An Investigation of GPGPU Optimisations for High Bandwidth Network Intrusion Detection Systems

Andrew Calder
School of Design and Informatics
Abertay University
DUNDEE, DD1 1HG, UK
Word Count: 3414

ABSTRACT
Context
As computer networks have experienced rapid growth, so too has the volume of data processing required; it is increasingly difficult to provide both security and performance. NIDS (Network Intrusion Detection Systems) are one of the first lines of defense in network security, as such, they are subject to increasingly intensive loads. While availability and ease of deployment has improved, existing open source solutions are heavily reliant on CPU-based detection engines which are expensive to scale appropriately for high bandwidth networks. However, using GPGPUs (General Purpose Graphics Processing Units), the same data could be processed in a far more scalable manner at a fraction of the cost

Aim
The aim of this project is to develop a GPGPU based detection engine for NIDS, and thus provide significant performance improvements on commodity hardware. A signature based solution is proposed, focusing on optimizing key components to maximize throughput.

Method
Project execution will begin with extensive research into the design and implementation of NIDS, as well as optimization for common -and subject specific- GPGPU design patterns. The findings will be used to cohesively develop a NIDS for linux-based operating systems, utilising CUDA and C++. Finally, performance of the solution will be evaluated and tested against packet generation tools and publically available datasets.

Results
The finished solution will be evaluated against a variety of publicly available datasets and packet generation tools. Performance of the application, measured as throughput, will be used to compare the base application, optimisations, and existing solutions. Worthwhile optimisations (those that produced substantial improvements) will be highlighted to guide future work.

Conclusion
As the demand for high performance networks faces exponential growth due to ever-improving data-links, so too does the need for high performance Network Intrusion Detection Systems. The system proposed in this paper will make use of both GPGPU and optimisations to maximise throughput on commodity hardware.

Keywords
Network Intrusion Detection Systems, Optimisation, Pattern Matching, CUDA, GPGPU, Network Security

1. CONTEXT
Attempts to breach networks and information systems have skyrocketed in recent years. According to the *Cyber Breaches Survey 2018*, 43% of businesses have experienced some form of cyber attack this year (Department for Digital, Culture, Media & Sport, 2018), 19% higher than 2016. Improvements in network infrastructure have benefitted genuine and malicious traffic alike. With the advancement of data links, average network throughput has risen to 10Gbps, with 40Gbps set to become the defacto standard in the near future (Khalil, 2015). As such, the need for equally high throughput network security systems will only grow.

Network Intrusion Detection Systems are a software solution for automatically identifying possible security incidents. They act as the first line of defence in many network configurations; performing malware detection, data categorization, and other use-specific rulesets on traffic. There are three main detection methodologies; signature-based, anomaly-based, and stateful protocol analysis (NIST, 2007). Signature detection involves comparing patterns called *signatures* against known bad signatures. Signature detection is very effective at detecting known threats (NIST, 2007) but due to the pattern matching process, it is computationally expensive.

Pattern matching itself is a relatively simple task, the issue lies in the sheer number of comparisons that need to be made to find a match. The obvious solution is to split the task across multiple threads - executing the comparisons in parallel. Two notable CPU-based open source Network Intrusion Detection Systems -*Snort* and *Suricata*- have exploded in popularity. Despite the adoption of multithreaded processing techniques, neither is able to keep up with a fully saturated 10 Gbps data link (Khalil, 2015). Without specialized -often expensive- hardware, these open source solutions are incapable of achieving the level of throughput expected of modern networks.

As per operational requirements in modern computing, each core in a CPU can execute instructions independently and support a complex instruction set. As such, they are rather large; limiting the number of cores a CPU can have. In contrast, a GPU is made up of many tiny simple cores, making it appear perfectly suited for pattern matching. Pattern detection can be described as a highly parallel task; it should scale well with GPGPU parallelism.

The aim of this project is to investigate whether a highly-parallel GPGPU-optimized signature-based Network Intrusion Detection System can outperform existing CPU-based solutions. The proposed solution focuses on maximising throughput and improving availability through support for commodity hardware.

As signature detection is a form of pattern matching, the teachings may be relevant beyond the scope of this project. High throughput pattern matching can be applied to other areas such as DNA sequencing or Hard Drive and Memory Forensics.

