the issues related to programming the GPU, we then describe our efforts to port Snort's string-matching to the GPU and we conclude with a description of PixelSnort's parallel-packet string-matching algorithm.

3.1 GPU Programming With Cg

Programming the GPU can be difficult for developers who have never written graphics code. The architecture of the GPU and the nature of the graphics APIs used to control the state of the GPU place restrictions on the types of algorithms that will run efficiently on the GPU. Indeed, the first task before porting any application to the GPU is to determine how to map the CPU algorithm, which implicitly assumes a Von Neumann architecture, to the parallel, stream-architecture of the GPU. For this reason, before introducing PixelSnort, we provide a brief overview of GPU programming.

Cg[1] runs as an embedded GPU language in conjunction with another graphics API, either OpenGL or DirectX (Microsoft). OpenGL and DirectX are approximately equivalent graphics programming APIs and differ primarily in the governing bodies which dictate the changes that will be made to the respective API's. Our research was done exclusively using OpenGL because it is an open-standard API. Cg differs from CPU programming languages such as C in that: 1) Cg lacks pointers, and 2) Cg supports certain datatypes that are optimized for GPU execution such as matrices and vectors.

The GPU consists of two sub-processors: the vertex processor and the fragment processor. Cg programs can be written for either processor, called respectively vertex shaders and fragment shaders. The vertex processor is responsible for transforming the input vertices according to changes to the geometry of the vertices. For example, the input vertices may represent a square that the vertex shader will transform into a circle. The output of the vertex processor are "fragments" that represent potential pixels. The fragment shader takes these fragments and can apply a texture and/or possibly discard the fragment if some criterion is met. For example, if the coordinates for a fragment lie outside of some region, the fragment processor/shader can discard the current fragment. The output of the fragment processor is pixels that are written to the framebuffer, which is a memory region that can either be written to the output device (e.g., the monitor) or written to a texture which can then be returned to the CPU (in the case of a general-purpose CPU computing algorithm). In Figure 1 we show the graphics pipeline in terms of the major components.

General-purpose GPU (GPGPU) research has focused on using the fragment processor (and hence the fragment shader) [33] [34] [35]. There are several reasons for this, however, the most important reason is that the fragment shader has direct access to the framebuffer (as can be seen in Figure 1) and thus is in a better position to discard pixels from being written to the framebuffer. The vertex processor has no such ability to discard vertices. In our research, we have used the fragment processor exclusively.

[1]There are two GPU programming languages in general use (BrookGPU [9] is an example of a non-graphical GPU programming language): Cg [11] (C for Graphics) and GLSL[12] (OpenGL Shading Language). Cg was designed principally by NVidia while GLSL was developed by the OpenGL Architecture Board [10]. Although Cg in principle can run on ATI hardware, NVidia is optimized for Cg making it the preferred language for NVidia hardware.

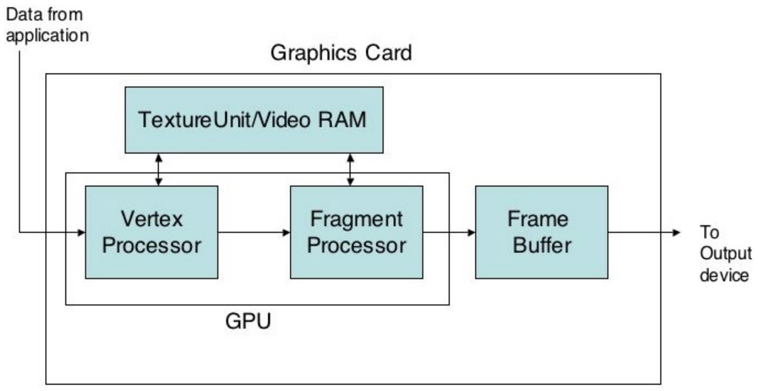

Figure 1. The graphics pipeline

3.2 Porting Snort to Cg

Our research consists of porting part of the Snort Intrusion Detection System (IDS) to the GPU. Snort is a signature-matching opensource IDS. Signature-matching refers to the technique of using a set of bytes (or strings) that are known to appear in packets that are propagating a particular attack across the network. This set of bytes and/or characters serve as a signature for the attack. During operation, the IDS scans the network traffic for packets containing the attack signatures (an IDS will scan each packet for many different attack signatures). If an attack is detected, the IDS will raise an alarm of some sort depending on how Snort is configured.

