CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Network Intrusion Detection in Embedded/IoT Devices using GPGPU

Increasing throughput while reducing power consumption with an integrated GPU

Master's thesis in Computer Systems and Networks

SIMON KINDSTRÖM

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2018

Master's thesis 2018

Network Intrusion Detection in Embedded/IoT Devices using GPGPU

Increasing throughput while reducing power consumption with an integrated GPU

SIMON KINDSTRÖM

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2018

Network Intrusion Detection in Embedded/IoT Devices using GPGPU
Increasing throughput while reducing power consumption with an integrated GPU
SIMON KINDSTRÖM

Supervisor: Magnus Almgren, Department of Computer Science and Engineering
Supervisor: Charalampos Stylianopoulos, Department of Computer Science and Engineering
Examiner: Marina Papatriantafilou, Department of Computer Science and Engineering

Master's Thesis 2018
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2018

Network Intrusion Detection in Embedded/IoT Devices using GPGPU
Increasing throughput while reducing power consumption with an integrated GPU
SIMON KINDSTRÖM
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Internet of Things (IoT) devices are low-powered and network connected embedded computers that collect sensor data and perform computations at the edge of a network. These Internet-connected devices often lack sufficient security, with the Mirai botnet being the most highlighted incident to date. To detect attacks, a Network Intrusion Detection System (NIDS) may be used. Intrusion detection is often performed with the costly method of pattern matching, where predefined patterns are matched against observed network traffic, requiring up to 70% of a NIDS's computational power.

This thesis evaluates the suitability of using an embedded device with an integrated GPU as an NIDS. Direct Filter Classification, a state of the art pattern matching algorithm, is improved by moving part of the execution to a GPU. This implementation is then optimized, keeping the quirks of embedded systems in mind. Surprisingly, some optimizations that would intuitively result in an improved execution time, instead increases it. Further attempts at optimizations are performed in the heterogeneous design domain where the CPU and GPU cooperate extensively.

Evaluation is performed by comparing the throughput of network traffic possible to analyze per second, and energy consumption of the algorithm in its different forms: CPU-only, GPU-only and a heterogeneous variant. These are later compared to another state of the art pattern matching algorithm.

By utilizing a GPU, the throughput was increased by more than $2\times$ while reducing the total energy consumption by more than 50%, compared to a CPU-only variant of DFC. The GPGPU variant of DFC was able to improve the throughput of the widely used pattern matching algorithm Aho-Corasick by more than 50% while only requiring 50% of the energy.

Keywords: Pattern matching, NIDS, Network Intrusion Detection System, IoT, GPU, GPGPU, OpenCL, Heterogeneous design

Acknowledgements

I would like to acknowledge the support I have received by thanking my two supervisors: Charalampos Stylianopoulos and Magnus Almgren. Further thanks are extended to Sam Halali, Fredrik Rahn and Margot Brunet for their support and insights. Also, thank you to my parents, Heléne and Sören Kindström, for supporting me on my journey thus far. The final gratitude is extended to my grandma, Birgitta Skeppstedt, for her delicious half past four cinnamon buns.

Simon Kindström, Gothenburg, June 2018

Contents

Acronyms

AC Aho-Corasick.

APU Accelerated Processing Unit.

CPU Central Processing Unit.

DDoS Distributed Denial of Service.

DF Direct Filter.

DFC Direct Filter Classification.

FFBF Feed-Forward Bloom Filter.

FSA Finite State Automata.

GPGPU General-Purpose computing on Graphics Processing Units.

GPU Graphics Processing Unit.

IoT Internet of Things.

NIDS Network Intrusion Detection System.

OpenCL Open Computing Language.

PFAC Parallel Failureless-AC.

RAM Random Access Memory.

SIMD Single Instruction, Multiple Data.

SIMT Single Instruction, Multiple Threads.

List of Figures

List of Tables

1

Introduction

This chapter introduces an existing problem to grant the reader insight in why this thesis holds importance. It then presents the goal of the thesis and clarifies the motivation behind the goals. The chapter later limits the scope of the thesis and concludes by describing the structure of this report.

1.1 Problem background

Internet of Things (IoT) devices are also called *smart* devices. IoT devices are considered smart because they are connected to the Internet and often perform mundane tasks, but with some added functionality and remote control. Some commercial examples are a smart lock for home owners allowing remote unlock and alerts when certain key codes are used [3], and a smart thermostat that tracks your location through your phone allowing the heat to be turned off when the user is not home [4].

IoT devices are also used in factories where they monitor production lines [5, 6]. The many machines in the factory are monitored by IoT devices, which collect data about the machines' usage. Solutions by companies such as Microsoft and IBM offer promises of optimizing equipment performance, accurate views into product quality and increased efficiency by insights from data analytics [5, 6]. By utilizing analytics early in the process, machines can be serviced before they break and product quality can be tracked and kept consistent.

Internet connected deploy-and-forget devices have proven themselves to be harmful. The most notable example is the Mirai worm that infected enough devices to build the world's (at the time) largest botnet [7]. The Mirai worm spread during 2016 by taking advantage of the weak default credentials of IoT devices, by simply connecting remotely using common credentials such as *admin* and *password*. Once many devices had been infected, multiple Distributed Denial of Service (DDoS) were launched. Notable targets of the DDoS attacks include Twitter, Netflix, Reddit and many others. The author of Mirai later published the source code online, presumably to evade identification. This caused many variants of the worm to continue plaguing the internet, one of them causing 900.000 home routers to be put out of service [8].

Malware such as Mirai are prevalent because many IoT devices have no way of being updated. If they do possess the possibility of being updated, it is often a complicated and fragile process. One example is when an over-the-air update to a smart lock rendered 500 people no longer able to unlock their door through their device [9].

A Network Intrusion Detection System (NIDS) is a common way of protecting network connected systems. an NIDS analyzes all network traffic and may send alerts if it detects malicious network traffic. Some NIDS use *pattern matching* between a predefined corpora of patterns against observed network traffic to determine if a network packet is malicious or not [10]. Pattern matching is computationally expensive, accounting for more than 70% of the load in an NIDS [11]. With an increase in network traffic, the importance of efficient pattern matching also increases.

Graphics Processing Units (GPUs) trade processing power through higher core and thread count with simplified control logic and less memory per core. The tradeoff allows GPUs to execute in a highly parallel fashion.

The increased parallelism offered by GPUs has been used for pattern matching, where they allowed the analysis of gigabits of network traffic per second [12, 13, 14]. GPUs have also shown to decrease both execution time and energy consumption when performing pattern matching in embedded systems [15, 16].

1.2 Goals

This thesis will evaluate what effects a cache-efficient, vectorizable, pattern matching algorithm has upon the suitability of using an IoT device with a GPU as a Network Intrusion Detection System. The impact of optimizations and how they differ between embedded systems and their more powerful counterparts will additionally be looked at. Lastly, the effects of heterogeneous computing through cooperation between CPU and GPU will be shown.

To evaluate the suitability, the throughput of network traffic that is possible to analyze per second and the required energy consumption will be compared to another state of the art algorithm.

1.3 Motivation

The lack of security and issues with updates in IoT requires a solution. To help secure these devices, one could deploy a Network Intrusion Detection System (NIDS). The defacto intrusion detection systems, such as Snort [17], are less mobile and energy efficient than the common IoT device, making the deployment of an NIDS unwieldy. Instead of a discrete server performing intrusion detection, an IoT device could perform it. It could either be a discrete IoT device deployed in the network, solely for the purpose of being an NIDS or the NIDS could be introduced on a device already part of the network with computational power to spare.

1.4 Limitations and Scope

Some limitations are necessary to reduce the scope of the project. There are numerous embedded devices being manufactured, not all equally fitted for the suggested workload. In this thesis, the *ODROID-XU3* [18] will be used when performing experiments. The XU3 is a reasonable choice as it consists of an ARM processor (commonly used in embedded devices) and a GPU supporting general purpose computing. It also has a high speed Ethernet interface, making it viable for inspecting network traffic.

In this thesis, only the detection phase of an NIDS will be considered, using pattern matching. an NIDS has many phases, most notably packet acquisition followed by a detection phase. Packet acquisition is also important for the performance in an NIDS [13, 14] but is not equally important in embedded systems where a CPU and GPU share memory, reducing the need for costly copies and memory transfers [19]. Therefore it is reasonable to focus only on the performance and energy consumption of the pattern matching phase in this thesis.

Regular expressions will not be considered, instead only fixed string matching will be looked at. Regular expressions increase the complexity, and as this thesis is more interested in the impact GPGPU has, there is no need to increase the complexity needlessly.

1.5 Report structure

After this introductory chapter, Chapter 2 starts by giving the reader an introduction to the required background information including NIDS and pattern matching, among others. Related works are then presented, to give the reader a better understanding of recent advances within the field, found in Chapter 3. Once all preliminary information has been granted, Chapter 4 introduces the design of the algorithm implemented in this thesis and all modifications and optimizations are explained. How experiments are performed and evaluated, and their results are then presented in Chapter 5. A discussion regarding the results is held in Chapter 6 and Chapter 7 then presents brief concluding remarks.

2

Background

This thesis touches upon multiple topics: Internet of Things, Network Intrusion Detection Systems, pattern matching and GPU-programming. The purpose of this chapter is to familiarize the reader with these topics. To give the reader a better understanding of the environment the thesis relates to, the chapter starts with introducing IoT. The purpose of a Network Intrusion Detection Systems, how it works and a defacto NIDS application is presented. Pattern matching, the core of an NIDS, is then explained. The chapter finishes by describing how GPU programming is performed using OpenCL.

2.1 Internet of Things

As described in Section 1.1, the Internet of Things are Internet-connected embedded devices. These devices often add remote control to previously mundane tasks in homes, such as door locks [3] and thermostats [4]. IoT devices are also being deployed in factories, where the offerings promise increased efficiency and quality tracking [5, 6].

The amount of Internet of Things devices are increasing at a rapid rate, and are forecasted to reach 29 billion devices by the year 2022 [20]. In 2016 there existed 0.4 billion IoT devices, resulting in 72.5 $\times$ more IoT devices in merely six years. The increase is forecasted to mainly come from new use cases for IoT devices, which fits well with the idea that IoT devices are previously dumb devices turned smart. The rapid increase of smart devices furthers the importance of finding ways of keeping them secure.

2.2 Network Intrusion Detection Systems

A Network Intrusion Detection System (NIDS) is used to detect network infiltrations and attacks in a network. This section will introduce the reader to the two approaches to network intrusion detection, *signature detection* and *anomaly detection* [10]. Signature detection uses pattern matching for detection, and is the detection method used in this thesis. To give the reader a better understanding of what features an NIDS offers, the popular open source NIDS Snort [17] will be briefly introduced.

2.2.1 Detection techniques

Signature detection uses pattern matching to match a set of patterns against incoming network packets [10]. Signature detection have two different modes: *white-listing* and *black-listing* [21].

White-listing is when an administrator defines what network traffic is allowed, based on packet content, IP address, and others. Any network packet that does not match at least one pattern in the white-list is blocked. Such configurations are prone to false positives as an application may edit how their network packets look at a whim, but will be able to stop most new attacks. Considering the many applications used inside a modern network, white-listing is only feasible for highly specific networks. Such a network could for example be the interior network of a car, where the manufacturer would be able to know exactly what messages are supposed to be sent.

Black-listing is when a user defines patterns to be blocked [21]. Any network packet that matches a pattern in the black-list will be blocked. Black-listing has fewer false-positives, but has an increased risk for false-negatives. Fewer false-positives is important for the system administrators to not see an NIDS as the boy who cried wolf. Black-listing has little to no effect without an extensive corpora of malicious patterns. That is why popular NIDS applications include many patterns by default, exemplified with Snort [17].

In contrast to signature detection, anomaly detection does not use a predefined set of allowed (or denied) patterns. Instead it *learns* what the common behavior of the system is [10]. During operation the NIDS would then flag any behavior outside of the norm. Anomaly detection may be performed using statistics, machine learning and other similar techniques.

Anomaly detection is good at learning a common network traffic pattern such as an administrator logging into a remote system. However, learning uncommon but benign patterns is much harder. Another issue would be if the network is under attack during learning, where the attack pattern would then be deemed the norm.

2.2.2 Application Example: Snort

Snort [17] is an open source Network Intrusion Detection System, using signature based detection. It uses a custom language for defining rules, as can be seen in Listing 2.1. The rule will send an alert with the message *We're under attack* when all the following properties are true:

- A message is received from any IP, from the port 1025
- The message destination is the IP 12.13.14.10 at port 8096
- The message contains the content *You're under attack*

The example in Listing 2.1 raises an alert, while other actions include logging and dropping the offending packet [22]. In addition to such simple string patterns, Snort allows for many other rule configurations. Such configurations include regular expressions, decryption of SSL/TLS connections, base64 decoding and others [22].

```
alert tcp any 1025 -> 12.13.14.10 8096 \
(msg: "We're under attack!"; content: "You're under attack!")
```

Listing 2.1: A simple Snort rule. When a message with a source port of 1025 is sent to the 12.13.14.10:8096, and contains the sentence "You're under attack", an alarm with the message "We're under attack" will be raised

2.3 Pattern matching

Pattern matching is the process of determining the position of one or more *patterns* in an input text. There are many applications for pattern matching, many of them related to computer security. Antivirus, firewalls, network intrusion detection, all of these may use pattern matching to detect malicious activity.

There are two major classes of pattern matching algorithms: single pattern matching and multiple pattern matching [23]. As the name suggests, the former matches only a single pattern at a time, while the latter matches multiple at a time. With single string matching, the input must be traversed once per pattern. This may be done in sublinear time, not having to compare each character of the input [23]. However, with a large corpus of patterns, the overhead of traversing the input many times may become unreasonable. For that reason, multiple pattern matching algorithms are used in applications when the amount of patterns are large, such as Network Intrusion Detection Systems.

2.3.1 Single string matching

Single string matching matches a single string against an input. The Boyer-Moore [23] is a common single string matching algorithm. It provides sublinear time complexity through three key observation.

The first out of three observations is that comparisons can be performed starting at the end of the pattern instead of at the beginning. This observation enables large time savings when the input and the pattern share a long common prefix. An example of this is when a pattern shares a common path but different parameters, as can be seen in Listing 2.2. If one would start searching from the beginning of the string, 24 characters would have to be compared before a mismatch occurred. In this biased example, only a single comparison would have to be made at position 25 before noticing that they are different.

```
/cgi-bin/admin?username=admin
/cgi-bin/admin?username=user
```

Listing 2.2: Two strings with a common path but different parameters

The second observation is called the *bad character heuristic*. This observation notes that if there is a mismatch, it is possible to skip to the next matching character in the pattern. If a mismatch occurs at position i in the pattern P, and at position j in the input I, then it is possible to skip until $P_k = I_j$ where k is the first position of I_j in P. If there is no k where $P_k = I_j$, the entire length of the pattern may be skipped. An example of the bad character heuristic may be found in Figure 2.1.