2. BACKGROUND

Despite the potential of GPGPU, notable open source solutions still favour CPU detection engines. *Snort* only recently received multithreaded support, with no GPGPU support in sight. Meanwhile *Suricata* did at least explore GPGPU support. However, it is scarcely documented and development seems to have stalled; on the support page CUDA support is listed as “unmaintained, currently in various stages of brokenness” (openinfosecfoundation.org, n.d.). While it is possible that there has been some commercial development, there have been no notable open source GPGPU NIDS advancements since 2013.

2.1 Operation of NIDS

The operation of Network Intrusion Detection Systems can be broken down into three main processes; packet acquisition, ruleset processing, and response. Most of the processes involved in packet acquisition are very much hardware reliant. Typically (on linux based operating systems), packets will be fetched directly from the kernel buffer using libraries such as *libpcap*, *netmap*, or abstraction layers based upon these libraries. As the name suggests, ruleset processing interprets and enforces rules on traffic, for example performing an action when a file that matches a specified signature is seen. Responses quite simply dictate what actions are taken, such as an alert email.

2.2 Use of GPGPUs in NIDS

Prior research into GPGPU based NIDS has yielded promising results, however, said research has explored constituent optimizations, whereas this project seeks a more cohesive approach.

In *‘Gnort: High Performance Network Intrusion Detection Using Graphics Processors’* (Vasiliadis et al. 2008), Vasiliadis achieves a throughput of 2.3Gbps through the utilisation of GPGPUs and a buffered memory design to “offload pattern matching computation”. *Gnort* uses separate buffer types for different classifications of packets; once a buffer is full, all packets are transferred to the GPU in one operation, if a buffer is not full after 100ms it is transferred anyway - preventing stranded packets. Additionally, a “double buffering scheme” is used, which ensures packets can be stored for processing, even if the GPU is currently operating on packets matching that classification.

The paper ‘*Kargus: a highly-scalable software-based intrusion detection system*’ (Jamshed et al. 2012) describes another Snort based NIDS. Kargus utilizes packet acquisition techniques from “the PacketShader software router” which also uses a buffered approach to prevent frequent costly kernel and userspace context switches. Opportunistic load balancing is used to “prevent excessive power consumption”; should the rate of incoming traffic be within a predetermined throughput threshold, packets will be processed by the CPU instead of the GPU. Kargus is not open source meaning the exact methods of performing the above can not be reviewed or further developed:“The IDS source code is not available to the public, as it contains a derivation from industry-transferred code”.

In ‘*NBA (network balancing act): a high-performance packet processing framework for heterogeneous processors*’ (Kim, J et al. 2015) -a continuation of the works presented in *Kargus*, the authors achieve “near 30 Gbps” throughput using the techniques described previously with a sixteen core dual Intel Xeon system equipped with two Nvidia GTX 680 GPUs, 32GB of RAM, and four dual-port 10GbE network interface cards. The paper reveals that the packet parsing library used in *Kargus* ‘Click-Parser’ is open-source. However, no further documentation is provided and the parser has not been updated since 2015, limiting its usefulness.

3. METHOD

Project execution will begin with extensive research into the design and implementation of NIDS, as well as optimization for common -and subject specific- GPGPU design patterns. This research will be broken down into three main areas; GPU Architecture, Algorithms, and GPU Optimisation. The findings will be used to cohesively develop a NIDS for linux-based operating systems, utilising CUDA and C++. The final solution will be available as open source to encourage adoption of the explored techniques.

3.1 GPU Architecture

Before delving into algorithms and optimisations, the GPU architecture must first be considered. Consideration of the underlying architecture could make a big difference to performance. GPUs allow for a great deal of user specification, especially so for memory access. In CUDA there are three main types of device memory; Global, Shared, and Texture. Each type has pros and cons, and thus some are better suited for certain tasks than others, as proof,each type will be implemented for comparison. Consideration of the GPU architecture will also help maximise potential of algorithm implementations.

3.1.1 Global Memory

In CUDA, global memory refers to off-chip DRAM (Iandola, F.N. et al, 2013). Just like in a CPU, the global memory is backed by a L1 and L2 cache. Currently, as far as the author is aware, global memory can be anywhere from 0-12GB - depending on the GPU. As global memory is off-chip, there is a high cost to read from it (compared to other methods); global memory requires up to 600 cycles to be accessed. However, it is also the simplest storage method to implement, and thus will likely be included as part of a feasibility demo.