Porting the string-matching portion of Snort to the GPU consists of adding OpenGL function calls to the Snort source code and re-writing Snort's string-matching function as a Cg fragment shader. The OpenGL calls are responsible for loading the shader. In Figure 2 we show Snort's architecture. The upper right hand portion of the figure indicates the point at which the Cg fragment shaders were incorporated into the rest of Snort. The GPU presents us with a number of architectural features that must be considered before any code may be ported from the CPU to the GPU.

Parallel Rendering Pipelines. The rendering pipeline (also known as the graphics pipeline) is the processing pipeline on the graphics card that produces (i.e., renders) pixels to be displayed by the output device (e.g., by the monitor). Modern GPUs have multiple, parallel rendering pipelines (16, for the NVidia 6800 GT). The inherent parallelism in graphical data enables a computational model in which each data-point can conceivably be handed to a different rendering pipeline. Although there are not

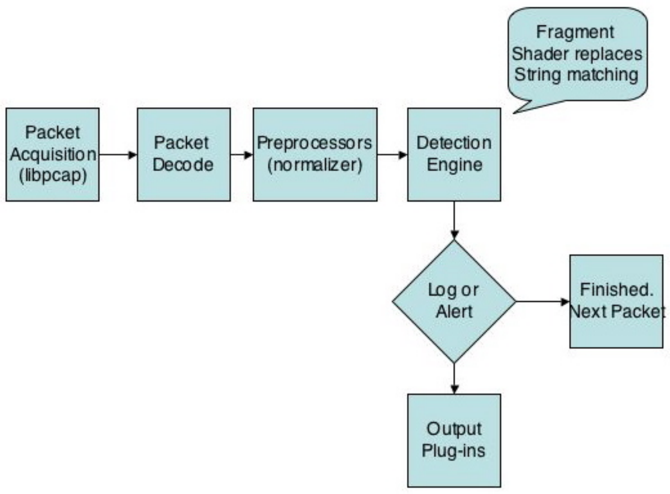
Fragment Shader replaces String matching
Packet Acquisition (libpcap)
Packet Decode
Preprocessors (normalizer)
Detection Engine
Log or Alert
Finished. Next Packet
Output Plug-ins

Figure 2. Snort's architecture

enough rendering pipelines to process each data-element one per pipeline (assuming that there are more than 16 data-points under consideration), this is the abstraction that the developer should use in thinking about how the rendering pipeline will treat the data. This architectural parallelism requires that any algorithm and data to be ported to the GPU must be inherently parallel. In our case, we have used this parallelism to implement a parallel string-matching algorithm which we describe in detail in Section 3.3.

Passing Data to the GPU. OpenGL provides a number of mechanisms to pass data to a Cg shader. However, in applications in which large volumes of data are to be handled, there are two possibilities: 1) via vertices and 2) via textures. Vertices are the standard geometric primitives that the graphics pipeline uses to render pixels. They represent the vertices of shapes that appear on the output devices. However, vertices must be passed to the GPU one-by-one. Textures, on the other hand, can be used to send large amounts of data to the GPU all at once. Textures are two or three dimensional arrays of points that are used to give vertices a textured appearance. For example, a jpeg texture of a stone-wall could be applied to a sphere to give it the appearance of being made of stone. Higher-end graphics cards have larger on-card video RAM and as such can hold larger textures. Texture image-sizes of up to tens of megabytes are now routinely used in some graphical applications. In the case of general-purpose computing, a texture can be used to encode data that the shader will use as input. In the present solution we encode packet-data and string-matching rules as textures and pass them both to the shader that then performs string-matching on the input textures.

Memory Model. Each data element in a texture (texel) is independent, enabling the GPU to use a simplified memory model that eliminates overhead due to memory management and data synchronization. The ideal GPGPU application would be able to perform all of it's computation in a single rendering pass. However some circumstances require multiple rendering passes in which the output of one pass is used as the input to the next rendering pass through the pipeline. Textures as described above are used to encode data and are then passed into the shader. Shaders are not able to read and write to the same texture in the same rendering pass. In graphics jargon, we say that shaders are capable of gather (the shader can gather information about the points surrounding the current point) but not scatter (the shader can not update surrounding points in the texture). Thus, textures can be used as input, but not directly as output. Output from the shader is written to the framebuffer.

Download/Upload Asymmetry. In conventional graphics applications, vertices, textures and other variables are passed (downloaded) to the shader for computation, but the output from the framebuffer is rarely read back (uploaded) into main memory. The result of this fact is that developers of graphics drivers have optimized download bandwidth to the GPU but not upload from the GPU. This asymmetry in download/upload bandwidth means that general-purpose computing applications must be careful in designing applications that make use of the data computed at the end of the pipeline (i.e., in the framebuffer). This is not an issue in PixelSnort because the output in the framebuffer is used as a conditional array indicating whether or not a particular string-matching operation is successful which is then passed back to the CPU for post-processing.

