Abertay University

School of Design and informatics

An Investigation into GPGPU Optimizations for Network Intrusion Detection Systems

Andrew R. Calder

supervised by
Dr. Xavier Bellekens

21st March 2022

Acknowledgements

First and foremost, I would like to thank my supervisor, Dr Xavier Bellekens, whose guidance, expertise and support made it possible to explore a topic of great interest to me. It was a pleasure working with you.

I would also like to thank Dr Adam Sampson and Dr Ethan Bayne for their assistance and advice throughout this year; you both helped shape key areas of this project and for that I am grateful.

Thank you to all my friends at HackSoc these past four years; to my mum and dad for putting up with my rambling phone calls, to Crystal for her patience and unwavering support and proofreading, and (of course) to my flatmates who have been like a second family to me these past few years.

Abstract

With continual advancements in network infrastructure, vast amounts of data is communicated every second. These advancements have benefited genuine and malicious traffic alike; attempts to breach networks and information systems have skyrocketed in recent years. It is increasingly difficult to provide both security and performance. Network intrusion detection systems (NIDS) are one of the first lines of defense in network security, as such they are subject to increasingly intensive loads. Existing open source solutions are heavily reliant on CPU-based detection engines which are expensive to scale appropriately for high bandwidth networks. However, using GPUs (Graphics Processing Units), the same data could be processed in a far more scalable manner at a fraction of the cost. The research presented aims to demonstrate that a GPU-based solution can be just as effective, on commodity hardware.

This paper details the implementation of a signature-based solution utilising the HEPFAC algorithm, which focuses on the optimisation of key components to maximise throughput. Modern networks typically operate up to 40Gbps, as such a throughput target of at least 40Gbps was set. This target must be met to ensure security incidents can be identified in real time.

To accomplish the imposed target, a CUDA/C++ implementation of the HEPFAC algorithm was developed. The HEPFAC_CPP solution identifies locations of matches within a provided input stream. In a full NIDS solution, these matches would correspond to specific packets. However, in its current form HEPFAC_CPP serves as as a demonstration of the throughput that can be achieved using the specified optimisations. Said optimisations are not only applicable to NIDS, but also to other applications including (but not limited to); Anti Virus, File Carving, Memory Forensics and DNA sequencing.

Evaluation of benchmark results show that the techniques and optimisations used to develop HEPFAC_CPP allow it to exceed modern network throughput requirements, by a significant margin - laying the foundations for an effective yet affordable real-time network intrusion detection system.

Contents

1 Introduction **1**
1.1 Background . . . 1
1.2 Pattern Matching . . . 1
1.3 Scope . . . 1

2 Literature Review **3**
2.1 Use(s) of GPGPUs in Information Security & Networking . . . 3
2.2 Optimizations & Algorithms . . . 4
2.2.1 Multi Pattern Matching . . . 4
2.2.2 Memory Optimization . . . 7
2.2.3 Transfer and Launch Minimization/Maximization . . . 8
2.2.4 Intrinsic Functions . . . 9
2.2.5 Bitwise Operations . . . 9
2.2.6 Control Flow . . . 11
2.3 Summary . . . 11

3 Methodology **12**
3.1 Overview . . . 12
3.2 Optimisations . . . 13
3.2.1 Selection Process . . . 13
3.2.2 Selected Optimisations . . . 13
3.3 Development . . . 16
3.3.1 Design . . . 16
3.3.2 Implementation . . . 17
3.3.3 Testing . . . 20

4 Results **21**
4.1 Unit Tests . . . 21
4.2 Test System Specifications . . . 21
4.3 Search Optimisation Benchmarks . . . 22
4.3.1 Individual Optimisations . . . 22
4.3.2 Variable Signature Length . . . 23
4.3.3 Variable Signature Count . . . 24
4.3.4 Variable File Size . . . 25
4.4 Other Benchmarks . . . 26
4.5 Overview . . . 27

5 Discussion **28**
5.1 Overview . . . 28
5.2 Analysis of Results . . . 28
5.3 Recommendations . . . 30
5.4 Problems Faced . . . 31
5.5 Algorithm . . . 31

6 Conclusions **32**
6.1 Future Work . . . 32

7 References **33**

Appendices **35**

List of Figures

1 Merged Suffixes (Bellekens 2016) . . . 5
2 Memory Scheme Transformation (Bellekens 2016) . . . 6
3 Population Count (Warren 2012) . . . 10
4 Pageable vs Pinned Memory . . . 12
5 Population Count Benchmark (Nanoseconds) . . . 14
6 Visualisation of Bitwise Indexing Method . . . 14
7 Synchronous vs Asynchronous Execution . . . 16
8 Command Line Interface Output Example . . . 17
9 Individual Optimisations - 100x6 Trie vs 1GB Random Data . . . 22
10 Fixed Count Variable Length - 25xN Trie vs 1GB Random Data (Unoptimised) . . . 23
11 Fixed Count Variable Length - 25xN Trie vs 1GB Random Data (Optimised) . . . 24
12 Variable Count Fixed Length - Nx6 Trie vs 1GB Random Data (Unoptimised) . . . 24
13 Variable Count Fixed Length - Nx6 Trie vs 1GB Random Data (Optimised) . . . 25
14 Variable File Size - 100x6 Trie vs N MB Random Data (Unoptimised) . . . 25
15 Variable File Size - 100x6 Trie vs N MB Random Data (Optimised) . . . 26
16 Variable Result Array Size - Bitwise Indexing vs Boolean Indexing . . . 27
17 Variable Pattern Characters - Build Trie vs Reduce Trie . . . 27
18 Nvidia Visual Profiler - Unoptimised Global Search . . . 28
19 Nvidia Visual Profiler - Global Search W/ Concurrency . . . 28
20 Nvidia Visual Profiler - Optimised Shared Search Occupancy . . . 29
21 Nvidia Visual Profiler - Optimised Texture Search . . . 29
22 Bitwise vs Boolean Indexing Comparison (No Multi-threading) . . . 35

List of Tables

1 Unit Tests . . . 21
2 Test System Specifications . . . 22
3 Individual Optimisations - 100x6 Trie vs 1GB Random Data . . . 23
4 Fixed Count Variable Length - 25xN Trie vs 1GB Random Data (Unoptimised) . . . 38
5 Fixed Count Variable Length - 25xN Trie vs 1GB Random Data (Optimised) . . . 38
6 Variable Count Fixed Length - Nx6 Trie vs 1GB Random Data (Unoptimised) . . . 39
7 Variable Count Fixed Length - Nx6 Trie vs 1GB Random Data (Optimised) . . . 39
8 Variable File Size - 100x6 Trie vs N MB Random Data (Unoptimised) . . . 40
9 Variable File Size - 100x6 Trie vs N MB Random Data (Optimised) . . . 40

1 Introduction

1.1 Background

With continual advancements in network infrastructure, vast amounts of data is communicated every second. With these advancements, typical network throughput has risen to 10Gbps, with 40Gbps set to become the de facto standard in the near future (Khalil 2015). However, improvements have benefited genuine and malicious traffic alike; attempts to breach networks and information systems have skyrocketed in recent years. According to the Cyber Breaches Survey 2018, 43% of businesses experienced some form of cyber attack last year (*Cyber Security Breaches Survey 2018: Statistical Release* 2018), 19% higher than 2016. 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 rule-sets on traffic. There are three main detection methodologies; signature-based, anomaly-based, and stateful protocol analysis (Scarfone and Mell 2012). Signature detection involves comparing patterns called signatures against known bad signatures. Signature detection is very effective at detecting known threats (Scarfone and Mell 2012) but due to the pattern matching process, it is computationally expensive.

1.2 Pattern Matching

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 multi-threaded 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 independently and support a complex instruction set. As such, they are rather complex and thus large, limiting the number of cores a CPU can have. In contrast, a GPU is made up of many smaller simple cores. Each GPU core runs significantly slower and does not support nearly as many features as its CPU counterpart. Despite a slower per core speed, the sheer number of available cores in a GPU allows for extremely fast parallel execution through many task subdivisions. This behaviour is a reflection of the principles of strong scaling, which in turn is a measure of how (for a given problem specification) the time to solution decreases as more processors are added (*CUDA C Best Practices Guide* n.d.). As pattern matching can be described as a highly parallel task; it should scale well with GPGPU parallelism.

1.3 Scope

Despite the potential of GPGPU, notable open source solutions still favour CPU detection engines. Snort only recently received multi-threading 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" (*Suricata Support Status* 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.

Prior research is simply not documented well enough nor portable enough to enable widespread adoption. This research aims to develop a prototype highly-parallel GPGPU-optimized signature-based Network Intru-

sion Detection System that can outperform existing CPU-based solutions. The proposed solution focuses on maximising throughput, minimizing required operations, and ensuring support for commodity hardware. The primary focus of this investigation can be summarized with the following research question:

> *How can GPGPU-based Network Intrusion Detection Systems be optimized for high throughput networks?*

In similar papers, optimization efforts focus on memory layout and padding. Said optimizations are often device or architecture specific, and thus limited in relevance to previous generation and lower-end hardware configurations. In contrast, the optimizations that this research aims to demonstrate will be device and architecture agnostic (where possible), ensuring the findings are more widely applicable, and -at least to some degree- portable.

Despite the scope of this research being limited to NIDS, as signature detection is a form of pattern matching, the optimizations and other teachings may be relevant to other applications. GPGPU accelerated multi pattern matching can be applied to other areas such as DNA sequencing, Digital Forensics, and Anti-Virus. While the detailed optimizations are applied specifically to the pattern matching process, they may be applicable to other GPGPU-based applications.

2 Literature Review

With the onset of fibre optic networking, computer networks have increasingly fast connectivity speeds. Higher throughput network traffic procures a requirement for higher throughput defense and monitoring - Network Intrusion Detection Systems. The following chapter aims to provide insight into existing NIDS solutions, the uses of GPGPU in information security - be it detection, prevention or forensic based, as well as optimization techniques that could be broadly applied. Additionally, the strengths and shortcomings current solutions will be detailed.

2.1 Use(s) of GPGPUs in Information Security & Networking

Intrusion Detection

As mentioned in section 1.3 there have not been any notable open source GPGPU NIDS advancements since 2013, and none of the big three (Snort, Suricatta, Zeek) currently support GPU acceleration. Despite the lack of implementation, prior research into GPGPU based NIDS has yielded promising results. 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".

Packet Processing

In *NBA (network balancing act): a high-performance packet processing framework for heterogeneous processors* (Kim 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 documentation is provided and the parser has not been updated since its initial (prototype) submission, limiting its usefulness.

Digital Forensics

String/Pattern searching is as critical for intrusion detection as it is for digital forensics. In *Accelerating digital forensic searching through GPGPU parallel processing techniques* (Bayne 2017), Bayne details an OpenCL-based file carving tool - '*Open Forensics*'. *Open Forensics* utilized the Parallel Failureless Aho-Corasick (PFAC) algorithm for multi-string searching which "demonstrates significantly greater processing improvements from the use of a single, and multiple, GPUs", and "minimised the amount of time required to search for greater amounts of patterns". Additionally, empirical testing suggests that this method (PFAC) may be "more efficient than the widely-adopted Boyer-Moore algorithms when applied to string searching". The presented tool appeared to limited by the storage device read speed and thus performance was on par with a multi-threaded CPU solution;

both achieved around 800 MiB/s. With an intrusion detection system the only limiting factor would be network throughput, which can be substantially faster than storage read speeds.

2.2 Optimizations & Algorithms

2.2.1 Multi Pattern Matching

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.

Aho-Corasick

Aho-Corasick (AC) is an efficient multi-pattern matching algorithm which can compare a number of patterns to a source in a single pass (Aho and Corasick 1975). It is widely known for its performance and has become the de facto standard of multi-pattern matching. The standard Aho-Corasick algorithm employs a failure trie that allows it check other possible patterns on no match states with a single thread. With many thread graphics processors, the failure trie is far less useful, and actually just leads to unnecessary overhead.

Parallel Failureless Aho-Corasick

In '*Accelerating string matching using multi-threaded algorithm on GPU*' (Lin et al. 2010), Lin describes *Parallel Failureless Aho-Corasick* - a evolution of Aho-Corasick specifically for parallel processors. As the name suggests, this version does not use the failure trie and through GPU architecture considerations, manages to achieve up to a "4000 time speed-up compared to AC algorithm on CPU". Additionally, compared to other GPU approaches, PFAC performs 3 times faster with significant improvements to memory efficiency. Aho-Corasick's requirement for failure nodes stems from the single pass requirement, it meant the algorithm did not need to go back when only a partial match was found. Instead, PFAC assigns a thread to every index in the source; all locations are compared simultaneously.

It is worth noting that the results from PFAC are likely positively skewed, in the "comparisons with previous gpu approaches" section the Lin states that the PFAC was tested with a GeForce GTX 295, the next most modern GPU listed is the GeForce 9800GX, which according to userbenchmark is around 30% less powerful. The comparison to the method listed as "Huang et al. Modified WM" is particularly unfair as according to userbenchmark the GeForce 7600GT is around 2000 times slower the the GTX 295 (*User Benchmark Comparison - GTX 295 vs 7600GT* 2019). However, even considering the best case 30% hardware/architecture improvement, PFAC is not just 30% faster than the "Vasiliadis et al. DFA" compared method, it is closer to 300% faster as stated. While the performance improvement is impressive overall, some of the comparisons are unfair.

Highly Efficient Parallel Failureless Aho-Corasick

In 'High performance pattern matching and data remanence on graphics processing units' (Bellekens 2016), Bellekens presents the 'Highly Efficient Parallel Failureless Aho-Corasick (HEPFAC) algorithm, which -as the name suggests- builds upon PFAC. Version one of this algorithm stores the required trie structure rather differently to PFAC; as a one dimensional row major ordered breadth-first array of structures (containing an offset and bitmaps). 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.