3.1.1.1 Coalesced Memory Accesses

To achieve optimal memory access speeds, reads and writes should be coalesced. When a thread requests access to a location in memory it is given access to multiple locations which collectively form a chunk; all threads in the same bock have access to this chunk. Fetching new blocks costs additional memory transactions which ultimately reduce performance. As such, the requested data should be optimised for adjacent storage - it should be coalesced.

3.1.2 Shared Memory

When shared memory is used, a portion of the cache is semi-permanently allocated. The portion is what is referred to as shared memory. As the cache is on-chip, it is significantly faster than global memory.

In CUDA, there are four cache preferences that can be applied; these cache preferences state how the cache should be divided up. For example, *CudaFuncCachePreferShared* will allocate most of the space to Shared memory and leave at least 16kb of the space for the L1 cache. While shared memory has the potential to improve the solution efficiency, reducing the available cache may have knock-on effects in other areas of the application. A version utilising shared memory will be implemented to demonstrate the performance benefits and costs.

3.1.3 Texture Memory

Texture memory is off-chip just like global memory, however, unlike global memory it is read-only. The key area of interest is the texture cache which is optimised for read-only access. Making use of it does not require sacrificing any of the main L1 or L2 cache. Unlike global memory, it does not require accesses to be coalesced for optimal performance. In '*Communication-minimizing 2D convolution in GPU registers*' (Iandola. F.N et al, 2013) the authors demonstrated a 7 fold improvement in bandwidth from shared memory to texture memory. The implementation using texture memory will likely be the fastest version, if the read access speed is consistent.

3.2 Algorithms

In the area of pattern matching there are two process types; single pattern matching and multi pattern matching. Single pattern matching involves checking each pattern individually against a string, whereas in multi pattern matching, all patterns are compared simultaneously. Multi pattern matching is well suited to comparing multiple signatures, and thus, only multi pattern algorithms will be implemented.

3.2.1 HEPFAC Version 1

In '*High performance pattern matching and data remanence on graphics processing units*' (Bellekens, X. 2016), Bellekens presents the 'Highly Efficient Parallel Failureless Aho-Corasick (HEPFAC) algorithm. Version one of this algorithm stores the trie structure required by Aho-Corasick as a one dimensional row major ordered breadth-first array of structures (containing an offset and bitmap/s). Use of the trie structure allows for comparison of multiple patterns simultaneously as each node can represent multiple patterns. For instance, any patterns that start with 'A' shall share the same first 'A' node. The trie structure is also compressed to save further space; patterns with exactly the same suffixes are merged as can be seen in *Figure 1*.

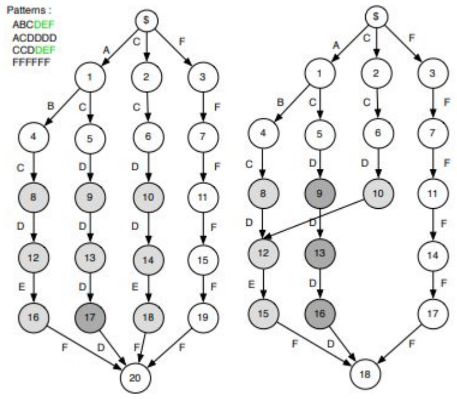

Figure 1: Merged Suffixes (Bellekens, X. 2016)

As traditional methods of representing the trie were not very space efficient, a new approach was required. In HEPFAC, each node is not aware of their identity, only that of their children. Each node is made up of a bitmap (or bitmaps depending on the alphabet size) which identify the children this node has, and an integer which represents the index of the first child. As the patterns are alphabetically sorted before the trie is built, the children of each node will be placed in alphabetical order too. By performing a population count on the bitmaps within each node, the number of children can be found. Equally, the location of a given child can be found by performing a population count *upto* the given child; if the child exists, its location will be the sum of the population count and offset of the first child. This method ensures each node will always be of a fixed size, which reduces the complexity of the GPU memory allocation. However, both the patterns and text strings are stored in global memory which is relatively simple to implement regardless.