Vertex Processor versus Fragment Processor. Most research into GPGPU has been done using the fragment processor and fragment shaders [35] [34] [33]. There are several reasons for this:

1. The fragment processor is able to discard pixels.
2. The vertex processor traditionally has not been able to access the texture unit (this is no longer the case in the most recent GPUs).
3. The graphics pipeline imposes limits on the number of instructions in a shader. The shader instruction-length limits have traditionally been different for vertex shaders and fragment shaders with vertex shaders being the smaller of the two (in newer GPUs this difference is decreasing).

GPGPU Execution and Graphical Process Execution. The graphic device driver is designed to drive the GPU in an optimal fashion. It maintains state for each graphical process running on the GPU at any given time. As such,

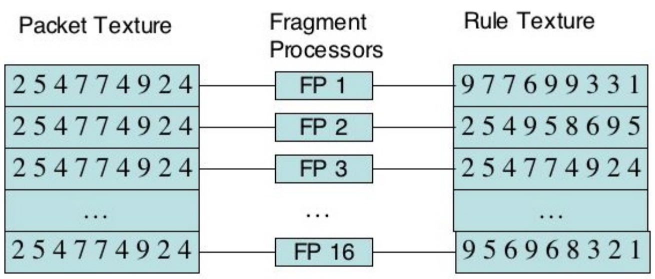

Figure 3. Packet-texture, Rules-texture and fragment processor.

the device driver also causes the GPU to update the output display when it is given an execution context by the host operating system during multi-tasking.

3.3 String-Matching on the GPU

In order to perform string-matching on the GPU, there are two issues that must be considered: 1) how strings will be encoded, and 2) what algorithm to use to perform the string-matching.

OpenGL textures are n-by-n "arrays" of 32-bit values that are interpreted by the graphics pipeline as the Red, Green, Blue and Alpha (transparency) components for a given texture element (texel). Each of the RGBA components is 8-bits in length (8-bits * 4-components = 32-bits). Eight-bits is sufficient to encode an ASCII character in one of the RGBA components. Thus in a single texel we can encode four ASCII characters. The GPU/Cg can perform comparisons on each of the components in a pixel in parallel (i.e., in a single comparison we can check whether the R, G, B and A components of two pixels match). The rule-data may be longer than a row in the two-dimensional texture. In this case, the remaining characters of the rule-data are written to the next line of the texture. Once the Snort-rules have been converted to a texture, the texture is downloaded to the GPU using OpenGL API functions. Snort rules are of the form:

```
log tcp 130.64.1.83 110 ->
192.168.1.12 111 (content: "000186a5";
msg: "external mountd access";)
```

This rule says: For any TCP packets originating from 130.64.1.83 whose destination is port 111 on 192.168.1.12 and whose payload contains '000186a5', write to the log-file the message 'external mountd access'. The source and destination IP, source and destination port numbers, and the content string are all ASCII-encoded and then written to a row of the texture. If the rule-string is longer than the current row, the ASCII character CR (carriage return) is written to the end of the current row which indicates to the fragment shader that the rule continues on the next row. The remainder of the rule-string is then written to the next row. The rule is terminated with the ASCII character EOT (end of text).

Using an ASCII encoding, the source IP address of the above rule would be converted to:

```
 1   3   0   .   6   4   .   1   .   8   3...
061 063 060 056 066 064 056 061 056 070 063...
```

The same technique would be used to encode the rest of the Snort-rule (i.e., the source port, the destination port, the destination IP and the content string) into the texture. The packet is encoded in the same way and with these same fields.

Snort uses a Finite State Machine [16] as part of the Aho-Corasick [4] algorithm to perform string-matching. Implementing such a data-structure entirely on the GPU would be difficult given the GPU's simple memory model. Thus we opted instead to use the GPU's parallel rendering pipelines to build a parallel-packet string-matching algorithm.

On PixelSnort system initialization we first load the rules-texture by:

1. Taking the strings from the Snort-rules and encoding them using the standard (octal)ASCII encoding, and
2. Writing the encoded ASCII strings into a two-dimensional texture in which each row of the texture corresponds to a Snort-Rule. The rules-texture is then downloaded to the GPU.