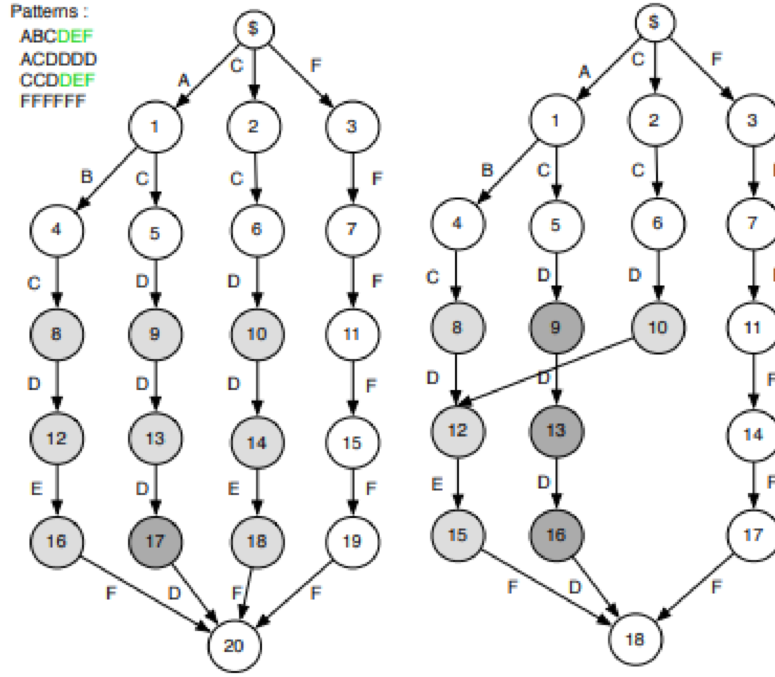

Figure 1: Merged Suffixes (Bellekens 2016)

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. Similarly, the location of a node's given child can be found by performing a population count up to the given child; if the child exists, its location will be the sum of the population count and offset of the first child, if it doesn't exist the search ends. This method ensures each node will always be of a fixed size, which reduces the complexity of the GPU memory allocation, and compared to PFAC, the size. In version one, both the patterns and text strings are stored in global memory (simplest & slowest on-GPU storage) and thus is relatively simple implement.

The search process is as follows (for each thread); start comparisons at the root node, check if the current compared character is any of the children of this node. If it is, let that node be the current node, otherwise exit. This process is repeated until the current node is the end node; the last character in a pattern has been reached - a match has been found. Bellekens also details a second version of the algorithm that makes use of the texture cache -which is described in more detail in section 2.2.2.

Use of the texture cache allows for even greater performance improvements over PFAC. In HEPFACv2, 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 (256 total characters), the first 8 columns of each row contain the 8 32-bit bitmaps of each node, while the 9th column contains the offset.

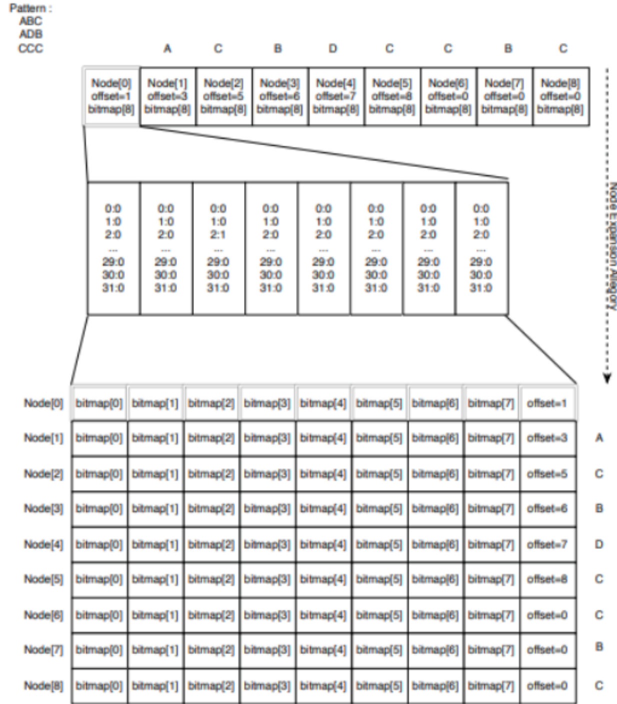

Figure 2: Memory Scheme Transformation (Bellekens 2016)

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 et al. 2014, in which the authors compared the use of the texture cache to global memory. Regarding a comparison of the two types, it was found that "the implementation using the texture memory achieves double throughput compared to the implementation that uses global memory". In HEPFACv2, the patterns are stored in texture memory and the comparison source is stored in global memory. However, if the comparison source was small enough, it may fit in constant memory which could produce even greater results due to the significantly lower access latency.

For all the performance improvements offered by HEPFACv2 over the likes of PFAC, there is one caveat - the identification system. By merging trie suffixes, identification of individual patterns on the GPU is impossible as there is no way to tell which pattern a shared node belongs to - it could belong to any number of patterns. However, the significant size reduction would allow for more patterns to be compared at once -each pattern has a significantly smaller footprint. This in turn could reduce both the copy time (less bytes to copy) and the number of copies required - if using a large enough signature list. Another point of note is that without GPU identification (only locating), the return array can be significantly smaller; with identification the return array would be an Integer (4 bytes), whereas with location only a Boolean (1 byte) is required. Furthermore, in the case of an IDS there is already going to be some form of packet identification required, so it is possible that the overhead would be negligible in comparison to packet identification. Given the design of HEPFAC, it may be possible to test both the non-compressed and compressed versions; the reduction merely rearranges nodes in the trie, the non-compressed trie is search-able. Thus both GPU-identification and CPU-identification could be tested, in terms of flexibility and potential for optimization HEPFAC seems to be the best candidate.

2.2.2 Memory Optimization

Aside from algorithms and optimisations, 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, each type will be implemented for comparison. Consideration of the GPU architecture will also help maximise potential of algorithm implementations.

Coalesced Memory Access

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 (*CUDA C Best Practices Guide* n.d.). As such, the requested data should be optimised for adjacent storage - it should be coalesced.

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 512MB-12GB - depending on the GPU. As global memory is off-chip, there can be 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 at very least be used as a point of comparison.

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 (*CUDA C Programming Guide* n.d.). 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.

Texture Memory

Texture memory (or rather the texture cache) is off-chip just like global memory, however, unlike global memory it is read-only. The texture cache which is optimised for read-only access, and making use of it does not require sacrificing any of the main L1 or L2 cache. Similarly to global memory, it does not require accesses to be coalesced for optimal performance. In 'Communication-minimizing 2D convolution in GPU registers' (Iandola et al. 2013) the authors demonstrated a 7 fold improvement in bandwidth from shared memory to texture memory. If their success can be replicated, an implementation using texture memory will likely be the fastest version - if the read access speed is consistent. Improvements were also seen in 'Adaptive Optimization l1 -Minimization Solvers on GPU' (Gao et al. 2017), in which *Gao et al* explored the use of the texture cache (as opposed to shared/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%.

Pinned Memory

By default, host allocated memory is pageable; "The GPU cannot access data directly from pageable host

memory, so when a data transfer from pageable host memory to device memory is invoked, the CUDA driver must first allocate a temporary page-locked, or 'pinned', host array" (Harris 2012). The data is copied to the pinned host array, then to device memory. This introduces an unnecessary copy which in turn reduces throughput. However, the additional transfer can be avoided all together by directly allocating arrays in pinned memory.

Pointer Aliasing

Pointers are said to *alias* if the memory they point to overlaps, if a compiler cannot determine whether or not pointers overlap, it must assume they do - this can limit performance. In the C99 standard the *restrict* keyword provides a way for programmers to tell the compiler that a *restrict* pointer will not modify the value pointed to by any other *restrict* pointer; effectively the value pointed to is read-only and as such can be cached. In the C++ standard there is no such keyword, however, most compilers allow the keyword to be used for the same purpose. In GCC (and subsequently NVCC) this is denoted by **¨restrict¨**. In (*CUDA C Programming Guide* n.d.) it is noted that "the compiler might not always be able to detect that the read-only condition is satisfied for some data." However, "Marking pointers used for loading such data with both the `const` and `__restrict__` qualifiers increases the likelihood that the compiler will detect the read-only condition". This should introduce performance improvements in all three types of memory identified, but will likely introduce the biggest improvement to Global Memory as it is the only method without any programmatically specified cache access.

2.2.3 Transfer and Launch Minimization/Maximization

According to the official CUDA developers optimization guide one of the highest priority optimizations that should be made is transfer minimization & maximization; the overall size of the copied data should be as small as possible *"The peak theoretical bandwidth between the device memory and the GPU is much higher than the peak theoretical bandwidth between host memory and device memory. Hence, for best overall application performance, it is important to minimize data transfer between the host and the device"*. At the same time it is better to copy one large block than lots of small blocks *"because of the overhead associated with each transfer, batching many small transfers into one larger transfer performs significantly better than making each transfer separately"* (*CUDA C Best Practices Guide* n.d.).

Concurrent Execution

In CUDA concurrency allows multiple operations to be performed simultaneously; kernel execution, host to device copies, device to host copies, and CPU operations (Rennich 2011). Concurrency can be implemented through the use of *Streams* which are effectively first-in first-out execution queues on the GPU; operations in different streams may run concurrently - depending on the flags used at initialization. The number of possible concurrent streams is limited by the GPU architecture used, for example *Fermi* was limited to 16. While many streams can be created and used, GPU occupancy (total resource usage i.e. cache, registers) may restrict how they can be used, although with compatible algorithms it should always produce improvements as it "effectively removes memory size limitations" (Luitjens 2015).

Persistent Kernel

In 'GPU I/O persistent kernel for latency bound systems' 'GPU I/O persistent kernel for latency bound systems', a method of minimizing launch latency is presented. Martinelli found that by creating a persistent kernel (as opposed to relaunching with each dataset) 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, the implementation method is not described so implementation within this this project is unlikely.

2.2.4 Intrinsic Functions

Intrinsic functions are way to tell the compiler to use its own implementation of a function. In both CUDA (nvcc) and GCC, as well as other modern compilers, intrinsic -or builtin- functions are offered. These functions are often directly implemented in the compiler in assembly, rather than belonging to a library. Builtin methods often provide very good implementations of common functions, such as *Find First Set* or *Find First Zero*. The HEPFAC algorithm discussed in section 2.2.1 makes extensive use of population counts; in the building, reduction, and searching processes. The CUDA C Best Practices Guide states that the intrinsic functions should be used "whenever speed trumps precision" (*CUDA C Best Practices Guide* n.d.). As HEPFAC functions only need integer accuracy (not floating point) these functions could be used to provide performance improvements over manual implementation - although this is something that will have to be tested.

2.2.5 Bitwise Operations

Many common bitwise optimizations are implemented automatically by the compiler -or at least should be; there are edge cases - particularly when non-standard types are used. Due to the possibility of such cases, it is still be worth implementing the optimizations manually on CPU. On GPU the behaviour is similar, however there has been mixed reports about if the the compiler implements these automatically or not, the CUDA C Best Practices Guide states that "The compiler will perform these conversions if n is literal" (*CUDA C Best Practices Guide* n.d.), where n is an integer used in a arithmetic computation such as the modulo and division in listing 1 below.

```
int popcto_before(unsigned int bitmap[], int idx){
    int i = 0,        // index
        count = 0;   // number of set bits
    do {
        if(bitmap[i/32] & 1 << (i % 32)){
            count++;
        }
        i++;
    } while (i < idx);
    return count;
}
```

Listing 1: Unoptimized Population Count

Integer Division & Modulo (Powers of Two)

Integer division and modulo operations are known for being particularly costly, taking several additional operations over their bitwise counterparts. There is good reason for this; arithmetic operations use several safety measures such as sign preservation and value accuracy. However, in this use case, there is no need for such safety measures and so the bitwise version is generally better. The bitwise operations in listing 2 below are functionally similar to the arithmetic operations in listing 1; right shifting five times is the same as dividing by 32, and a bitwise '&' with a power of two minus 1 is equivalent to modulo that power of two.

```c
int popcto_after(unsigned int bitmap[], int idx){
    int i = 0,       // index
        count = 0;   // number of set bits
    do {
        if(bitmap[i >> 5] & 1 << (i & 31)){
            ++count;
        }
        ++i;
    } while (i < idx);
    return count;
}
```

Listing 2: Bitwise 'Optimized' Population Count

Binary Magic Numbers

The term *Binary Magic Numbers* first appeared in an article published in Dr Dobbs Journal 1983. In the article, Freed states that similarly to mathematical magic numbers - binary numbers with strange and useful properties exist (Freed 1983). Most magic numbers are mere mathematical or programmatic curiosities and no real world applications, however there a few outliers in base 10 arithmetic. Freed notes that the binary numbers *0b0101010101010101*, *0b0011001100110011*, *0b0000111100001111*, and *0b0000000011111111* could be used to perform divide-and-conquer-esque sideways addition - which is another name for a population count; summing the total set bits in a given variable. Hacker's Delight (Warren 2012) further explains how the divide and conquer method can be applied; the task subdivided into many smaller tasks; first every two bits are summed, then every four, then every eight. The example given in figure 3 is a great demonstration of this technique.

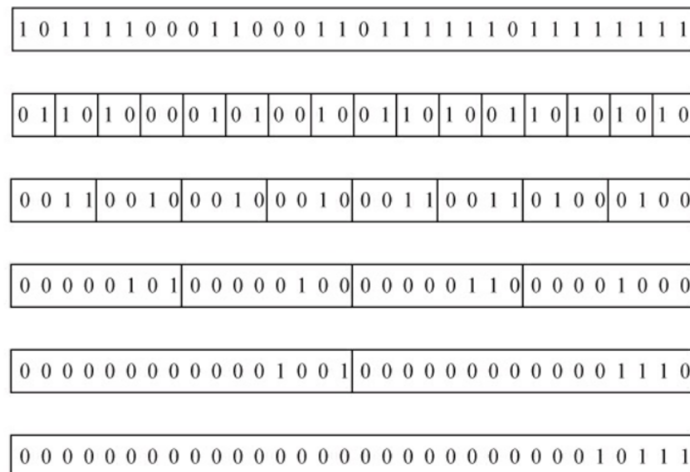

Figure 3: Population Count (Warren 2012)

A highly efficient version of the divide and conquer algorithm appears in the AMD64 optimization guide (*Software Optimization Guide for AMD64 Processors* 2005). This version focuses on minimizing register use and further reducing the number of required operations - requiring only 12 operations which is significantly less than previous versions. An implementation of the version described by AMD can be seen in listing 3 which relies on the same principle but uses the hexadecimal representation of the *binary magic numbers*.

```
int popcount(unsigned int temp){
    // Implementation of Population Count as
    // described in software optimization guide for amd64 processors
    temp = temp - ((temp >> 1) & 0x55555555);
    temp = (temp & 0x33333333) + ((temp >> 2) & 0x33333333);
    return (((temp + (temp >> 4)) & 0xF0F0F0F) * 0x1010101) >> 24;
}
```

Listing 3: AMD64 Population Count

Most & Least Significant Bit

Finding the most and least significant bits can be very useful in bitwise operations. For example, if searching the for all set bits in a given data type knowing where to start and where to end could save many operations. In *Matters Computational* (Arndt 2010), Arndt details two methods of isolating the most significant bit, one uses an assembly instruction which may be device specific, the other repeatedly bitwise OR's the variable with itself shifted right by 1, 2, 4, 8, and 16 places - for 32 bit types, and then adds 1 and finally shifts right once more. Say the variable started out at 0b00100101, after the initial OR shifts it would be 0b00111111, adding one would take it to 0b01000000, and finally right shifting takes it to 0b00100000 - the isolation of the most significant bit.

The technique Arndt demonstrates to isolate the least significant bit is far simpler; performing a bitwise *n & -n* will isolate the least significant bit of *n*. The same technique also appears in Hackers Delight (Warren 2012), in which warren describes the technique as a "formula to isolate the rightmost 1-bit, producing 0 if none (e.g. 01011000 to 00001000)" which makes this very useful in other bitwise operations.

2.2.6 Control Flow

GPUs utilize a single instruction multiple data architecture (SIMD). This means it "takes an operation specified in one instruction and applies it to more than one set of data elements at the same time" (Furht 2008), which introduces some control flow complications. In (Chen 2016) Chen discusses why branch divergence can cause significant performance degradation. Threads are bundled into blocks, which are in turn bundled into warps which "follow the same instruction synchronously". If a branching statement is encountered - for example an if else block that causes a different outcome on different threads- a branch divergence occurs. Threads in a warp cannot diverge; all threads meeting the condition must be executed first and only then can the warp go back to execute any divergent threads; "the branch divergence serializes all the possible execution paths" - this can have a serious impact on execution time. To reduce execution time, use of branching statements should be minimised. This also means branching statements should be constructed so that the 'true evaluated' threads can take a direct path - i.e. avoid nested branching statements to prevent unnecessary divergence serialization.

2.3 Summary

With ever increasing network bandwidth capabilities, it is apparent that there is a need for high throughput network security solutions. Of the reviewed multi string/pattern matching implementations and algorithms, the current best utilized GPGPU processing and was able to achieve 27Gbps throughput (Bellekens 2016). However, modern networks are commonly capable of 40Gbps which -during high traffic situations- presents a losing battle that would result in a backlog of unprocessed or dropped packets. Previous research has generally sought algorithmic improvements to increase throughput. However, through consideration of the techniques discussed above, implementation optimizations may yield the improvements required to close the gap between processing and network throughput.

3 Methodology

3.1 Overview

Algorithm

Due to time constrains only a single algorithm will be focused on; it is unlikely that optimisations could be applied with the same finesse if multiple algorithms were implemented. Before any optimisations could be selected a suitable algorithm was identified as to most effectively direct optimisation efforts. While Parallel Failure-less Aho-Corasick (PFAC) is a commonly used (if not the most commonly used) GPGPU multi-pattern matching algorithm that is somewhat documented, it isn't the fastest - not by a long way. The HEPFAC algorithm presented in (Bellekens 2016) was able to achieve 27Gbps, the closest to the 40Gbps minimum required of modern networks. Additionally, an early *C* implementation of the HEPFAC algorithm was provided by the author (hereby referred to as ' HEPFAC_CPP') which will ease understanding and aid optimisation efforts.

The HEPFAC algorithm introduces a number of quirks. For example, the trie reduction method -while guaranteeing a size reduction- removes the ability to determine the specific pattern identified on the GPU, instead providing the location that at which a match was identified - effectively acting as match sieve. HEPFAC also truncates the given patterns to a user-specified length; in (Bellekens 2016) it was found that 99% of signatures tested could be individually identified even with a significant length reduction. However, it is able to identify significantly more patterns than PFAC (per search) due to lower space complexity, and do so faster.

Basic Implementation

Before an optimised version could be developed, a basic implementation was created. This provides a baseline for testing. Although this version is intended to be the 'basic implementation' it; includes global, shared and texture memory based search methods, adheres to branch minimization principles, and uses pinned memory. Generally, these optimisations/techniques were selected due to their prevalence in similar papers - the results would show a misleading improvement if these relatively basic additions were not included. Pinned memory is an exception to this, according to (*CUDA C Programming Guide* n.d.) the texture cache can only copy from Pinned memory. If only the texture memory implementation had access to pinned memory it may give it an unfair advantage; pinned memory reduces the number of copies required and in turn increases potential throughput as can be seen in figure 4. To maintain consistency all basic search types utilize pinned memory.

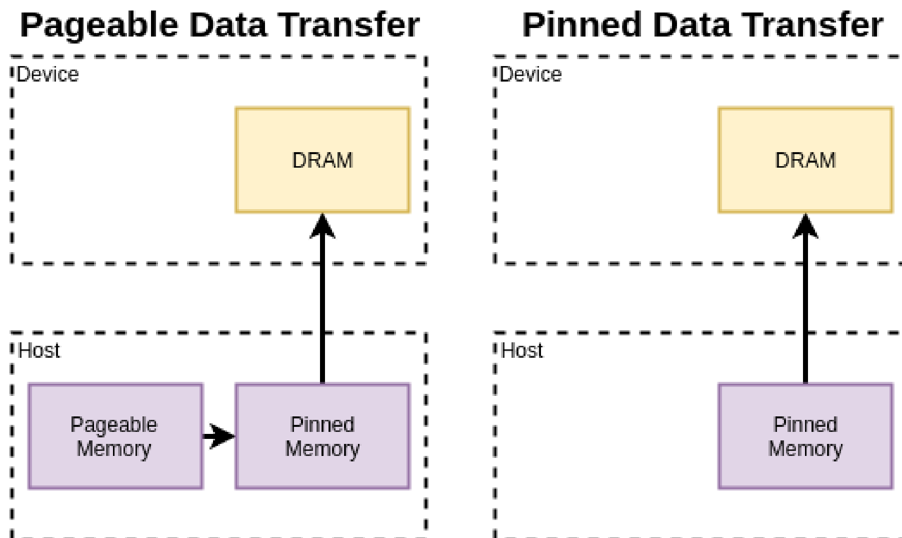

Figure 4: Pageable vs Pinned Memory

3.2 Optimisations

The primary purpose of optimisation is to increase throughput. This section will cover; how types of possible optimisation were identified, how they were compared, and what version was implemented.

3.2.1 Selection Process

Identify

By analysing key components of the provided HEPFAC_C implementation and the HEPFAC algorithm itself, optimisation research could be focused on commonly used functions and operations. For example, population counts are a critical component of the build, reduce, and search processes. For each character placed, and each characters searched a population count takes place. As such, minimizing the time taken to complete a population count could prove extremely beneficial to throughput. The CUDA documentation (*CUDA C Programming Guide* n.d.), best practices guide (*CUDA C Best Practices Guide* n.d.) and Nvidia developer articles such as (Harris 2012) provided suggestions and tips that lead to several other optimisations including (but not limited to); Read Only, Copy Minimization, and Concurrency.

Compare

Some types of operations had several alternate methods and/or techniques that could be used exclusively. Of the reviewed literature; generally speaking, research into these methods tends to focus on the number of operations required to complete the task, or the time it took to complete. However, very few provided direct comparisons to alternate methods and those that did, failed to select reasonable comparisons; positively skewing the results.

To select the most appropriate method, simple benchmarks implementing each were setup. These benchmarks not only demonstrated a performance comparison but acted as method verification, some methods did not work as expected -which will be discussed in more detail in *Selected optimisations*. The benchmarks used a strict timing system; they measured the average completion time for each method (in nanoseconds) and accounted for the timer launch period (28 nanoseconds). Averages were based on 1,000,000 runs -ensuring highly accurate results. The best performing method in each of the benchmarks was then selected for full implementation in the project prototype (hereby referred to as ***HEPFAC_CPP***).

3.2.2 Selected Optimisations

By studying the CUDA Programming Guide and analyzing the HEPFAC algorithm; key areas of potential optimisation were identified. These areas are as follows;

- Population Count
  - Counting the number of binary 1's in a given variable
- Copy Minimization
  - Reducing the size and/or required number of memory copies
- Read Methods
  - Using various types of memory effectively
- Concurrency
  - Utilising streams, asynchronous copies and asynchronous kernel launches where appropriate
- Bitwise
  - Bitwise operations are often thought to be faster than their arithmetic counterparts -are they?

Population Count

As previously discussed, there are several ways to perform a population count. Although prior research has touched on the number of operations required to complete each population count method, no direct side by side measured comparison was found. As such, a small benchmark demo was created to identify the best performer, which could then be implemented in the prototype. The AMD64 Divide and Conqeur method (*Software Optimization Guide for AMD64 Processors* 2005) proved to be the best performer -as can be seen in figure 5- and so was highlighted as a possible GPU optimisation. Validity of the *Actual* value is forced by decorating the function with "`__attribute__((optimise("0")))`" which ensures that no optimisations are applied; it will always provide compile to a completely unoptimised version. Similar tests were performed on GPU which found the CUDA (nvcc) intrinsic method '__popc' to be significantly faster than alternatives. As figure 5 shows, surprisingly G++ 7.3.1-6's intrinsic method fails to provide the correct result at optimisation level 2. The issue was also verified on another system using G++ 8.3.1-2.

```
[andrew@localhost tests]$ g++ popcount_tests.cpp -o popctest -O2
[andrew@localhost tests]$ ./popctest
~Population Count Tests: running 1000000 times to get average ns~
Population Count | Optimization Type          | Time (nanoseconds)
Actual      = 76 | No Optimizations(Actual)   | 546
Test Case 1 = 76 | Bitwise Only               | 185
Test Case 3 = 85 | __builtin_popcount         | 18
Test Case 4 = 76 | Peter Wegner               | 55
Test Case 5 = 76 | Div&Conq AMD64             | 7
```

Figure 5: Population Count Benchmark (Nanoseconds)

Copy Minimization

To reduce the size of copies to and from the device, the layout and structure of copied data was considered. The data structure representing nodes (in the trie) were already minimized in the HEPFAC specification; each node contains bitmaps representing the alphabet in use and an offset value. There is no need for nodes to self identify beyond trie verification -e.g. in search methods- as each parent knows the identity of a given child.

As mentioned previously, due to the reduction process the HEPFAC algorithm identifies whether a match was found at a given location but not which specific match was found. Subsequently, only a Boolean is needed to store the match result for a given location. However, Boolean values are rather wasteful - they only use a single bit, resulting in seven unused bits per byte. If each bit could act as one location instead of a full byte per location, a significant reduction to output size could be made. A visualization of the proposed Bitwise indexing method can be seen in figure 6.