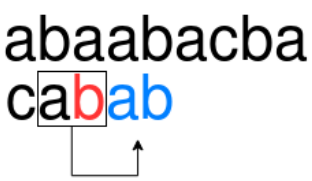
(a) When a mismatch occurs move to the next character

(b) Result after bad character hop

Figure 2.1: Example of the bad character heuristic. As the **e** of the pattern does not match the **a** of the input, hop to the next **a** of the pattern

The final observation is called the *good suffix heuristic*. The good suffix heuristic states that when there is a mismatch between a pattern and the input, there should be some suffix in the pattern that has already matched the input. The heuristic notes that one may then jump to the next occurrence of that suffix in the pattern. An example of the good suffix heuristic may be found in Figure 2.2.

As previously mentioned, single string matching is not fit for applications where a large amount of patterns are needed, such as Network Intrusion Detection Systems. Single string matching is included for completeness of the description of pattern matching techniques. The observations mentioned here have also been used to improve multiple string matching algorithms [24].



(a) When a mismatch occurs move to the next equal suffix

(b) Result after good suffix hop

Figure 2.2: Example of the good suffix heuristic. As the **b** in the pattern does not match the **a** of the input, hop to the suffix equal to whatever was already matched (**ab**)

2.3.2 Multiple string matching

Multiple string matching, in contrast to single string matching, is able to match multiple patterns with only a single iteration over the input. This is a valuable property when there are many patterns and a long input. There are many ways of performing multiple pattern matching [25, 24, 26, 27, 28, 29]. One of the most common algorithms is Aho-Corasick (AC) [25], and a variant of AC is also used in Snort.

Aho-Corasick has a preprocessing stage where it builds a Finite State Automata (FSA), in other words a state machine, with all patterns. However the FSA differs from a normal FSA as *failure transitions* are also added. These failure transitions occur when there is a mismatch between the input and the state machine, and points to the state sharing the longest common prefix to current state. During processing, the input is traversed one character at a time and a corresponding state transition occurs. If there is no common prefix, the state machine starts over. An example of how a state machine for AC might look may be found in Figure 2.3. The arrows between each branch is a failure transition.

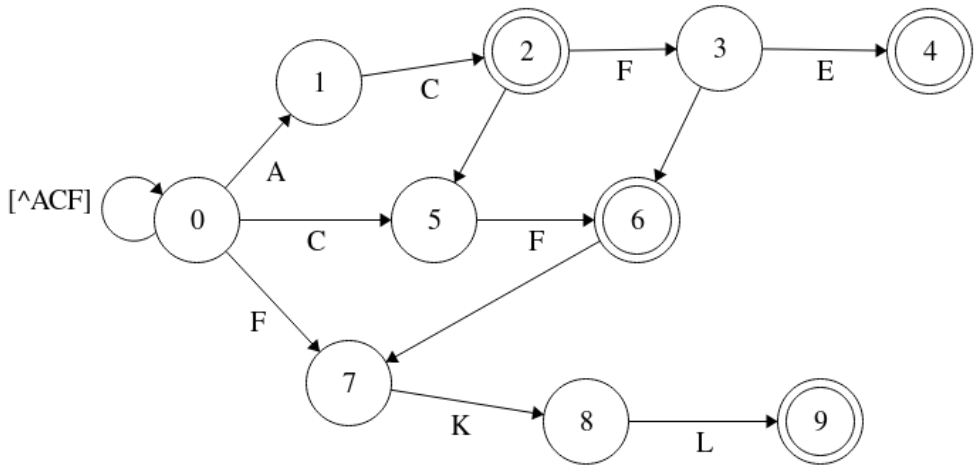


Figure 2.3: Aho-Corasick state machine for the patterns AC, ACFE, CF, FKL

Aho-Corasick is a simple and efficient way of performing multiple pattern matching. However, storing all the states and their transitions requires significant memory space [30, 29]. Because of the large memory requirements, many cache misses oc-

cur during state transitions [29]. AC has for that reason been improved since its inception, partly through reduced storage [30].

2.4 Graphics Processing Units in intrusion detection

Graphics Processing Units (GPUs) were initially introduced to handle computations related to displaying pixels on a screen. These computations often have to be performed for each pixel, and are often independent of each other. Such tasks are sometimes referred to as being *embarrassingly parallel* [31]. Embarrassingly parallel tasks are computations that can be converted from a serial computation to many parallel computation with little to no effort.

The importance of GPUs has increased, as GPUs become more powerful and advances in General-Purpose computing on Graphics Processing Units (GPGPU) are made. As the name suggests, GPGPU allows for general purpose programs on GPUs instead of only graphics computations. GPGPU has had a big impact in many areas, such as deep learning [32], weather forecasting [33] and Network Intrusion Detection Systems [34, 12, 35, 19, 14, 13]. GPGPU is often performed through the heterogeneous computing framework Open Computing Language (OpenCL) [36], described in detail in Section 2.4.1.

The GPU in an embedded system compared to a GPU in a PC or a server is different. The differences will be explained in Section 2.4.2.

2.4.1 Programming Graphics Processing Units

There are two popular GPGPU frameworks, CUDA [37] and Open Computing Language (OpenCL) [36]. CUDA is developed by NVIDIA and only works on their GPUs. OpenCL is a *specificiation* by the Khronos Group, allowing for heterogeneous computing on both GPUs and CPUs. As OpenCL is more portable, supporting both NVIDIA, ARM and AMD GPUs, it will be used in this thesis.

This section introduces the concepts of OpenCL for discrete GPUs, and the differences compared to integrated GPUs are discussed in the following section.

In OpenCL, there exists a single *host* that may have multiple (or no) *devices*. An OpenCL compliant CPU or GPU may be used as a device. Within a device there are multiple *compute units* divided into one or more *processing elements*. These concepts are physical, and rarely affects the programmer using OpenCL. Instead the OpenCL programmer is concerned with *work-groups* consisting of *work-items*. Work-items within the same work-group are able to synchronize through barriers and memory fences, but any cross work-group synchronization is not possible.

Each work-item executes a *kernel*. A kernel is simply a function that is executed with

OpenCL. Kernels are written in a language derived from C99, with some restrictions and additional types. Some of these restrictions are that no function pointers, no recursion and no variable length arrays and structures are allowed. Some of the additional data types are vector and image types.

OpenCL has a fine grained memory model. The first layer, separated from the device, is called host memory and is the memory used by a CPU. On the device there are at least three layers of memory: global, local and private. *Global memory* may be read from any work-item, but requires the largest number of cycles to access. Access to the global memory should therefore be minimized. *Local memory* is shared among work-items in the same work-group. Each work-item then has *private memory* which it may access very quickly. However, as the access speed increases the size of the corresponding memory type decreases. While the global memory may be measured in gigabytes, the private memory is measured in bytes. In addition to the mentioned memory hierarchy, there is image memory. Image memory is faster than global memory but has some additional constraints, such as having to represent the memory as a vector during execution. Image memory has been used in pattern matching to store read-only data used in pattern matching, such as the state transition table of Aho-Corasick [1] or a commonly accessed Bloom-filter [29].

Desktop GPUs gather threads (work-items) into groups, sometimes called *warps*, and each thread within the thread group execute in lock-step. When there is conditional code, such as *if this ... then that ...*, some threads have to pause as the others execute their branch. Instead of using conditionals, one may instead use clever tricks as can be seen in Listing 2.3, where bit operations were used to get the absolute value of an integer instead of a more straight-forward branch.

```
__kernel
void absolute_diverging(__global int *input,
                        __global int *absolute_value) {
  if (*input < 0) {
    *absolute_value = -1 * input;
  } else {
    *absolute_value = input;
  }
}

__kernel
void absolute(__global int *input,
              __global int *absolute_value) {
  int const mask = *input >> 32 - 1;

  *absolute_value = (*input + mask) ^ mask;
}
```

Listing 2.3: Two OpenCL kernels for calculating absolute values. The first one results in divergent execution, as it contains a conditional execution. The second one does not, as it instead uses bit operations

2.4.2 Integrated Graphics Processing Units

Section 2.4.1 presents OpenCL concepts from the point of view of a discrete desktop or server GPU. However, embedded systems often have an integrated GPU, where the CPU and GPU share memory. This architectural change may lead to several differences when compared to discrete GPUs, most notably in the memory architecture.

With a discrete GPUs, the CPU does not share memory with the GPUs. However, an integrated GPU share memory between the host (CPU) and the device (GPU). The shared memory reduces the need for memory transfers, which is often the bottleneck of GPU computations instead of the actual processing power [19]. However, it also means that the device will share the slower memory of the host and memory congestion might occur. Furthermore, accessing off-chip memory, such as global memory, is a costly operation in terms of energy [38, 39, 40], increasing the importance of reducing memory accesses e.g. through high utilization of the available caches.

Integrated GPUs may not have local or private memory. What OpenCL believes to be local and private memory are instead stored in global memory, with the same high access cost as global memory accesses has [41]. The lack of memory hierarchy removes many potential optimizations related to storing frequently accessed data in local memory and influences the design decisions explained in Chapter 4.

2.5 Heterogeneous design

Merriam Webster's dictionary defines *heterogenity* as 'the quality or state of consisting of dissimilar or diverse elements' [42]. The two important words to note here are *dissimilar* and *diverse*. In this thesis, and in many other works [43, 44, 45], the diverse elements are a CPU and GPU. Further mentions of heterogeneous design in this thesis refers to the cooperation between a CPU and GPU.

Heterogeneous design has been the topic of some research with a focus on embedded systems. Utilizing a heterogeneous design has been shown to reduce execution time [43, 44, 45] and energy consumption [43, 44]. The most notable heterogenous design implementation, a heterogeneous implementation of a computer vision algorithm, resulted in a speedup of nearly 12× with energy savings of 3.25× compared to a CPU-only implementation. The same implementation compared with a GPU-only implementation showed a slight speedup but with a 22% energy reduction.

Having multiple phases is crucial in a heterogeneous design, as otherwise the work can not be distributed among the devices. Both Huang et al. [43] and Rister et al. [44] implemented the computer vision algorithm SIFT, and divided SIFT into eight and five stages respectively. These many stages allow for many opportunities at which the control may be given to either CPU and GPU, whichever is the best suited for the task at hand. By measuring the execution time of each stage when

executed on a CPU and a GPU respectively, they could make an informed decision about which stages should be executed on what computing device. It is worth noting is that a stage may become faster when executed on a GPU while still requiring more power [43].

Only one of the previously mentioned heterogeneous designs implemented a pattern matching algorithm, and in that work the heterogeneous design was simply used to improve the *preprocessing* phase of a pattern matching algorithm [45]. In this work, the heterogeneity will be utilized in the main execution path of the pattern matching.

3

Related work

This chapter brings forth related work within the field of pattern matching and its applications. Each work will be shortly described and related to the work in this thesis. The chapter starts by presenting recent advances in pattern matching using cache-efficient data structures, later followed by advancements in pattern matching using GPGPU. The chapter concludes by discussing recent applications of pattern matching and relates it to IoT.

3.1 Cache-efficient pattern matching

The pattern matching algorithms presented in Section 2.3, Boyer-Moore [23] and Aho-Corasick [25], use data structures with unpredictable memory access patterns, causing frequent cache misses and stalls. IoT devices are resource constrained, further increasing the importance of using a resource-friendly pattern matching algorithm. Two recent memory-efficient pattern matching algorithms are Feed-Forward Bloom Filter (FFBF) [29] and Direct Filter Classification (DFC) [28]. Both of these algorithms have an initial approximate matching phase that discards most data and patterns, and does so efficiently. This initial phase uses a cache-efficient data structure for quick lookups. This thesis implements a resource friendly algorithm to gain as much performance as possible from a resource constrained IoT device.

3.1.1 Feed-Forward Bloom Filter

Feed-Forward Bloom Filter, like the original Bloom filter [46], creates a bit vector indexed by more than one hash function [29]. If the bit referenced by an index equals 0, the input is guaranteed to not exist in the original pattern corpus. If the position referenced by an index equals 1, the input *may* exist in the pattern corpus. As such, accuracy is exchanged for space, where FFBF is able to reduce the memory consumption by 50 $\times$ compared to GNU grep which uses an improved version of Aho-Corasick [24].

Bloom filters have an inherent false positive rate. To remove the error rate of the Bloom filter, FFBF uses a multi-stage approach to pattern matching. In the first stage, designed to be highly efficient, a sliding window rolls over the input and checks if a bit in the bit vector, indexed by a hash of the input, is equal to 1. If it

is, the input is saved for later processing, otherwise it is discarded. The next stage fetches the full patterns that matched in the previous stage. In the final, and most computationally expensive, stage exact matching is performed between the matched input from stage one and the full patterns from stage 2, using any exact pattern matching algorithm e.g. Aho-Corasick [25] as described in Section 2.3.2. Such multi stage pattern matching results in great improvements if most input gets filtered in the first stage.

There are two major drawbacks with Feed-Forward Bloom Filters. The first drawback is that FFBF require all patterns to be of the same length, where any shorter patterns must be matched separately. The second drawback is that computationally expensive hash computations are required for each sliding window [29].

3.1.2 Direct Filter Classification

DFC is a memory and cache efficient pattern matching algorithm, using Direct Filters (DFs) [28]. A Direct Filter is a bitmap that summarizes some consecutive bytes from the pattern, and are small enough to be cache resident. A 2 byte DF will use 2 consecutive bytes from a pattern, e.g. the first or last two bytes, to index a bit in the bitmap.

Like FFBF, DFC performs matching in multiple phases: *filtering* and *verification*. In the filtering phase, a window of two bytes is slid over the input, summarized and matched to the filter. If the window matches, additional DFs requiring more bytes for indexing may be used to check that it really is a match. The verification phase performs exact matching using a compact hash table with efficient indexing.

DFC does not have the same constraints that FFBF does. First of all, it may match patterns of any length. This is crucial for an NIDS, as one may otherwise not be able to detect some attacks. Secondly, 2.5 $\times$ fewer instructions are required for the indexing used in DFC compared to the indexing by hash functions used in FFBF [28]. The reduced instruction count results in a 2.4 $\times$ overall speedup [28].

GPUs offer similar vectorization as CPUs, but with better memory latency hiding, potentially allowing even greater speedups on GPUs [2]. Since its inception, DFC's throughput has been increased through CPU vector instructions [2]. The vectorized version of DFC increased the throughput with 3.6 $\times$ compared to the original DFC algorithm. Vectorization is enabled by separating the filtering and verification phase of DFC. Instead of instantly verifying any window that got a hit in the DFs, matches during the filtering phase are stored for later verification. Such distinct phases also allow for better cache locality during each phase.

3.2 Pattern matching with Graphics Processing Units

In this thesis a GPU on an embedded system is utilized to both improve the throughput and decrease the energy consumption compared to a CPU-only implementation. However, GPUs offer performance improvements only when the many cores and threads of the GPU are utilized. Therefore one must carefully design an algorithm to be run on a GPU. Naive pattern matching is usually a sequential operation, comparing one character at a time to one or more patterns. As character comparisons have to be performed even in more advanced pattern matching algorithms such as Aho-Corasick [25], how to fully utilize the parallelism of GPUs is not obvious. What, if any, considerations have to be made when performing pattern matching on embedded systems will be presented later in the section.

3.2.1 Work segmentation

Feed-Forward Bloom Filter (FFBF) [29], described in Section 3.1.1, perform matching on GPUs by letting each thread hash a n-gram of characters and then match it to the Bloom-filter in the matching phase. A similar procedure is done in the pattern filtering phase of FFBF, where each thread filters the patterns matching a single n-gram. This results in each byte of the input, on average, being processed n times.

Parallel Failureless-AC (PFAC) is an implementation of Aho-Corasick with focus on GPUs [1]. PFAC improves upon the previous designs by assigning each thread a unique starting point. Previous GPU-implementations of AC assigned a fixed amount of characters equal to the length of the longest pattern per GPU-thread, simplifying the division of work, as seen in Figure 3.1. In PFAC, each thread will traverse the input starting from its starting point, until no match is found, and then immediately terminate. PFAC removes any failure transitions and self-loop transitions to simplify the state machine, also because these states are no longer needed. While there are many threads spun up, most will quickly terminate due to the lack of a next state. Figure 3.2 illustrates how PFAC assigns each input character its own thread and state automata.

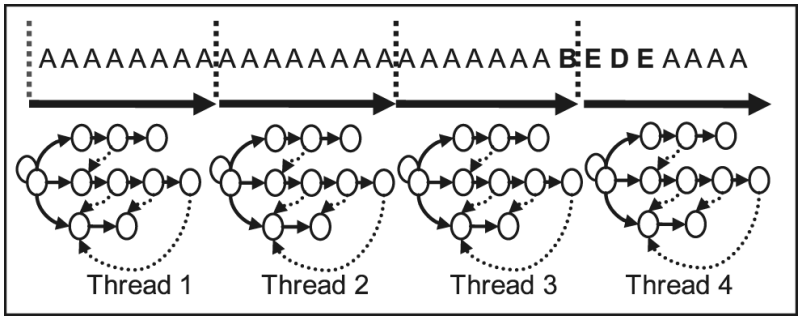


Figure 3.1: A naive work segmentation among threads *as seen in [1] © 2013 IEEE*

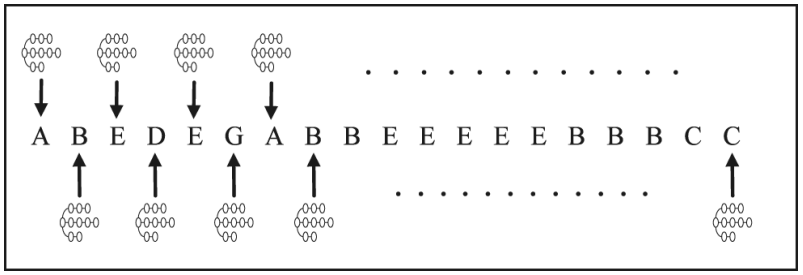


Figure 3.2: Work segmentation in PFAC *as seen in [1] © 2013 IEEE*

3.2.2 General-Purpose computing on Graphics Processing Units in embedded systems

GPUs have been shown to decrease energy consumption in embedded systems [15]. The improvements may not be gained for all applications, but rather for tasks well-suited for the parallelization that GPUs offer. Pattern matching is one such application, where energy consumption was $3\times$ larger when executing on a CPU compared to an embedded GPU [15].

Care has to be taken when performing optimizations aimed to improve the execution time and energy consumption in embedded GPUs [15, 16]. The software optimizations common for more powerful GPUs may instead cause performance degradation. The degradation is caused by the differences in the set of available features as well as hardware differences. Differences include shared memory between the CPU and GPU, and limited or lack of L2 cache, among others [19]. Having shared memory between the two devices is a double-edged sword. It removes the need for transferring data between the CPU and GPU, but results in the GPU using the slower DRAM instead of the GDDR memory used in high-end GPUs [19]. Shared memory also results in contention on the shared memory bus and controller.

3.3 Pattern matching applications

This section will reveal how pattern matching is currently used in some recent works. At first it will briefly discuss how GPUs have been used in Network Intrusion Detection System and what impact integrated GPUs may have. The section later explores SplitScreen [47], a malware detection tool that utilizes a client-server architecture to enable malware detection on resource constrained devices. While SplitScreen's methods are not in the same direction as explored in this thesis, it is a technique worth mentioning.

3.3.1 Network Intrusion Detection System

Using GPUs for pattern matching in an NIDS is nothing new [34, 35, 14, 13, 19]. More recent solutions are focused on powerful machines using multiple graphics cards [14, 13]. The authors of Kargus [14] show that even if GPUs are extremely powerful, they are not always superior to the CPU. There is an overhead involved

with transferring packets and executing kernels on the GPU, that is more costly than what is gained when packets are small.

Embedded devices often use integrated GPUs. APUNet [19] is a more recent work that utilizes an integrated GPU in contrast to a discrete GPU. Integrated GPUs share memory with the CPU, eliminating the memory transfer overhead. The authors of APUNet show that while the throughput is lower on an Accelerated Processing Unit (APU), the performance-per-dollar is significantly higher than discrete GPUs when used for pattern matching. IoT devices with GPUs often use an integrated GPU, and APUNet shows that these devices might be highly cost-effective. APUNet also includes optimizations that is used in this thesis.

3.3.2 Malware detection

Malware detection is similar to intrusion detection as it also performs pattern matching. Instead of matching network traffic, malware detection matches file content to patterns. By having a client-server architecture, SplitScreen [47] is able to perform malware detection on the iPhone 3GS. SplitScreen uses FFBF, see Section 3.1.1, and a client only stores the bloom filters from FFBF, reducing the memory requirements.

The client in SplitScreen performs pattern matching between its local files and the compressed patterns. In this stage, up to 90% of false-positives files and 99% of signatures are discarded [47]. Once a potential match has been detected, the client requests the full version of the matched pattern from the server, to later use the full pattern for exact matching. As the files never leave the device, the cost of the network transfers is low and confidentiality can be kept.

Utilizing a client-server architecture is an attractive choice as it offloads some of the burden from the client to the server. SplitScreen managed to reduce the size of the pattern corpus that the client holds from 9.9MB to 0.77MB. These techniques allowed the authors to perform malware detection with millions of patterns using an iPhone 3GS. Similar techniques could be used for network intrusion detection on IoT devices as well, trading lower memory requirements for increased latency. While a client-server architecture is not used in this thesis, a similar multi-phase architecture where the initial filtering requires little memory is used, and could utilize a client-server architecture in some future work.

4

Design & Implementation

In this thesis an ODROID-XU3 [18] is used to execute all tests. To be able to motivate some of the optimizations presented in this chapter, the choice of hardware will first be motivated and its specifications will be presented. Once the reader is familiar with the device, the algorithm best suited for the evaluation will be presented, followed by the motivation behind the selection and how the algorithm is implemented. Finally, optimization performed upon the algorithm are introduced.

4.1 Hardware platform

The hardware to be used in this thesis requires a GPU, preferably an integrated GPU as these are common in embedded systems. The GPU must also support general purpose programming.

In this thesis the ODROID-XU3 [18] is used to execute all tests. The XU3 uses the Exynos 5 Octa (5422) [48] chip that has a quad-core ARM Cortex-A15 [49], a quad-core ARM Cortex-A7 [50] and an ARM Mali-T628 GPU [51]. The XU3 also has 2GB of Random Access Memory (RAM). A figure of the XU3's architecture and its ports is shown in Figures 4.1 and 4.2. The Exynos 5422 is used in one variant of the 4 year old Samsung Galaxy S5, showing that the XU3 is a reasonable choice for a more powerful IoT device today. The most important reason as to why the XU3 is used it because it possesses a GPU that allows General-Purpose computing on Graphics Processing Units (GPGPU). Further reasons are that it has a high speed ethernet port, allowing for high speed network sniffing.

The XU3's variant of the GPU is a Mali-T628 MP6 that has 6 cores. These shader cores may be programmed using OpenCL as described in Section 2.4.1. The cores are divided into two separate OpenCL devices, one device with four cores and one device with two cores. The GPU does not have any OpenCL shared or local memory, instead placing these in the global memory that in turn is stored in the RAM shared with the CPU [41]. Each core has L1 and L2 memory caches to remedy the cost of always accessing the global memory. These caches have a 64-byte cache line. There are two 16kB L1 caches for each core, one used for generic memory accesses and one for texture memory.

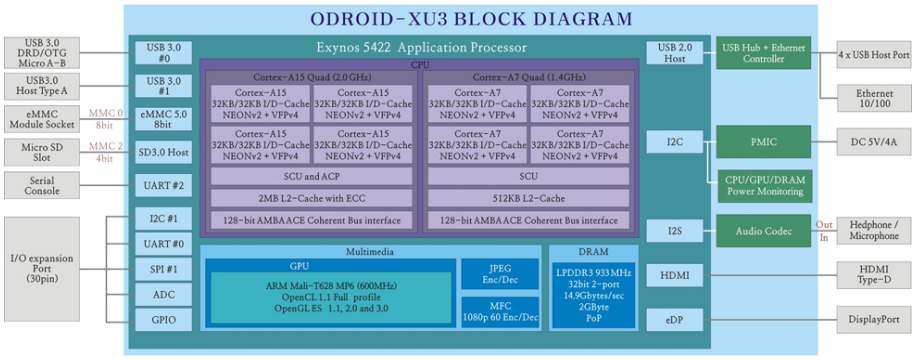


Figure 4.1: Block diagram of the ODROID-XU3 *from hardkernel.com*

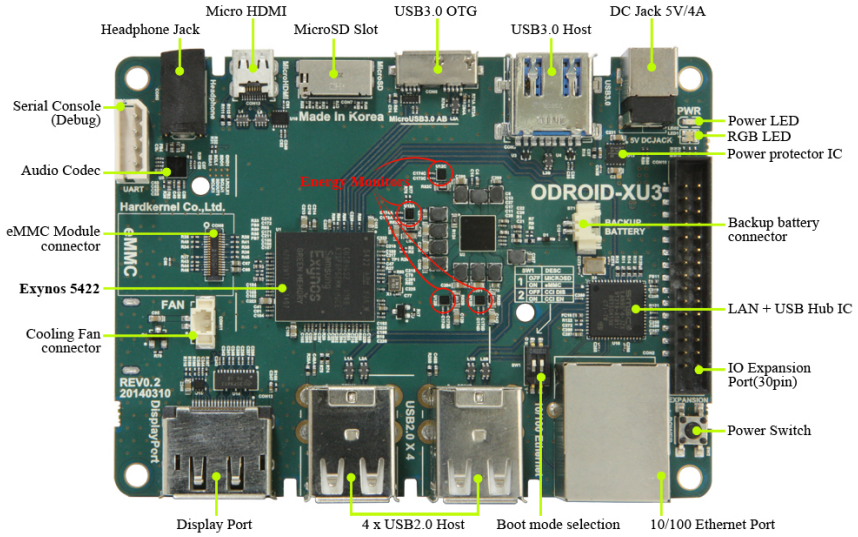


Figure 4.2: Labeled ODROID-XU3 board *from hardkernel.com*

Divergent execution, when the code contains conditional statements, is not a problem with the GPU in XU3 [41]. Desktop GPUs gather threads into groups, sometimes called *warps*. All threads in a warp execute the same instruction in lock-step, meaning that if diverging code occurs only some threads may continue while others are stalled. However, each thread in a Mali GPU has an individual program counter, meaning that diverging code does not stall threads. This is an important property as it allows for more complex kernel code without worries of stalling threads. However it may negatively affect data locality causing more frequent cache misses.

ODROID-XU3 is a relatively powerful embedded device in both memory and processing power, more so when compared to devices used in sensor networks. GPUs are rarely used in today's IoT devices, but are common in smartphones. Even if GPUs are uncommon in IoT devices today, it does not exclude it from being common in future devices if it is shown that utilizing GPGPU may reduce energy consumption and improve execution time.

4.2 The algorithm: Direct Filter Classification

This section describes why Direct Filter Classification (DFC) [28], see Section 3.1.2, was chosen as the most fitting algorithm for the use case and how DFC was implemented and customized.

4.2.1 Motivation

Improving cache hit rate is important in IoT devices, as access to off-chip memory is a costly operation in terms of energy [38, 39, 40]. DFC is a memory and cache efficient pattern matching algorithm, using Direct Filters (DFs) [28]. DFC is a good choice for a pattern matching algorithm in IoT devices because the majority of the pattern matching may be performed with few cache misses, reducing the number of accesses to off-chip memory. That is also the reason why some other GPU optimized algorithms such as PFAC [1] might not be a wise choice, as PFAC has many unpredictable memory accesses. Feed-Forward Bloom Filter (FFBF) [29], described in Section 3.1.1, is another cache friendly pattern matching algorithm, but may only be used for patterns of the same size and requires a costly hash function [28].

An additional reason why DFC was chosen as the most suitable algorithm is that DFC has been shown to be vectorizable on a CPU. GPUs offer similar vectorization as CPUs, but with better memory latency hiding, potentially allowing even greater speedups on GPUs [2]. Furthermore, GPUs in embedded systems have shown to reduce power consumption for pattern matching [16, 15]. Combining these two properties might result in an improved execution time and energy consumption.

4.2.2 Implementation

This thesis uses a different filter setup compared to the one used in the initial DFC publication by Choi et al. [28]. Instead the thesis utilizes the same filter setup used in the CPU vectorized version by Stylianopoulos et al. [2]. What follows is a brief description of that filter design, and is shown in Figure 4.3.

In the altered Direct Filter setup there are three filters. *Filter 1* summarizes patterns of length one, two and three characters and is indexed by two bytes. A filter indexed by two bytes is of size 8kB, which easily fits in the 16kB L1 cache of the XU3's GPU. *Filter 2* is an intermediate filter to match any patterns with a length longer than three characters. Filter 2 allows for an efficient way of checking if the input matches any patterns longer than three characters, by simply using the same index as calculated for filter 1. *Filter 3* however does not use the same indexing technique as the first two filters, but keeps the same size of the DF. Filter 3 uses a multiplicative hash function to index a bit in its bitmap. The use of a hash function allows the use of more characters for filtering without increasing the size of the filter, allowing it to be cache resident. A 3B indexed filter would require 2MB, much larger than any cache available on the XU3, making a filter indexed by 4B impossible without

a hash function.

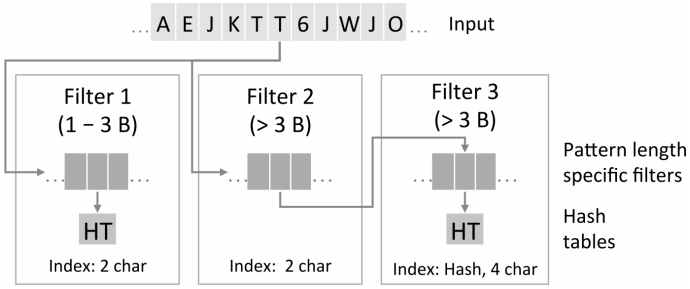


Figure 4.3: The filter design used. HT is the abbreviation of hash table *as seen in [2] © 2017 IEEE*

Due to memory limitations, all network traffic can not be processed at once. Therefore, the input has to be split into *chunks*. However, naively splitting the input may lead to missing some patterns due to part of the pattern existing in the first chunk, and the remainder in the second chunk. To remove any such omissions, each chunk will overlap with the previous chunk by the size of the longest pattern. While such an overlap results in no misses, it may cause extra matches detected by DFC.

What follows are more in-depth details regarding how the GPU is utilized when implementing DFC with GPGPU.

Graphics Processing Unit variant

What in this thesis is called the GPU variant of DFC, is a version of DFC that executes the entire algorithm on the GPU. It works just like the CPU version would, where exact matching is performed right as a hit is found in a DF. The gains to be had from using a GPU in this fashion is simply that it allows the many cores of a GPU to work simultaneously.

The CPU is still required to some degree, even in the GPU variant of DFC. It performs all preprocessing where the DFs and compact tables are set up and transferred to the device. The GPU then perform the matching. Once matches are found, the CPU has to process the matches by calling a user-defined callback function for each match.

There is a problem related to tracking matches on the GPU, to later be processed by the CPU. When a CPU performs matching, the callback is called as soon as a match was found, without having to store any matches. As a GPU can not call a function on the host, nor are function pointers allowed at all in OpenCL, matches have to be stored for later processing. Kernels in OpenCL may not dynamically allocate memory either, instead requiring one large static allocation used to host all potential matches before any matching takes place.

Further issues regarding matches found on the GPU are related to the parallelism in which the matching occurs. One could use a shared variable among all threads to

index a vector containing the results, where the index gets atomically incremented when a match occurs. By doing so, a reasonable upper bound for maximum number of matches may be set, the result vector will be ordered and the CPU will have an exact count of amount of matches. However, using atomic function in OpenCL may lead to drastic slowdowns [52].

Because of the potential problems with atomic operations, the implementation in this thesis allocates enough memory to track matches occurring for each input character, e.g. if there was a match at input location X, the match would be stored in a result vector at index X. However, there may be many matching patterns for a certain location in the input. To ensure reasonable memory constraints, the choice was made to to only store at most two matches per input location, but counting all matches. If more matches occurred than the ones stored, a warning message is printed.

OpenCL does not allow for variable sized arrays, and therefore memory allocated for the text representation of a pattern must be equal for all patterns. This means that every representation of a pattern used in OpenCL require an equal amount of characters to the longest pattern in the data set. Therefore a significant memory overhead exists if there is a single long pattern and many more much smaller patterns. That is why a reasonable upper bound for pattern length must be used. Further restrictions of OpenCL can be found in Section 2.4.1.

Heterogenous variant

To enable a heterogeneous design where the CPU and GPU cooperate extensively, it is important for an algorithm to be performed in more than one stage. If it is not, there is simply no work that may be cooperated upon. More information regarding heterogeneous design may be found in Section 2.5.

The heterogeneous design of DFC in this thesis will perform the filtering on a GPU, to later perform exact matching on a CPU. The reasoning for this is that the highly specified application of filtering is well-fitting for the simplified control logic of a GPU. While branching is a minor problem for the GPU used for the tests in this thesis, they are not in more powerful GPUs. Furthermore, the amount of instructions required for exact matching is significant, causing evictions in the instruction cache.

As the tasks required of the GPU is simplified in the heterogeneous variant, there are fewer issues surfaced by the use of OpenCL. First of, patterns does not have to be stored on the GPU, as no exact matching is performed. Secondly, less memory is required to track the result from the GPU. The reason is that the heterogeneous variant only has to set a bit if a certain location of the input caused a hit in the Direct Filters, requiring only as many bytes as the input. In this thesis, the GPU sets the first bit if a hit was found in the first filter, and sets the second bit if a found in both the second and third filter.

4.3 Optimizations

Simply translating a CPU version of Direct Filter Classification to OpenCL will not result in optimal throughput or energy consumption. Instead it is important to consider both the specifications of the hardware platform and the possibilities of the algorithm. This section introduces optimizations techniques that will be used to try to improve the throughput of both the GPU and heterogeneous variant of DFC.

4.3.1 Reducing memory transfers

Memory transfers between a OpenCL host (CPU) and OpenCL device (GPU) is often the bottleneck in GPGPU applications [19]. This overhead can be reduced by minimizing the data transferred between the two units. There are two ways of reducing memory transfers. The first is to compress the data, and then performing extra computations on the GPU to decompress it [53]. This may cause an overall slowdown because of the additional computations required.

The second way is only available in the case where the host and device shares memory, such as the case of an integrated GPU, which is used in this thesis. In such cases, OpenCL allows for a buffer to be created on a GPU, to be shared with a CPU [19, 16]. This is called *memory mapping*. Memory mapping removes most of the transfer overhead, leaving some work for OpenCL to handle the memory still.

In this thesis, only memory mapping will be used to reduce memory transfers.

4.3.2 Increasing work per thread

In most GPUs, all threads in a *warp* (collection of threads), execute in lockstep. Even if one of the threads in a warp finished executing, the others will continue, and the finished one will idle until all threads are done. In the case of DFC, this is likely to happen if one thread finds a match in a Direct Filter and has to perform exact matching, while the others does not. To relieve this problem, each thread may perform matching on multiple locations.

Another issue with having each thread handle a single character, is that a lot of threads are spawned. This is not a free operation, costing some time and energy. By having each thread perform more work, less threads are needed. By performing more work on each thread, each thread may reuse previously fetched data and gain better cache locality.

The XU3 in this thesis does not have the issue where threads execute in lockstep, as each thread has a separate program counter. However, divergent execution may still disturb the cache, having a negative impact on performance.

A side-effect of having each thread handle multiple input locations is that instead of allocating a buffer to handle the matches found at *each position of the input*,

the buffer can instead be used to track the number of matches found *per thread*. While this does not change the memory requirements, it affects how the CPU later processes the matches. Instead of the CPU having to loop an equal amount of times to the bytes in the input, it may now instead only loop through the amount of GPU threads spawned. Lets consider an example to make the impact clearer. Imagine that each GPU thread handles 8 characters of the input, and only a single match was found. Instead of the CPU having to access 7 memory locations without a match, and one with, the CPU now only has to handle the one match without having to perform any work for the 7 misses as the information for this is now stored in the same location. The heterogeneous version does not utilize this method of storing multiple matches per thread, and instead stores it per location.

4.3.3 Altering OpenCL workgroup size

Another variable is the size of the OpenCL workgroup. Pattern matching has no natural workgroup size, compared to e.g. image down scaling. That is why it makes sense to configure it to whatever works best for the device. Fewer workgroups should result in a lower overhead for maintaining them. The size of a workgroup is usually a maximum of 256, as is the case on the ODROID-XU3.

4.3.4 Utilizing local memory

Local memory is shared between each work-item in a workgroup, and is often used to improve the execution time by reducing the amount of accesses to global memory. The memory model of OpenCL is described in detail in Section 2.4.1. As described in Section 2.4.2, more constrained GPU devices may not have local memory. As such, neither does the XU3 used in this thesis.

In this thesis, a local memory approach is implemented, to showcase how a traditional optimization approach may not be fit for embedded systems. The local memory version of DFC stores all three Direct Filters in local memory, which should be the memory area accessed the most often.

4.3.5 Storing Direct Filters in texture memory

An additional optimization strategy used in this thesis is to utilize the texture memory. As there is a separate L1 cache for textures in the XU3, one may increase the cache hits further by storing one or more DF as a texture. It is worth noting that any access to the texture memory requires vector loads, loading at least 16 bytes at a time. This results in the need for some additional registers and computations to retrieve the one bit of interest, potentially reducing the gain from better cache locality.

In this thesis, the two first Direct Filters will be stored in texture memory. The reason is that each DF is 8kB, fitting nicely in the 16kB L1 cache, theoretically resulting in a 100% cache hit rate.

4.3.6 Vectorized design

Normally a GPU works in lockstep, where multiple threads execute the same instruction. Such a design is sometimes called Single Instruction, Multiple Threads (SIMT). SIMT can be likened to Single Instruction, Multiple Data (SIMD), where instead only a single thread processes multiple data. In both cases, execution is vectorized.

The Mali GPU of the XU3 uses SIMD instructions instead of SIMT. As mentioned before, this allows for branching code without large penalties, while still allowing vectorized execution.

DFC has been improved through the usage of SIMD instructions on a CPU [2]. This version uses two different SIMD instructions: *shuffle* and *gather*. *shuffle* is used to change the order of bytes in a vector register. *gather* fetches data from multiple, non-adjacent locations, with a single instruction.

In the vectorized version of DFC, *shuffle* is used to massage the input into the correct format, on which the bit operations are then performed to calculate the bitmask and the index to the Direct Filters. These indices are used by the *gather* instruction to fetch the relevant bytes of the DF. That means that a *gather* instruction has to be performed for each lookup in a DF, namely two or three times per input location.

Sadly, *gather* is not supported in OpenCL. Instead one is forced to perform a loop, fetching the data one-by-one, without any vectorization. One may try to improve the effectiveness of the data loading through coalescing fetches located in adjacent memory to reduce the amount of fetches and reduce cache trashing [54]. Such an optimizations may lead to performance improvements of up to $7\times$ [54]. However, as the Direct Filters are expected to be cache resident, any such coalescing will have a minor, or no, impact.

The vectorized version of DFC on the GPU used in this thesis is similar to the one on the CPU [2]. As previously discussed, the *gather* instruction is however replaced with a sequence of scalar fetches. In the vectorized version, all filtering is done and temporarily stored before exact matching is performed. This is done in an attempt to reduce the interleaving of vector and scalar instructions, which may remove much of the benefits had from SIMD instructions [2]. Psuedo code of the vectorized version may be seen in Listing 4.1.

```
# input: Data to be matched against
# DfSmall: DF for short patterns
# DfLarge: DF for longer patterns
# THREAD_GRANULARITY: The amount of characters to be
    checked per GPU thread
# matchesSmall[THREAD_GRANULARITY]: temporary array for
    matches in DF for short patterns
# matchesLarge[THREAD_GRANULARITY]: temporary array for
    matches in DF for longer patterns

for i < (THREAD_GRANULARITY / 8); ++i {
  input = vector_load(input, i)
  shuffled = shuffle(input, SHUFFLE_MASK)

  // abstracted away. These are also vectorized
  indices = get_indices(shuffled)
  masks = get_masks(shuffled)

  vectorDfSmall[8];
  vectorDfLarge[8];
  // scalar "gather" instruction
  for (k < 8; ++k) {
    vectorDfSmall[k] = DfSmall[indices[k]]
    vectorDfLarge[k] = DfLarge[indices[k]]
  }

  // vector instructions once more
  matchesSmall[i] = vectorDfSmall & masks
  matchesLarge[i] = vectorDfLarge & masks
}

// exact matching on matchesSmall and matchesLarge using
    scalar instructions
...
```

Listing 4.1: Psuedo code of a GPU vectorized DFC

4.3.7 Overlapping execution

In the original implementation, the CPU will stall until result has been produced by the GPU, process the matches and then start a new round of matching on the GPU. When overlapping execution, the CPU and GPU will instead work simultaneously. Overlapping execution is granted by having one extra buffer for input and result respectively, that may be processed on the CPU while the GPU process the next batch.

The benefit of overlapping execution between the two execution devices is that, in

theory, the total execution time should become the maximum of the CPU's and GPU's execution time, instead of the sum of them.

The downside is that extra memory has to be allocated, and extra bookkeeping is required for OpenCL as there are now multiple commands with different buffers being executed.

5

Evaluation

How the experiments were performed and their results will be presented in this chapter.

5.1 Evaluation method

Evaluating whether the algorithm implemented in this thesis is a good candidate for intrusion detection in IoT, its execution time and energy consumption must be compared to another state of the art pattern matching algorithm. The impact of each optimization method is evaluated through the act of trying to reach the best possible configuration, where each optimization is configured for maximum throughput. Once the best configuration has been found for DFC, it will then be matched against another pattern matching algorithm, namely Aho-Corasick (AC) [25]. When evaluating the algorithms, three different pattern data sets will be matched against four different traffic data sets.

5.1.1 Metrics & Challenges

Execution time, and therefore throughput, can easily be measured and evaluated. Energy consumption is however not as straightforward. Luckily, the ODROID-XU3 has energy sensors measuring voltage, current and energy consumption for the two CPUs, the GPU and the memory. These energy sensors can be seen in Figure 4.2. A deeper understanding of the energy consumption may be realized by separating the algorithm into multiple phases, having a timeout between each phase. Such a division will allow patterns in the current draw to be mapped to each phase.

5.1.2 Other algorithms

DFC [28] will be matched up against Aho-Corasick (AC) [25], which is used by default in the popular NIDS application Snort. In this thesis, a version of AC extracted from the Snort source code is used, and is therefore highly optimized, and is to be executed on a CPU. There are variants of Aho-Corasick with tradeoffs between search speed and memory efficiency [30], but may come at an increased search cost. The experiments will be performed with the full version of AC.

5.1.3 Data sets

Two different type of data sets are required for evaluating the algorithms. These two types are *patterns* to be matched against *network traffic*. To enable a discussion regarding how data affect the execution time and energy consumption of DFC, it is important to test different data sets for both patterns and network traffic.

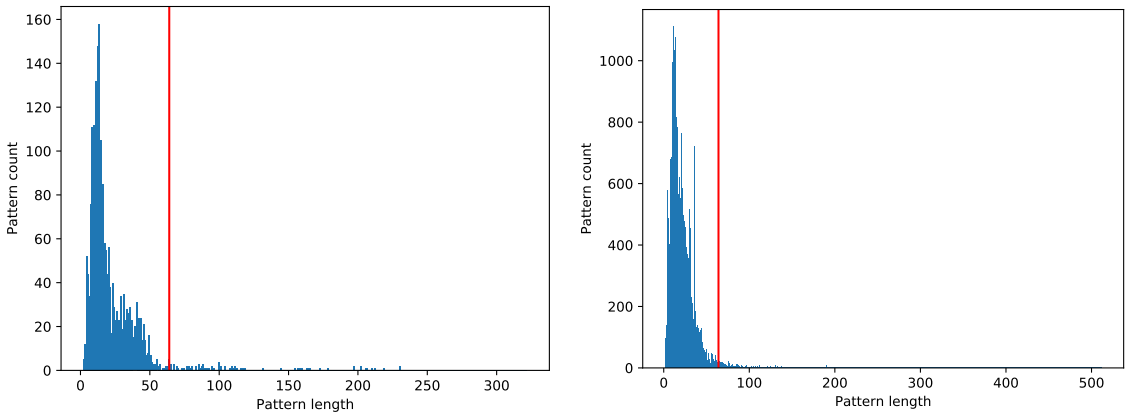
Four *traffic* data sets are used in the experiments. The first data set contains 300MB of network traffic from the DARPA 2000 network capture. There are concerns regarding the suitability of using the DARPA 2000 capture to measure the detection performance of an NIDS [55], but since this thesis is only concerned with throughput and energy consumption, it does not invalidate the data set. The next two are part of the ISCX 2012 data set [56], where approximately 1GB of network traffic is extracted from day 2 and day 6 respectively. The final traffic data set is 1GB of randomly generated data.

Accompanying the traffic data sets are three *pattern* data sets. The first data set is one containing patterns from Snort [17], containing approximately $2,500$ HTTP related patterns. The second data set is downloaded from `emergingthreats.net` and contains $20,000$ patterns. Thirdly, the set of patterns from `emergingthreats.net` will be culled to only contain patterns related to HTTP traffic, leaving approximately $9,500$ patterns. Only filtering on HTTP traffic makes sense because in NIDS systems, like Snort, traffic is grouped by protocol before being matched against only related patterns. As the data sets used contain mostly HTTP traffic, filtering using only HTTP patterns is reasonable and will reflect the usage in a real application.

Finally, any patterns longer than 64 characters are removed. This is due to the restriction in OpenCL where any struct may not contain pointers, instead arrays must have their size defined at compile time. That means that each pattern must allocate enough data to host the largest pattern in a corpus of patterns. In the case of the patterns from `emergingthreats.net`, the longest pattern is 513 characters, while most are much much shorter than that. Removing any pattern longer than 64 characters removes approximately 80 and 500 from the first and second pattern data set respectively. The distribution of pattern lengths in the first and second pattern data set may be seen in Figure 5.1a and 5.1b respectively.

5.1.4 Hardware setup

An ODROID-XU3 will be used for any experiment measuring execution time. The motivation behind this choice and the specifications of the XU3 is described in detail in Section 4.1. In short, an XU3 is a reasonable choice for running experiments because it has an integrated GPU that may be programmed with OpenCL. Furthermore, an ODROID-XU3 allows for easy energy measurements as it has energy sensors for the CPUs, GPU and memory.



(a) Snort HTTP patterns **(b)** All patterns from `emergingthreats.net`

Figure 5.1: Distribution of pattern lengths. Red line signifies 64 characters

5.1.5 Experiments

Many DFC executions are performed to find the optimal configuration for the two variants of DFC. There are many optimization parameters implemented, and exploring all permutations is not possible. The final configuration will therefore be found greedily, where a single configuration parameter is optimized before moving onto the next one.

When exploring the effect of optimizations, the average value from five runs of matching Snort's HTTP related patterns against the DARPA network traffic dump is used.

The initial configuration, before any optimizations were performed, did not map memory, had a thread granularity of one, did not use local or texture memory, did not use vector instructions, had a workgroup size of 128 and a read chunk size of 10MB.

The first optimization performed is *mapping* memory, where the CPU and GPU share memory. Mapping memory should reduce need for memory transfers between CPU and GPU, and therefore decrease the total execution time.

The second parameter to configure is the work per thread, called *thread granularity* in this thesis. An increased granularity increases the work per thread, therefore reducing the total amount of threads. A side-effect of increasing the thread granularity is that the work performed by the CPU reduced, see details in Section 4.3.2.

The third optimization is trying to use a different form of memory other than the normal (global) memory. The two mutually exclusive options are local and texture memory. An improvement is not expected while using local memory, as the XU3 does not have local memory. Using texture memory the cache hit rate should be increased, resulting in a reduced execution time.

Vectorization is the fourth optimization. By vectorizing the filtering part of DFC an improvement of up to $8\times$ is theoretically possible for this phase, as it is possible to operate on 8 input locations with a single instruction. Vectorized instructions should be better for the SIMD cores of the XU3.

The fifth configuration parameter is the OpenCL workgroup size. The impact of workgroup size is highly dependant on the device, but a larger workgroup size should result in fewer workgroups, and therefore less overhead.

The second to last configuration parameter is the amount of input that is processed at once by the GPU. A larger chunk results in more threads and a fewer number of executions of the kernel. However, there is also a memory limit and more memory may be more costly.

The very final optimization tested is overlapping the execution of GPU and CPU. As the execution time of the different sections of the algorithm can no longer be fairly separated, it is not displayed in the same fashion as the previous optimization parameters. When evaluating the effects of overlapping execution, a test with Snort's HTTP patterns are matched against all traffic data sets. A decreased execution time, and therefore also a reduced total energy consumption, is expected to come of overlapping the execution.

To better understand the difference in energy consumption between the CPU, GPU and heterogeneous version of DFC, each version was executed with a one second sleep between major phase of the execution. The phases are input validation and environment setup, reading patterns from file and storing them intermediately, compiling patterns into direct filters and compact tables, matching, and finally environment teardown. The GPU and heterogeneous variant is also displayed when the work between the CPU and GPU overlap. The overlapping execution should require more instantaneous energy, but have a lower execution time.

Once the best configuration for both the GPU and heterogeneous variant has been greedily found, all three variants of DFC will be compared against one another. The comparison will use all traffic and pattern data sets, to see how different data impact the variants. Once more, five iterations are executed. The variants will then be compared to Aho-Corasick, where all data sets are once more used.

The latency of when some input starts to get analyzed and when it finishes, as well as the impact this latency has upon potential packet drops, is *not* tested. While the latency is easy to test, it is not of interest in this thesis where the focus is on the throughput and energy consumption. Such latency is mostly a balancing act between the amount of data analyzed at once, and throughput. If a large number of bytes are analyzed at once, some time is required to fill up the buffer, therefore increasing latency. However, with smaller buffers the full potential of the GPU may not be utilized, resulting in a poor throughput.

5.2 Results

This section presents the results of all tests.

5.2.1 Effect of optimizations

Tables 5.1 and 5.2 shows the effect mapping memory had on the GPU and heterogeneous variant respectively. *EXE* and *ENE* are the abbreviations for execution time and energy consumption, respectively. The required time for reading from the GPU decreased to 20% and 13% of the original, for the GPU and heterogeneous version respectively, showing a tremendous speedup for a single configuration.

Configurations	Results						Improvement	
MAP MEMORY	WRITE (ms)	READ (ms)	KERNEL (ms)	CPU (ms)	TOTAL (ms)	ENERGY (J)	EXE	ENE
NO	431	4870	2596	2730	11170	2867	1	1
YES	198	945	2750	2661	6787	1806	1.65	1.59

Table 5.1: Impact of mapping memory for the GPU variant

Configurations	Results						Improvement	
MAP MEMORY	WRITE (ms)	READ (ms)	KERNEL (ms)	CPU (ms)	TOTAL (ms)	ENERGY (J)	EXE	ENE
NO	436	1313	646	2509	5195	1438	1	1
YES	183	165	807	2475	3858	1192	1.35	1.21

Table 5.2: Impact of mapping memory for the heterogeneous variant

The results from increasing the thread granularity is shown in Tables 5.3 and 5.3. The GPU variant improves execution time by almost 2 $\times$ and reduces energy consumption by even more. Both the CPU and kernel execution time is reduced by about 400ms, equalling approximately 30% and 20% respectively. In the heterogeneous variant, the kernel execution time is just barely improved.

Configurations	Results						Improvement	
THR GRAN	WRITE (ms)	READ (ms)	KERNEL (ms)	CPU (ms)	TOTAL (ms)	ENERGY (J)	EXE	ENE
1	198	945	2750	2661	6787	1806	1.00	1.00
8	195	678	1829	1191	4118	991	1.65	1.82
16	195	654	1599	1119	3790	923	1.79	1.96
24	196	653	1488	1108	3670	875	1.85	2.06
32	194	644	1427	963	3451	854	1.97	2.11
40	196	648	1406	899	3440	845	1.97	2.14
48	196	647	1412	764	3464	845	1.96	2.14

Table 5.3: Impact of thread granularity for the GPU variant

Configurations	Results						Improvement	
THR GRAN	WRITE (ms)	READ (ms)	KERNEL (ms)	CPU (ms)	TOTAL (ms)	ENERGY (J)	EXE	ENE
1	183	165	807	2475	3858	1192	1.00	1.00
8	181	163	763	2482	3815	1200	1.01	0.99
16	185	167	1027	2902	4509	1277	0.86	0.93

Table 5.4: Impact of thread granularity for the heterogeneous variant

Using local or texture memory results in great slowdowns, seen in Tables 5.5 and 5.6. The use of local memory, not surprisingly, causes the greatest slowdown. Texture memory also causes a slowdown, but not as great as local memory.

Configurations	Results						Improvement	
MEM TYPE	WRITE (ms)	READ (ms)	KERNEL (ms)	CPU (ms)	TOTAL (ms)	ENERGY (J)	EXE	ENE
NORMAL	196	648	1406	899	3440	845	1.00	1.00
LOCAL	194	643	6267	900	8411	2238	0.41	0.38
TEXTURE	197	646	1777	899	3897	996	0.88	0.85

Table 5.5: Impact of local and texture memory for the GPU variant

Configurations	Results						Improvement	
MEM TYPE	WRITE (ms)	READ (ms)	KERNEL (ms)	CPU (ms)	TOTAL (ms)	ENERGY (J)	EXE	ENE
NORMAL	181	163	763	2482	3815	1200	1.00	1.00
LOCAL	183	165	22274	3016	25866	8138	0.15	0.15
TEXTURE	185	166	1070	2903	4548	1320	0.84	0.91

Table 5.6: Impact of local and texture memory for the heterogeneous variant

Vectorization, expected to have a large positive impact, instead had a negative one, displayed in Tables 5.7 and 5.8. The kernel execution time increased significantly in both the GPU and heterogeneous variant.

Configurations	Results						Improvement	
VECTORIZED	WRITE (ms)	READ (ms)	KERNEL (ms)	CPU (ms)	TOTAL (ms)	ENERGY (J)	EXE	ENE
NO	196	648	1406	899	3440	845	1.00	1.00
YES	191	637	2065	876	4172	1095	0.82	0.77

Table 5.7: Impact of vectorized design for the GPU variant

Configurations	Results						Improvement	
VECTORIZED	WRITE (ms)	READ (ms)	KERNEL (ms)	CPU (ms)	TOTAL (ms)	ENERGY (J)	EXE	ENE
NO	181	163	763	2482	3815	1200	1.00	1.00
YES	183	164	1084	2940	4594	1285	0.83	0.93

Table 5.8: Impact of vectorized design for the heterogeneous variant

Workgroup size was not expected to have a large impact, and did not, seen in Tables 5.9 and 5.10.

Configurations	Results						Improvement	
WORKGROUP	WRITE (ms)	READ (ms)	KERNEL (ms)	CPU (ms)	TOTAL (ms)	ENERGY (J)	EXE	ENE
128	196	648	1406	899	3440	845	1.00	1.00
64	196	645	1674	892	3784	905	0.91	0.93
256	196	645	1435	892	3501	870	0.98	0.97

Table 5.9: Impact of workgroup size for the GPU variant

Configurations	Results						Improvement	
WORKGROUP	WRITE (ms)	READ (ms)	KERNEL (ms)	CPU (ms)	TOTAL (ms)	ENERGY (J)	EXE	ENE
128	181	163	763	2482	3815	1200	1.00	1.00
64	185	167	783	2465	3828	1186	1.00	1.01
256	183	165	763	2500	3839	1202	0.99	1.00

Table 5.10: Impact of workgroup size for the heterogeneous variant

As the number of bytes processed at once increases, the GPU variant becomes faster but requires some extra energy, seen in Table 5.11. However, Tables 5.12 shows how

the heterogeneous variant becomes slower both when increasing and decreasing the number of bytes processed at once. The reason why a larger data processing chunk was not tested for the GPU variant, was because the memory could not be allocated and the program crashed.

Configurations	Results						Improvement	
CHUNK (MB)	WRITE (ms)	READ (ms)	KERNEL (ms)	CPU (ms)	TOTAL (ms)	ENERGY (J)	EXE	ENE
10	196	648	1406	899	3440	845	1.00	1.00
20	246	604	1399	693	3212	871	1.07	0.97
25	286	580	1394	641	3187	872	1.08	0.97

Table 5.11: Impact of read chunk size for the GPU variant

Configurations	Results						Improvement	
CHUNK (MB)	WRITE (ms)	READ (ms)	KERNEL (ms)	CPU (ms)	TOTAL (ms)	ENERGY (J)	EXE	ENE
10	181	163	763	2482	3815	1200	1.00	1.00
5	187	173	781	2494	3857	1189	0.99	1.01
20	201	163	765	2709	4080	1207	0.94	0.99

Table 5.12: Impact of read chunk size for the heterogeneous variant

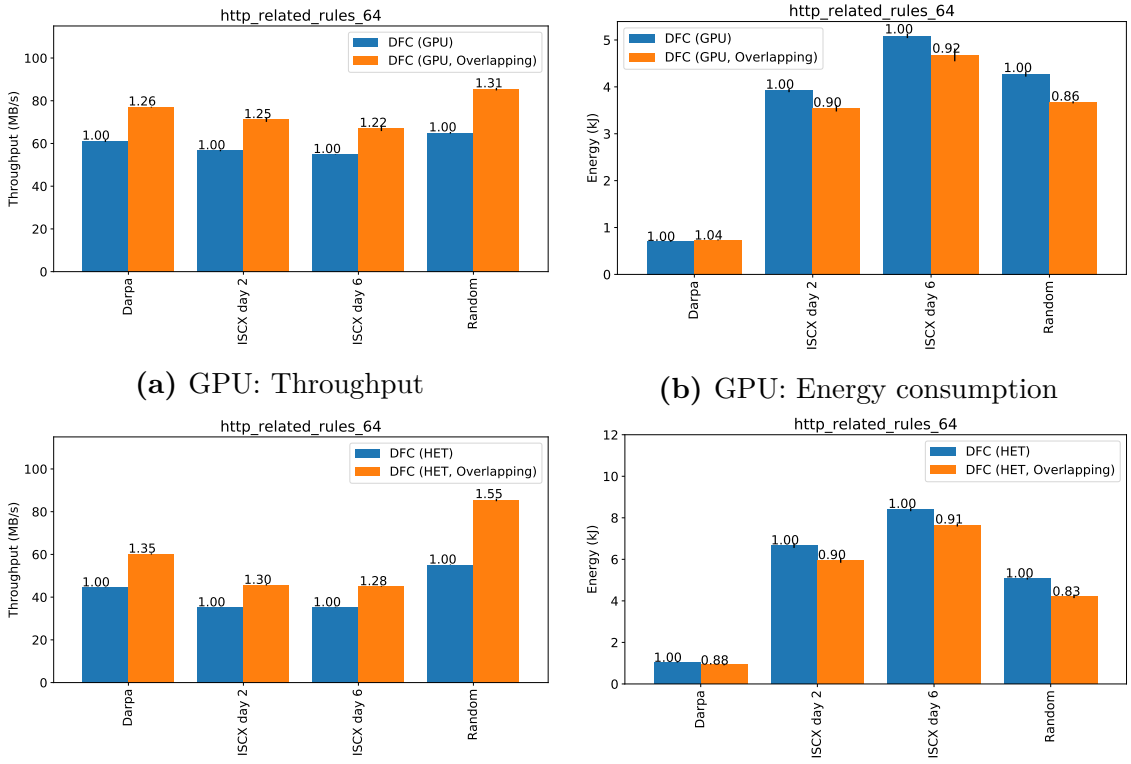
A summary of all optimizations performed and their impact are displayed in Appendix A.

In the end, the optimizations performed so far onto the GPU and heterogeneous version of DFC resulted in a speedup of 3.5 × and 1.36 × respectively.

The final optimization performed is using overlapping execution between the CPU and GPU. As the execution becomes overlapped, it is no longer possible to easily count the required time for the different components, and instead the total throughput and energy is used to measure the effectiveness. Figure 5.2 shows how overlapping execution is better both in terms of execution time and energy, for both variants of DFC. The throughput of the GPU variant is improved by an additional 1.26 × on real network traffic, whereas the throughput of the heterogeneous variant is improved by an additional 1.35 × on the same data set.

The energy consumption for the different phases of DFC for the CPU, GPU and heterogeneous version can be seen in Figure 5.3a, 5.3b and 5.3c respectively. The plotted values are the energy consumptions for the two CPUs, A15 and A7, the GPU, the RAM and finally the total consumption, when comparing Snort's HTTP patterns against the DARPA network dump.

The two versions using the GPU has a large spike at the start for a short duration, as the OpenCL environment is initialized. Only the GPU version has a large spike during the compilation phase, where the patterns are compiled into the Direct Filters (DFs) and compact tables used for exact matching. The reason is that in this phase, these compiled data structures are also transferred to the GPU in preparation for the matching phase. A small spike can be seen at the same time for the heterogeneous version. The spike is much smaller as there is no need to store the memory intensive compact tables, nor a large result vector, causing less overhead for OpenCL.



(a) GPU: Throughput

(b) GPU: Energy consumption

(c) Heterogeneous: Throughput

(d) Heterogeneous: Energy consumption

Figure 5.2: Effect of overlapping execution

Following the compilation phase is the matching phase. As expected, the CPU only version increases the energy consumption solely of the CPU, while the two others offloads some of the processing, and with it the energy consumption, to the GPU.

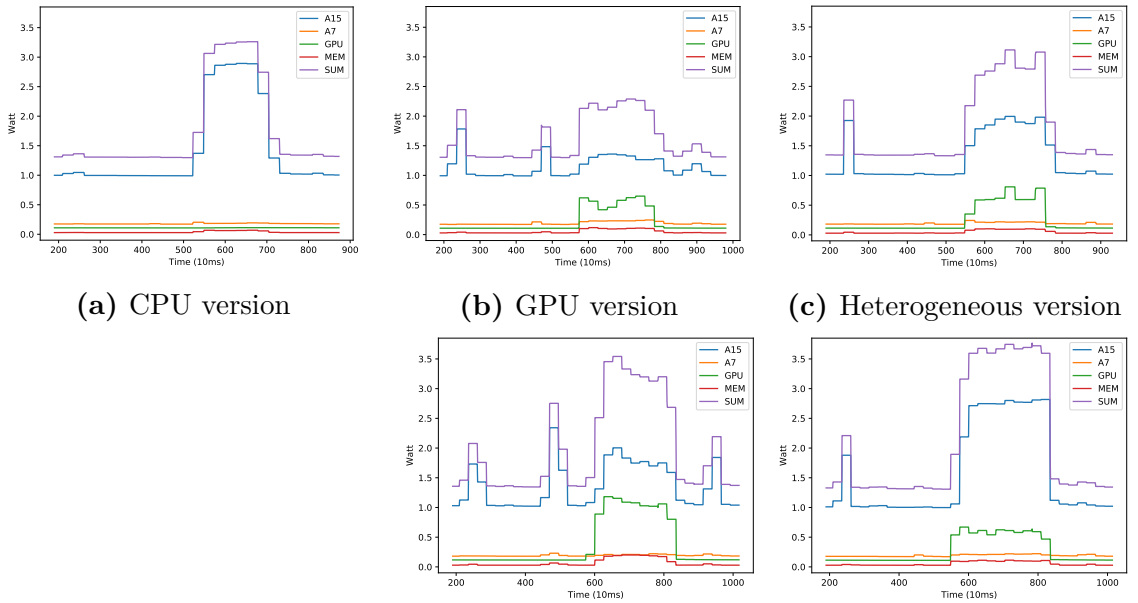
Both version using OpenCL has a spike at the end, where the GPU version requires more energy than the heterogeneous version. Once again, this is because there are fewer allocations required for the heterogeneous variant.

The spikes within the the matching phase that exists within the GPU and heterogeneous version of DFC comes from the multiple executions switching between kernel execution and CPU execution. The spikes are smaller in the heterogeneous version as the chunk size used is smaller, requiring less time for each execution.

While the memory has a barely noticeable increase in power usage in the CPU version, it is more apparent in the two version using the GPU, especially in the GPU heavy version, and even more so when overlapping the execution. Even when the memory is mapped between the CPU and GPU to reduce memory transfers, some extra power is shown to be required.

It can be seen how both the GPU and heterogeneous version of DFC without overlapping execution requires less energy than the CPU one. However, the configurations using overlapping execution has a larger instantaneous energy consumption than the

others, as both the GPU and CPU gets utilized more.



(a) CPU version

(b) GPU version

(c) Heterogeneous version

(d) GPU version (overlapping execution)

(e) Heterogeneous version (overlapping execution)

Figure 5.3: Phases of energy consumption for DFC

5.2.2 Comparison of Direct Filter Classification variants

As the overlapping configuration improved both the GPU and heterogeneous, they will be the ones used in further tests. The GPU version of DFC is faster than either the CPU variant or the heterogeneous in all cases, followed by the heterogeneous variant, as seen in Figure 5.4, 5.5 and 5.6.

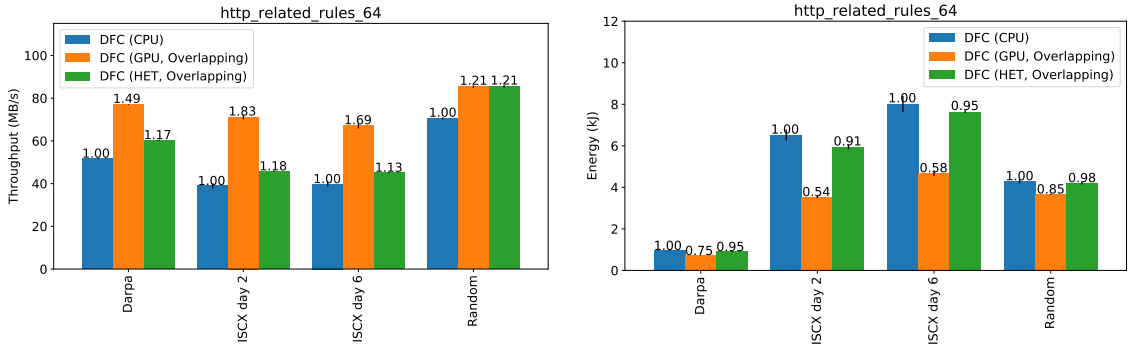
The variants using the GPU requires the least energy in all cases, where the GPU only version requires significantly less energy.

The greatest gains in total energy consumption are when the GPU version is significantly faster, at times reaching a more than 2 $\times$ speedup compared to the second fastest version, seen in Figure 5.5a. This causes the energy consumption to be only half that of the second fastest version.

When matching against random data with few patterns, the heterogeneous and GPU version have a similar throughput, seen in Figure 5.4a. However, as the amount of patterns increase, the GPU variant becomes significantly faster.

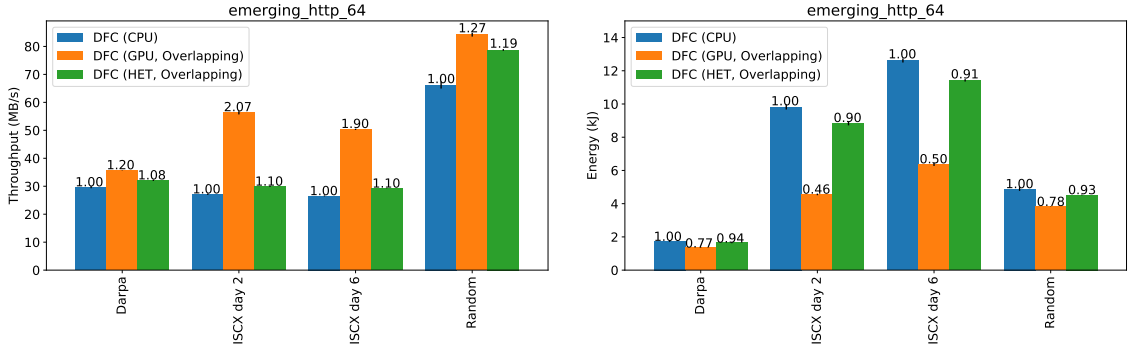
5.2.3 Direct Filter Classification compared to Aho-Corasick

Because the CPU version is slower than the other DFC variants, it is not included in the comparisons against AC. Furthermore, the XU3 ran out of memory when using



(a) Throughput **(b)** Energy consumption

Figure 5.4: Comparison of DFC variants: Snort HTTP patterns (2k)



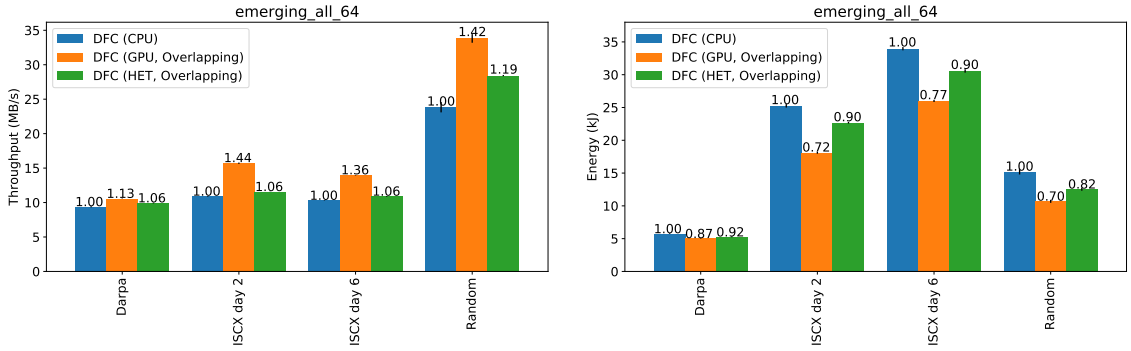
(a) Throughput **(b)** Energy consumption

Figure 5.5: Comparison of DFC variants: `emergingthreats.net` HTTP patterns (9k)

Aho-Corasick (AC) with 20,000 patterns and large traffic data set, and is therefore not included in the results.

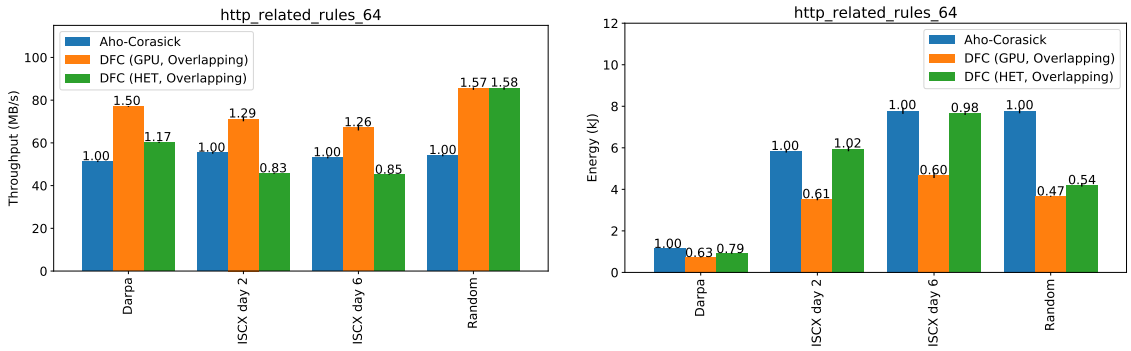
Figures 5.7 and 5.8 show how AC is equally as fast, or faster, than the CPU variant of DFC, except for when matching against random traffic. One may see how AC keeps a somewhat consistent throughput for each traffic data set, whereas DFC is much more dependant upon the amount of hits during the filter phase.

The GPU is once more the fastest option, beating the throughput of AC by 1.67 × while only requiring 0.54 × of the energy. The heterogeneous variant is faster than AC when matching against the Darpa network traffic and the randomly generated traffic, otherwise AC is faster than the heterogeneous variant. When matching against random data, the variants of DFC are up to 2.5× faster, while only requiring 0.4 × as much energy.



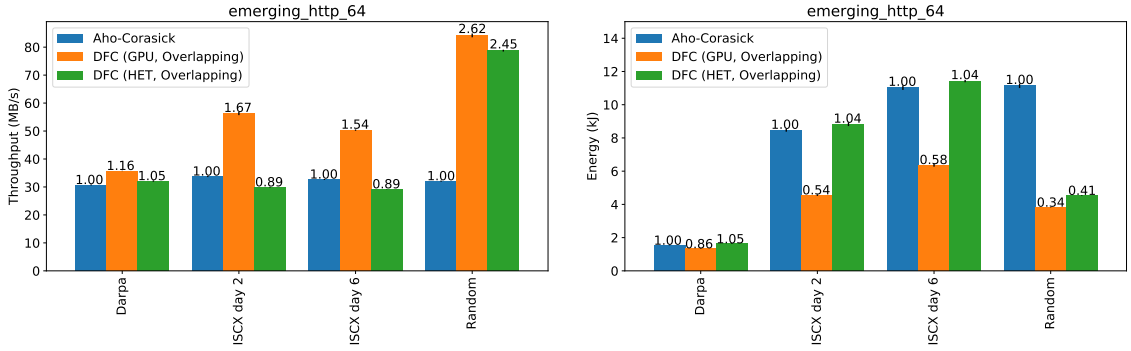
(a) Throughput **(b)** Energy consumption

Figure 5.6: Comparison of DFC variants: All `emergingthreats.net` patterns (21k)



(a) Throughput **(b)** Energy consumption

Figure 5.7: DFC compared to AC: Snort HTTP patterns (2k)



(a) Throughput **(b)** Energy consumption

Figure 5.8: DFC compared to AC: `emergingthreats.net` HTTP patterns (9k)

6

Discussion

This chapter will start by discussing the result shown in Section 5.2. It will then briefly discuss any ethical and sustainability impacts the areas touched upon in this thesis may have. The chapter concludes by discussing potential future work that may be done in the area of GPGPU to improve IoT applications.

6.1 Effect of optimizations

Many tests were run in order to greedily find the best configuration for the two variants of DFC using a GPU. The many tests showed the importance of constant measurements, as what one might assume to be an obvious optimization might not be, and even differs between the two variants. The summary of the final results can be seen in Appendix A.

As expected, mapping memory had a great impact upon the total time, seen in Table 5.1 and 5.2. This is because there is no longer need to copy data between the CPU and GPU.

Thread granularity had a surprisingly large effect upon the GPU version, seen in Table 5.3. Both the kernel and CPU execution time is greatly reduced, but the CPU time even more so. The reduced execution time on the CPU is because it does not have to needlessly iterate over many positions in the result vector, as the matches are bundled per thread. Barely any improvement was recorded for the heterogeneous variant, see Table 5.4, but the small improvement seen were most likely because of fewer wasted memory loads.

As expected, utilizing local memory did not result in any gains for either variant, but instead caused great slowdowns, seen in Tables 5.5 and 5.6. As mentioned in Section 4.3.4, this was expected as the Mali GPU does not have any local memory, but instead emulates local memory by simply storing it in global memory. Using local memory therefore only increases the amount of memory accesses in vain, causing the slowdown.

By storing the Direct Filters (DFs) in texture memory, a reduced execution time was hoped to be achieved thanks to an increased cache hit rate. Instead it resulted

in an increased execution time of the kernel. The cause is possibly that textures in OpenCL have to be loaded as vectors. This means that a 16B load has to be performed, and the index with that contains the one relevant bit has to then be calculated, causing extra computations. Since this computations is in the hot path of the kernel execution, it may be the cause of the slowdown. The hot path is very important in DFC, since DFC's speed is the result of quickly being able to filter out matches.

Utilizing vector instructions did not speed up the execution either, displayed in Tables 5.7 and 5.8. The potential $8\times$ throughput increase instead turned into a 40% increase of kernel execution time for the GPU version. The most likely reason is that the amount of memory accesses increase, causing the slowdown. The filtering phase of the vectorized version requires many stores, as the previously not stored filter result have to be both stored. These filters have to later be read, causing unnecessary memory accesses. It is highly likely that a GPU with local memory could benefit from storing the filter matches, as writing and reading from local memory is far less costly than accessing global memory.

There are two further reasons as to why the vectorized kernel is slower than the scalar one. The first is the lack of a *gather* instruction, causing the filtering phase to not be entirely vectorized and has to instead become scalar in the middle. However, the slow down should still not cause the execution time to be worse than the simple scalar version. Secondly, the vectorized version requires slightly more complex logic for indexing, requiring extra instructions in the hot path of the execution.

Overlapping execution further improved execution time and energy consumption. It can be seen in Figure 5.3 how the to overlapping variant require a larger instantaneous energy consumption, but reduce the total energy consumption due to the reduced execution time. The reduced execution time is most likely because the overlapping execution better utilizes the CPU and GPU, where frequent stalls do not interrupt the execution behavior.

The rest of this section will go into more detail regarding the two respective variants.

6.1.1 Graphics Processing Unit version

The speedups gained from optimizing the parameters of the GPU version of DFC were great, resulting in a speedup of $3.5\times$ and an energy improvement almost as big. Two optimizations had the biggest impact upon the improvements: mapping memory and increasing the thread granularity. The improvements gained from mapping memory were expected, as the memory transfer overhead is often the bottleneck for GPGPU applications [19]. Mapping memory had a much greater impact upon the GPU version compared to the heterogeneous variant because the buffer containing the results of the matching is much greater in size for the GPU version.

Increasing the thread granularity to 8 had the largest relative increase, while the

remaining increases up to 40 had diminishing returns. Somewhat surprisingly, the CPU execution time was affected the most by increasing the thread granularity. The reason is because the array containing the matches is stored per thread instead of per input character, greatly decreasing the work that has to be performed if there are no matches, as discussed in Section 4.3.2. Why the kernel execution time was reduced is not as obvious, but may have to do with fewer threads being spawned, causing less competition for execution. Fewer threads might also lead to worse latency hiding during a memory load, which is why it might lead to a slowdown in the end. Another possible reason for the speedup of the kernel execution is that since any memory load always fetches 16B from memory [41], memory might have been fetched unnecessarily when only a few bytes were accessed. This contradicts somewhat with the idea of wasted memory loads, since 40 is not divisible by 16.

Further improvements were seen when increasing the amount of data processed at once. Larger data chunks results in more threads being spawned, and fewer kernel executions. While the kernel and CPU execution time both decreases by a small margin, the total time does not decrease by much, and the energy consumption actually increases. A possible reason is that there is some overhead related to tracking the many threads, or working with a larger allocation of memory.

6.1.2 Heterogeneous version

The heterogeneous version did not see great improvements from most optimizations. Mapping memory and overlapping the execution of the CPU and GPU had the largest impact. Any other configuration caused a slowdown or increase in energy consumption.

Why increasing the thread granularity to 16 caused a slowdown compared to 8 in kernel execution time is not obvious, as it had a positive effect for the kernel in the GPU version. One reason might be that it results in fewer threads, not being able to hide the many memory loads required for the simple execution model of the heterogeneous kernel. However, that line of reasoning does not seem to hold as increasing the chunk size increases the amount of running threads, and with the previous reasoning it should then improve the speed further.

6.2 Comparison of Direct Filter Classification variants

The GPU variant of DFC is faster, or equally as fast, than either the CPU and heterogeneous in all cases. When matching against few patterns, the GPU and heterogeneous variant have the same throughput for random data. The reason is that the few patterns result in few matches in the filters, resulting in the CPU not having to perform many exact matches. But as the number of pattern increases, the work that has to be performed by the CPU also increases, slowing down the heterogeneous variant. With the same reasoning one may see how the CPU variant

is much closer to the throughput of the GPU variant when comparing few patterns to random data, but the gap increases with the number of patterns.

Another intriguing finding, also suggested by Figure 5.3, is that even when the throughput of the GPU variant and the heterogeneous version is equal, the energy required for the GPU version is lower. Once again, this is because the GPU requires less energy than the CPU.

As to why the GPU version is faster than the heterogeneous version when matching against real traffic is simply because of the increased concurrency granted through the many threads of the GPU is better suited for exact matching. As seen with the random network traffic, filtering is cheap but exact matching is not. Therefore, by offloading both filtering and exact matching to the many threads of the GPU, a speedup is achieved. This is not necessarily true for all GPUs, as the many branches required for exact matching may be too costly on some architectures, discussed in Section 2.4.1.

What is interesting is that filtering was believed to be the phase better suited on a GPU, while it seems as if exact matching is the better suited phase.

Another finding is the great improvements that were had upon the CPU's execution time with the GPU variant when the thread granularity was increased. This finding pointed towards how costly it is to simply loop through hundreds of megabytes of data. A similar scheme may be used for the heterogeneous variant, but was not implemented in this thesis. However, an equally drastic improvement is not expected as it is still believed that exact matching is what requires the most time for the CPU in the heterogeneous variant.

6.3 Direct Filter Classification compared to Aho-Corasick

The biggest difference between Aho-Corasick (AC) and DFC is arguably that the throughput of AC is consistent between traffic data sets. AC simply iterates over the characters in the input, transitioning between different states for each one, and is therefore not concerned with how many matches there are in the input. DFC is much more dependant upon the input data, able to very quickly remove true negative matches, but slows down once exact matching is performed. This is the most obvious when matching against random data, as DFC is able to quickly filter out all the non-matches.

The sensitivity DFC has towards the network traffic analyzed may open it up towards algorithmic attacks, where the attacker sends carefully crafted packets to cause resource exhaustion of the NIDS. If the attacker knew the patterns used in the NIDS, it could cause a great slowdown by sending the longest pattern in the set, as the exact matching is what requires the most time.

The GPU variants improves the throughput up to $1.67\times$ while only requiring $0.54\times$ the energy. The reason is once more that the GPU simply requires less energy for the same amount of work, and is able to do more work at once thanks to the many cores.

6.4 Ethics & Sustainability

Internet of Things has long been plagued by intrusions. At the time of writing, another $500,000$ malware infected home routers have been discovered [57]. In the article it is described how this malware has a kill-switch, being able to destroy any of the infected devices at will, potentially rendering hundreds of thousands without internet.

It is easy to see how solving this issue would be ethical. If embedded IoT devices were able to run an efficient NIDS, maybe some of the many attacks may have been avoidable. It might also bring further security education to the masses, as they may now be notified if someone tries to access their system without having been granted access.

Decreasing the energy consumption needed for an NIDS is obviously a step in the right direction towards sustainability. However, if it was deemed that the best implementation of an NIDS in a home is a separate device, there might be a need to produce many NIDS devices. And as the bandwidth requirements increase, these IoT Network Intrusion Detection Systems have to be replaced, causing unnecessary waste. Hopefully the material in these devices could somehow be recycled, removing most of the concerns regarding unnecessary waste.

6.5 Future Work

The opportunities for future work within the cross section of GPGPU, IoT and NIDS are many. One way forward is to attempt to improve DFC in a GPGPU context further. Potentially by improving the communication pattern between the CPU and GPU, as the largest gains seemed to be had there. Another way is to further improve the kernel execution time, as the largest time spent is currently there. Yet another improvement to be made is to let the CPU and GPU work at the same time, instead of having the GPU wait for the CPU to process the matches before starting to process the next chunk of input data.

Further improvements to DFC may be looked into where the cost of the exact matching is reduced, as it is obviously a bottleneck considering that when matching against random data the throughput can be almost doubled. One way might be to reverse the design of the heterogeneous variant, where the filtering is instead performed on the CPU and the exact matching on the GPU.

Another way forward is to search for other pattern matching algorithms better suited

for being executed on a GPU, improving the execution time and energy consumption further.

Finally, it has been shown that GPGPU can improve the execution time for some applications. Further research should be done to see what kind of applications these are. Such research would not only aid IoT devices, but could also have a big impact upon mobile devices. As many smartphones nowadays have powerful GPUs, being able to take advantage of it to not only decrease execution time but also energy consumption is an attractive possibility.

7

Conclusion

In this thesis the suitability of using a low-powered and network connected embedded device with a GPU as a network intrusion detection system was evaluated. The suitability depends on the energy consumption and the throughput of network traffic that the system can analyze. A GPGPU implementation of DFC is used to improve the throughput of network traffic that may be analyzed per second and lower the energy consumption.

The thesis shows that utilizing a GPU may not only reduce the execution time, but also the energy consumption compared to only utilizing a CPU. By utilizing GPGPU, the Direct Filter Classification algorithm's throughput was increased by more than $2\times$ while reducing the energy consumption by more than 50%. The GPGPU variant of DFC was able to outperform the widely used pattern matching algorithm Aho-Corasick by more than 50% while only requiring 50% of the energy.

Even if the execution time is not reduced, a reduced energy consumption may be enjoyed. Utilizing a GPU as much as possible is therefore shown to be the best solution. However, optimizing GPGPU code requires great care, and requires many measurements to reach optimal performance. Failure to do so will result in a sub-optimal throughput and energy consumption.

GPGPU in embedded systems may be used to gain significant speedups while reducing energy consumption. It is therefore reasonable to assume that for Network Intrusion Detection Systems in IoT, GPGPU is the path forward.

Bibliography

[1] Cheng-Hung Lin, Chen-Hsiung Liu, Lung-Sheng Chien, and Shih-Chieh Chang. Accelerating Pattern Matching Using a Novel Parallel Algorithm on GPUs. *IEEE Transactions on Computers*, 62(10):1906–1916, Oct 2013.

[2] Charalampos Stylianopoulos, Magnus Almgren, Olaf Landsiedel, and Marina Papatriantafilou. Multiple Pattern Matching for Network Security Applications: Acceleration through Vectorization. In *2017 46th International Conference on Parallel Processing (ICPP)*, pages 472–482, Aug 2017.

[3] RemoteLock 7i WiFi Lock. `https://shop.remotelock.com/product/lockstate-remotelock-7i/`. Accessed: 2018-02-20.

[4] tado° Smart Thermostat Starter Kit v3. `https://www.tado.com/se/products/smart-thermostat-starter-kit`. Accessed: 2018-02-21.

[5] Microsoft IoT. `https://www.microsoft.com/en-us/internet-of-things/connected-factory`. Accessed: 2018-02-20.

[6] IBM Watson IoT. `https://www.ibm.com/internet-of-things/spotlight/iot-zones/smart-factories`. Accessed: 2018-02-20.

[7] A New Era of Internet Attacks Powered by Everyday Devices. `https://nytimes.com/2016/10/23/us/politics/a-new-era-of-internet-attacks-powered-by-everyday-devices.html`. Accessed: 2018-03-04.

[8] New Mirai Worm Knocks 900K Germans Offline. `https://krebsonsecurity.com/2016/11/new-mirai-worm-knocks-900k-germans-offline/`. Accessed: 2018-03-04.

[9] Update bricks smart locks preferred by Airbnb. `https://techcrunch.com/2017/08/14/wifi-disabled/`. Accessed: 2018-03-04.

[10] Hervé Debar, Marc Dacier, and Andreas Wespi. Towards a taxonomy of intrusion-detection systems. *Computer Networks*, 31(8):805 – 822, 1999.

[11] Spyros Antonatos, Kostas G. Anagnostakis, and Evangelos P. Markatos. Generating Realistic Workloads for Network Intrusion Detection Systems. *SIGSOFT Softw. Eng. Notes*, 29(1):207–215, January 2004.

[12] Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. Parallelization and characterization of pattern matching using GPUs. In *2011 IEEE International Symposium on Workload Characterization (IISWC)*, pages 216–225, Nov 2011.

[13] Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. MIDeA: A Multi-parallel Intrusion Detection Architecture. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 297–308, New York, NY, USA, 2011. ACM.

[14] Muhammad Asim Jamshed, Jihyung Lee, Sangwoo Moon, Insu Yun, Deokjin Kim, Sungryoul Lee, Yung Yi, and KyoungSoo Park. Kargus: A Highly-scalable Software-based Intrusion Detection System. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 317–328, New York, NY, USA, 2012. ACM.

[15] Arian Maghazeh, Unmesh D. Bordoloi, Petru Eles, and Zebo Peng. General Purpose Computing on Low-Power Embedded GPUs: Has It Come of Age? Technical report, Linköping University, Software and Systems, 2013.

[16] Elena Aragon, Juan M. Jiménez, Arian Maghazeh, Jim Rasmusson, and Unmesh D. Bordoloi. Pattern Matching in OpenCL: GPU vs CPU Energy Consumption on Two Mobile Chipsets. In *Proceedings of the International Workshop on OpenCL 2013 & 2014*, IWOCL '14, pages 5:1–5:7, New York, NY, USA, 2014. ACM.

[17] Martin Roesch. Snort - Lightweight Intrusion Detection for Networks. In *Proceedings of the 13th USENIX Conference on System Administration*, LISA '99, pages 229–238, Berkeley, CA, USA, 1999. USENIX Association.

[18] ODROID-XU3 Product Page. `http://www.hardkernel.com/main/products/prdt_info.php?g_code=g140448267127&tab_idx=1`. Accessed: 2018-05-25.

[19] Younghwan Go, Muhammad Asim Jamshed, YoungGyoun Moon, Changho Hwang, and KyoungSoo Park. APUNet: Revitalizing GPU as Packet Processing Accelerator. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 83–96, Boston, MA, 2017. USENIX Association.

[20] Ericsson AB. Ericsson Mobility Report. `https://www.ericsson.com/assets/local/mobility-report/documents/2017/Ericsson-mobility-report-june-2017.pdf`, 06 2017. Accessed: 2018-03-28.

[21] Chris W. Johnson. Barriers to the Use of Intrusion Detection Systems in Safety-Critical Applications. In Floor Koornneef and Coen van Gulijk, editors, *Computer Safety, Reliability, and Security*, pages 375–384, Cham, 2015. Springer International Publishing.

[22] Snort User Manual. `https://snort.org/downloads/snortplus/snort_manual.pdf`. Accessed: 2018-03-06.

[23] Robert S. Boyer and J. Strother Moore. A Fast String Searching Algorithm. *Commun. ACM*, 20(10):762–772, October 1977.

[24] Beate Commentz-Walter. A String Matching Algorithm Fast on the Average. In *Proceedings of the 6th Colloquium, on Automata, Languages and Programming*, pages 118–132, London, UK, UK, 1979. Springer-Verlag.

[25] Alfred V. Aho and Margaret J. Corasick. Efficient String Matching: An Aid to Bibliographic Search. *Commun. ACM*, 18(6):333–340, June 1975.

[26] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, March 1987.

[27] Sun Wu and Udi Manber. A fast algorithm for multi-pattern searching. Technical Report TR-94-17, University of Arizona. Department of Computer Science, May 1994.

[28] Byungkwon Choi, Jongwook Chae, Muhammad Jamshed, KyoungSoo Park, and Dongsu Han. DFC: Accelerating String Pattern Matching for Network Applications. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 551–565, Santa Clara, CA, 2016. USENIX Association.

[29] Iulian Moraru and David G. Andersen. Exact Pattern Matching with Feed-forward Bloom Filters. *J. Exp. Algorithmics*, 17:3.4:3.1–3.4:3.18, September 2012.

[30] Marc Norton. White paper: Optimizing pattern matching for intrusion detection. Technical report, 2004.

[31] Maurice Herlihy. *The art of multiprocessor programming*, page 14. Morgan Kaufmann, Waltham, MA, 2012.

[32] Rajat Raina, Anand Madhavan, and Andrew Y. Ng. Large-scale Deep Unsupervised Learning Using Graphics Processors. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pages 873–880, New York, NY, USA, 2009. ACM.

[33] John Michalakes and Manish Vachharajani. GPU acceleration of numerical weather prediction. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–7, April 2008.

[34] Giorgos Vasiliadis, Spiros Antonatos, Michalis Polychronakis, Evangelos P. Markatos, and Sotiris Ioannidis. Gnort: High Performance Network Intrusion Detection Using Graphics Processors. In Richard Lippmann, Engin Kirda, and Ari Trachtenberg, editors, *Recent Advances in Intrusion Detection*, pages

116–134, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[35] Nausheen Shoaib, Jawwad Shamsi, Tahir Mustafa, Akhter Zaman, Jazib ul Hasan, and Mishal Gohar. GDPI: Signature based Deep Packet Inspection using GPUs. *International Journal of Advanced Computer Science and Applications*, 8(11), 2017.

[36] OpenCL Overview. `https://www.khronos.org/opencl/`. Accessed: 2018-03-11.

[37] About CUDA. `https://developer.nvidia.com/about-cuda`. Accessed: 2018-03-11.

[38] Richard Fromm, Stylianos Perissakis, Neal Cardwell, Christoforos Kozyrakis, Bruce McGaughy, David Patterson, Tom Anderson, and Katherine Yelick. The Energy Efficiency Of Iram Architectures. In *Conference Proceedings. The 24th Annual International Symposium on Computer Architecture*, pages 327–337, June 1997.

[39] Kshitij Gupta and John D. Owens. Compute & memory optimizations for high-quality speech recognition on low-end GPU processors. In *2011 18th International Conference on High Performance Computing*, pages 1–10, Dec 2011.

[40] Kwang-Ting Cheng and Yi-Chu Wang. Compute & Memory Optimizations for High-Quality Speech Recognition on Low-End GPU Processors. In *Proceedings of 2011 International Symposium on VLSI Design, Automation and Test*, pages 1–4, April 2011.

[41] ARM Mali GPU OpenCL, Version 3.0, Developer Guide. `https://static.docs.arm.com/100614/0300/arm_mali_gpu_opencl_developer_guide_100614_0300_00_en.pdf`. Accessed: 2018-03-14.

[42] Merriam Webster definition of heterogeneity. `https://www.merriam-webster.com/dictionary/heterogeneity`. Accessed: 2018-05-31.

[43] Miaoqing Huang and Chenggang Lai. Accelerating Applications Using GPUs on Embedded Systems and Mobile Devices. In *2013 IEEE 10th International Conference on High Performance Computing and Communications 2013 IEEE International Conference on Embedded and Ubiquitous Computing*, pages 1031–1038, Nov 2013.

[44] Blaine Rister, Guohui Wang, Michael Wu, and Joseph R. Cavallaro. A fast and efficient sift detector using the mobile GPU. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 2674–2678, May 2013.

[45] Shima Soroushnia, Masoud Daneshtalab, Juha Plosila, and Pasi Liljeberg. Heterogeneous Parallelization of Aho-Corasick Algorithm. In Julio Saez-Rodriguez, Miguel P. Rocha, Florentino Fdez-Riverola, and Juan F. De Paz Santana, editors, *8th International Conference on Practical Applications of Computational*

Biology & Bioinformatics (PACBB 2014), pages 153–160, Cham, 2014. Springer International Publishing.

[46] Burton H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM*, 13(7):422–426, July 1970.

[47] Sang Kil Cha, Iulian Moraru, Jiyong Jang, John Truelove, David Brumley, and David G. Andersen. SplitScreen: Enabling efficient, distributed malware detection. *Journal of Communications and Networks*, 13(2):187–200, April 2011.

[48] Samsung Exynos 5 Octa (5422) product page. `http://www.samsung.com/semiconductor/minisite/exynos/products/mobileprocessor/exynos-5-octa-5422/`. Accessed: 2018-03-14.

[49] ARM Cortex-A15. `https://developer.arm.com/products/processors/cortex-a/cortex-a15`. Accessed: 2018-03-14.

[50] ARM Cortex-A7 product page. `https://developer.arm.com/products/processors/cortex-a/cortex-a7`. Accessed: 2018-03-14.

[51] ARM Mali-T628 product page. `https://www.arm.com/products/multimedia/mali-cost-efficient-graphics/mali-t628.php`. Accessed: 2018-03-14.

[52] M. Elteir, H. Lin, and W. C. Feng. Performance Characterization and Optimization of Atomic Operations on AMD GPUs. In *2011 IEEE International Conference on Cluster Computing*, pages 234–243, Sept 2011.

[53] Toman Akenine-Moller and Jacob Strom. Graphics Processing Units for Handhelds. *Proceedings of the IEEE*, 96(5):779–789, May 2008.

[54] Bingsheng He, Naga K. Govindaraju, Qiong Luo, and Burton Smith. Efficient Gather and Scatter Operations on Graphics Processors. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, pages 46:1–46:12, New York, NY, USA, 2007. ACM.

[55] Matthew V. Mahoney and Philip K. Chan. An Analysis of the 1999 DARPA/Lincoln Laboratory Evaluation Data for Network Anomaly Detection. In Giovanni Vigna, Christopher Kruegel, and Erland Jonsson, editors, *Recent Advances in Intrusion Detection*, pages 220–237, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[56] Ali Shiravi, Hadi Shiravi, Mahbod Tavallaee, and Ali A. Ghorbani. Toward developing a systematic approach to generate benchmark datasets for intrusion detection. *Computers & Security*, 31(3):357 – 374, 2012.

[57] New VPNFilter malware targets at least 500K networking devices worldwide. `https://blog.talosintelligence.com/2018/05/VPNFilter.html`. Accessed: 2018-05-24.

A

Summary of configuration impact

Configurations						Results						Improvement	
MAP	THR	MEM	VEC	WG	CHUNK	WRITE (ms)	READ (ms)	KERNEL (ms)	CPU (ms)	TOTAL (ms)	ENERGY (J)	EXE	ENE
NO	1	NOR	NO	128	10	431	4870	2596	2730	11170	2867	1	1
YES	1	NOR	NO	128	10	198	945	2750	2661	6787	1806	1.65	1.59
YES	8	NOR	NO	128	10	195	678	1829	1191	4118	991	2.71	2.89
YES	16	NOR	NO	128	10	195	654	1599	1119	3790	923	2.95	3.11
YES	24	NOR	NO	128	10	196	653	1488	1108	3670	875	3.04	3.28
YES	32	NOR	NO	128	10	194	644	1427	963	3451	854	3.24	3.36
YES	40	NOR	NO	128	10	196	648	1406	899	3440	845	3.25	3.39
YES	48	NOR	NO	128	10	196	647	1412	764	3464	845	3.22	3.39
YES	40	LOC	NO	128	10	194	643	6267	900	8411	2238	1.33	1.28
YES	40	TEX	NO	128	10	197	646	1777	899	3897	996	2.87	2.88
YES	40	NOR	YES	128	10	191	637	2065	876	4172	1095	2.68	2.62
YES	40	NOR	NO	64	10	196	645	1674	892	3784	905	2.95	3.17
YES	40	NOR	NO	256	10	196	645	1435	892	3501	870	3.19	3.29
YES	40	NOR	NO	128	20	246	604	1399	693	3212	871	3.48	3.29
YES	40	NOR	NO	128	25	286	580	1394	641	3187	872	3.51	3.29

Table A.1: Summarized configuration impact for GPU version of DFC

Configurations						Results						Improvement	
MAP	THR	MEM	VEC	WG	CHUNK	WRITE (ms)	READ (ms)	KERNEL (ms)	CPU (ms)	TOTAL (ms)	ENERGY (J)	EXE	ENE
NO	1	NOR	NO	128	10	436	1313	646	2509	5195	1438	1	1
YES	1	NOR	NO	128	10	183	165	807	2475	3858	1192	1.35	1.21
YES	8	NOR	NO	128	10	181	163	763	2482	3815	1200	1.36	1.20
YES	16	NOR	NO	128	10	185	167	1027	2902	4509	1277	1.15	1.13
YES	8	LOC	NO	128	10	183	165	22274	3016	25866	8138	0.20	0.18
YES	8	TEX	NO	128	10	185	166	1070	2903	4548	1320	1.14	1.09
YES	8	NOR	YES	128	10	183	164	1084	2940	4594	1285	1.13	1.12
YES	8	NOR	NO	64	10	185	167	783	2465	3828	1186	1.36	1.21
YES	8	NOR	NO	256	10	183	165	763	2500	3839	1202	1.35	1.20
YES	8	NOR	NO	128	5	187	173	781	2494	3857	1189	1.35	1.21
YES	8	NOR	NO	128	20	201	163	765	2709	4080	1207	1.27	1.19

Table A.2: Summarized configuration impact for heterogeneous version of DFC