To process packets we:

1. Retrieve a packet. Encode the packet as an ASCII-string. Write N copies of this ASCII-string to the packet-texture one per row for each of the N Snort rules.
2. Send the packet-texture to the GPU.
3. Each of the 16 fragment shaders takes a copy of the packet-string from the packet-texture and compares it to a Snort-rule-encoded string from the rules-texture using the Knuth-Morris-Pratt algorithm [14]. In this way, 16 Snort-rules are compared against the packet-data in parallel (See Figure 3).
4. The results of the comparison are written to the frame-buffer and then sent to the CPU.

The core string-matching algorithm is a simplified version of the Knuth-Morris-Pratt algorithm[14] which compares two strings, called the pattern and the text, by sliding the pattern across the text and recording the matches. In the worst case, this algorithm does O(nm) comparisons where n is the length of packet-string and m is the length of the Snort-rule string.

Step 4 consists of two sub-steps. The first sub-step is to determine whether there were any matches from the comparison. If a match is found, the fragment shader will write to the framebuffer, otherwise the shader will discard the packet. We use a graphics mechanism called occlusion-query to determine whether the shader wrote anything to the framebuffer. An occlusion-query is an OpenGL-supported mechanism for gathering statistics on framebuffer activity. The second substep is to return the framebuffer to the CPU if a packet-signature match was found. This "read-back" is accomplished by another OpenGL technique called copy-to-texture (CTT). In CTT, the framebuffer is copied to a texture using the OpenGL glReadPixels() function (recall that in most graphical applications, the framebuffer contents is sent to the video-output device and is never seen by the CPU).

4 Experiments

The primary task for our experiments was to use different system workloads to characterize PixelSnort's full-range of runtime behavior. Generally, system benchmarks stress certain subsystems in order to determine how the system as a whole reacts. In a modern system, the video subsystem contains it's own bus (AGP, the Accelerated Graphics Port), processor (the GPU) and memory (on-card video RAM), making it fairly segregated from the rest of the system. Thus, because PixelSnort treats the video subsystem as a computational unit, our testing focussed on stressing the following subsystems:

1. CPU: PixelSnort's goal is to try and offload computation from the CPU to the GPU, so we tested this ability by loading the system with CPU-bound tasks.

2. Memory: Many applications make heavy use of memory, so we tested PixelSnort to determine how it reacts in the presence of memory-intense tasks.

3. IO: Filesystem operations are slow and as such should not affect PixelSnort (which makes little use of filesystem operations).

4. Network-load: PixelSnort processes data originating in the networking subsystem. This test was designed to determine its performance in the presence of heavy network traffic.

For the experiments all non-necessary services and daemons were turned off to measure IDS performance in relation to CPU-load. Specifically, the only daemons left running were the various kernel daemons. At the time of our research, the NVidia 6800 GT (256MB RAM) graphics card was the highest-end commercial graphics card on the consumer market. The features and capabilities of graphics cards improve significantly with each new graphics card. These new features result in new language features for Cg. Our proposed solution of running Snort on the GPU required language features that the previous family of NVidia cards did not support, requiring us to use the NVidia 6800 GT.

The string-matching time was used as a measure of performance because string-matching was identified as being the major time-consuming operation in the Snort IDS. String-processing time was measured using wall-clock time. For Snort, wall-clock time was captured using Snort's built-in timing function. For PixelSnort, wall-clock time was measured by taking a system-time snapshot before and after a batch of packets was downloaded GPU and the the results were uploaded to the CPU. There is no means to perform timing measurements within the GPU, so downloading and uploading of data to and from the GPU must be included in all our timing measurements.

4.1 CPU Load

The CPU-load experiments were designed to test PixelSnort's performance as a function of the general CPU-load on the system. CPU, memory and filesystem loading was induced using a number of benchmarks. Although benchmarks from Contest [19], LMBench [20], DBench [21] and Linux Test Project [22] we conducted, we show results from Contest due to space limitations. The results for the other benchmarks showed similar performance trends. From the Contest benchmark we chose the following representative sample Process, IO, Memory, Read and List Load (listed in Table 1).[2] Each of these experiments took approximately thirty minutes to execute. Figure 4 shows the results of these experiments.

In order to sample the CPU-load we periodically captured the contents of the file /proc/loadavg. This file is part of the Linux virtual filesystem. The proc filesystem provides access to various aspects of the running kernel. The load average value in /proc/loadavg is the average number of processes in the kernel run queue. A load value of 1 indicates that the CPU's time slices are completely being used.

The first observation is that for the Process (Figure 4a) and Memory (Figure 4b) PixelSnort's string-matching outperforms Snort as CPU load increases beyond 50%. For the Process Load benchmark, there is a larger performance difference which we attribute to the fact that this is a CPU-intensive task (creating new processes consists primarily of

[2] Note that these are the names given in Contest and "IO" is misleading, because it copies from /dev/zero to memory and is not a test of IO from the network.

Table 1. Test Matrix

Test Name	Description	Goal
Process Load	Fork and exec process	Process fork'ing and exec'ing is performed by the kernel and consists of memory operations and CPU operations.
Memory Load	Repeatedly reference 110% of RAM in a pattern designed to cause cache misses.	Memory-intense task.
IO Load	Copies /dev/zero continually to a file the size of the physical memory	Memory and Filesystem-heavy task
Read Load	Reads a file the size of the physical memory	Filesystem-heavy task.
List Load	Lists the entire file system (ls -lRa /).	Walking the data-structures related to the filesystem (Inodes) is memory-read intense and the filesystem reads are IO-intense tasks.
RootFu	Snort/PixelSnort process the packet-capture from the DefCon Capture-The-Flag contest	Networking-stack intense task where PixelSnort will be detecting numerous attack-signatures.
Random	Snort/PixelSnort process random packets generated by ISIC	Networking-stack intense task where PixelSnort should detect few or no attacks-signatures.
Random and Process-Load	Test how networking-load. and CPU-load affect PixelSnort	Test combined load

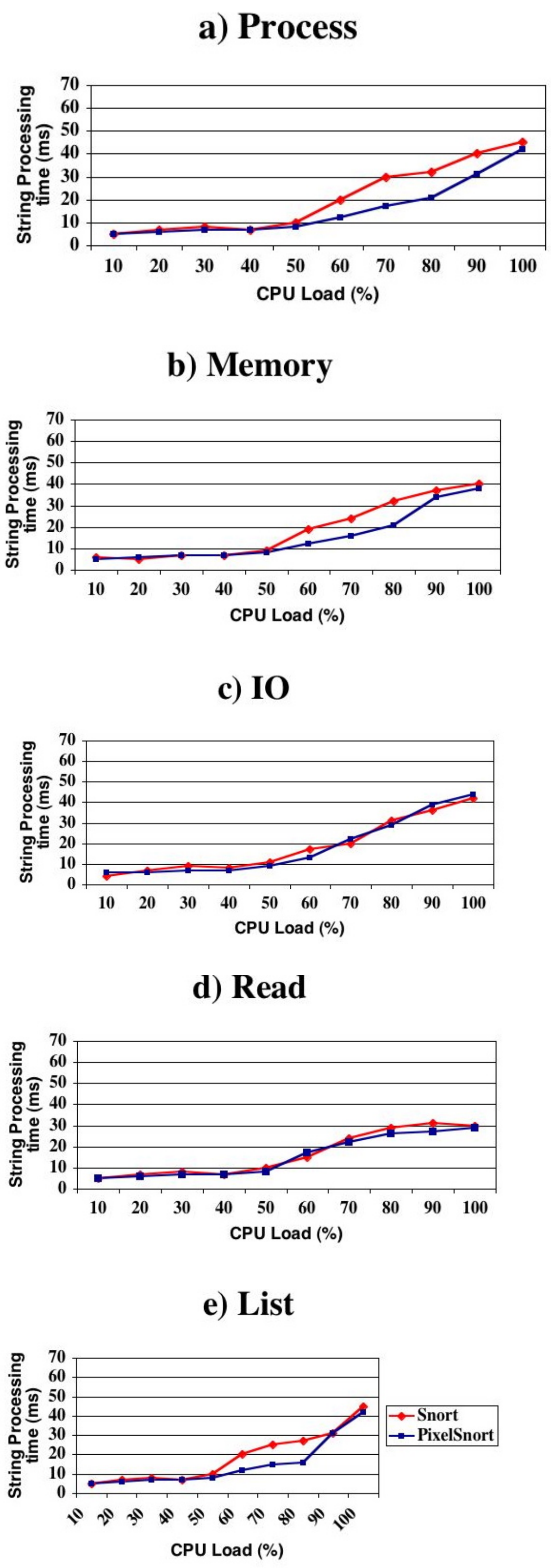

Figure 4. CPU Load Results

CPU operations). PixelSnorts performance increase over Snort for the Memory Load benchmark occurs because PixelSnort does not use RAM for its string matching and therefore loading the RAM in the CPU has little affect – it does however affect PixelSnort's ability to load textures (which is done in the CPU) to the GPU and hence we see some performance degradation. However, note that the dedicated AGP bus between the CPU and the GPU means that the overhead of downloading/uploading data to and from the GPU does not incur any computational cost to the rest of the program (i.e., the CPU portion of PixelSnort).

For the IO(Figure 4c) and Read (Figure 4d) benchmarks the performances of Snort and PixelSnort are indistinguishable. Although the operation of the GPU is independent of file system operations, the performance of PixelSnort degrades because the performance of the entire system is affected. Under a heavy file operation load, the CPU portion of PixelSnort is not able to get enough time slices to even send any computation to the GPU.

The final benchmark, List (Figure 4e), is interesting because its expected behavior would have been similar to IO and Read. However, the data-structures corresponding to locating files on the disk (i.e., the Inodes) are in memory for the most part, making the major operation of this benchmark walking the in-memory data-structure. Therefore the performance of PixelSnort relative to Snort is similar to that of Memory (Figure 4b).

PixelSnort degradation in performance at extremely high loads converges to that of Snort. This suggests that the CPU-bound aspects of PixelSnort eventually dominate the overall behavior. In particular, the overhead of setting-up of the textures before the GPU computation and the decoding of the results after the GPU computation are likely suspects for performance degradation. In conclusion, the experiments illustrate that in the presence of filesystem-intensive tasks, PixelSnort provides no measurable benefit. However, in the presence of CPU or memory-intensive tasks, PixelSnort can be used effectively to provide a performance advantage over Snort under loads from 50-85%..

4.2 IO Load Tests

The second set of experiments were designed to test PixelSnort's performance as a function of the packet-load on the system. Packet load was induced using 1)ISIC [23] network benchmarking tool, which can craft random packets at varying rates (as determined by the user) and inject them into a network, and 2) the DefCon 11 RootFu packet-capture, which was re-played and injected onto the test network [15]. The two benchmarks are listed in Table 1 as Random and RootFu respectively. Packet-load here is a measure of the throughput on the network. The test network was a 100Mb/s Ethernet with a consumer-grade network-switch.

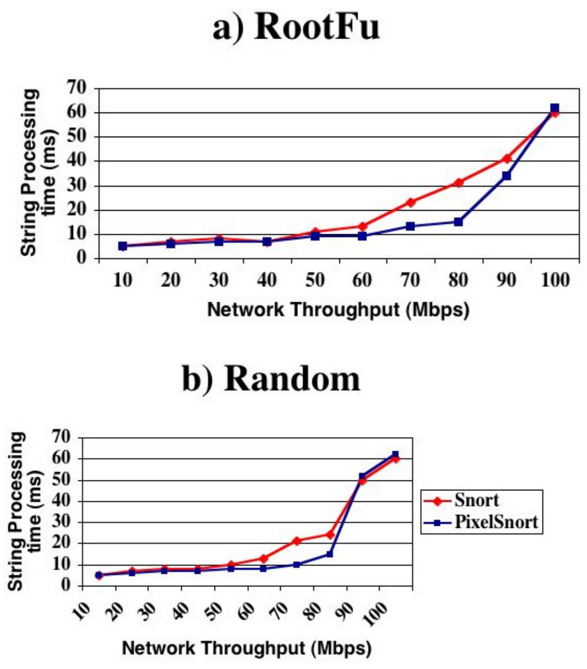

Figure 5. Network Load Results

The goal of our experiments is to measure PixelSnort's performance as a function of packet load. The results of the experiment are shown in Figure 5. Note that each experiment took approximately thirty minutes to execute.

We observe that for both Random and RootFu, as the packet load increases, the wallclock time of both Snort and PixelSnort remains fairly constant until the throughput reaches approximately 80Mbps. We monitored the network throughput on the switch and noted that at about 80Mbps packets began to be dropped. Notice that both Snort and PixelSnort have the same performance, with PixelSnort having a small advantage in the 60-80Mbps range.. By the 80Mbps point, the system was generally becoming non-responsive in a qualitative sense. This seems to be related to the previous experiment in that there is a certain point at which the CPU-bound behavior of PixelSnort dominates the application as a whole.

4.3 CPU and IO Load

The final experiment involved combining CPU-load and packet-load. For this test we combined the Process-Load scenario with the Random packet load. The results are shown in Figure 6. The results are consistent with the results of both the Process-Load and Random-Packet-Load tests. Here PixelSnort experienced superior performance as compared to Snort for the same reasons that PixelSnort outperformed Snort for Process and Random.