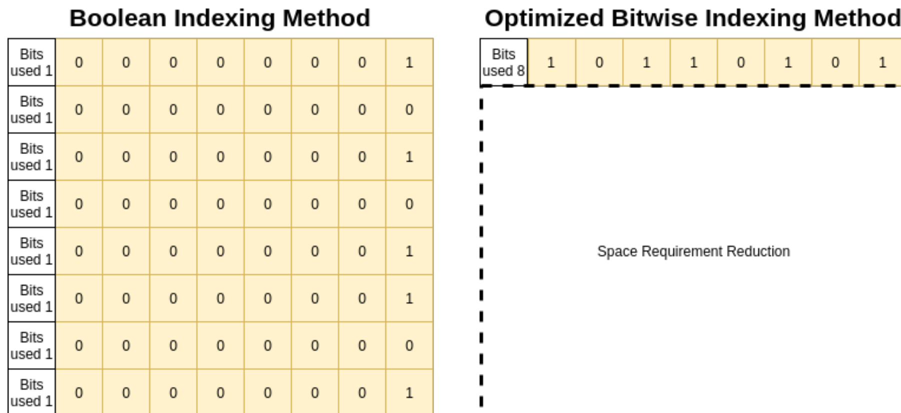

Figure 6: Visualisation of Bitwise Indexing Method

By utilizing each bit in the byte the footprint of the output is reduced, this has a few of knock-on effects; firstly - the time taken to allocate and copy the output is reduced and secondly - due to the reduction there is even more space for the comparison data, which could increase throughput. Additionally, the host-side result

identification may be faster; listing 5 would have to compare 32 times the number of locations as listing 4 for every 32 locations with 0 matches. A performance demo of the two methods can be seen in section A. The data type used for the Bitwise indexing could be altered for different pattern frequencies. For example, if patterns aren't expected to be within 64 characters of each other, a 64 bit type such as a double could be used to further improve performance.

```
int Hepfac::get_matches(unsigned output[]) {
    int matches = 0;
    for (int i=0; i < ELEMENTS/32; ++i){
        if(output[i]){
            for(int j=0; j<32; ++j){
                if (output[i] & (1 << (j % 32))){
                    ++matches;
                    match_callback((i*32)+j);
                }
            }
        }
    }
    return matches;
}
```

Listing 4: Bitwise Indexing Method

```
int Hepfac::get_matches(unsigned output[]) {
    int matches = 0;
    for (int i = 0; i < ELEMENTS; ++i){
        if(output[i]){
            ++matches;
            match_callback(i);
        }
    }
    return matches;
}
```

Listing 5: Boolean Indexing Method

Shared Memory

As an explicitly user-managed L1 cache of sorts, Shared memory effectively allows control of what is cached -the trie can be directly copied to cache which will improve the access speed. Shared memory is not a perfect solution, when shared memory is allocated it reserves sections of the primary cache. Subsequently, the GPU will be less able utilize the primary cache which may reduce the performance of other search operations. However, potentially lower cache utilization is the least of Shared memory performance concerns. As per (*CUDA C Programming Guide* n.d.) Shared memory has "the lifetime of the block"; attempting to access previously copied data in a separate kernel launch will produce undefined behaviour. To ensure this does not happen the trie must be copied to shared memory at every kernel launch, including concurrent kernel launches.

Texture Memory

Texture Memory is cached in a dedicated *Texture Cache*, it "costs one memory read from device memory only on a cache miss, otherwise it just costs one read from texture cache" (*CUDA C Programming Guide* n.d.). As previously mentioned, texture memory does not require reads to be coalesced and copies to texture memory are handled automatically through the texture object API. It is not clear how the texture objects are bound, or whether they are persistent between kernel launches; however, the fast read speed and (more) flexible access patterns should make this the fastest unoptimised search type.

Global Memory

Global memory is often made read/write when the compiler cannot determine if pointers overlap. In this mode, it requires a high number of clock cycles to access. By comparison, Shared and Texture memory are effectively managed cache space and have the access speeds to match; with no optimisations it is expected that they would be faster. Global memory doesn't have to be read/write, with a minor modification listing 9 the compiler can be strong-armed into making it read-only. Using as does `const` with the C99 restrict qualifier effectively tells the compiler that a given pointer's data is probably read-only. If the compiler takes the hint, global memory will be able to make much better use of its cache and will likely perform better than shared memory due to the trie copy overhead.

Concurrency

The rules of concurrency in CUDA are quite simple; host to device (HtoD) copies may occur at the same time as device to host (DtoH) copies (on supported hardware), at the same time as kernel execution. If there are

multiple HtoD transfers they will be queued and executed sequentially, the same is true for DtoH transfers. Kernels will only be queued if there is insufficient occupancy for concurrent execution. Since each stream is effectively a work queue a common design pattern involves splitting up required work into many '*HtoD, Kernel, DtoH*' streams. By splitting up copies into smaller parts a kernel can start processing data sooner and in turn the result can be copied back sooner. Repeating this processes can massively reduce the overall processing time and maximises GPU occupancy throughout the given task, an example of this can be seen in figure 7.

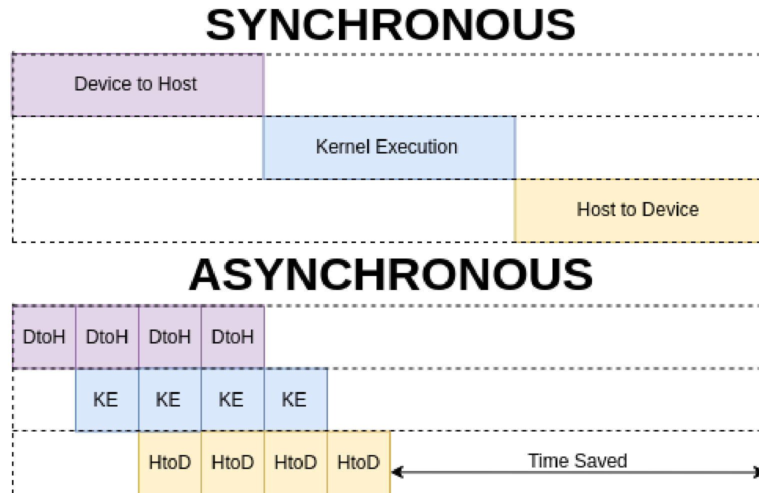

Figure 7: Synchronous vs Asynchronous Execution

Bitwise

Generally, bitwise optimisations are a topic of debate for many developers; bitwise should be faster due to fewer operations but the compiler should optimise situations where arithmetic can be swapped for a bitwise equivalents. This topic is briefly discussed in (*CUDA C Programming Guide* n.d.): "Integer division and modulo operation are costly as they compile to up to 20 instructions. They can be replaced with bitwise operations in some cases: If n is a power of 2, (i/n) is equivalent to (i¿¿log2(n)) and (i%n) is equivalent to (i&(n-1)); the compiler will perform these conversions if n is literal". These *optimisations* are not expected to improve performance but rather demonstrate that there is no difference due to compiler conversions.

3.3 Development

3.3.1 Design

Language & Platform

The development process of ***HEPFAC_CPP*** followed two main directives: maximise performance and (where possible) adhere to coding principles & standards. Typically readable code structures are preferred over more complex ones, however, some optimisations cannot be implemented cleanly. Thus adhering to readability principles such as KISS (Rich 1995) is not always possible. As ***HEPFAC_CPP*** will be made open source, it important that complex optimisations either reference the relevant material or offer inline explanations within comment blocks.

When considering GPGPU development there are two main platforms to consider, OpenCL and CUDA. While OpenCL is the more portable option -most graphics cards support OpenCL, CUDA seems more thought out and the documentation is more comprehensive. Additionally, previously discussed GPU solutions often used CUDA - providing a more direct comparison. Both C and C++ are supported by CUDA extensions. For both C & C++ the NVCC compiler -which is an extension of GCC- can be used. C++ is a widely used language, according to (Diakopoulos and Cass 2016) it is the 4th most popular programming language. C++ () was

selected for development as its standard library functions and data types reduce boilerplate code, and it is more readable than C.

Expandable & Flexible

In abiding by C++ coding standards and best practices, HEPFAC_CPP utilizes object orientated programming principles throughout. All algorithm related methods are part of the *Hepfac* class. This class abstracts complex method parameters; it provides a simple interface for interacting with core components and hides unnecessary details. A conscious decision was made to implement functions that were not needed specifically for testing but may encourage use of the presented work. For example, a function reference can be provided to the *set_callback* function, this function is then passed the index of each match found upon completion of each search. Other examples include the Texture Object template function section C and BitIndex type declaration section B. These makes it easy to utilize the optimisations presented in HEPFAC_CPP in other applications.

Interface

As the final application is effectively a test suite for HEPFAC optimisations, there is no need for a fancy GUI - a simple text based interface will do. That being said, when debugging, verifying and benchmarking; a clean, easily discernible output is beneficial. As such, an output system was designed around the Bootstrap colours (*Colors*) - primary, success, failure (danger), warning, and info. Each of the output types can be used similarly to *std::cout* except the appropriate colour and formatting is applied to each. An example of the colour system in action can be seen in figure 8.

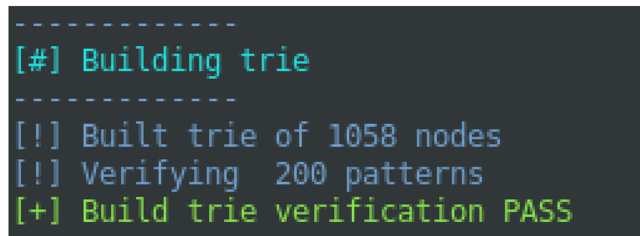

Figure 8: Command Line Interface Output Example

Extensive Verification Methods

Instead of manually verifying trie construction, reduction and search methods; verification functions such as the one seen in figure 8, provide fast, repeatable and conclusive results. Trie construction and reduction can both be verified by searching the trie for each pattern - if any of the patterns cannot be found then it's classed as a fail, otherwise it's a pass. The search methods are verified by searching the pattern file -if the number of matches is the same as the number of patterns then it passes verification, otherwise it fails.

3.3.2 Implementation

This section details feature implementations that deviated from design, expanded upon design or have not otherwise been mentioned. Features that were implemented exactly as outlined will not be discussed further.

HEPFAC Trie Construction

The method used to build the trie in HEPFAC_CPP is quite different to the original in HEPFAC_C; in HEPFAC_C, the patterns must partially searched for the previously placed character before the next character of that pattern can be placed. In HEPFAC_CPP, each pattern is represented by a struct containing the pattern string and an integer 'id' which is the last node associated with the pattern. The addition of 'id' allows the build trie implementation be to greatly simplified as each pattern's next character can be added to it's previously placed character. Simplified pseudocode for the build process can be seen in listing 6 below - as a reminder, a

node consists of 8 bitmaps and an offset which is the index of the first child of that node, a trie is an array of those nodes.

```
trie[pattern_length*pattern_count]
for i=0 to set_pattern_length:
    for pattern in patterns:
        character = pattern.string[i]
        node = trie[pattern.id]
        if node.bitmap contains character:
            pattern.id = index of that character
        else:
            put character into node.bitmaps
            create new node
            if node.offset == root
                node.offset = index of new node
            pattern.id = index of new node

return trie
```

Listing 6: Build Trie Pseudocode

HEPFAC Trie Reduction

Due to the complexity of the trie reduction algorithm, there is was little that could be changed to improve the HEPFAC_CPP implementation. Trie reduction has a strict requirement for merging nodes; merge candidates and each merge candidate's parent node must contain exactly the same children. If this condition is not met, pattern suffixes will incorrectly overlap and the trie will be invalidated.

HEPFAC Search

In HEPFAC_C, the pattern reduction level was hard coded meaning it could only be changed at compile time. Because the pattern reduction level was known at compile time, the search method loops could be unrolled for a performance improvement. As Bellekens noted the loop unrolling provided minimal improvement (Bellekens 2016), it was decided that this technique would not be utilised in HEPFAC_CPP. Instead the reduction level could be set at run time, as part of the Hepfac class initialisation. This approach allows greater flexibility when testing different lengths of signatures - it can be specified using the CLI rather than requiring a separate build of the application.

In CUDA, there is no 'Calloc' (allocate and zero) equivalent for device memory; it must be allocated and then separately set to 0 -either iteratively or using memset- before any searching can begin. If the memory was not zeroed, it is possible that false 'matches' could appear. The HEPFAC_CPP search utilizes a relatively novel idea that removes the zeroing requirement, while introducing no additional operations. All successful pattern matches will begin at the *root* node and finish at the *end* node, the *end* node's offset will always be '-3' - which is how the search detects a match. By setting the output for a given index to the Boolean evaluation "offset == -3", all output locations will be set appropriately and any erroneous data will be overwritten. Pseudocode demonstrating the search process can be seen below in listing 7. Both the build trie and reduce trie functions use a verification system based on this search; the signatures/patterns are provided as the 'input_text'.

```
tid = blockIdx.x * blockDim.x + threadIdx.x
character = input_text[tid]
node = trie[0]

for i = tid+1 to textsize && node.bitmap contains character:
    index of character = popcount of node.bitmaps up to character
    node = trie[index of character]
    character = input_text[i]

// Boolean version
output[tid] = (node.offset == -3)
//Bitwise version
output[tid/32] |= (node.offset == -3) << (tid % 32)
```

Listing 7: Search Trie Pseudocode

Bitwise Indexing

Due to the compile time size requirements, the bitwise indexing template could not be used with the variably sized search output. It may still have uses in other applications where the output size is fixed but it is not suitable for HEPFAC_CPP. A very similar solution was implemented that makes uses malloc to allocate the host output array. The new function used to determine how many bytes should be allocated for the output array can be seen in listing 8, the read method remains the same as section B.

```
void Hepfac::set_output_size(int size){
    #if defined(OPTIMIZATION_OUTPUT)
    result_elements = ((size / sizeof(int)*8) + !!(size & (sizeof(int)*8-1)));
    result_size_bits = size;
    #else
    result_elements = size;
    result_size_bits = size*sizeof(int)*8;
    #endif

    #if defined(OPTIMIZATION_OUTPUT)
    outp_size = result_elements*sizeof(int);
    #endif
}
```

Listing 8: Bitwise Index Size Calculation

Preprocessor Directive Toggles

To easily enable and disable the various optimisations, preprocessor directives were used to swap optimised and unoptimised lines of code, depending on if the appropriate '#define optimisation_name' was set or not. Using this method does reduce readability of the code a little, however it means that a optimisations can be enabled or disabled simply by commenting or uncommenting a line at the top of the file. Preprocessor directives are also used to set the level of debug output required, in the HEPFAC_CPP there are three options for this; off, basic and verbose. To make each build easily discernible, another one of the directives places code that prints out the enabled optimisations when a Hepfac object is constructed. The example shown in listing 9 demonstrates how the read-only optimisation is enabled or disabled for the global memory search.

```
__global__ void
#ifdef READ_ONLY_CONST
search_trie_global(NodeReduced* __restrict__ trie, const char* __restrict__ input_text,
                   const unsigned size_text, const int text_offset, unsigned* out) {
#else
search_trie_global(NodeReduced* trie, char* input_text, unsigned size_text,
                   int text_offset, unsigned* out) {
#endif
```

Listing 9: Preprocessor Directive to Enable Read optimisation

3.3.3 Testing

To ensure that each optimisation, search type, and class method performs as expected, unit tests were created to comprehensively test every aspect of HEPFAC_CPP. While tests now primarily rely on the output of the validation functions, initially unit tests were validated manually - pen & paper algorithm execution versus program output, but as more test cases were added this proved impractical. When testing both the files and patterns were randomly generated to maximise the possibility of erroneous results. Test cases were as follows;

- 100x 6 length signatures vs 1000MB file
  - W/ No optimisations
  - W/ Bitwise optimisation
  - W/ Popcount optimisation
  - W/ Read-only optimisation
  - W/ Asynchronous execution
  - W/ Bitwise Output
  - W/ All optimisations
- 25x 10,20,30,40,50 length signatures vs 1000MB file
  - W/ No optimisations
  - W/ All optimisations
- 25x,50x,100x,200x 6 length signatures vs 1000MB file
  - W/ No optimisations
  - W/ All optimisations
- 100x 6 length signatures vs 1MB,10MB,100MB,1000MB files
  - W/ No optimisations
  - W/ All optimisations

4 Results

4.1 Unit Tests

During development, unit tests were created to ensure that all optimisations, search types, and class methods perform as expected. HEPFAC_CPP passed all unit tests; no failures were encountered in any configuration of the application. The following table details the unit tests and their pass status:

100x 6 length signatures vs 1000MB	Status
W/ No optimisations	PASS
W/ Bitwise optimisation	PASS
W/ Popcount optimisation	PASS
W/ Read-only optimisation	PASS
W/ Asynchronous execution	PASS
W/ Bitwise Output	PASS
W/ All optimisations	PASS
25x 10,20,30,40,50 length signatures vs 1000MB file	Status
W/ No optimisations	PASS
W/ All optimisations	PASS
25x,50x,100x,200x 6 length signatures vs 1000MB file	Status
W/ No optimisations	PASS
W/ All optimisations	PASS
100x 6 length signatures vs 1MB,10MB,100MB,1000MB files	Status
W/ No optimisations	PASS
W/ All optimisations	PASS

Table 1: Unit Tests

4.2 Test System Specifications

The performance of a given optimisation will at least in part be determined by the system on which it is benchmarked. This can make comparing to other implementations challenging as it is very unlikely that exactly the same system will be used for testing. While side by side comparisons on the same system cannot be performed, providing detailed system specifications will enable comparisons based on theoretical system performance to be made. For example, the GTX 1080 that was used to test a later build of HEPFAC_C is estimated to be around 25% faster than the GTX 980Ti used for HEPFAC_CPP testing.

The full system specifications of the test system can be seen in table 2 below. To ensure consistent performance all benchmarks shown were performed in a terminal environment - the graphical user interface was disabled to avoid any overhead which could negatively impact results.

Form Factor	Desktop
Operating System	Fedora 29
Kernel	Linux 5.0.5-100
Processor	Ryzen 7 1700
Processor Specifications	8 Core @ 3.0GHz, 16 Threads
CPU Memory	16GB DDR4 3200MHz CL15
GPU	Nvidia GTX 980Ti
CUDA Cores	2816
GPU Memory	6GB GDDR5 7000MHz
Storage Device	Plextor 256GB NVME SSD

Table 2: Test System Specifications

4.3 Search Optimisation Benchmarks

All benchmark results are based on averages. Due to the sheer number of builds to evaluate, it was not practical to perform any more than 100 runs of each configuration. The gathered results are are split into four main sections, one for each test case. To maintain readability only graphs are displayed in this section, however, the corresponding measurements can be found in section D, section E, and section F.

4.3.1 Individual Optimisations

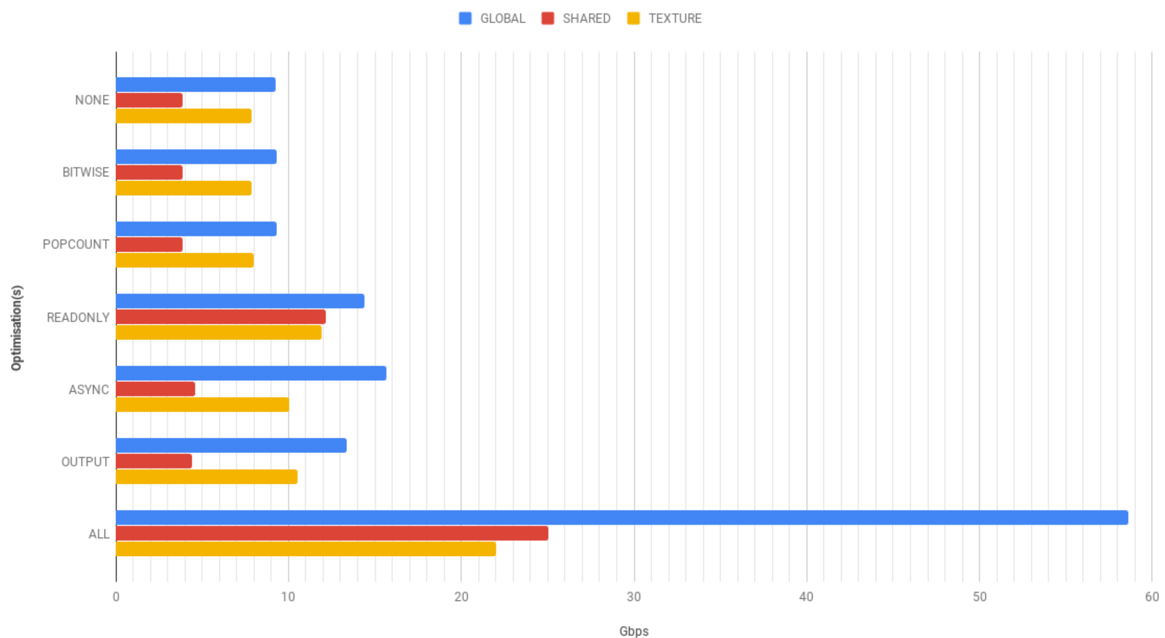

Figure 9: Individual Optimisations - 100x6 Trie vs 1GB Random Data

To identify which optimisations were the most (and least) effective, a build using no optimisations was compared to builds with an individual optimisation enabled, and to a build with all optimisations enabled. figure 9 demonstrates how each optimisation affected throughput, and how combining them can produce a significant improvement. These results can be used to recommend *worthwhile* implementations for time limited

projects. Using all optimisations, throughput exceeds the imposed 40Gbps target, achieving 58.628Gbps - more than twice what prior research achieved. Surprisingly, the results also indicate that Bitwise optimisations marginally improve throughput as can best be seen in table 3 below.

Optimisation	GLOBAL	SHARED	TEXTURE
NONE	9.27025	3.85691	7.84752
BITWISE	9.30484	3.86138	7.88799
POPCOUNT	9.30887	3.8788	7.95518
READONLY	14.3663	12.1498	11.8897
ASYNC	15.676	4.60139	10.0294
OUTPUT	13.3722	4.4215	10.5487
ALL	58.629	25.0288	22.0038

Table 3: Individual Optimisations - 100x6 Trie vs 1GB Random Data

4.3.2 Variable Signature Length

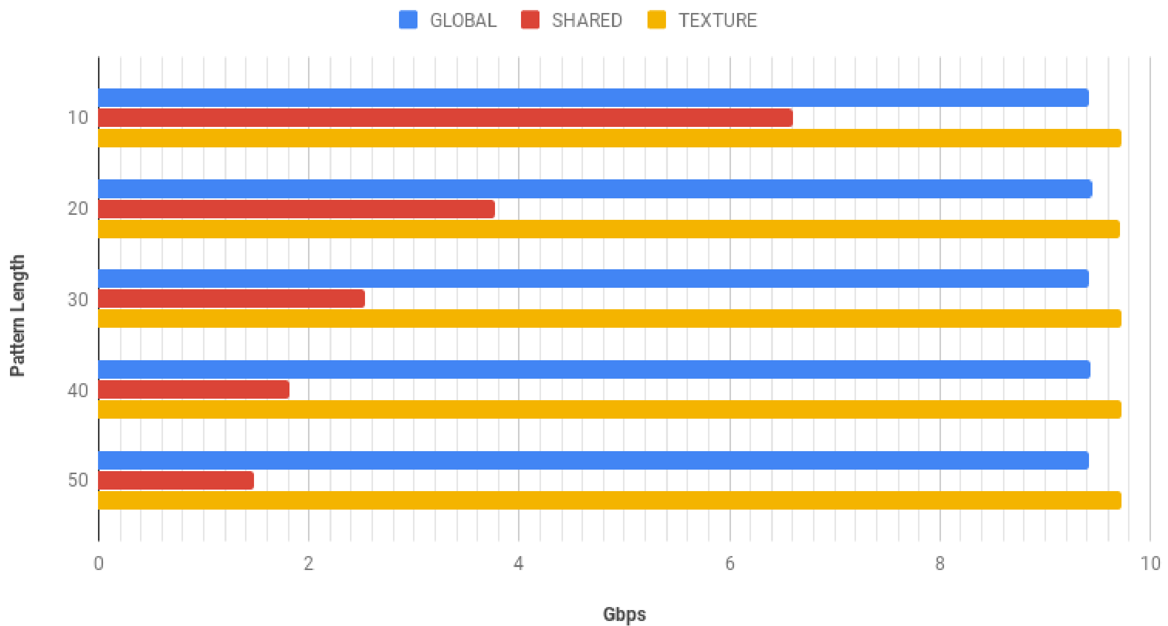

Figure 10: Fixed Count Variable Length - 25xN Trie vs 1GB Random Data (Unoptimised)

As figure 10 and figure 11 show, as signature length increases the number of nodes that must be compared before a match can be identified also increases. As was previously mentioned, when using shared memory the trie must be copied each time; as signature length increases so to does trie size, and thus copy time. The results for variable signature length are mostly as expected; throughput of shared search decreases as pattern length increases, throughput of global and texture memory remains mostly the same. It is possible that with longer patterns, global and texture memory would exhibit similar throughput reductions, however, due to cache size limitations these could not be tested. Throughput measurement data can be seen in section D.

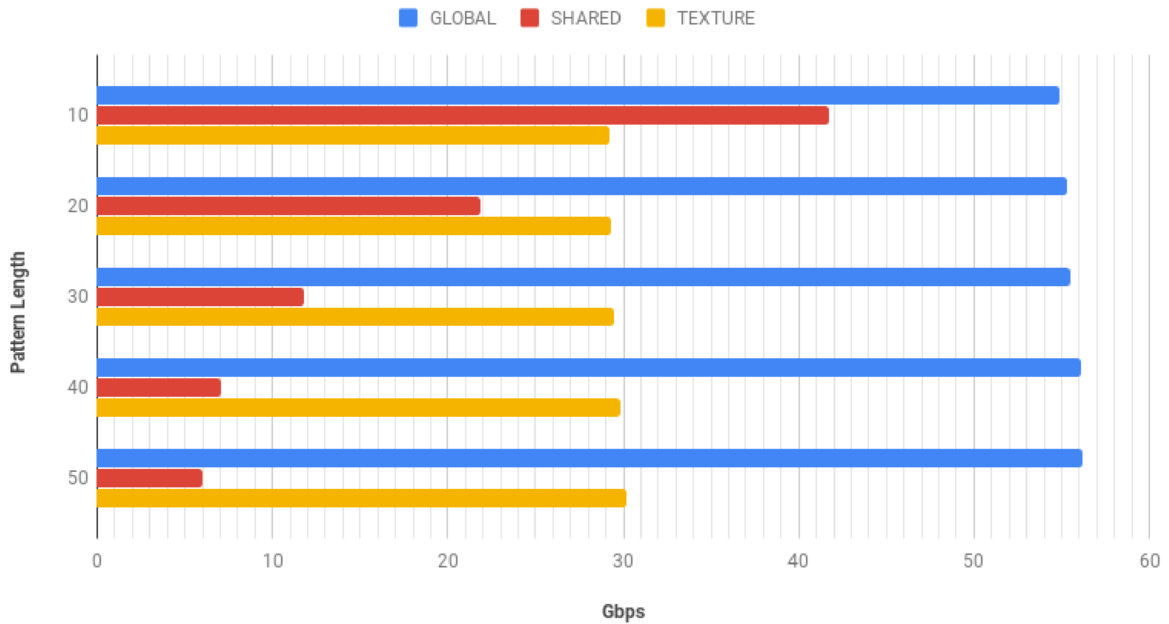

Figure 11: Fixed Count Variable Length - 25xN Trie vs 1GB Random Data (Optimised)

4.3.3 Variable Signature Count

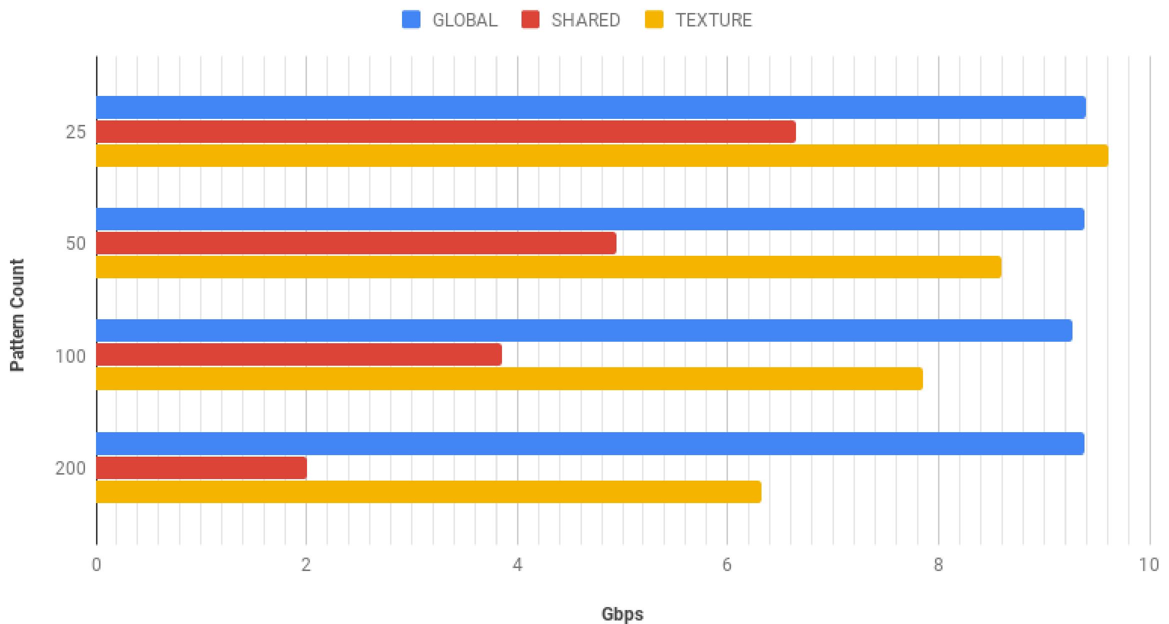

Figure 12: Variable Count Fixed Length - Nx6 Trie vs 1GB Random Data (Unoptimised)

figure 12 and figure 13 demonstrate how throughput changes as pattern count changes. As increasing the pattern count while maintaining fixed pattern length also increases the trie size, shared memory is once again

slower the larger the trie gets. Both texture and shared memory show similar throughput reductions across the optimised and unoptimised builds. It appears that global could have benefited from further testing -e.g. 400, 800 patterns- but due to time limits these tests could not be conducted.

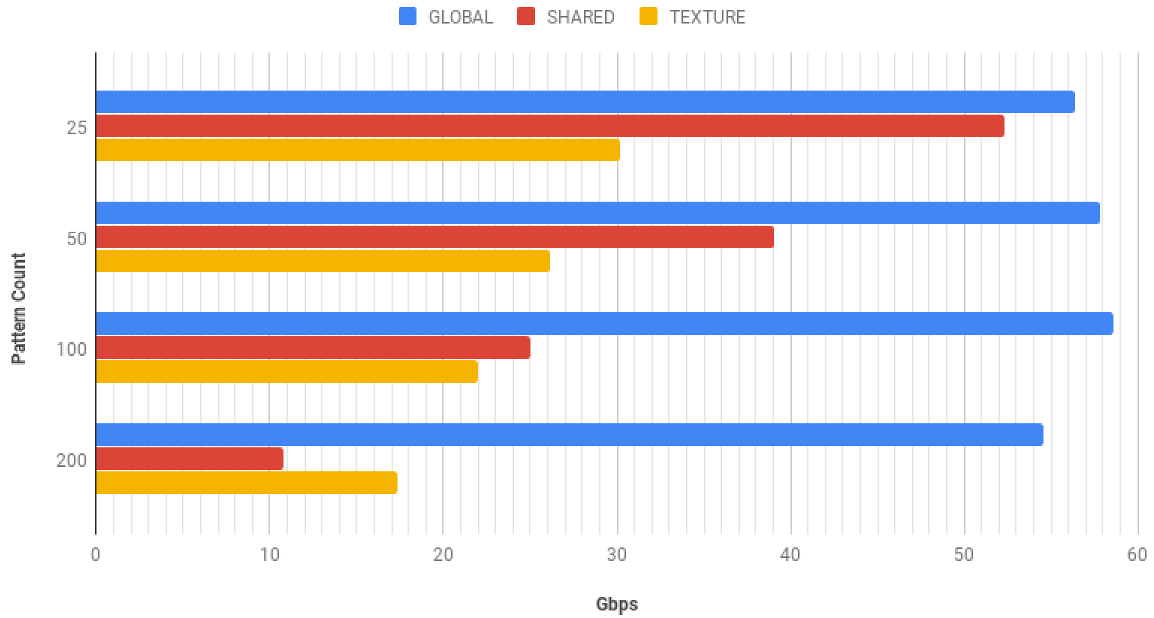

Figure 13: Variable Count Fixed Length - Nx6 Trie vs 1GB Random Data (Optimised)

4.3.4 Variable File Size

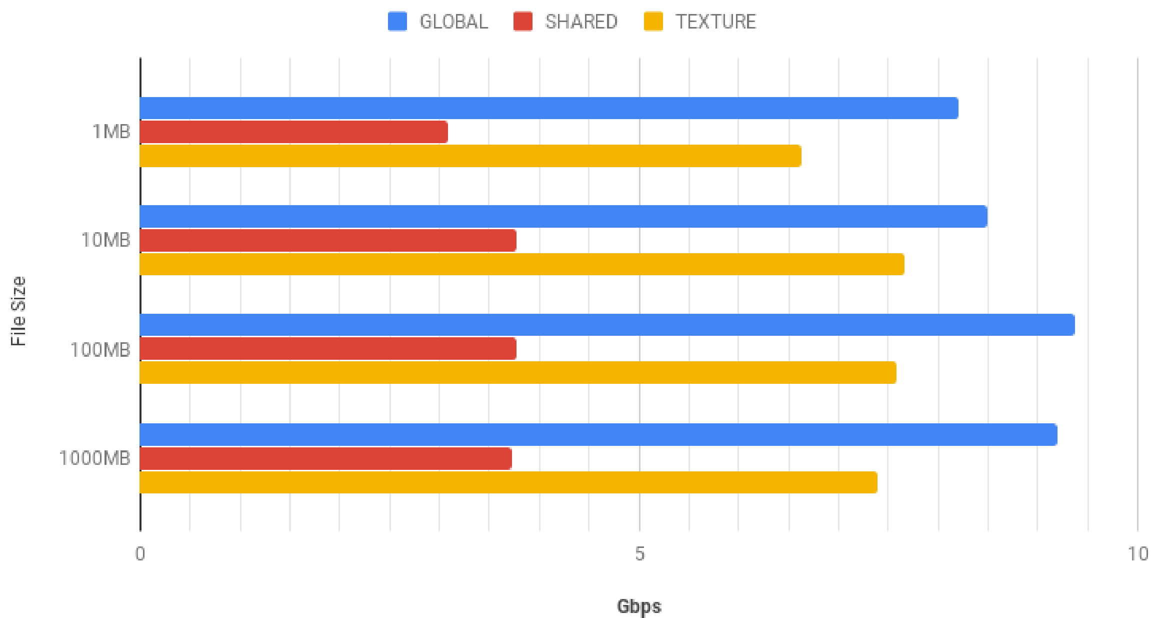

Figure 14: Variable File Size - 100x6 Trie vs N MB Random Data (Unoptimised)

Figure 14 and figure 15 demonstrate how throughput changes as file size changes. For all memory types, throughput is lowest on smaller file sizes. For all unoptimised searches, the throughput sweet spot exists somewhere between 10MB and 100MB. For optimised searches, throughput increases as file size increases, though the improvement is less and less each time. It is expected that if a 10GB/10000MB file was used that throughput would start to trend downwards.

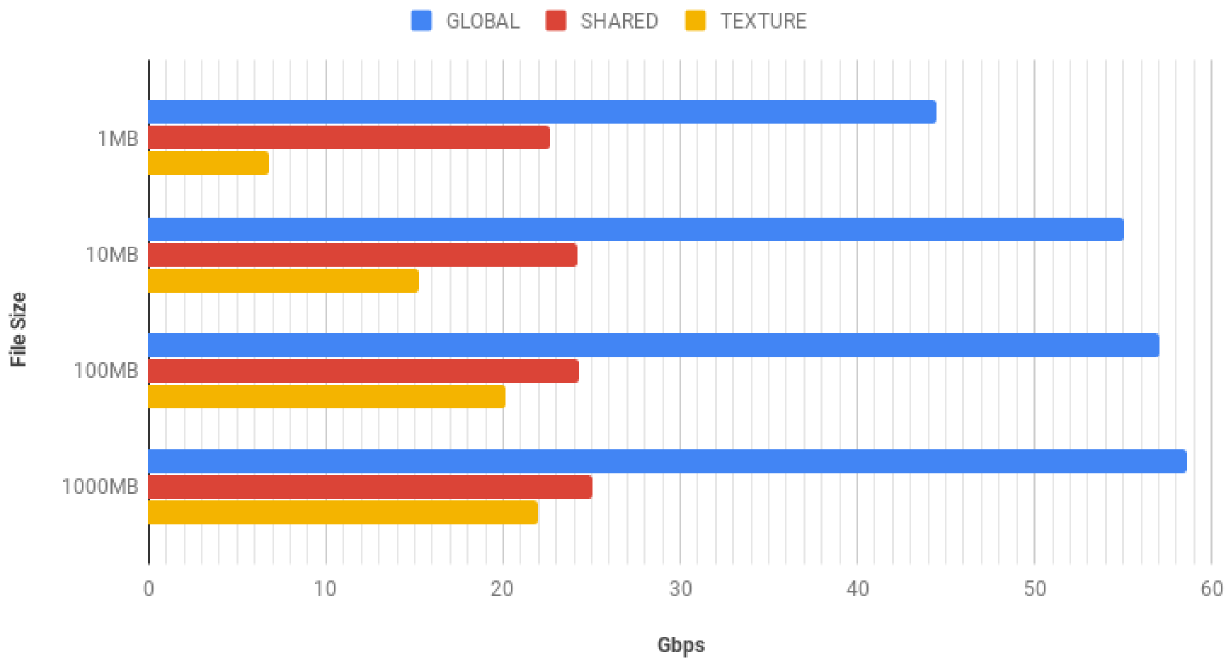

Figure 15: Variable File Size - 100x6 Trie vs N MB Random Data (Optimised)

4.4 Other Benchmarks

Two other benchmarks were also performed - *Bitwise indexing vs Boolean indexing* and *Trie Construction vs Trie Reduction*. The bitwise indexing benchmark shows that the output optimisation isn't just faster due to the decreased copy size, but also due to the 32 at a time search enabled by checking if the storage integer at each location is equal to zero - as can be seen in figure 16 below. If a CPU identification/full pattern verification system was being used, this would drastically improve access time. The Trie construction benchmark shows that the build trie process performs well regardless of the number of patterns added, but the reduction process takes significantly longer as can be seen in figure 17 below. This is likely due to the added complexity of merging and copying nodes.

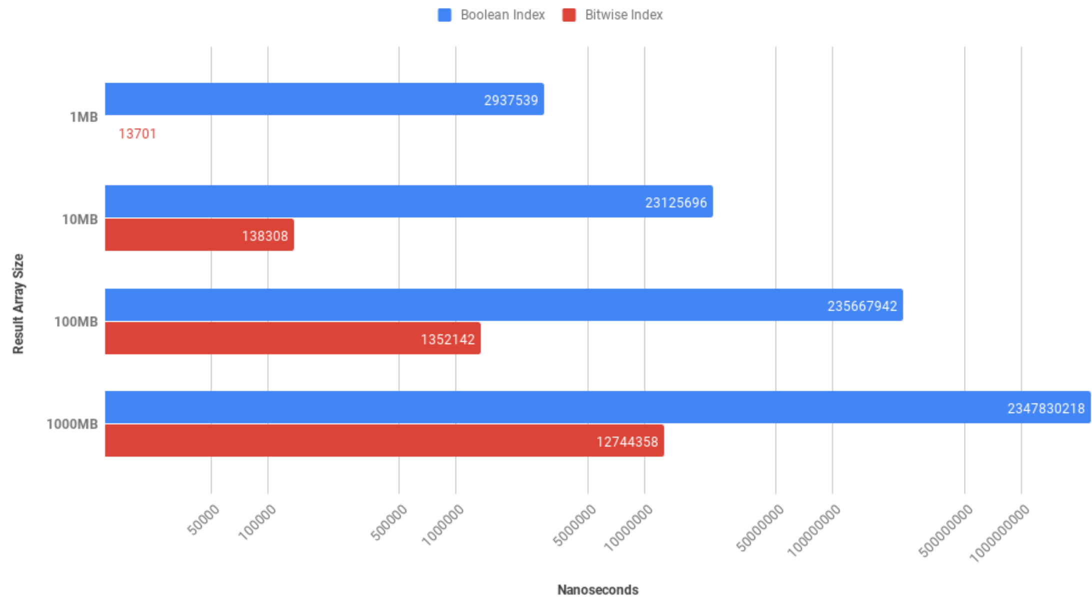

Figure 16: Variable Result Array Size - Bitwise Indexing vs Boolean Indexing

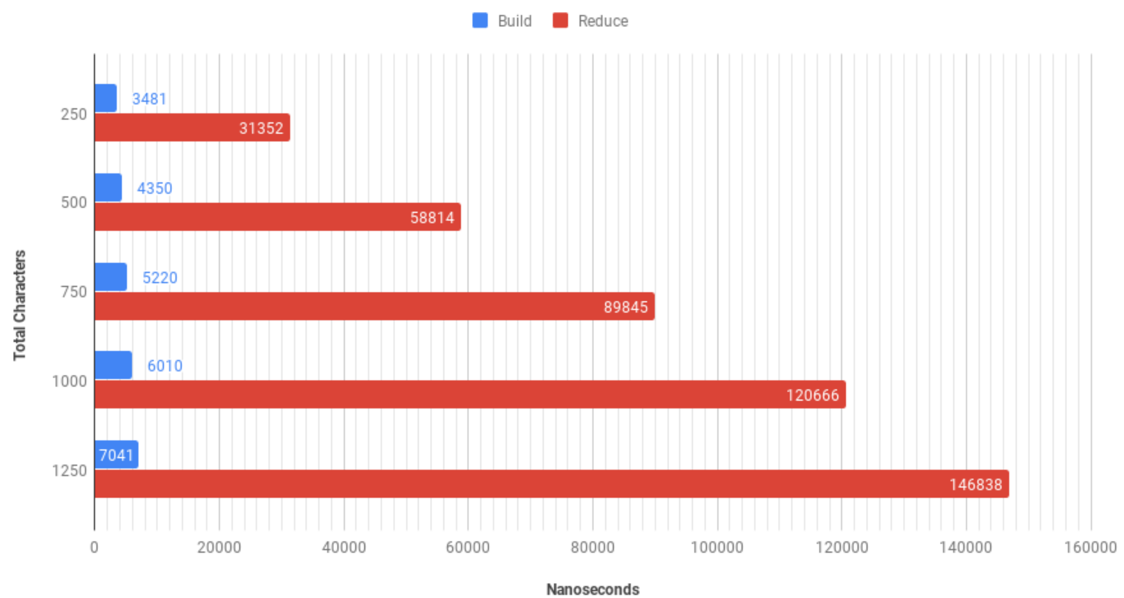

Figure 17: Variable Pattern Characters - Build Trie vs Reduce Trie

4.5 Overview

As all test cases have clearly show, the optimisations provide a clear advantage over the basic implementation. All of the optimisations implemented provided at least marginal improvements over the original methods. In most tests, each memory type performed as expected with and without optimisations; however, there were some outliers which will be discussed in more detail in section 5.

5 Discussion

5.1 Overview

The research question asked at the beginning of this project was; "How can GPGPU-based Network Intrusion Detection Systems be optimised for high throughput networks?". When reviewing existing solutions such as Suricatta it was found that there was nothing to optimise, not because the implementation was perfect but because it had never been finished. As such, the primary focus of the project changed.

The project focus became the implementation and optimisation of a pattern matching algorithm for use in signature based network intrusion detection systems; that could in turn meet throughput requirements of modern networks(40Gbps). HEPFAC_CPP has succeeded in this using commodity hardware to achieve throughput that would otherwise only be seen in high end commercial solutions. As benchmark results have demonstrated, the HEPFAC_CPP implementation is more than capable - using optimisations it consistently achieves over 54Gbps.

While some of the optimisations focus on operations specific to the processes of the HEPFAC algorithm, many are widely applicable, and the results demonstrate that they are worth implementing. Some of these provided huge performance bonuses and required only minimal source code modifications, for example the use of __restrict__ and const to force fast read-only access.

5.2 Analysis of Results

During initial research, global memory appeared to be the most disadvantaged type and it was expected that it would be the worst performer. However, even in the unoptimised build, global outperforms texture and shared memory. These results conflict with the CUDA documentation which states that texture and shared memory are significantly faster than global (*CUDA C Programming Guide* n.d.). However, these results can (mostly) be explained.

Global

Global memory performs the best and benefits the most from optimisations. It was clear that it stood to benefit the most from the read-only optimisation as it otherwise has very limited access to cache. It is assumed that this fast, automatically managed cache access is what gives it an edge over other search types. Global also benefited the most from concurrency, figure 18 shows global search profiled without concurrency and figure 19 shows it profiled again but with concurrency. No other optimisations were enabled between these two tests and yet there is a 4ms execution time difference.

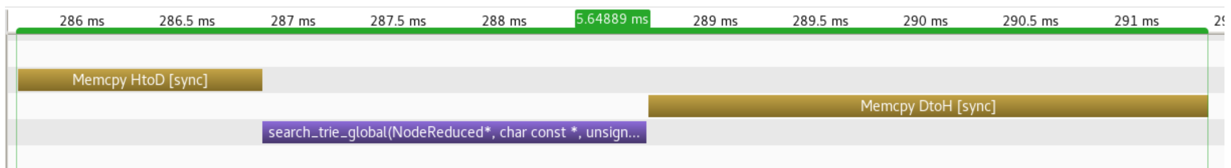

Figure 18: Nvidia Visual Profiler - Unoptimised Global Search

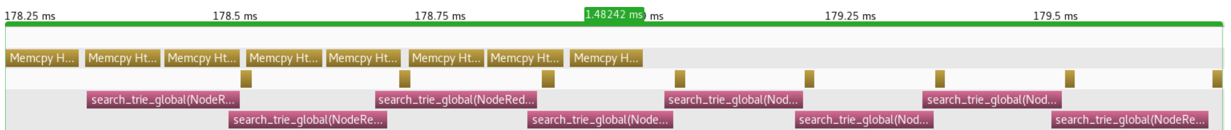

Figure 19: Nvidia Visual Profiler - Global Search W/ Concurrency

Shared

Shared memory performs the worst on average across all tests. The suspected cause of this lackluster performance is the trie copy. As the data copied from global memory to shared memory is only guaranteed for the lifetime of the kernel (*CUDA C Programming Guide* n.d.), the trie must be copied from global to shared memory each time the kernel is launched. As can be expected, this introduces additional overhead. Currently there is no way to copy directly from the host to shared memory. Another possible cause is the substantial reduction to occupancy; using shared memory reserves a significant portion of the cache which can then not be used to quicken other aspects. While 100% occupancy does not demonstrate that the GPU is being used to its full potential, low occupancy (such as in figure 20) is a tell tale sign that the GPU is **not** being used to its full potential.

search_trie_shared(NodeReduced*, unsigned int, char const *, unsigned int, int, unsigned int*)

Queued	n/a
Submitted	n/a
Start	186.56677 ms (186,566,765 ns)
End	186.9989 ms (186,998,902 ns)
Duration	432.137 µs
Stream	Stream 22
Grid Size	[8789,1,1]
Block Size	[128,1,1]
Registers/Thread	32
Shared Memory/Block	15.855 KiB
Launch Type	Normal
Occupancy	
Theoretical	37.5%

Figure 20: Nvidia Visual Profiler - Optimised Shared Search Occupancy

Texture

The texture memory results are rather confusing; in reviewed literature, texture memory typically provided the fastest solution. To better investigate this strange behaviour, the Nvidia Visual Profiler was employed. As can be seen in figure 21, the first texture search kernel execution takes significantly longer than the rest. It is theorized that this initial overhead is the texture object is being bound to the texture cache. In the older 'texture bind' method, the texture would be bound to the texture cache well in advance of the kernel execution - explaining the performance discrepancy. It is possible that if the 'texture bind' method was used, the texture search could have performed better.

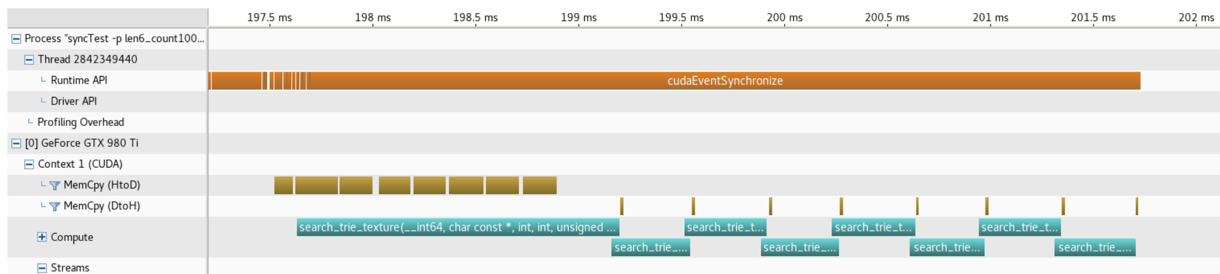

Figure 21: Nvidia Visual Profiler - Optimised Texture Search

5.3 Recommendations

As each of the implemented optimisations has already been explained in great detail, this section will instead suggest when and why said optimisations should be used (if at all). The following suggestions are evidenced by the results presented in section 4:

- Bitwise
  - Usually not worth implementing, provides insignificant improvements
- Population Count
  - AMD64 Population count is significantly faster than other CPU methods
  - CUDA's intrinsic __popc() method is faster than other GPU methods
  - These methods are valid for 32-bit types, but could be altered for 64-bit types
- Read-Only Data
  - Use of const and __restrict__ suggest to the compiler that data should be read-only
  - Read-only data can be accessed significantly faster than read/write data
  - Should only be used on data that will not be modified for duration of the kernel
- Concurrent Execution
  - Generally applicable and worth implementing, provides significant improvements
  - Effectively removes device memory limits by queuing tasks in streams
- Bitwise Indexing
  - Applicable when only location is required; use in place of a Boolean array
  - Significantly reduces allocation and copy time (8x reduction)
  - Significantly speeds up index identification (200x faster on single thread)
- Branch Divergence Minimisation
  - Generally applicable and worth implementing, provides significant improvements
  - Avoid nesting branching statements, prefer single multiple-condition statements
  - Expect true-evaluated statements to be executed first, structure accordingly
- Pinned Memory
  - Applicable for long-term allocations and frequently used host/device structures
  - Reduces number of copies required to reach device, improving performance
  - Do NOT frequently allocate/deallocate as this introduces significant overhead
- Structure Minimisation
  - Generally applicable, can provide significant improvement
  - Structures containing only essential members are smaller - less to copy
  - Smaller copies equal less time spent copying, improving performance

5.4 Problems Faced

While the CUDA documentation is relatively good, there are still areas of it that are significantly lacking. The texture memory API documentation was pretty much useless; it doesn't specify how all read modes are used, and while it does list all members, it doesn't mention which are optional and which are required. Texture memory was implemented through trial and error; even now it is possible that the method used is incorrect. By comparison, shared memory had many usage examples and the documentation was clear on when and how it should be used.

The current implementation of the trie reduction function will only work if there are two or more patterns. A single pattern unit test was not considered as the functions purpose is to merge multiple pattern suffixes. The materialisation of this error is quite humorous; it deletes all nodes except root. The function attempts to merge similar nodes that have similar parents; when a single pattern trie is reduced 'current node' is always the same as 'previous node', and of course the parent would also match. As it ascends the trie it deletes these *duplicate* nodes until it reaches root. The issue is easily resolved by adding a unique condition for single pattern tries (do not attempt to reduce - there is nothing to merge with).

One of the biggest issues faced was finding relevant, quantifiable literature. Many supposedly relevant GPGPU papers didn't provide comparable results, did not fully detail the algorithms or implementation used because of corporate interests, or only tested very specific cases. These factors made it difficult to discern what techniques should be included in the basic implementation, and at what point should a technique should be considered an *optimisation*.

5.5 Algorithm

The main limitation of the current version of the HEPFAC algorithm is its rigidity; with the additional understanding implementation brings, some aspects of the algorithm become frustrating to work around. For example; trie reduction prevents signatures of differing lengths - all signatures are trimmed to a set size, and even in the build process, prefix patterns cannot exist.

Trie Reduction

As the trie reduction function has no way of calculating the final size of a given reduction by the count or length of patterns, there is no way to calculate how many patterns can be added to the trie for a given memory type. Unless the pattern file has been specifically crafted for trie reduction, the worst case has to be assumed which is *2 + (count * length) - count*. Reduction cannot be reliably used as a way to fit more patterns into a given memory type unless the worst case size is assumed.

Prefix Patterns

Another issue with the HEPFAC algorithm is the lack of support for prefix patterns - patterns that exist within the first few characters of another pattern. Adding support for prefix patterns would require major modifications to the build and search processes. These modifications would introduce more branching and thus reduce peak performance. HEPFAC performs best as an pattern index search, rather than a pattern identification search; it is best suited for finding where (rather than what) patterns are. Considering the strengths of the algorithm, another possible solution to the prefix support issue was devised; keep only the prefix patterns. This seems like an anti-solution, as it effectively discards larger patterns. However, they never mattered anyway; if a location is a match of a prefix pattern or of a full size pattern the result is the same, the index is returned. The index results could then be used to direct a fast CPU-based identification which can then determine which pattern specific pattern was matched.

6 Conclusions

Being able to quickly and accurately detect attacks is paramount to many organisations; initial research found that network intrusion detection systems are less and less able to keep up with modern network throughput. Current solutions require specialized hardware that often costs several thousands, and while there have been attempts to utilise GPU, support has dwindled. The pattern matching process is an essential part of signature based intrusion detection, HEPFAC_CPP was developed throughout this project as a fast signature detection prototype. With all optimisations enabled, HEPFAC_CPP achieved a peak throughput of 58.629Gbps -nearly 20Gbps more than modern network requirements- and for that reason it is considered a success.

This research has demonstrated how various optimisations and design decision can improve the throughput of a signature detection prototype -and potentially other GPU based applications. Of the eight optimisations tested, all were found to improve the performance -although to varying degrees. Some of the optimisations significantly reduce the readability of the code, and in turn may deter developer interest. To ease and encourage developer implementation, optimisations have been extensively documented, simplified through method abstraction or provided as template functions that can be used in any application - not just HEPFAC_CPP.

It is hoped that the results of this investigation can be utilised in current and future development; be it a network intrusion detection system, digital forensics file carver, or even DNA sequence matching. The signature detection prototype, data used to test it, and optimisation proof of concepts will be made open source to encourage implementation and further research. New research often focuses on algorithmic improvements, this investigation has demonstrated that the performance of an algorithm is only as good as its implementation.

6.1 Future Work

HEPFAC_CPP was designed as a showroom of sorts, for various optimisations. The objective was never to create a full network intrusion detection system but to demonstrate how a crucial component of one could be implemented. However, it was intended that its teachings could be used to further the development of current open source solutions such as Suricatta or Snort. Given the only external library requirement is CUDA, it should be relatively easy to apply the necessary changes - by altering current methods or by replacing them with equivalents from HEPFAC_CPP. Either way, HEPFAC has laid the foundations for a highly effective yet affordable real-time network intrusion detection system.

While this paper focused on NIDS, future research could explore other applications. It is believed that the prefix support solution outlined in section 5.4.1 would be particularly effective in forensic file recovery applications. If forensic file recovery was explored, removal of the trie reduction function (as per the reasons outlined in section 5.4.1) would be tested. However, the larger trie size would increase cache occupancy, which may in turn reduce performance rather than improve it.

All tests of HEPFAC_CPP were performed on the same system as outlined in section 4.2, it would be beneficial to test on other systems and platforms such as windows. The GTX 980Ti used in testing is approximately 25% slower than the GTX 1080 (*User Benchmark Comparison - GTX 980Ti vs GTX 1080* 2019) which was used in testing of HEPFAC_C. It would be interesting to see if performance scales linearly; HEPFAC_CPP could reach throughput around 72Gbps.

7 References

Aho, AV and MJ Corasick (1975). 'Fast pattern matching: an aid to bibliographic search'. In: *Communications of ACM* 18.6, pp. 333–340.

Arndt, Jörg (2010). *Matters Computational: ideas, algorithms, source code.* Springer Science & Business Media.

Bayne, Ethan (2017). 'Accelerating digital forensic searching through GPGPU parallel processing techniques'. In: *hgpu.org.*

Bellekens, Xavier JA (2016). 'High performance pattern matching and data remanence on graphics processing units'. PhD thesis. University of Strathclyde.

Bellekens, Xavier JA et al. (2014). 'Glop: Enabling massively parallel incident response through gpu log processing'. In: *Proceedings of the 7th International Conference on Security of Information and Networks.* ACM, p. 295.

Chen, Y. (2016). 'Advances in GPU Research and Practice'. In: ed. by Hamid Sarbazi-Azad. Morgan Kaufmann. Chap. Augmented Block Cimmino Distributed Algorithm for solving tridiagonal systems on GPU.

CUDA C Best Practices Guide (n.d.). URL: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html.

CUDA C Programming Guide (n.d.). URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

Cyber Security Breaches Survey 2018: Statistical Release (2018). Tech. rep. Department for Digital, Culture, Media & Sport.

Diakopoulos, Nick and Stephen Cass (2016). *Interactive: The Top Programming Languages 2016.* URL: https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2016.

Freed, Edwin E (1983). 'Binary Magic Numbers Some Applications and Algorithms'. In: *Dr. Dobb's Journal* 8.78, 176–182. URL: http://archive.6502.org/publications/dr_dobbs_journal/dr_dobbs_journal_vol_08.pdf.

'SIMD (Single Instruction Multiple Data Processing)' (2008). In: *Encyclopedia of Multimedia.* Ed. by Borko Furht. Boston, MA: Springer US, pp. 817–819. ISBN: 978-0-387-78414-4. DOI: 10.1007/978-0-387-78414-4_220. URL: https://doi.org/10.1007/978-0-387-78414-4_220.

Gao, Jiaquan et al. (2017). 'Adaptive Optimization l1-Minimization Solvers on GPU'. In: *International Journal of Parallel Programming* 45.3, pp. 508–529.

Harris, Mark (2012). 'How to optimize data transfers in CUDA C/C++'. In: *NVIDIA Developer Zone.*

Iandola, Forrest N et al. (2013). 'Communication-minimizing 2D convolution in GPU registers'. In: *2013 IEEE International Conference on Image Processing.* IEEE, pp. 2116–2120.

Jamshed, Muhammad Asim et al. (2012). 'Kargus: a highly-scalable software-based intrusion detection system'. In: *Proceedings of the 2012 ACM conference on Computer and communications security.* ACM, pp. 317–328.

Khalil, George (2015). *Open Source IDS High Performance Shootout.* URL: https://www.sans.org/reading-room/whitepapers/intrusion/open-source-ids-high-performance-shootout-35772.

Kim, Joongi et al. (2015). 'NBA (network balancing act): A high-performance packet processing framework for heterogeneous processors'. In: *Proceedings of the Tenth European Conference on Computer Systems*. ACM, p. 22.

Lin, Cheng-Hung et al. (2010). 'Accelerating string matching using multi-threaded algorithm on GPU'. In: *2010 IEEE Global Telecommunications Conference GLOBECOM 2010*. IEEE, pp. 1–5.

Luitjens, Justin (2015). 'Cuda Streams: Best Practices and Common Pitfalls'. In: *GPU Techonology Conference*.

Martinelli, Michele. 'GPU I/O persistent kernel for latency bound systems'. In:

Otto, Mark and Jacob Thornton. *Colors*. URL: https://getbootstrap.com/docs/4.0/utilities/colors/.

Petriconi, Felix et al. URL: https://isocpp.org/.

Rennich, S (2011). 'Webinar: CUDA C/C++ streams and concurrency'. In: *Online at https://developer. nvidia. com/gpucomputing-webinars*.

Rich, Ben R (1995). 'Clarence Leonard (Kelly) Johnson: 1910–1990: A Biographical Memoir'. In: *Biographical Memoirs* 67, pp. 221–241.

Scarfone, Karen and Peter Mell (2012). *Guide to intrusion detection and prevention systems (idps)*. Tech. rep. National Institute of Standards and Technology.

Software Optimization Guide for AMD64 Processors (2005). URL: https://www.amd.com/system/files/TechDocs/25112.PDF.

Suricata Support Status (n.d.). URL: https://redmine.openinfosecfoundation.org/projects/suricata/wiki/Support_Status (visited on 08/10/2018).

User Benchmark Comparison - GTX 295 vs 7600GT (2019). URL: https://gpu.userbenchmark.com/Compare/Nvidia-GeForce-GTX-295-SLI-Disabled-vs-Nvidia-GeForce-7600-GT/m7987vsm8028.

User Benchmark Comparison - GTX 980Ti vs GTX 1080 (2019). URL: https://gpu.userbenchmark.com/Compare/Nvidia-GTX-980-Ti-vs-Nvidia-GTX-1080/3439vs3603.

Vasiliadis, Giorgos et al. (2008). 'Gnort: High performance network intrusion detection using graphics processors'. In: *International Workshop on Recent Advances in Intrusion Detection*. Springer, pp. 116–134.

Warren, Henry S. (2012). *Hacker's Delight*. 2nd. Addison-Wesley Professional. ISBN: 0321842685, 9780321842688.

Appendices

A Bitwise vs Boolean Indexing Comparison

```
[andrew@andrew-pc GPU Accelerated Security]$ ./exectime 1
Generating Randoms...
1 in 1 match

# Starting bit index search...
Time micro= 10830
Time nano= 10830895

# Starting boolean index search...
Time micro= 10758
Time nano= 10758805
[andrew@andrew-pc GPU Accelerated Security]$ ./exectime 10
Generating Randoms...
1 in 10 match

# Starting bit index search...
Time micro= 2906
Time nano= 2906240

# Starting boolean index search...
Time micro= 12233
Time nano= 12233278
[andrew@andrew-pc GPU Accelerated Security]$ ./exectime 100
Generating Randoms...
1 in 100 match

# Starting bit index search...
Time micro= 652
Time nano= 652845

# Starting boolean index search...
Time micro= 11279
Time nano= 11279056
[andrew@andrew-pc GPU Accelerated Security]$ ./exectime 1000
Generating Randoms...
1 in 1000 match

# Starting bit index search...
Time micro= 366
Time nano= 366411

# Starting boolean index search...
Time micro= 11772
Time nano= 11772272
[andrew@andrew-pc GPU Accelerated Security]$ ./exectime 10000
Generating Randoms...
1 in 10000 match

# Starting bit index search...
Time micro= 344
Time nano= 344309

# Starting boolean index search...
Time micro= 12229
Time nano= 12229222
```

Figure 22: Bitwise vs Boolean Indexing Comparison (No Multi-threading)

B BitIndex Type Decleration

```
#include <functional>
#include <array>

template <typename _type, unsigned int _bitsize>
struct BitIndex {
    typedef _type                value_type;
    typedef value_type&          reference;
    typedef unsigned int         size_type;

    static constexpr size_type
    array_size = (_bitsize / (sizeof(value_type)*8)) + !!(_bitsize & ((sizeof(value_type)*8)-1));
    std::array<value_type, array_size> data;
    std::function<void(int)> match_found_callback;

    // Capacity
    const unsigned int           block_size = sizeof(value_type)*8;
    constexpr size_type size() const noexcept { return data.size(); }
    constexpr size_type bit_size() const noexcept { return _bitsize; }
    constexpr bool empty() const noexcept { return 0 == size(); }

    // Helpers
    void set_callback(std::function<void(int)> callback_func){
        match_found_callback = callback_func;
    }

    constexpr size_type util_popc(size_type temp){
        // AMD64 Optimization Guide - Popcount
        temp = temp - ((temp >> 1) & 0x55555555);
        temp = (temp & 0x33333333) + ((temp >> 2) & 0x33333333);
        return (((temp + (temp >> 4)) & 0xF0F0F0F) * 0x1010101) >> 24;
    }

    constexpr size_type util_popcto(size_type& temp, size_type& idx){
        const int bmi = idx >> 5;        // max data index
              int count = 0;             // number of set bits
        for (int i = 0; i < bmi; ++i){
            count += util_popc(data[i]);
        }
        return count + util_popc(data[bmi] & ((1<<idx)-1));
    }

    constexpr size_type popcto(size_type& idx){
        return util_popcto(data, idx);
    }

    // Getters
    constexpr size_type get_idx(size_type idx) {
        return data[idx/(8*sizeof(value_type))] & 1 << (idx & block_size -1);
    }

    constexpr size_type get_all() {
        // must have set a lambda or function reference as callback_func
        for (int i=0; i<data.size(); ++i){
            for(int j=0; data[i] && j<block_size; ++j){
                if (data[i] & 1 << (j &  block_size -1)){
                    match_found_callback((i*block_size)+j);
                }
            }
        }
    }

    // Setters
    constexpr void set_idx(size_type idx) {
        data[idx/block_size] |= 1 << (idx & (block_size -1));
    }

    constexpr void set_all(size_type idx) {
        data[idx/block_size] |= ((1 << (block_size - 1)) | ~(1 << (block_size - 1)));
    }

    constexpr void unset_idx(size_type idx) {
        if (data[idx/block_size] & 1 << (idx & block_size-1))
            data[idx/block_size] ^= 1 << (idx & block_size-1);
    }

    constexpr void unset_all(size_type idx) {
        data[idx/block_size] &= 0;
    }
};
```

Listing 10: BitIndex.cpp

C Texture Object Template Function

```
template<typename _type>
cudaTextureObject_t * createTextureObject(_type array1D[], size_t array1D_size){
    cudaTextureObject_t * tex_p = new cudaTextureObject_t();
    struct cudaResourceDesc resDesc = {};
    resDesc.resType = cudaResourceTypeLinear;
    resDesc.res.linear.devPtr = array1D;
    resDesc.res.linear.sizeInBytes = array1D_size;
    resDesc.res.linear.desc = cudaCreateChannelDesc<_type >();
    // Create texture description
    struct cudaTextureDesc texDesc = {};
    texDesc.readMode = cudaReadModeElementType;

    cudaCreateTextureObject(tex_p, &resDesc, &texDesc, NULL);
    return tex_p;
}
```

Listing 11: TextureTemplate.cu

D Fixed Count Variable Length Results

Pattern Length	GLOBAL	SHARED	TEXTURE
10	9.42435	6.60067	9.73495
20	9.44196	3.76957	9.71931
30	9.41899	2.53527	9.72632
40	9.43964	1.81093	9.72426
50	9.41622	1.47842	9.72465

Table 4: Fixed Count Variable Length - 25xN Trie vs 1GB Random Data (Unoptimised)

Pattern Length	GLOBAL	SHARED	TEXTURE
10	54.9064	41.75	29.2316
20	55.3337	21.8267	29.3134
30	55.4667	11.8125	29.4745
40	56.1452	7.014	29.8152
50	56.1676	5.9999	30.1763

Table 5: Fixed Count Variable Length - 25xN Trie vs 1GB Random Data (Optimised)

E Variable Count Fixed Length Results

Pattern Count	GLOBAL	SHARED	TEXTURE
25	9.40249	6.65085	9.60766
50	9.38718	4.94591	8.589
100	9.27025	3.85691	7.84752
200	9.3885	2.00948	6.31987

Table 6: Variable Count Fixed Length - Nx6 Trie vs 1GB Random Data (Unoptimised)

Pattern Count	GLOBAL	SHARED	TEXTURE
25	56.3984	52.3513	30.1517
50	57.8171	39.0266	26.1455
100	58.629	25.0288	22.0038
200	54.6009	10.7612	17.3438

Table 7: Variable Count Fixed Length - Nx6 Trie vs 1GB Random Data (Optimised)

F Variable File Size Results

File Size	GLOBAL	SHARED	TEXTURE
1MB	8.21102	3.08955	6.62684
10MB	8.49311	3.76471	7.66153
100MB	9.37084	3.7784	7.58254
1000MB	9.19066	3.72071	7.39829

Table 8: Variable File Size - 100x6 Trie vs N MB Random Data (Unoptimised)

File Size	GLOBAL	SHARED	TEXTURE
1MB	44.4311	22.6206	6.78204
10MB	55.0186	24.1451	15.2671
100MB	57.0328	24.2733	20.1553
1000MB	58.629	25.0288	22.0038

Table 9: Variable File Size - 100x6 Trie vs N MB Random Data (Optimised)