The search process is as follows (for each thread); starting at the root node, check if the current compare character is any of the children of this node. If it is, let that node be the current node, otherwise exit. If the current node is the end node, the last character in a pattern has been reached - a match has been found.

3.2.2 HEPFAC Version 2

In HEPFAC version 2, the build process is mostly the same, although some modifications are made to the storage method "The modification helps finding the first child of the current node while using a two dimensional matrix instead of the row major ordered array". The main difference is the structure of the nodes; in version 1 *(Figure 2 - top)* each node was self contained, in version 2 (*Figure 2 - bottom*) the nodes coexist in a two dimensional matrix. Each row of the matrix is one Node in the trie. Assuming the ASCII alphabet is in use, the first 8 columns of each row contain the 8 bitmaps of each node, while the 9th column contains the offset.

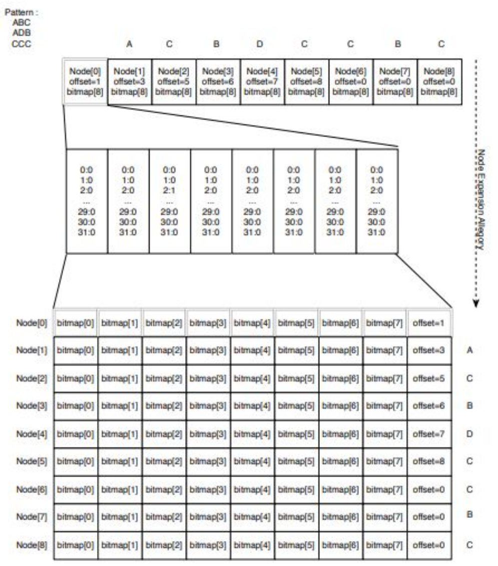
Pattern :
ABC
ADB
CCC
A C B D C C B C
Node[0] offset=1 bitmap[8]
Node[1] offset=3 bitmap[8]
Node[2] offset=5 bitmap[8]
Node[3] offset=6 bitmap[8]
Node[4] offset=7 bitmap[8]
Node[5] offset=8 bitmap[8]
Node[6] offset=0 bitmap[8]
Node[7] offset=0 bitmap[8]
Node[8] offset=0 bitmap[8]
0:0 1:0 2:0 ... 29:0 30:0 31:0
Node Expansion Allocation
bitmap[0] bitmap[1] bitmap[2] bitmap[3] bitmap[4] bitmap[5] bitmap[6] bitmap[7]
offset=1
offset=3 A
offset=5 C
offset=6 B
offset=7 D
offset=8 C
offset=0 C
offset=0 B
offset=0 C

Figure 2: Memory Scheme Transformation (Bellekens, X. 2016)

In '*Trie Compression for GPU Accelerated Multi-Pattern Matching*' (Bellekens, X. et al, 2017), the author describes the HEPFACv2 algorithm as a GPGPU multi-pattern matching that demonstrates "85% less space requirements than the original highly efficient parallel failure-less Aho-Corasick", and demonstrates "over 22 Gbps" throughput. *Kargus*, *Gnort* and *NBA* all made use of the original Aho-Corasick algorithm, an implementation of this algorithm should prove even faster.

Utilising HEPFACv2, excellent results were seen in '*GLoP: Enabling Massively Parallel Incident Response Through GPU Log Processing*' (Bellekens, X. et al, 2017), in which the authors compared the use of the texture cache to global memory. Regarding a comparison of the two types, the authors found that "the implementation using the texture memory achieves double throughput compared to the implementation that uses global memory".

In this version, the patterns are stored in global memory and the comparison text is stored in texture memory, which provides the performance benefits of the texture cache. However, by utilising texture memory for both, or utilising a combination of texture and shared memory, it may be possible to produce even greater results.

3.3 GPGPU Optimisation

By considering the architecture and algorithms utilised, optimisations from other works could be implemented where applicable. For instance, the buffered memory transfers as seen in Kargus would reduce the number of memory transactions required to move the same amount of data, thus reducing overhead.

In '*Adaptive Optimization l1 -Minimization Solvers on GPU*' (Gao, J. et al, 2017), the authors explored the use of the texture cache (as opposed to shared or global memory). They were able to reduce the execution time significantly as described: "We see that for all test cases, the execution time ratios have been sustained at around 1.2"; execution time was reduced by 20%.