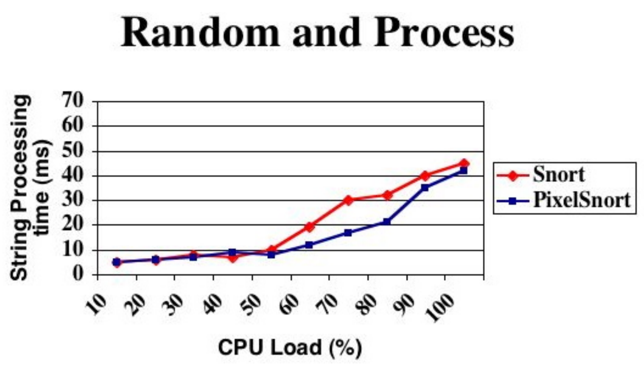

Figure 6. Combined Process-Load and Random-Packet-Load

An interesting question not explored in this paper is the security of PixelSnort itself. More generally, GPUs have received little attention as to their vulnerability to attack. Simple denial-of-service attacks are possible against the GPU by running fragment/pixel shaders that contain tight loops with no operations performed in the loop body. This has the effect of preventing the display (as was discovered accidentally during development) from updating. In order to take advantage of such an attack, the attacker would have to design a system that could load custom shaders into the GPU by comprising some aspect of the graphical subsystem such as the device driver.

The results of our experimentation may seem to suggest that GPUs do not show much promise as offloading co-processors. However, the purpose of this work was to show that GPU's can be used for tasks that are of interest to the security community. Beyond this, it becomes a question of code-optimization.

GPU internal-architecture and device driver source code have till this point in time been closed. As a result, developers have had very limited ability to optimize GPU code for general purpose computing. Shader code compilation and 'optimization' are device driver functions and are specifically tuned for graphics workloads. However, one area of development/research in particular that promises to improve GPGPU code optimization is hardware-virtualizing languages/frameworks (eg: BrookGPU [9]). These programming languages are designed to provide a more general set of programming primitives to programmers. They also provide a virtualization layer to the graphics hardware so that programers do not have to manually manage how registers, etc are being used by the code.

5 Conclusion and Future Work

We have presented a solution signature-matching Intrusion Detection System performance issues. Our solution is based on offloading packet-processing from the CPU to the GPU thus using the GPU as an auxiliary processor. Our experimentation involved porting a well-known IDS to the GPU using the Cg programming language. The stream-based architecture of the GPU imposes certain constraints that presented various technical challenges. Our experimental results demonstrate that for CPU-intensive workloads, the GPU can successfully be used to off-load computation from the CPU.

Our work was a fairly straightforward port of Snort's string-matching operations to the GPU. Further development and re-architecting of PixelSnort would undoubtedly result in a software architecture better suited to taking advantage of the GPU's architecture. For example, GPU programming APIs such as OpenGL are sophisticated and thus provide many opportunities for optimizations of various sorts that would almost certainly improve the runtime characteristics of PixelSnort. One area in particular in need of optimization is the string-matching algorithm itself. The string-matching algorithm we used is fairly simple [14] and may not be the most optimal solution for performing string-matching on the GPU.

One of the initial assumptions in our research was that the GPU may be able to accelerate Snort's packet processing given the GPU's 54 GFlops versus the approximately 5.6 GFlops of the Intel Pentium 4. However, we noticed during experimentation that there was no appreciable speed-up in packet processing under normal-load conditions. It is likely that there are further optimizations that could be carried out to adapt Snort to run on the GPU that would further improve the packet-processing time.

References

[1] "Snort Intrusion Detection System," http://www.snort.org/

[2] S.A. Wallach and D.S. Wallach, "Denial of Service via Algorithmic Complexity Attacks", *Usenix Security 2003*

[3] R.S. Boyer and J.S. Moore, "A fast string searching algorithm," *Communications of the ACM* 20, pages 762-772, 1977.

[4] A.V. Aho and M.J. Corasick, "Efficient String Matching: An aid to bibliographic search," *Communications of the ACM*, pages 333-340, 1975.

[5] W. Fenner, G. Harris and M. Richardson, "TCPDump," http://www.tcpdump.org/

[6] T.H. Ptacek and T. N. Newsham, "Insertion, evasion and denial of service: Eluding Network Intrusion Detection," Technical report, Securenetworks.com, January 1998.

[7] G. Torres, "GeForce 7800 GTX Launch Coverage" http://www.hardwaresecrets.com/article/156

[8] Intel Pentium 4 Xeon, http://www.top500.org/

[9] I. Buck, T. Foley, D. Horn, J. Sugerman, P. Hanrahan, M. Houston and K. Fatahalian, "BrookGPU," http://graphics.stanford.edu/projects/brookgpu/

[10] OpenGL, http://www.opengl.org/

[11] Cg, http://developer.nvidia.com/page/cg_main.html

[12] GLSL, http://www.opengl.org/documentation/oglsl.html

[13] AGP, http://www.devhardware.com/c/a/Video-Cards/AGP-8X-A-Closer-Look/

[14] D. E. Knuth, J. H. Morris, Jr., and V. R. Pratt, "Fast pattern matching in strings," *SIAM J. Computing* 6, 323-350, 1977.

[15] DefCon Capture The Flag, http://www.shmoo.com/cctf/ctl.shtml

[16] A. Jalote, T.N. Vijaykumar and C.E. Brodley, "Advanced Memory Optimizations for String Matching in Intrusion Detection Systems," Purdue University Technical Report, 2004.

[17] H. Debar, M. Dacier and A. Wespi, "Towards a Taxonomy of Intrusion-Detection Systems," *Computer Networks*, Vol. 31, No. 8, 23, pp. 805-822, 1999.

[18] M.J. Ranum, "Experiences Benchmarking Intrusion Detection Systems," NFR Security White Paper, December, 2001.

[19] Contest, http://members.optusnet.com.au/ckolivas/contest/

[20] LMBench, http://www.bitmover.com/lmbench/

[21] DBench, http://www.dbench.org/home/index.php

[22] Linux Test Project, http://sourceforge.net/projects/ltp/

[23] ISIC – IP Stack Integrity Checker, http://www.packetfactory.net/Projects/ISIC/

[24] R. Lippmann, J.W. Haines, D.J. Fried, J. Korba and K. Das, "The 1999 DARPA Off-Line Intrusion Detection Evaluation," *Computer Networks*, pp. 579-595, 2000.

[25] L. Schaelicke, T. Slabach, B. Moore and C. Freeland, "Characterizing the Performance of Network Intrusion Detection Sensors," *Recent Advances in Intrusion Detection: 6th International Symposium*, RAID 2003, Pittsburgh, PA, USA, September 8-10, 2003.

[26] D. Banks and M. Prudence, "A High-performance Network Architecture for a PA-RISC Workstation," *IEEE Journal on Selected Areas in Communications*, vol. 11, no. 2, pp. 191-202, Feb. 1993.

[27] - D. Clark, V. Jacobson, J, Romkey and M. Salwen, "An Analysis of TCP Processing Overhead," *IEEE Communications Magazine*, vol. 27, pp. 23-29, June 1989.

[28] J. Coit, S. Staniford, J. McAlerney, "Towards Faster String Matching for Intrusion Detection or Exceeding the Speed of Snort," *Proc. DARPA Information Survivability Conference and Exposition* (DISCEX II ?02), IEEE CS Press, Los Alamitos, Calif., pp. 367-373, 2001.

[29] K. G. Anagnostakis , S. Antonatos , E. P. Markatos and M. Polychronakis, "A Domain-Specific String Matching Algorithm for Intrusion Detection," *Proceedings of the 18th IFIP International Information Security Conference*, pp. 217-228, 2003.

[30] C. Giovanni, "Fun With Packets: Designing a Stick. Draft White Paper on Stick," http://www.eurocompton.net/stick/

[31] SNOT Discussion on Security Focus IDS Mailing List, http://www.securityfocus.com/ids

[32] S. Patton, W. Yurcik, D. Doss, "An Achilles? Heel in Signature-Based IDS: Squealing False Positives in Snort," *Proceedings of RAID*, pp. 95-114, 2001.

[33] Jesse D. Hall John C. Hart, "GPU Acceleration of Iterative Clustering," *The ACM Workshop on General Purpose Computing on Graphics Processors*, pp. 45-52, 2004.

[34] NK Govindaraju, B Lloyd, W Wang, M Lin, D Manocha, "Fast Computation of Database Operations using Graphics Processors," *Proceedings of the 2004 ACM SIGMOD Conference*, pp. 77-86, 2004.

[35] MJ Harris, G Coombe, T Scheuermann, A Lastra, "Physically-based visual simulation on graphics hardware," *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pp. 45-53, 2002

[36] J. C. Rabek, R. I. Khazan, S. M. Lewandowski, R. K. Cunningham, "Detection of Injected, Dynamically Generated, and Obfuscated Malicious Code," *Proceedings of the 2003 ACM Workshop on Rapid Malcode*, pp. 231-241, Oct. 2003

[37] Puma Processor, http://www.ubinetics.com

[38] Lynx Processor, http://www.eonic.com/processingplatforms/lynx-sar.html

[39] Tiger Processor, http://www.us.design-reuse.com/news/news10176.html