In '*GPU I/O persistent kernel for latency bound systems*' (Martinelli. M, 2017), a method of minimizing launch latency is presented. Martinelli found that by creating a persistent kernel (as opposed to relaunching with each dataset) the user/kernel space switch latency was eliminated; the kernel launch overhead is reduced to a single launch. Preliminary results demonstrated a consistent improvement. The persistent kernel optimisations will be implemented if time permits, however, it is not an essential optimisation.

3.4 Measuring Performance

Performance of the solution will be evaluated and tested against packet generation tools such as *Pktgen*, or publically available datasets. Performance evaluations will primarily focus on effective throughput, in GBps. Baselines will be established for packet collection speed and memory write speeds. Performance can only be measured when there are no other tasks. If possible, the final solution will be tested on multiple hardware configurations to provide further data sets for comparison.

4. Summary

With attempts to breach networks and information systems skyrocketing in recent years, the need for high throughput Network Intrusion Detection Systems will only grow. Despite the potential of GPGPU, notable open source solutions still favour CPU detection engines. Prior research is simply not documented well enough to enable implementation.

The proposed project aims to develop a GPGPU signature-based detection engine for NIDSs, and thus provide significant performance improvements on commodity hardware. As signature detection is a form of pattern matching, the results will be relevant to other areas of high performance computing; pattern matching can be applied to areas such as DNA sequencing or Hard Drive and Memory Forensics. Unlike prior research, the proposed solution will not use any privately owned components, and thus will be open sourced to encourage adoption of the explored techniques.

5. REFERENCES

Bellekens, X., Seeam, A., Tachtatzis, C. and Atkinson, R., 2017. Trie compression for GPU accelerated multi-pattern matching. arXiv preprint arXiv:1702.03657.

Bellekens, X.J., Tachtatzis, C., Atkinson, R.C., Renfrew, C. and Kirkham, T., 2014, September. Glop: Enabling massively parallel incident response through gpu log processing. In Proceedings of the 7th International Conference on Security of Information and Networks (p. 295). ACM.

Bellekens, X.J., 2016. High performance pattern matching and data remanence on graphics processing units (Doctoral dissertation, University of Strathclyde).

Department for Digital, Culture, Media & Sport (2018). *Cyber Security Breaches Survey 2018: Statistical Release*. Department for Digital, Culture, Media & Sport, p.1.

Gao, J., Li, Z., Liang, R. and He, G., 2017. Adaptive Optimization $$ l_1 $$ l 1-Minimization Solvers on GPU. International Journal of Parallel Programming, 45(3), pp.508-529.

Khalil, G. (2015). Open Source IDS High Performance Shootout. [Blog] SANS Institute InfoSec Reading Room. Available at: https://www.sans.org/reading-room/whitepapers/intrusion/open-source-ids-high-performance-shootout-35772 [Accessed 1 Oct. 2018].

Kim, J., Jang, K., Lee, K., Ma, S., Shim, J. and Moon, S., 2015, April. NBA (network balancing act): A high-performance packet processing framework for heterogeneous processors. In Proceedings of the Tenth European Conference on Computer Systems (p. 22). ACM.

NIST (2007). Guide to Intrusion Detection and Prevention Systems (IDPS). Gaithersburg, MD: National Institute of Standards and Technology, U.S. Department of Commerce, pp.3, 4.

Redmine.openinfosecfoundation.org. (n.d.). Support Status - Suricata - Open Information Security Foundation. [online] Available at: https://redmine.openinfosecfoundation.org/projects/suricata/wiki/Support_Status [Accessed 8 Oct. 2018].

Vasiliadis, G., Antonatos, S., Polychronakis, M., Markatos, E.P. and Ioannidis, S., 2008, September. Gnort: High performance network intrusion detection using graphics processors. In International Workshop on Recent Advances in Intrusion Detection (pp. 116-134). Springer, Berlin, Heidelberg.

Iandola, F.N., Sheffield, D., Anderson, M.J., Phothilimthana, P.M. and Keutzer, K., 2013, September. Communication-minimizing 2D convolution in GPU registers. In Image Processing (ICIP), 2013 20th IEEE International Conference on (pp. 2116-2120). IEEE.

Martinelli, M., 2017, GPU I/O persistent kernel for latency bound systems. In Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing