Efficient Scheduling of Concurrently Executed Network Packet Processing Applications using Heterogeneous Hardware

Giannis Giakoumakis

Thesis submitted in partial fulfillment of the requirements for the

Masters' of Science degree in Computer Science and Engineering

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes University Campus, 700 13 Heraklion, Crete, Greece

Thesis Advisors:
Assoc. Prof. Sotiris Ioannidis
Asst. Prof. Polyvios Pratikakis

This work has been performed at the Foundation for Research and Technology - Hellas (FORTH), Institute of Computer Science (ICS), Distributed Computing Systems and Cybersecurity Laboratory (DiSCS).

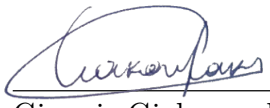
The work has been supported and received funding from the European Union's Horizon 2020 Research and Innovation program under grant agreement No 780787.

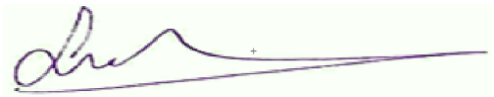
University of Crete
Computer Science Department

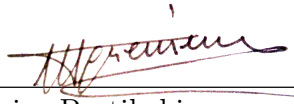
Efficient Scheduling of Concurrently Executed Network Packet Processing Applications using Heterogeneous Hardware

Thesis submitted by
Giannis Giakoumakis
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

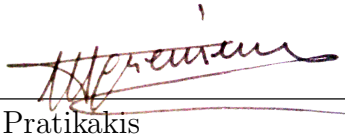
THESIS APPROVAL

Author: 
Giannis Giakoumakis

Committee approvals: 
Sotiris Ioannidis
Associate Professor, Thesis Supervisor


Polyvios Pratikakis
Assistant Professor, Thesis Supervisor


Xenofontas Dimitropoulos
Associate Professor, Committee Member

Departmental approval: 
Polyvios Pratikakis
Assistant Professor, Director of Graduate Studies

Heraklion, October 2020

Efficient Scheduling of Concurrently Executed Network Packet Processing Applications using Heterogeneous Hardware

Abstract

Network packet processing is a field of research that has been well studied during the past decade, yet the implementation of efficient and top performing middleboxes is far from being considered trivial. The difficulties mainly derive from the multiple levels of heterogeneity that have to be addressed, such as the different types of underlying hardware architecture, each one with its own strengths and weaknesses, the diversity of the typical network applications and the interference that is observed when those are executed concurrently and compete for shared resources and the fluctuations in the network traffic rate and characteristics. In this work we identify the bottlenecks and causes of those inefficiencies and propose a scheduling schema that maps packet processing applications to heterogeneous processing devices and adjusts the mapping at real time based on the traffic fluctuations in order to sustain the best possible performance. Through the evaluation phase, we show that our system is able to detect changes and adapt in order to remain as efficient as possible in terms of throughput, latency or power consumption. Finally, we identify the limitations of our proposed system and we tackle some of them by upgrading the hardware setup and applying software optimizations. Also, we benchmark the new architecture and we show that it is capable of line rate packet processing with less power consumption and reduced end-to-end latency.

Αποδοτικός Καταμερισμός Πολλαπλών Παράλληλα Εκτελούμενων Εφαρμογών Επεξεργασίας Πακέτων του Διαδικτύου χρησιμοποιώντας Ετερογενείς Συσκευές

Περίληψη

Η επεξεργασία των πακέτων του διαδικτύου αποτελεί ένα ερευνητικό τομέα, ο οποίος έχει μελετηθεί εκτενώς την τελευταία δεκαετία. Παρ΄ όλ΄ αυτά, η αποτελεσματική υλοποίηση συστημάτων επεξεργασίας πακέτων που προσφέρουν κορυφαία απόδοση δεν είναι τετριμμένη. Οι δυσκολίες προκύπτουν από τα πολλαπλά επίπεδα ετερογένειας που πρέπει να ληφθούν υπόψη και να αντιμετωπιστούν. Ένα από αυτά είναι η διαφορετικότητα των αρχιτεκτονικών του υλικού που συνθέτουν ένα σύγχρονο υπολογιστικό σύστημα. Κάθε μία από αυτές τις αρχιτεκτονικές έχει ξεχωριστά πλεονεκτήματα αλλά και αδυναμίες. Ένα δεύτερο επίπεδο ετερογένειας είναι η διαφορετικότητα των εφαρμογών που χρησιμοποιούνται για την επεξεργασία των πακέτων καθώς και οι παρεμβολές που δημιουργούνται όταν πολλαπλές εφαρμογές εκτελούνται παράλληλα και διαμοιράζονται τους πόρους μίας συσκευής. Επίσης, οι διακυμάνσεις στα χαρακτηριστικά της κίνησης του διαδικτύου και στην ταχύτητα μετάδοσης των πακέτων αποτελούν ένα τρίτο επίπεδο ετερογένειας σε τέτοιου είδους συστήματα. Σε αυτήν τη δουλειά εντοπίσαμε τους κύριους λόγους που οδηγούν σε μη-αποδοτικές υλοποιήσεις και προτείνουμε ένα σύστημα προγραμματισμού και ανάθεσης των εφαρμογών στις κατάλληλες συσκευές έτσι ώστε να επιτευχθεί υψηλή απόδοση μέσω αποτελεσματικής χρήσης των διαθέσιμων επεξεργαστικών πόρων. Επιπροσθέτως, το προτεινόμενο σύστημα έχει την δυνατότητα να αλλάζει την χαρτογράφηση σε πραγματικό χρόνο για χάρη μιας καλύτερης, αν το κρίνει σωστό, με βάση τα χαρακτηριστικά της εισερχόμενης κίνησης. Μέσω της πειραματικής διαδικασίας, δείχνουμε ότι το σύστημά μας μπορεί όντως να εντοπίσει τις αλλαγές και να προσαρμόσει κατάλληλα την χαρτογράφηση έτσι ώστε να καταφέρει να παραμείνει όσο πιο αποτελεσματικό γίνεται, με βάση τυπικές μεθόδους μέτρησης της απόδοσης (ταχύτητα διεκπεραίωσης, καθυστέρηση, κατανάλωση ενέργειας). Τέλος, αλλάξαμε ένα μέρος του υλικού, προσαρμόσαμε την αρχιτεκτονική της προσέγγισης μας και εφαρμόσαμε ορισμένες βελτιστοποιήσεις για να λύσουμε κάποιους περιορισμούς που υπάρχουν. Μέσα από μια σειρά από εκτενή πειράματα παρατηρούμε ότι στην βελτιωμένη του εκδοχή το σύστημά μας είναι ικανό να διατηρήσει ρυθμό επεξεργασίας ίσο με τον ρυθμό άφιξης των πακέτων, ενώ παράλληλα πετυχαίνει εξαιρετικά χαμηλή καθυστέρηση και σχετικά χαμηλή ενεργειακή κατανάλωση, ανεξαρτήτως των χαρακτηριστικών της εισερχόμενης κίνησης.

Acknowledgments

First and foremost, I would first like to thank my supervisors Prof. Sotiris Ioannidis and Prof. Polyvios Pratikakis for the trust, support and guidance towards the completion of this thesis and also Prof. Xenofontas Dimitropoulos for being part of the committee and for his thoughful comments and suggestions.

Furthermore, I would like to thank Eva Papadogiannaki and Giorgos Vasiliadis for their support, feedback and helpful discussions on this thesis, as well as each and every member of the Distributed Computing Systems and Cybersecurity Laboratory not only for the collaboration, the knowledge sharing and the technical support but also for hanging out, making fun and creating a friendly working environment; it has been a pleasure to work with those people from the very first day.

Additionally, I would also like to mention the valuable contribution of my friends, who are always willing to spend their limited spare time to either hang out, take a couple of shots, have fun and enjoy the moment or to help with any problem, even in the context of this work.

Last but not least, special thanks to my family for their full and unconditional support, encouragement and willingness to contribute in every way they could when I was in need for anything related to this work or not.

This work would not have been possible without the help and the contribution of each one of the aforementioned people.

Part of this work has been published in the 2020 6th IEEE Conference on Network Softwarization (NetSoft) [34].

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Over the last decade, Internet traffic has been significantly increased. Network experts also speculate that constant increase of network traffic rate is going to be a trend for the next years, mostly due to the rising number of connected users and devices and the growth and popularity of video streaming platforms and services.

However, traditional network architectures and principles are becoming outdated and under-performing in a wide range of network situations. To overcome security and performance problems, the deployment of network middlebox processing services is ubiquitous among modern enterprises, autonomous systems (ASes), network providers and operators in general. A *middlebox* is an intermediate device (between a source host and a destination host) that performs some actions, such as monitoring, filtering and/or modification of the network packets under surveillance [25]. Those actions are typically different from common IP router functionality. Instead, a middlebox is usually deployed to enhance security (e.g by emulating an intrusion detection system (IDS) or a NAT), or to boost network performance (e.g. by acting as a load balancer).

With the continuous advance of computer architecture, modern processors tend to have more computing power by either exposing more processing units, or by being faster or both. On top of that, more heterogeneous hardware processors are becoming widely available and are quite cheap as well, allowing network specialists to explore the possibility of deploying high performance middleboxes using commercial devices.

Recent works in the field strongly support that the exploitation of modern commercial off-the-shelf (COTS) processors and accelerators can lead to highly performing systems. However, such systems tend to combine many processing devices which often remain idle or underutilized in many cases, failing to sustain peak performance or to remain efficient. Those cases are extremely common in networking systems primarily because the behavior of the input is

not constant (e.g. variable traffic rate, fluctuations in traffic characteristics, etc.) and secondarily because of the heterogeneity in either the workloads or in the underlying hardware.

This thesis aims to prove that (i) commodity heterogeneous hardware can be exploited to accelerate many typical network packet processing workloads in a highly efficient manner and (ii) proper interleaving of the execution of applications on the devices results in more efficient device utilization and higher overall system performance in multiple ways, such as increased throughput, reduced latency, etc. In order to support those statements, we design and implement a system which schedules the work to a subset of all the available processing devices (e.g. multi-core CPUs, different types of GPUs) with respect to a performance or power efficiency policy. It is able to make real-time scheduling decisions when traffic rate changes are detected to remain efficient. The major contributions of this thesis are the following:

- We extensively characterize the performance of typical software network packet processing applications, as well as the interference effects when multiple instances are executed on parallel on a variety of heterogeneous, off-the-shelf hardware devices.

- Motivated by the current gap in the state-of-the-art, we present a scheduling approach that, given a set of network packet processing applications, can effectively and efficiently utilize the most appropriate device or group of devices, based on the current system and network conditions, using a predefined policy that specifies the performance goal. The scheduler is able to dynamically respond to system and performance fluctuations and provide consistently good performance for concurrently running applications.

- We propose an optimized version of our system that is able to tackle some of its limitations, mainly regarding the maximum traffic rate it can handle and benchmark it to prove that it can provide line rate packet processing at 50 Gbps, with low latency and very good performance to power score, using only one target device, even when the most intensive network applications are being executed or network traffic characteristics exhibit large fluctuations.

The rest of the thesis is organized as follows: Chapter 2 provides the reader with essential background information about the topic of interest, while Chapter 3 lists and briefly analyzes already published works which are closely related to the topic of the current thesis. Chapter 4 extensively describe our system, the design decisions, the implementation challenges, as well as the benchmarking process, the corresponding quantitative results, its main limitations and

some suggestions for the future. In Chapter 5 we describe the optimizations we apply to further increase the performance and efficiency of our system and the benchmarks which prove our statements. Finally, Chapter 6 summarizes the main concepts that are being discussed in this thesis.

Chapter 2

Background

In this chapter we provide a summary of essential background information related to the topic that is described in this thesis. Our purpose is to make sure that even if the reader's research interests are not closely related to this topic, by reading this chapter it will become easier to comprehend all the details of this work.

2.1 Userspace I/O

Nowadays, modern, general-purpose operating systems (OSes) provide an enriched network stack, which also expresses great flexibility to the software developers. This is why its exploitation could be considered a good solution to build and run network applications, such as middleboxes, on top of it. However, the performance requirements of such applications had not been taken into consideration when OS network stack had been originally designed. Additionally, hardware advances could not but positively affect network interface cards; 10GbE and 25GbE links are both very common, while 100GbE or even 200GbE links are the cornerstone of high-end network cards. The current implementation of the network stack, which is integrated into the OS, is the bottleneck of network applications and a cause of major performance problems. In the following subsections, we explain the reasons of those problems and list a number of widely used solutions that tackle them.

2.1.1 Kernel Network Stack

First and foremost, a brief introduction on packet processing flow in the traditional network stack is needed. Figure 2.1 shows the layers through which a packet has to go before reaching its final destination (i.e. the userspace

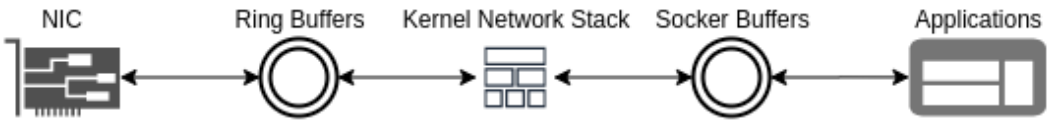


Figure 2.1: Packet flow in a traditional network stack.

application if it is a received packet). The major parts of this flow are the following:

- Upon arrival to the NIC, the packet is transferred via DMA to the ring buffer (which is managed by the kernel module of the network adapter).
- An interrupt is generated to notify the CPU that a newly available packet is ready to be handled.
- After the interrupt is handled by the OS, dynamic memory allocation occurs to provide enough space for the new packet.
- Then the packet is processed by the kernel network stack.
- After the processing procedure is completed, the packet is transferred from the kernel space to the userspace through the socket buffer.
- Finally, the application is able to access the packet though the standard POSIX calls.

The flow remains exactly the same in case of transmitted packets, the only change is the direction of the steps.

It should be obvious by now, that the traditional kernel stack has some limitations when the ultimate goal is to handle as many packets as possible. First of all, too many system calls are being generated which, in turn, lead to a large number of costly context switches from the userspace to the kernel space.[1] Second, a packet needs to be copied either from the kernel space to the userspace or vice versa. Those kind of memory copies are considered very expensive. Third, this per-packet processing approach, although being good for reducing the end-to-end latency of each packet, under-performs significantly when the goal is maximum throughput. Constantly generating interrupts, allocating memory and manipulating kernel data structures adds unnecessary overhead, which could be easily amortized.

[1]Context switching is considered expensive as it has to save/restore the context of the current running process, may, and most probably will, cause cache pollution, has to flush the TLB, etc.

Many solutions have been proposed in order to bypass the kernel and gain performance improvements by implementing packet I/O in userspace [26]. In the next two subsections, we provide background information about two different solutions, which are very popular for tackling the aforementioned problems and have been used in this thesis.

2.1.2 Netmap

The first solution, named Netmap [55], has been proposed several years ago by Luigi Rizzo. It is a framework specifically designed to address the previously discussed issues. It bypasses the kernel to reduce or even eliminate unwanted overheads, gains huge performance improvements and still remains secure. Netmap does not follow the per-packet processing approach; instead packet batching is used, so the system calls overhead can be amortized. Next, Netmap allows applications from userspace to directly access the NIC by exposing the appropriate memory address space. As a result, expensive data copies are greatly reduced. Finally, each hardware ring has a corresponding Netmap ring, which is pre-allocated and static, so the problem of dynamically allocating memory no longer exists.

A distinctive feature of Netmap is the design and implementation of an API that does not rely on specific hardware or device mechanisms, but solely on those of existing OSes. Additionally, it is very simple and easy to use, giving Netmap an edge over other similar solutions.

From the evaluation, Netmap can achieve line rate packet sink or transmit on a 10Gbps network interface, even with the smallest packet size (64 Bytes).

2.1.3 DPDK

Intel's Data Plane Development Kit [1] is also a framework for fast packet I/O, very similar to Netmap. A main difference is that DPDK uses more optimizations in order to achieve even higher performance, or put it in other words, consume as few clock cycles as possible to make a packet available from the NIC to the userspace and vice versa. Another difference resides in the fact that drivers in DPDK run completely in userspace, while the corresponding modules in Netmap still run in kernel space. Furthermore, DPDK uses polling mode drivers instead of interrupt-based Netmap drivers. This is a double-edged sword as it increases performance when packets burst, but keeps the CPU absurdly busy even when traffic rate drops significantly.

Evaluation result shows that DPDK can easily achieve line rate performance on either 10Gbps or 40Gbps network interfaces. In general, when maximum performance is desired, DPDK seems to be superior to Netmap [32]. In

addition, DPDK is scalable, more flexible and enriched with more features. However, DPDK has a steep learning curve, which makes Netmap an equally good solution if reasonable performance is required without investing too much time and effort.

2.2 Types of Processing Devices

Turning the clock back to the early 2000s, researchers and computing industry realized that techniques like increasing clock frequency and cache sizes in order to gain computation speedups were about to hit the wall. Although transistors continued getting smaller and more dense, their power density stopped being constant due to current leakage and chip heating problems, leading the silicon industry to a so-called powerwall [31]. To tackle this problem, a minor but important field of research, named **parallel computing**, had to be explored. Parallelism quickly dominated the market and affected the way software was, and still is, designed.

However, not a single way to achieve parallelism exists, and not every way is examined by this thesis. In general, parallelism is divided into four levels or categories, bit-level, instruction-level, task-level and data-level [27]. **Packet processing**, which is the main focus of this work, is achievable by taking action on each packet independently from the others, without any restriction apart from one; at the end of the processing, the ordering of the packets of each different flow shall not be changed. Having those details in mind, packet processing falls into the categories of primarily data parallel problems and secondarily task parallel problems. We only focus on those two levels of parallelism and on processing devices that can be exploited to accelerate such tasks as efficiently as possible. Those devices are multi-core processors and graphics processors. Note that there are more device types that could be exploited for the specific task, such as FPGAs and ASICs, or even embedded solutions on network cards but we solely want to focus on cheap, off-the-self, commercial devices that are commonly used and fairly easy to be programmed.

To better understand why the aforementioned types of devices fit for the packet processing model, some information regarding the architectural characteristics, the benefits and the disadvantages is necessary.

2.2.1 CPU

Central Processing Unit is the core of any modern machine, whether it is a desktop, a laptop, a server, a mobile or even a wearable or a smart device. The range and the diversity of applications that will be hosted on such systems

is extremely wide. For that matter, CPUs are designed to be general purpose processors able to ideally run any common application while having tolerable performance. For more than a decade, CPU architectures show constant improvements, despite the previously mentioned powerwall, due to two main reasons.

Firstly, architectural advances such as branch prediction, out-of-order execution and super-scalar have all contributed in boosting per-core performance. However, in this work there is no intention to put more focus on intra-core advances.

Secondarily, but most importantly, each chip nowadays packs many physical cores, thus allowing task and even data level parallelism. A chip that is packed with more than one cores can run more applications in parallel and do it in a completely independent manner, or run a single application on more than one core to promote data parallelism, if the application allows it [43]. Generally, packing more physical cores implies that there are more processing units available (e.g. ALUs, FPUs, SIMD units), which also implies higher performance. Modern CPUs pack anywhere from four cores (mid-end devices) to even half hundred physical cores (server devices).

On top of that, Intel introduced Hyper-Threading technology some years ago, which is a more efficient way of scheduling core resources across the applications. Each core is able to run up to two threads at the same time, sharing its resources between them when this is possible which leads to improved utilization and throughput [4].

Moreover, each CPU also integrates a memory controller, which enables it to directly access the main memory of the system; this is an extremely important aspect. Previous works may have managed to avoid redundant copies when performing network I/O (see Subsections 2.1.2 and 2.1.3), but the packets still reside in the main memory of a computer, so fast access is vital, especially when it comes to latency-sensitive tasks.

As ideal as those advantages may seem, they come at the price of increasing the complexity or the area of the chip, which inevitably leads to higher power consumption. However, there are strict thermal and power constraints and by design a CPU should remain well within those envelopes. Furthermore, even if power consumption was not a problem, a CPU would still not be able to pack many more cores on the same die, due to design complications and synchronization problems; remember a CPU is a general purpose processor.

Figure 2.2 is a photograph of an Intel Core i7-4770S CPU [16]; the major internal blocks which have been described above (e.g. cores, memory controller, etc.) are depicted. Note that there is a huge block, named Processor Graphics, which has not been mentioned yet, however, you can find more details in Subsection 2.2.2.

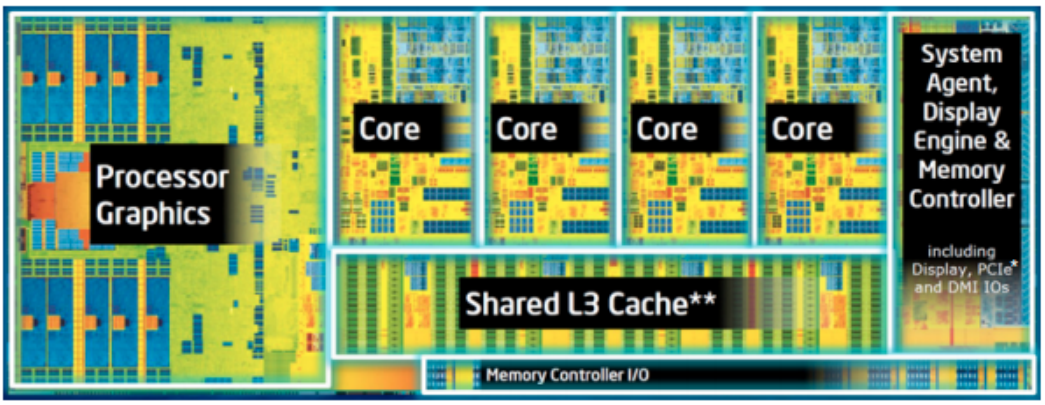


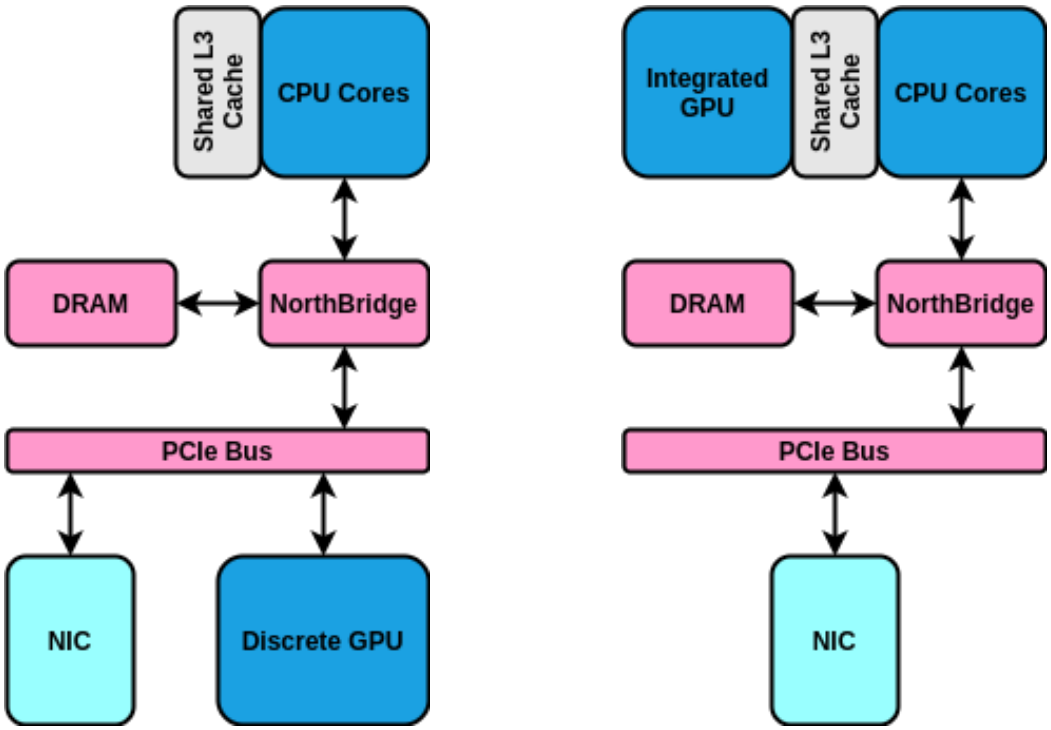
Figure 2.2: Photograph of the internal blocks of a CPU.

Overall, modern CPUs, especially high-end devices, pack so many processing units and it is extremely easy and flexible to develop applications based on them, making them ideal devices for packet processing frameworks.

2.2.2 GPU

GPU, which is the abbreviation for Graphics Processing Unit, has become an integral part of a computer nowadays, regardless of whether it is a mainstream or a server setup. Fundamentally, GPUs are built based on a totally different philosophy than CPUs, however they do share some common design parts and principles. A GPU is yet another processor, but it is made up of many smaller, more specialized cores compared to the CPUs and its own VRAM (or global memory), which is optimized to offer very high bandwidth. They first appeared as a response to graphically intense workloads that put a burden on the CPU and degraded the performance of the computer. GPUs became the standard solution to offload those workloads from the CPU, but as the amount of graphics workloads increased and so did their demands, those processors quickly became extremely powerful in performing mathematical calculations for other purposes as well apart from the initial one, which is rendering.

It is a fact that a modern GPU offers tremendous processing power, resulting in outstanding performance and it is reasonable to explore the benefits of this type of devices and how packet processing operations could be accelerated. However, GPUs have some weaknesses that cannot be neglected. First of all, a GPU is not a standalone device, it needs to be hosted by a conventional machine and communicate with its CPU, so the latter is able to dispatch jobs to the former. On top of that, it is the CPU's responsibility to also instruct a copy of the input data from the main memory of the system to the global



(a) Discrete-GPU-based system. (b) Integrated-GPU-based system.

Figure 2.3: High level comparison of different system setups.

memory of the GPU, as the GPU itself has no read or write access to the RAM. Matter of fact, the GPU completely ignores the existence of any other peripherals. Although the necessary interconnection between the two devices is achieved via the PCIe bus, it comes at the price of slow and expensive copies. In Subfigure 2.3 (a), there is a high level overview of how a discrete GPU is interconnected to the rest of the system. Many times, even on embarrassingly parallel problems, if the computation is really simple, the cost of communication (launch the GPU and copy the data) is higher than the cost of the computation itself, so GPU offloading may be meaningless or even worse, it may affect the performance negatively. Another disadvantage of GPUs is power consumption, as a modern graphics card can easily consume four times more energy than a CPU, if loaded. However, if it is underutilized due to either code inefficiencies or lack of tasks, the amount of meaningful work over time does not justify the high energy expenditure.

Research sector and industry both realized that GPUs are widely deployed and tried to propose another intermediate solution to tackle or at least mitigate the negative effects of data copying and power consumption. Instead of having

a large discrete graphics card always active, they integrated a small GPU into the CPU die (see 2.3 (b)). The advantages of this solution are multiple. First of all, a customer is no longer forced to buy a discrete graphics card for simple rendering processes. Secondly, the integrated solution may be less powerful than its discrete competitor, but it is also less energy hungry. Thirdly, it does not have dedicated memory, instead it uses the same shared L3 cache that the CPU cores use as well and is also mapped to the main memory of the system and shares it with the CPU, which is not necessarily negative because there is no more need for those expensive data copies over the PCIe bus. Finally, even in the presence of a discrete card, depending on the complexity and the quantity of tasks, the offload can take place on either the integrated or the discrete card. However, this flexibility adds one more level of heterogeneity on the systems and one more concern to the programmer. Overall, an integrated GPU is not an alternative to a discrete GPU, by any means, but more of an intermediate solution and a device that is designed to support and slightly offload the CPU cores when there are simple, mainly rendering, jobs.

2.2.2.1 GPU Architecture

Figure 2.4 reveals a detailed overview of the GF100 architecture, designed by Nvidia [68]. Although this architecture is a bit outdated, the key features are well depicted. The hierarchical organization of each modern GPU (see Figure 2.4) is the following:

- First of all, there is the GigaThread engine, which actually manages all the work that is going on.
- The biggest block is the GPC, or Graphics Processor Cluster. Each GPU is partitioned into multiple GPCs; they can best be described as standalone graphics chips in the sense that even if a GPU only had a single GPC, it would still be functional but less powerful.
- Each GPC contains multiple Streaming Multiprocessors (SMs) which are the components that perform the actual computations. Each one of them has its own control units, set of hardware registers, caches and execution pipelines.
- Each SM consists of an instruction buffer, two Warp Schedulers and Dispatch Units, a common Register File of several thousand entries and the so-called CUDA cores[2]; the exact number depends on the architecture,

[2]The names Warp and CUDA are used by Nvidia to refer to those parts within an SM, other vendors may just call them otherwise (e.g processing cores or GPU cores instead of CUDA cores, but the meaning and the functionality is the same.

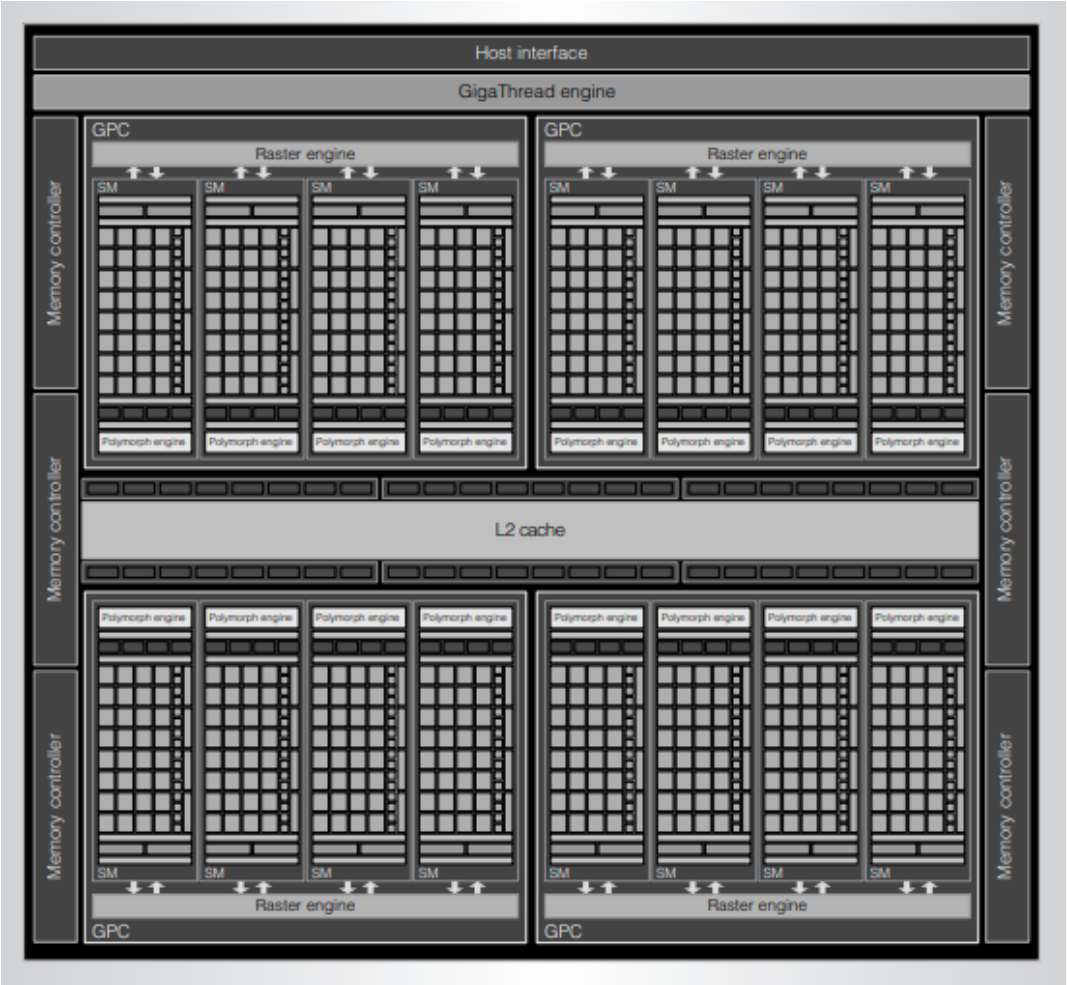


Figure 2.4: Detailed overview of the internal blocks of a GPU.

but typical numbers in modern GPUs are 32, 64, or even as many as 128 CUDA cores per SM [17]. A core is the basic compute unit, it has an ALU and a floating point unit and is responsible for hosting and executing the instructions of GPU threads, which are the basic abstract entities in the GPU programming model (discussed below).

- Finally, each GPU contains its own global memory (VRAM) and several memory controllers around the GPCs. The relationship between the VRAM and the GPCs is analogous to that of the DRAM and the CPU.

After reading the following subsection, it would hopefully be clear how the hardware and the software work together to accelerate parallel problems while also handling massive amounts of data and computations.

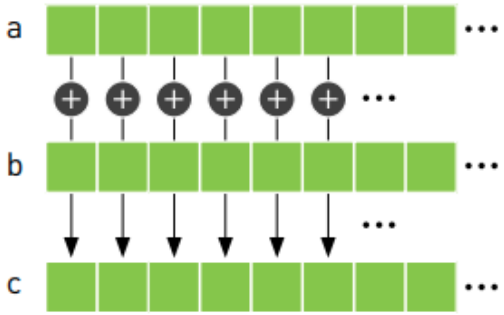


Figure 2.5: Summing two vectors.

2.2.2.2 GPU Programming Model

In order to understand why GPUs have such great impact on the performance, it is mandatory to explain how those features work together to achieve the desired data-level parallelism. There is a different programming model based on dummy threads, compared to the CPU, where all the program logic is managed by the CPU cores. We decide to use Nvidia's CUDA [50] paradigm to explain how this model works and why it boosts data-level parallelism.

We will contrive an example to illustrate threads, the basic software entity in this programming model, which is called SIMT (Single Instruction Multiple Threads) and is closely related to the SIMD (Single Instruction Multiple Data) model. Imagine having two vectors (arrays of integer values) where we want to sum corresponding elements of each array into a third array (see Figure 2.5). If this job was to be accomplished by a CPU, the source code would most probably look like the top part of Figure 2.6, while on the other hand, the corresponding GPU code would look like the bottom part of the same figure.

Such functions in GPU terminology are called **kernels** which are actually sets of simple C-style operations that are executed on the target device (the GPU in that case). Note that there is no loop inside the kernel, although it can be executed in parallel with the aid of CUDA threads. A **thread** is an abstract entity that represents the execution of the kernel. Threads are like dummy workers, they all run the same kernel (add) but each thread has a unique id, which is used to index the vectors, calculate memory address locations and take control decisions in order to complete the job without interference or synchronization issues between different threads. However, the hierarchy is more complex as multiple threads are organized in thread blocks by either manual intervention from the programmer for optimization reasons or by the compiler, which tries to find automatically the best working number of blocks and number of threads per block. A **thread block** is a set of concurrently

```
void add(int *a, int *b; int *c, int vsize) {
        for (int i = 0; i < vsize; i++)
                c[i] = a[i] + b[i];
}
```

Listing 2.1: CPU code

```
__global__ void add(int *a, int *b, int *c) {
        int idx = threadIdx.x + blockIdx.x * blockDim.x;
        c[idx] = a[idx] + b[idx];
}
```

Listing 2.2: GPU code

Figure 2.6: Simple source code example for summing two vectors.

executing threads that can share memory and cooperate among themselves through barrier synchronization. Each thread block has its own id (just like threads) which is unique within its block grid. Blocks are independent and must be able to execute in any order. A **block grid** is an array of thread blocks that execute the same kernel, read input data from global memory and write output results back and synchronize between dependent kernel calls, if needed. CUDA's hierarchy of threads maps to the hierarchy of the hardware of the GPU. One or more kernel grids are executed on a GPU at the same time, each SM executes one or more thread blocks and CUDA cores execute threads of the same thread block in parallel. Finally, within the SMs, the basic execution unit is the **warp**, which is a set of threads from the same thread block, which are launched together and execute the same instruction. Warps in modern implementations pack 32 threads and each SM integrated two warp schedulers, so two warps can be active at any time within the same SM. The SM implements a zero-overhead warp scheduler, so warps whose next instruction has its operands ready for consumption are eligible for execution, while warps whose next instruction need to wait for its operands, due to memory accesses, are not eligible for execution. In order to keep the cores busy all the time and amortize the cost of memory access overhead, the warp scheduler performs warp switches and schedules ready warps over the waiting ones [18].

While programmers can generally ignore the existence of warps for functional correctness and think of programming a single thread, it would be smarter if they manage to (i) split threads across blocks in multiples of 32 (ii) have threads in a warp execute the same code path, which means to minimize or even eliminate branches inside the source code and (iii) access memory

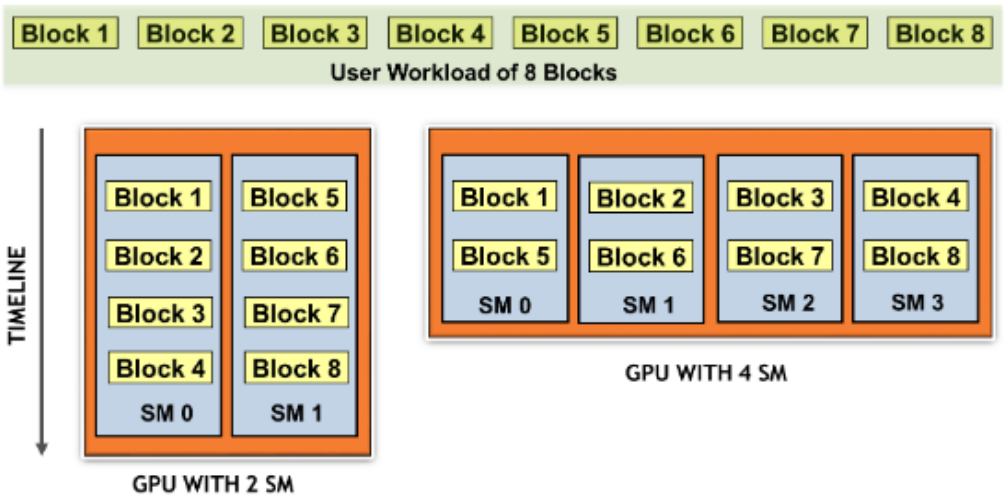


Figure 2.7: Correct workload distribution and its performance benefits on two different devices.

in nearby addresses. Figure 2.7 shows how the same workload distribution in threads and blocks, if it has been correctly performed, can scale out in a more powerful device and exploit its resources in order to reduce the overall execution time. So back to our example, to see how thread and block distribution can affect how the code will run on the GPU resources.

Imagine that the size of each vector is 1024, so we want to add two vectors of 1024 elements and save the result in a vector of equal size. Let us assume that each kernel will need to execute 10 GPU instructions until it is run to completion. In the first scenario, we will split the computation across 64 blocks of 16 threads each and suppose that our target device has only one SM for simplicity. The SM will create 64 warps of 32 threads each (1 warp for each block) and even though in each warp only half of its threads will be doing useful work, warps cannot migrate because each warp is responsible for threads from different blocks, which by design must be independent and should not be able to communicate. So, in this first scenario, a GPU with one SM, will need to execute 20480 total instructions (64 warps * 32 threads * 10 instructions = 20480). Even if more SMs were available or each SM had multiple Warp Schedulers, this would only affect the completion time of the job, not the bad utilization of GPU resources. Now, let us try to fix this first scenario by only changing the allocation of blocks and threads. We will split the computation across 16 blocks of 64 threads each this time, which will result in a total of 32 warps that need to be run (the GPU will split each of the 16 blocks in 2 warps of

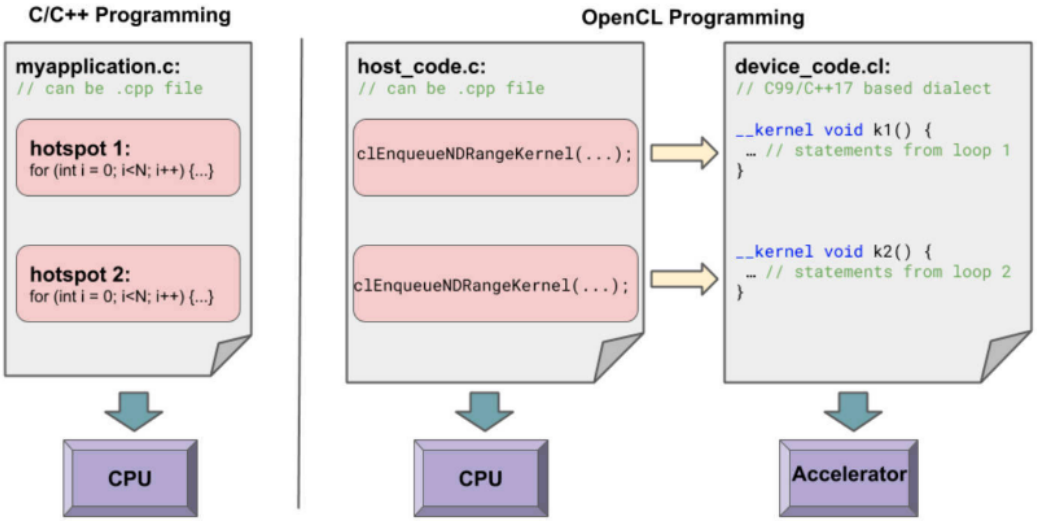


Figure 2.8: Traditional vs OpenCL programming paradigm.

32 threads). This time all threads of each warp will run in a meaningful manner, so the GPU will run a mere 10240 total instructions (32 warps * 32 threads * 10 instructions = 10240), half the number of the retired instructions of the first scenario. The same device will complete the same job two times faster in the optimized scenario, while a programmer who knows how to optimize the kernel launches will be able to scale out the computation and fully exploit any given device regardless of the number of SMs, the number of Warp schedulers, the number of CUDA cores or any other benefit or limitation the device has to offer.

2.3 OpenCL

Finally, in this section we will briefly explain the OpenCL framework and the reason for choosing to work with it. First of all, OpenCL, which stands for Open Compute Language, is a framework for writing programs that execute across heterogeneous platforms consisting of CPUs, GPUs, or basically any other accelerator (i.e. FPGAs, MICs, etc.) that supports this technology and is attached to a host processor [9]. However, in this thesis, its usage is restricted to CPUs and different types of GPUs, as previously described.

OpenCL defines a C-like language for writing programs, which is called OpenCL C and is based on C99. Programs are usually small functions, which are called kernels, that will be executed on one or more target devices. Usually kernels are the most computationally intensive parts of a bigger program and

OpenCL is used to offload them onto other devices, mostly accelerators, to enhance the overall performance of the application.

Apart from the language, OpenCL offers an application programming interface (API) which consists of two layers: (i) the platform layer that allows the programmer to run code on the host device to query, select and initialize any target device that may be present in a heterogeneous system and supports OpenCL and (ii) the runtime layer that allows the programmer to compile OpenCL kernels for a specific target device, load them on that device and execute them.

Figure 2.8 shows the differences between a traditional application that is written in C/C++ and only targets CPUs and an application that exploits OpenCL to offload its main computation to a target device. The OpenCL application is still written in a traditional language and runs partly on the host device which is the processor. The application programmer packs the application logic combined with the offloading logic in the host code but also exploits the OpenCL API in order to initialize the target device (which may be the exact same device as the host, a secondary processor, or any other accelerator), compile the OpenCL C code and finally execute it.

Yet, the only unanswered question is the reason for using OpenCL. As we described earlier, modern system setups consist of many heterogeneous devices of diverse type and compute resources. There are only two options in order to be able to target any of those devices, the traditional method, which requires too much unnecessary effort and the modern one. Using the traditional method, someone would write any application in a general purpose language, using standard compilers and only target the x86 CPUs. Then, to be able to exploit an Nvidia GPU for offloading purposes, the programmer would have to port the application to target the accelerator architecture, which means that computationally intensive parts would have to be rewritten as CUDA kernels to fit the SIMT programming model and changes in the original control logic would be mandatory in order to be able to launch those kernels. So far so good, but what if the programmer wanted to target a GPU from a different vendor that do not support CUDA, or a different CPU architecture, or a different accelerator type (e.g. FPGA). Constant application rewriting and porting is probably a very inefficient way to spend human resources; computer scientists, software engineers and developers aim to write readable, reusable, scalable and robust code. The OpenCL framework started as an attempt to tackle the majority of those problems and simplify heterogeneous computing by exposing a unified solution that targets many different underlying hardware architectures by different vendors. The maturity, flexibility and stability of this framework made it an ideal solution for the implementation of this thesis.

Chapter 3

Related Work

In this section, we briefly discuss previously published works related to the topic of this thesis, in an effort to list the most recent or the most high-impact advances on the field of interest.

Recently, heterogeneous systems have become very popular due to a substantial performance boost that GPUs provide to many individual network traffic inspection applications, such as intrusion detection [57, 62, 64, 65, 66], cryptography [39, 60, 44, 53] and IP routing [67, 38]. In addition, there have been proposed several programmable network traffic processing frameworks, such as Snap [59] and GASPP [63], that manage to simplify the development of GPU-accelerated network traffic processing applications. The literature clearly reveals an increased interest in middleboxes in general, and more precisely on acceleration techniques and their performance.

Many previously proposed approaches target a single application [52, 35, 24, 47] and try to load-balance its execution between multiple heterogeneous devices as needed, in order to efficiently increase the performance, which is not a common case nowadays. More and more solutions offer multiple application execution but some of them target homogeneous, CPU-based systems [22, 45] and in an effort to push the envelop of host processor capabilities, there is also a tendency to completely isolate a generous amount of system resources even from the host OS and dedicate them to packet processing engines [54]. Other works offer multi-device transparent execution but solely target homogeneous systems that consist only of discrete GPUs, leaving the host processor idle [41]. Additionally, there is ongoing work on providing performance predictability [29] and fair queuing [33] when running a diverse set of applications that contend for shared hardware resources and on packet routing [49] that draws power proportional to the traffic load, but all of them target homogeneous systems.

Furthermore, there have been proposed work-distribution systems that leverage load-balancing or scheduling schemes in order to choose the right placement

of multiple workloads to the appropriate devices. However, the majority of those approaches either tackle different problems instead of packet processing and fixed work size is perfectly adequate for evaluation purposes [28, 36, 58, 46], or address the packet processing problem but the evaluation methods are not ideal [61, 40]. For instance, static environment conditions, such as predefined sets of input data, are far from considered real-world network situations; only the maximum performance capacity of the system is presented under some fixed parameters that would normally fluctuate in a realistic scenario.

Finally, there are various approaches, many of them have been outlined above, which rely heavily on manual intervention or automated testing either prior to the execution, or during run-time, in order to find the best configurations and reach optimal performance, whether the measurement model is based on overall system throughout, latency, utilization, power efficiency or any other commonly used metric [52, 40, 35]. All those solutions are efficient, but when we address the problem of raw performance (i.e. how much traffic can be processed by a single middlebox at line rate), we run into approaches that try to figure out and alleviate the bottlenecks of implementing 100Gbps software packet processing frameworks [37, 51] but so far there are few, if none, concrete solutions based on commodity hardware; the majority of them are built using custom-made accelerators, like FPGAs [23, 70].

In this thesis, we approach the problem from two different angles. First, we extend a former solution [52] to create a software network packet processing framework that combines different heterogeneous devices and effectively map one or more applications to them, in an automated way, in order to provide more efficient execution in terms of throughput, latency and power consumption. Then, we tackle a major performance limitation regarding the way our system scales when the traffic rate is higher either because there are more network ports or because of contemporary network hardware that is able to deliver packets even faster.

Chapter 4

Efficiency using Workload Scheduling

Recently, multiple works related to packet processing have been proposed by the networking community, motivated by the evolution of the internet in conjunction with the evolution of hardware. The majority of these approaches often target a single computational device, such as a multi-core main processor or a powerful high-end GPU, excluding the remaining devices, leaving them completely idle. Developing a network processing application framework that can utilize multiple devices effectively, efficiently and consistently, between a wide range of diverse workloads running concurrently, is highly challenging. The main obstacle in fully utilizing a heterogeneous system, is to map the requested computations to the processing devices and have minimum interference (e.g. sharing resources that may negatively impact the performance) while also achieving it in the most automated way possible. Another challenge is to be able to handle variable traffic volumes and not making false assumptions of constant traffic, as this is not the case of a typical middlebox.

Driven by the gap in the field of packet processing to present an approach that efficiently utilizes multiple devices and at the same time take into consideration the traffic fluctuations and the lack of uniformity as regards the maximum performance of each device, we design and implement a scheduling approach for network packet processing applications that can be executed concurrently in a highly heterogeneous commodity base system. More specifically, our proposed scheduler is designed to explicitly focus on the heterogeneity that is introduced in (i) the underlying hardware architectures, (ii) the applications and (iii) the input network traffic rate. The scheduler can dynamically respond to dynamic performance fluctuations that can occur at any time during the runtime, such as traffic bursts, overloads and system changes. In the rest of the chapter, we are presenting the design and implementation details of our

system, some key points and challenges, as well as the evaluation results and its major limitations.

4.1 System Setup

First, we would like to briefly describe the hardware platform that we use during the design, implementation, testing and evaluation phase of this project. In addition, we present a unique (to the best of our knowledge) power consumption profiling tool and we also discuss the network packet processing applications, in short.

4.1.1 Hardware Setup

Our hardware setup consists of a high-end NVIDIA GeForce GTX 1080 Ti graphics card and an Intel Core i7-8700K hexa-core processor running at 3.7GHz, with enabled Hyper-Threading support that results in twelve hardware threads. The processor die is also equipped with an integrated GPU (UHD Graphics 630), which shares the 12MB L3 cache, the memory controller and the main memory (32GB dual-channel DDR4-2666 DRAM with 41.6 GB/s throughput) with the CPU cores. The GTX 1080 Ti has 3584 cores and 11GB of GDDR5X memory; it is rated at 11.34 TFlops and its Thermal Design Power (TDP) is 250 Watt. The UHD Graphics 630 has 24 execution units and its maximum performance is estimated at 423.2 GFlops (1150Mhz). The TDP for the whole processor is 95 Watt.

Overall, our machine contains three heterogeneous, commodity hardware devices: the multi-core CPU, the integrated GPU and the discrete GPU. Our three-way heterogeneous hardware setup presents an interesting trade-off: even though the integrated GPU has fewer and less powerful resources (e.g. execution units, hardware threads), when compared to a high-end discrete graphics card, the integrated GPU has satisfying performance, especially since it consumes much lower power mainly because it is directly connected to the processor and system's main memory via a fast on-chip ring bus.

Finally, our system is connected to the network via a PCIe v2.1 6-port Intel 82599ES 10 Gigabit Ethernet Adapter, with an estimated peak theoretical bandwidth of 60Gb/s. The gap between the actual and the theoretical peak bandwidth is huge mainly because of two limitations. The first one is that Netmap can handle up to four ports, so the remaining two ports of our adapter are always idle. The second one is that i7-8700K supports only up to 16 PCI Express lanes [2], so in our case those 16 lanes are shared between the set of PCIe cards (the discrete GPU and the network adapter), leaving the two

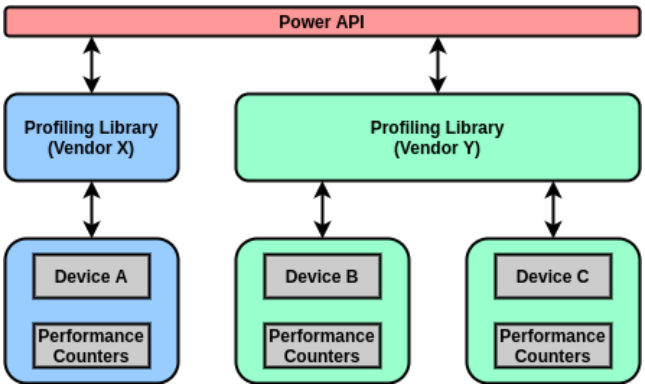


Figure 4.1: Overview of the power profiling tool.

devices with just 8 lanes each. For the discrete GPU this may not be an obstacle, because it takes advantage of the third PCIe generation, which is able to transfer up to 985 Megabytes per second per lane, but this is not the case with the older network card, which only supports the second generation of the PCIe standard and can only transmit or receive up to 500 Megabytes per second per lane [12]. Given those limitations, the network adapter acts as a bottleneck for our system, so the maximum overall throughput is limited to slightly more that 30Gbps (500MBytes/s/lane * 8 lanes * 8 bits/Byte = 32000Mbits/s = 31.25Gbits/s).

4.1.2 Power Profiling Tool

The divergence of the types and the vendors of the hardware devices introduces a problem when measuring hardware statistics, such as energy and power consumption. To overcome this obstacle, we implement a standalone tool that reads hardware performance registers of each device and periodically reports the consumed energy for a given time interval. The profiling tool is only dependent on the main hardware profiling library of each vendor, such as NVIDIA's NVML [7] and a customized version of Intel's PCM [5]. The latter does not make the API less usable, the client only needs slightly different PCM header files and a newly generated shared object file, but we combine all of them in the same bundle. The implementation of the profiling tool is straight forward. We provide an API with wrapper functions that internally decide which vendor library must be used, based on the type of the device that is being profiled. Our proposed tool is completely detached from the OpenCL framework, and thus, it can be used by any application as long as the client satisfies the mentioned requirements. Figure 4.1 presents an overview of the power API that we developed.

4.1.3 Kernel Multiplexing

In modern commodity hardware it is possible to achieve parallel execution of more than one kernels by using the OpenCL Framework. However the way of splitting hardware resources to multiple kernels is usually dependent on the type of the device. For instance, there is an OpenCL extension [8] (named device fission), which provides an interface for sub-dividing a CPU device into multiple sub-devices using many different ways. In other words, we have full control over how many and which exactly compute units will be assigned. We can also specify how many (if any) compute units will remain unused by the OpenCL framework, in order to be used by the host program.

On the other hand, with reference to the GPUs, there is not a unique way of multiplexing kernels, as for example one could create a single OpenCL context and a different command queue for each kernel and another could create a separate context with a single command queue per kernel, or even apply a hybrid solution. However the programmer has neither a fine grained way of splitting the resources of any type of GPU, in contrast with the CPU, nor any information or control over the execution of each kernel, such as how many compute units will be assigned or when context switches will occur; the hardware of the device takes care of such details. We put different, new generation GPU devices under test and found out that out of multiple kernel multiplexing solutions, the best one, in terms of performance and device utilization, is to create a discrete command queue inside a separate OpenCL device context for each different kernel. This is also the solution that leads to less interferences due to kernel multiplexing.

4.1.4 Software-based Packet Processing Applications

Taking advantage of the uniform execution that the OpenCL framework offers and its flexibility of writing kernels once and produce binaries for multiple devices, we implemented three typical packet processing applications, which are commonly deployed in network appliances and involve both processing and memory-intensive behaviors.

The first one is **deep packet inspection (DPI)** which is a very common operation in network traffic processing applications. It is used in traffic monitoring and classification tools, network intrusion detection and/or prevention systems, anti-viruses, etc. In our implementation, we use the Aho-Corasick algorithm [20] that offers simultaneous multi-pattern searching. We use 10,000 patterns of fixed strings that come from the latest signatures of Snort distribution [14]. The patterns are compiled into a DFA automaton. For our performance measurements below, we generate network traffic using the Netmap

tool, namely pkt-gen, and we inject it with content that results to around 30% match reporting.

Next one is **packet hashing**, which is used in redundancy elimination [19] and in-network caching systems [21]. Redundancy elimination systems traditionally maintain a "packet store" and a "fingerprint table". The packet store gets updated right after a packet reception, and the fingerprint table is being checked to determine whether the packet includes an important fraction of content cached in the packet store. In that case, an encoded version gets transmitted, eliminating this recently observed content. Specifically, we have implemented the MD5 algorithm. MD5 presents low probability of collisions and it is mainly used for checking data integrity [10] or deduplication [42].

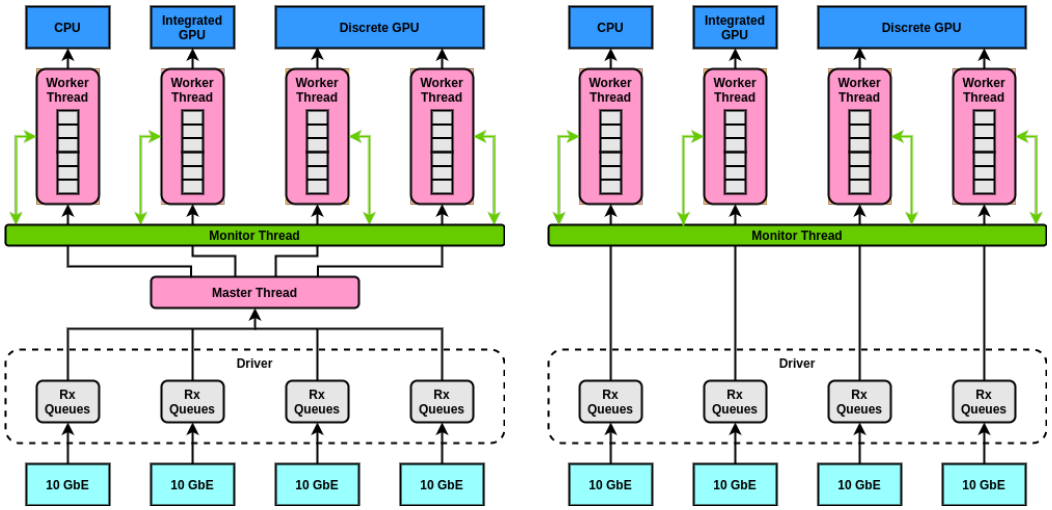
The third application that we develop is an **encryption** application using the AES-CBC mechanism with 128-bit key-size per connection. Encryption is used by protocols and services, such as SSL, VPN and IPsec, for securing connections by authenticating and encrypting the packets inside a communication channel. By employing end-to-end encryption, we can protect data flows between pairs of hosts, security gateways, etc. Due to its nature, this encryption technique is a representative form of computational-intensive packet processing workload.

4.2 System Design

Continuing with our approach, we describe two different architectural models for our system, the main differences between them and the limitations which led us to abort the first and proceed with the second design option. Furthermore, we provide implementation details and measurements to justify our design options. Finally, we explain the way that both phases (offline and online) of our approach work to deliver the desired results. Note that for the measurements and experiments of both models, we use the same Netmap application (named "pktgen") to generate custom traffic and Netmap modules to receive the generated traffic (as described in 2.1.2). We use a separate machine, which is directly connected to our testbed setup, to generate and deliver the network traffic.

4.2.1 Architecture

Firstly, we describe the two models, named "Master-Worker" and "Lock-Free" and then some generic design patterns which are common when implementing packet processing systems due to their ability to parallelize the packet processing job and amortize the cost of expensive operations.



(a) Master-worker architecture.

(b) Lock-free (master-less) architecture.

Figure 4.2: High level comparison of different architectural models.

4.2.1.1 Master-Worker

In this approach our system creates two sets of threads, with each set having a separate role. The first set of threads, the master threads, periodically poll the network interfaces, capture network packets from the driver's receiving queues and fill input data buffers. Each of those buffers is shared between a master thread and a worker thread, which is responsible for polling the corresponding data buffer and decide when there are enough packets in order to transfer it to a target device and spawn the device execution. In the meanwhile, the masters should keep filling buffers with new packets and repeat the procedure upon target device execution completion. In order to be able to achieve such pipeline, we create our buffers as follows: there is (i) an input buffer, (ii) a swap buffer and (iii) an output buffer. A master thread is responsible for filling the corresponding input buffers, then when a worker decides to spawn device execution, it swaps the contents of the input buffer to the swap buffer, it transfers the processed data from the target device back to the host program using the output buffer and finally transfers the swap buffer to the target device and spawn again its execution on the new input data. Using this structure for the buffers enables the master threads to keep capturing packets and filling buffers even when devices are busy with the processing procedure. While this is not the typical use of double buffering, it shows some performance benefits as the workers do not need to stall this pipeline of packet processing in order to wait for the capture procedure of the ingress packets. In our configuration, the

master thread is pinned in a CPU core and the utilization due to interrupts in order to capture network traffic is between 40% (if only one network interface is active) and 60% (if all four interfaces are transmitting). A single master is able to handle almost 100% of the incoming traffic basically because the PCIe limitation prevents our system from reaching its full potential. The rest of the CPU cores are either workers or perform actual processing, if CPU is also a target device. If we use a different module (e.g. DPDK) to capture network traffic and add some extra network interfaces this model introduces serious scalability problems as a single master thread is no longer capable of capturing the traffic from all network interfaces. More than one masters will be spawned and the processing capacity for the workers will decrease. The architecture is displayed in Figure 4.2 (a).

4.2.1.2 Lock-Free

A major disadvantage of the "Master-Worker" architecture is the need of synchronization between the master thread and each worker in pairs. In addition, the master thread is also in charge of handling the input traffic from the network interfaces. In our hardware setup, the master thread occupies 60% of its CPU core. However, adding even more network interfaces, would require the existence of multiple master threads that would lead to a demand of more complex synchronization between each other. Of course, scalability is of major importance for such systems. Thus, we implement a second model, with worker threads that do not rely on masters in order to consume the incoming network packets. In this new approach each worker is responsible for three main procedures: (i) capture network traffic from a discrete set of network interfaces, (ii) spawn the execution on the target device and (iii) collect statistics of the most recent execution cycle of the target device. Thus, instead of relying on a master thread to collect all the received network traffic and distribute it accordingly, we bind a single worker with a separate network interface, as shown in Figure 4.2 (b). Then, the number of the workers is equal to the number of the available network interfaces. This architecture splits the incoming traffic across the main processor cores, and each one is responsible to perform all the computations for every packet it receives in a lock-free manner. Compared to the previous architecture, the "Lock-Free" model alleviates the overhead caused by the synchronization burden, which is essential to ensure the normal execution of the workers, as well as the correctness of the results. However, the main disadvantage of this model is the inability to parallelize the pipeline; a worker should capture traffic, fill the corresponding buffers, spawn the execution and wait in order to collect statistics. We could tackle this problem by the introduction of actual double buffering (i.e. bind more than one worker

to each network interface), but this would eventually lead to synchronization issues between the workers that handle the same interface and would also make the implementation of the scheduler much more complicated. To tackle this limitation, we propose an architecture optimization based on the use of DPDK (see Chapter 5) that can scale up by leveraging more that one CPU cores to receive traffic from a single interface with zero synchronization overhead as it does not require any locking mechanism.

4.2.1.3 Design Patterns

Before we present the benchmark results, we share some generic design patterns that are commonly used among network packet processing approaches, including ours too.

First of all, we briefly present **work grouping** and how its usage can affect the performance of a system, especially if there are different types of target processors or accelerators, just like in our case. In OpenCL, an instance of a kernel is called work-item and a set of multiple work-items is called work-group. We follow a packet-per-thread approach (such as other relevant works [30, 38, 65]), which means that each work-item reads at least one packet from the memory and then performs the necessary processing action(s). Different work-groups can run concurrently on different hardware cores. The management of work-groups can present an interesting performance trade-off; a large number of work-groups offers more flexibility in scheduling, something that increases the switching overhead, though. Typically, GPUs contain a significantly faster thread scheduler, thus it is recommended to spawn a large number of work-groups, since it hides the latency that is introduced by heavy memory transfers through the PCIe bus. While a group of threads waits for data consumption, another group can be scheduled for execution with little or even zero overhead. On the other hand, CPUs perform more efficiently when the number of work-groups is close to the number of the available cores. Our system tackles the problem of work grouping by applying different work-group configurations based on the active device.

Memory accesses can be critical to the overall performance sustained by our applications. For instance, read and write operations in GPUs are more effective when data are being stored in a column-major order and accessed sequentially, which means that every adjacent thread is accessing contiguous memory locations. In GPU terminology, this setting is called **memory coalescing** [50] and leads to fewer memory transactions, which is beneficial for the performance of the device. CPUs, on the other hand, require row-major access order to benefit from cache locality within each thread. These two patterns are contradictory, so we could transpose the whole packets to column-major

order to benefit from memory coalescing, only when they reside within the GPU memory. However, the overall cost of the transpose procedure pays off only when accessing the memory with small vector types (i.e. char4), while it is not amortized in any of our representative workloads when using the int4 type (similar to [52]). Besides GPU, the CPU also offers support for single-instruction-multiple-data (SIMD) units when using the int4 type, since the vectorized code is translated to SIMD instructions [56]. Therefore, we access the packets using int4 vector types in a row-major order, for both hardware architectures (i.e. the CPU and the GPU).

One more memory-related issue is that OpenCL offers the so-called **local memory**, which is a memory region that is shared by every work-item inside a work-group. This local memory is implemented as an on-chip memory on GPUs, which is much faster than the off-chip global memory. Hence, OpenCL programmers can exploit this OpenCL abstraction to target the local memory of the GPUs and improve the performance of their applications. On the other hand, CPUs do not have any similar physical memory region to be used as local memory. As a result, all memory objects located inside the local memory are mapped to sections of the global memory, a procedure that causes performance overheads. To solve this, we explicitly put data to local memory only when the computations are meant to be performed on the discrete GPU.

Furthermore, the most common design is to place network packets into buffers, which are usually large-sized compared to the typical size of a single packet and send the whole batch for processing, instead of a single packet at a time. This technique, named **batching** is essential as it can lead to huge performance gains but at the same time it can be absurdly tricky, for a number of reasons. First of all, intuitively larger batch size leads to higher throughput and vice versa, but the end-to-end latency for each packet also increases. At some point, further increases in batch size have neutral or even negative impact on the throughput and certainly negative impact on the latency and the memory footprint. Secondly, traffic rate can directly affect the speed at which a fixed-sized buffer can be filled and be ready for processing. For those reasons, it is of equal importance to carefully decide the spectrum of the size of the buffers and to create a flexible way of adjusting the buffer size at runtime as needed.

Lastly, in typical packet processing approaches, we want to avoid **packet reordering**, which means that we shall not reorder packets from the same flow and process them in a different order than the one when received though the network interfaces. However, packets from different flows can arrive in a specific order and be processed in a different one. To prevent packet reordering we could synchronize devices using a barrier, enforcing all involved devices to execute in a lockstep fashion. This approach, though, would reduce the

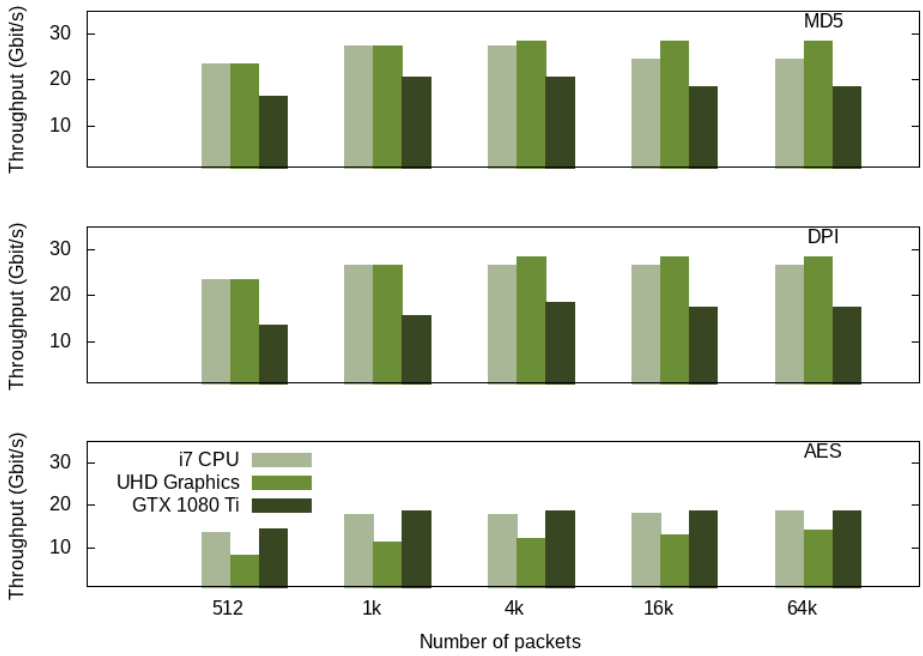


Figure 4.3: Throughput of different heterogeneous devices executing a single application each time for 1500-bytes network packets.

overall performance of a system, as fast devices would be forced to wait for the slow ones. This could be a major drawback in setups where the devices have high computational capacity differences. To bypass this problem, we firstly classify incoming packets by building the typical 5-tuple flows before creating the batches, and then we ensure that the packets that belong to the same flow will either be part of the same batch or will be placed in batches that will be processed sequentially by the same device and not simultaneously by different devices. Note that the flow classification is software-based, which causes an extra overhead that negatively affects the overall system performance. We propose an optimization to address this problem in the next chapter.

4.2.2 Performance Characterization

This section shows the overall picture of the achieved performance of different configurations running on our system. We use the term 'configuration' to describe which and how many applications are running concurrently, on which device or combination of devices and under which batch size setting. We create a pool of many different configurations and test all of them in an effort to

realize which are the dominant factors that have a major impact on the overall performance; this helps us build and optimize a scheduler that is able to take effective decisions at runtime. Note that the performance measurements and the characterization have both been conducted under the optimized "Lock-Free" architectural model; the very poor performance of the "Master-Worker" architecture especially when minimum-sized packets dominate the network traffic (as shown in [52]) is even more notable in this occasion, where the execution of multiple kernel leads to excessive resource interference. Furthermore, we use our power profiling tool (described in 4.1.2) to accurately measure the power consumption of each device that is part of our system, regardless of whether it is used by the active configuration or not. For example, when the discrete GPU is used for packet processing, the CPU is not idle, since it has to collect the necessary packets, transfer them to the device's dedicated memory, spawn the kernel for execution, and then transfer the results back to the host memory. On the other hand, when we use only the CPU (or the integrated GPU) for processing, although the discrete GPU is into an idle state, the energy expenditure of the device is not zero, so it should be taken into consideration when measuring the aggregated power consumption of the system.

In our first experiments, we run each network application on every device of our system with different batch size configurations, to justify the batch size effect in conjunction with the application and the device characteristics on the throughput of the system. Based on Figure 4.3 we observe throughput improvements as the batch size increases but different applications require a diverse batch size to reach their maximum throughput on the same device and also a single batch size is not equally good for a specific application across all different devices. Processing-intensive applications (i.e. AES) benefit more from large batch sizes, while memory-intensive applications (such as the DPI application) can reach the peak throughput using smaller batch sizes. We also claim that not only constant increments in batch size are useless from a point on but they can also even have negative impact on the throughput. For example, for the DPI we see that a working set larger than 4096 packets results in lower overall throughput for the CPU. Furthermore, increasing the batch size after the maximum throughput has been reached, results to linear increases in latency. Additionally, we also notice that the sustained throughput is not consistent across diverse devices. For instance, an integrated GPU seems to be a reasonable choice when performing MD5 and DPI on large batches of packets, compared to AES, where the integrated GPU results to low throughput. A CPU is the best option for latency-aware environments, using a small batch size. Apparently, there is not clear ranking between the devices, not even a clear winner. As a matter of fact, there are devices that can be the best fit for some applications, while at the same time, the worst option for another. If we

could give a general guideline regarding the batch size, it would be that bigger batches lead to higher throughput, but at the same time higher latency and vice versa. On top of that, if the target device uses the PCIe interconnection, batches should be larger to benefit from the processing capacity of the device, only if the traffic rate is high enough to allow the batch enlargement. The latter is extremely important when maximum throughput is the main goal and will be extensively discussed in the next chapter.

Tables 4.1, 4.2 and 4.3 summarize the results of the configuration testing and the achieved performance of each. Table 4.1 presents both the individual and the aggregated performance achieved by the DPI application, when executed either standalone or by sharing the device with 1 or 3 co-workers. The same benchmark executions are repeated for all the available devices of the system, i.e. i7-8700K, UHD graphics card and the GTX 1080 Ti. Similarly, Tables 4.2 and 4.3 display the individual and the aggregated performance achieved by the AES and MD5 applications respectively, when executed standalone or alongside some co-workers on the same devices as in Table 4.1. We note that the current implementation of our system supports the concurrent execution of every network packet processing application combined; for the purposes of simplicity though (as it is already extremely complicated to parse the results), we present only the combination of two different applications in each device at a time. When we utilize multiple devices or when we execute more than one concurrent applications, the batch of packets needs to be further split into sub-batches of different size that will be properly offloaded. We benchmark all possible combinations of packet batches, devices and applications for five times each and plot the average performance achieved for each case. In the case where the CPU processor is also a target device (i.e. performs the actual packet processing besides the packet receiving action), we include the results using all six cores (twelve logical threads) in parallel. There are other works that use single or dual Xeon processors configured with two NUMA nodes. Instead, we use a single-node desktop CPU to take advantage of the integrated GPU that is packed in the same processor die and prove that such a system can still reach satisfying performance in a very efficient way.

When executing concurrently more than one network packet processing applications in one device, we reach the challenge of unknown interference effects. These effects include but are not limited to: contention for hardware resources (e.g. shared caches, I/O interconnects, etc.), software resources, and false sharing of cache blocks. A major difference between single versus multiple concurrent application execution on the discrete GPU reveals an interesting finding. Batch size should be chosen wisely based on the number of applications concurrently executed as a large batch size (16K) has negative effects in cases where more than one applications are being offloaded to the GTX 1080 Ti. The

Table 4.1: Performance characterization of multi-device heterogeneous system using the DPI application in conjunction with different combinations of co-workers.

					Performance per Kernel				Aggregated performance		
Device	**Application**	**Buffer size**	**Co-workers**	**Co-worker**	**msec**	**Mpps**	**Gbps**	**Slow-down**	**msec**	**Mpps**	**Gbps**
GTX1080Ti	DPI	1024	1	MD5	1.0	1.099	12.9	14.6%	1.9	2.196	25.8
			3		1.7	0.588	6.9	54.3%	7.0	2.351	27.6
		4096	1		4.0	1.019	12.0	33.3%	8.0	2.038	23.9
			3		7.0	0.587	6.9	61.7%	27.9	2.346	27.6
		16384	1		22.7	0.722	8.5	49.4%	43.7	1.503	17.7
			3		41.0	0.399	4.7	72.0%	155.9	1.684	19.8
		1024	1	AES	0.9	1.133	13.3	11.9%	1.8	2.270	26.7
			3		1.7	0.588	6.9	54.3%	7.0	2.351	27.6
		4096	1		3.7	1.097	12.9	28.3%	7.5	2.195	25.8
			3		7.0	0.586	6.9	61.7%	27.9	2.346	27.6
		16384	1		26.9	0.610	7.2	57.1%	53.8	1.219	14.3
			3		39.2	0.418	4.9	70.8%	154.1	1.702	20.0
		1024	1	DPI	0.9	1.177	13.8	8.6%	1.7	2.353	27.6
			3		1.7	0.588	6.9	54.3%	7.0	2.352	27.6
		4096	1		3.7	1.112	13.1	27.2%	7.4	2.223	26.1
			3		7.0	0.587	6.9	61.7%	27.9	2.347	27.6
		16384	1		21.5	0.761	8.9	47.0%	43.1	1.521	17.9
			3		36.2	0.453	5.3	68.5%	145.4	1.803	21.2
i7-8700K	DPI	1024	1	MD5	0.9	1.204	14.1	45.8%	1.8	2.307	27.1
			3		1.8	0.567	6.7	74.2%	7.1	2.339	27.5
		4096	1		3.3	1.230	14.5	42.9%	7.1	2.318	27.2
			3		7.2	0.567	6.7	73.6%	28.3	2.323	27.3
		16384	1		13.2	1.245	14.6	42.7%	30.5	2.188	25.7
			3		28.9	0.567	6.7	73.7%	115.9	2.265	26.6
		1024	1	AES	0.6	1.586	18.6	28.5%	2.1	2.314	27.2
			3		1.5	0.689	8.1	68.8%	9.4	1.857	21.8
		4096	1		2.9	1.404	16.5	35.0%	8.3	2.171	25.5
			3		6.7	0.610	7.2	71.7%	38.4	1.776	20.9
		16384	1		12.6	1.299	15.3	40.0%	34.8	2.039	24.0
			3		29.8	0.549	6.5	74.5%	155.5	1.727	20.3
		1024	1	DPI	0.8	1.274	15.0	42.3%	1.8	2.290	26.9
			3		1.7	0.604	7.1	72.7%	7.1	2.327	27.3
		4096	1		3.3	1.230	14.5	42.9%	7.1	2.312	27.2
			3		6.8	0.601	7.1	72.0%	28.5	2.306	27.1
		16384	1		13.0	1.261	14.8	41.2%	30.9	2.178	25.6
			3		28.9	0.567	6.7	73.7%	120.5	2.180	25.6
UHDGraphics	DPI	1024	1	MD5	0.9	1.144	13.4	47.7%	1.8	2.288	26.9
			3		1.8	0.583	6.9	73.0%	7.0	2.333	27.4
		4096	1		3.5	1.176	13.8	50.0%	7.0	2.352	27.6
			3		7.0	0.588	6.9	75.0%	27.9	2.351	27.6
		16384	1		13.9	1.177	13.8	50.2%	27.8	2.353	27.7
			3		27.9	0.588	6.9	75.1%	111.4	2.352	27.6
		1024	1	AES	1.3	0.803	9.4	63.3%	2.6	1.607	18.9
			3		3.2	0.321	3.8	85.2%	12.8	1.283	15.1
		4096	1		4.3	0.963	11.3	59.1%	8.5	1.924	22.6
			3		11.2	0.367	4.3	84.4%	44.7	1.467	17.2
		16384	1		16.5	0.997	11.7	57.8%	33.0	1.990	23.4
			3		42.1	0.389	4.6	83.4%	169.2	1.549	18.2
		1024	1	DPI	0.9	1.129	13.3	48.0%	1.8	2.257	26.5
			3		1.8	0.576	6.8	73.4%	7.1	2.302	27.1
		4096	1		3.5	1.177	13.8	50.0%	7.0	2.353	27.6
			3		7.0	0.588	6.9	75.0%	27.9	2.352	27.6
		16384	1		13.9	1.177	13.8	50.2%	27.9	2.353	27.6
			3		27.9	0.588	6.9	75.1%	111.4	2.352	27.6

Table 4.2: Performance characterization of multi-device heterogeneous system using the AES application in conjunction with different combinations of co-workers.

					Performance per Kernel				Aggregated performance		
Device	**Application**	**Buffer size**	**Co-workers**	**Co-worker**	**msec**	**Mpps**	**Gbps**	**Slow-down**	**msec**	**Mpps**	**Gbps**
GTX1080Ti	AES	1024	1	MD5	0.9	1.176	13.8	26.2%	1.7	2.352	27.6
			3		1.7	0.588	6.9	63.1%	7.0	2.351	27.6
		4096	1		4.0	1.032	12.1	34.2%	7.9	2.064	24.3
			3		7.0	0.586	6.9	62.5%	27.9	2.347	27.6
		16384	1		22.9	0.715	8.4	49.4%	43.6	1.508	17.7
			3		40.9	0.400	4.7	71.7%	158.6	1.653	19.4
		1024	1	AES	0.9	1.177	13.8	26.2%	1.7	2.353	27.6
			3		1.7	0.588	6.9	63.1%	7.0	2.352	27.6
		4096	1		3.7	1.121	13.2	28.3%	7.3	2.242	26.3
			3		7.0	0.587	6.9	62.5%	27.9	2.347	27.6
		16384	1		21.6	0.758	8.9	46.4%	43.2	1.516	17.8
			3		36.2	0.452	5.3	68.1%	145.1	1.807	21.2
		1024	1	DPI	0.9	1.128	13.3	28.9%	1.9	2.249	26.4
			3		1.7	0.588	6.9	63.1%	7.0	2.350	27.6
		4096	1		3.6	1.128	13.3	27.7%	7.3	2.257	26.5
			3		7.0	0.587	6.9	62.5%	27.9	2.347	27.6
		16384	1		26.0	0.629	7.4	55.4%	52.1	1.259	14.8
			3		39.7	0.412	4.8	71.1%	158.6	1.653	19.4
i7-8700K	AES	1024	1	MD5	1.2	0.880	10.3	41.8%	1.9	2.331	27.4
			3		2.3	0.438	5.1	71.2%	7.3	2.320	27.3
		4096	1		4.8	0.854	10.0	45.4%	7.8	2.200	25.8
			3		9.4	0.439	5.2	71.6%	29.6	2.267	26.6
		16384	1		20.9	0.783	9.2	49.2%	33.5	2.085	24.5
			3		35.3	0.464	5.4	70.2%	117.3	2.265	26.6
		1024	1	AES	1.2	0.852	10.0	43.5%	2.6	1.611	18.9
			3		2.5	0.406	4.8	72.9%	10.7	1.559	18.3
		4096	1		5.0	0.827	9.7	46.7%	10.4	1.579	18.6
			3		10.5	0.391	4.6	74.9%	43.2	1.523	17.9
		16384	1		20.0	0.821	9.6	47.0%	42.0	1.563	18.4
			3		42.2	0.389	4.6	74.6%	174.7	1.506	17.7
		1024	1	DPI	1.3	0.807	9.5	46.3%	2.0	2.322	27.3
			3		2.5	0.407	4.8	72.9%	7.4	2.286	26.9
		4096	1		5.3	0.771	9.1	50.3%	8.3	2.164	25.4
			3		10.1	0.407	4.8	73.8%	30.5	2.215	26.0
		16384	1		21.4	0.764	9.0	50.3%	35.1	1.966	23.1
			3		38.3	0.428	5.0	72.4%	126.8	2.104	24.7
UHDGraphics	AES	1024	1	MD5	1.3	0.821	9.6	10.3%	2.5	1.641	19.3
			3		1.8	0.560	6.6	38.3%	7.3	2.241	26.3
		4096	1		4.4	0.931	10.9	6.0%	8.8	1.862	21.9
			3		7.1	0.576	6.8	41.4%	28.5	2.302	27.0
		16384	1		16.6	0.986	11.6	7.2%	33.2	1.972	23.2
			3		28.4	0.576	6.8	45.6%	113.8	2.303	27.1
		1024	1	AES	2.0	0.504	5.9	44.9%	4.1	1.008	11.8
			3		3.9	0.265	3.1	71.0%	15.5	1.059	12.4
		4096	1		7.3	0.560	6.6	43.1%	14.6	1.120	13.2
			3		13.9	0.296	3.5	69.8%	55.5	1.181	13.9
		16384	1		27.7	0.590	6.9	44.8%	55.6	1.178	13.8
			3		52.9	0.310	3.6	71.2%	212.3	1.235	14.5
		1024	1	DPI	1.3	0.801	9.4	12.1%	2.6	1.601	18.8
			3		1.9	0.552	6.5	39.3%	7.4	2.208	25.9
		4096	1		4.3	0.963	11.3	2.6%	8.5	1.926	22.6
			3		7.1	0.575	6.8	41.4%	28.5	2.300	27.0
		16384	1		16.5	0.993	11.7	6.4%	33.0	1.992	23.4
			3		28.5	0.576	6.8	45.6%	113.6	2.307	27.1

Table 4.3: Performance characterization of multi-device heterogeneous system using the MD5 application in conjunction with different combinations of co-workers.

					Performance per Kernel				Aggregated performance		
Device	**Application**	**Buffer size**	**Co-workers**	**Co-worker**	**msec**	**Mpps**	**Gbps**	**Slow-down**	**msec**	**Mpps**	**Gbps**
GTX1080Ti	MD5	1024	1	MD5	0.9	1.176	13.8	37.0%	1.7	2.353	27.6
			3		1.7	0.558	6.9	68.5%	7.0	2.352	27.6
		4096	1		3.5	1.159	13.6	32.3%	7.1	2.320	27.3
			3		7.0	0.587	6.9	65.7%	27.9	2.348	27.6
		16384	1		21.2	0.772	9.1	48.9%	42.4	1.545	18.2
			3		36.2	0.453	5.3	70.2%	144.9	1.810	21.3
		1024	1	AES	0.9	1.176	13.8	37.0%	1.7	2.353	27.6
			3		1.7	0.588	6.9	68.5%	7.0	2.351	27.6
		4096	1		4.0	1.034	12.2	39.3%	7.9	2.066	24.3
			3		7.0	0.587	6.9	65.7%	27.9	2.348	27.6
		16384	1		21.6	0.759	8.9	50.0%	43.7	1.499	17.6
			3		38.0	0.431	5.1	71.3%	160.1	1.639	19.3
		1024	1	DPI	1.0	1.082	12.7	42.0%	2.0	2.168	25.5
			3		1.7	0.588	6.9	68.5%	7.0	2.351	27.6
		4096	1		4.0	1.020	12.0	40.3%	8.0	2.038	23.9
			3		7.0	0.587	6.9	65.7%	27.9	2.347	27.6
		16384	1		20.9	0.783	9.2	48.3%	43.4	1.512	17.8
			3		38.9	0.421	4.9	72.5%	167.2	1.570	18.5
i7-8700K	MD5	1024	1	MD5	0.9	1.197	14.1	47.2%	1.8	2.335	27.4
			3		1.7	0.597	7.0	73.8%	7.1	2.344	27.5
		4096	1		3.5	1.174	13.8	49.5%	7.0	2.334	27.4
			3		6.7	0.610	7.2	73.6%	28.1	2.335	27.4
		16384	1		13.9	1.180	13.9	42.8%	28.4	2.306	27.1
			3		26.7	0.614	7.2	70.4%	113.9	2.307	27.1
		1024	1	AES	0.7	1.519	17.8	33.3%	2.0	2.317	27.2
			3		1.4	0.756	8.9	66.7%	9.1	1.946	22.9
		4096	1		3.0	1.379	16.2	40.7%	7.9	2.210	26.0
			3		6.2	0.665	7.8	71.4%	37.1	1.859	21.8
		16384	1		12.8	1.283	15.1	37.9%	33.7	2.066	24.3
			3		27.0	0.607	7.1	70.8%	149.7	1.810	21.3
		1024	1	DPI	0.9	1.176	13.8	48.3%	1.8	2.307	27.1
			3		1.7	0.594	7.0	73.8%	7.1	2.329	27.4
		4096	1		3.3	1.230	14.5	46.9%	7.1	2.317	27.2
			3		7.6	0.541	6.4	76.6%	28.5	2.306	27.1
		16384	1		14.5	1.130	13.3	45.3%	30.2	2.172	25.5
			3		28.4	0.578	6.8	72.0%	117.5	2.239	26.3
UHDGraphics	MD5	1024	1	MD5	0.9	1.158	13.6	49.3%	1.8	2.316	27.2
			3		1.8	0.583	6.9	74.3%	7.0	2.333	27.4
		4096	1		3.5	1.177	13.8	50.2%	7.0	2.353	27.6
			3		7.0	0.588	6.9	75.1%	27.9	2,352	27.6
		16384	1		13.9	1.177	13.8	50.2%	27.9	2.353	27.6
			3		27.9	0.588	6.9	75.1%	111.4	2.352	27.6
		1024	1	AES	1.2	0.824	9.7	63.8%	2.5	1.649	19.4
			3		3.1	0.327	3.8	85.8%	12.5	1.309	15.4
		4096	1		4.3	0.956	11.2	59.6%	8.6	1.912	22.5
			3		11.3	0.364	4.3	84.5%	45.1	1.453	17.1
		16384	1		16.6	0.986	11.6	58.1%	33.2	1.973	23.2
			3		42.3	0.387	4.6	83.4%	169.9	1.543	18.1
		1024	1	DPI	0.9	1.153	13.5	49.6%	1.8	2.306	27.1
			3		1.8	0.577	6.8	74.6%	7.1	2.309	27.1
		4096	1		3.5	1.176	13.8	50.2%	7.0	2.352	27.6
			3		7.0	0.588	6.9	75.1%	27.9	2.351	27.6
		16384	1		13.9	1.176	13.8	50.2%	27.8	2.353	27.7
			3		27.9	0.588	6.9	75.1%	111.4	2.352	27.6

reason behind this is the fact that both the discrete graphics card and the NIC share the same I/O interconnect (i.e. the PCI bus) and perform at reduced bandwidth, as explained earlier (in Subsection 4.1.1). Concurrently running application on a high-end GPU (just like the GTX 1080 Ti) and large batch sizes could be useful in the case of very dense traffic (very high traffic rate that is composed of minimum-sized packets), however this PCIe limitation should be tackled first. Another interesting fact has to do with multiple instances of AES on the same device. When this is the case, the aggregated performance is lower compared to the aggregated performance of every other kernel combination on a given device, as shown in Table 4.2. GTX 1080 Ti is an exception as it is not affected by the compute-intensive nature of AES and is able to sustain top performance even when four instances of the encryption workload are executed at the same time. On that note, despite that the integrated graphics card performs tolerably well on single AES execution when combined with a large batch size (Figure 4.3), it is the least suitable device when the desired scenario requires the concurrent execution of an AES instance alongside an instance of any other application, as shown in the bottom part of Table 4.2. Moreover, when DPI is coupled with MD5 (Tables 4.1 and 4.3), the GTX 1080 seems to be the worst fit; not only giving the worst performance but it does so by being the most energy-hungry device at the same time. The CPU and the integrated GPU both achieve similar performance results, but in the case of the UHD graphics card, top performance can be sustained regardless of the batch size. These observations lead us to a conclusion that in the presence of those two applications, the workload offloading to the integrated graphics card is a must, as it not only achieves top performance but it also leaves the CPU and the discrete GPU idle, which either promotes the energy efficiency of the system as a whole or provides room for the execution of at least one computation-intensive application, like AES, without sacrificing performance.

4.2.3 Offline Analysis

Our scheduling system consists of two phases, the offline analysis part, which we describe in this section and the online adaptive scheduling, which is analyzed in the following one. In the first phase, we implement an offline analysis tool that creates a pool of configurations, then tests each one and gathers the resulted performance statistics. Instead of creating a huge pool of configurations, we use the characterization results from the previous step (Subsection 4.2.2) to create a meaningful portion of all possible configurations in order to reduce the offline training time. By performing this offline analysis, every configuration of the system is systematically characterized and the results can be used later at real-time for deciding the most appropriate configuration

based on the corresponding conditions.

We provide an example for better understanding, so let us assume that we want to run two applications and poll two network interfaces, so each application will poll a single port. Imagine an example system with only two devices, a multi-core CPU and an integrated GPU. Those particulars provide enough knowledge, so our system can create application to device combinations (i.e. application A on device A and application B on device B, application A on device B and application B on device A, application A and B on device A, etc.) while skipping the meaningless combinations and then map each of those combinations to some effective batch size options. This process will create the pool of configurations that will then be tested for a given amount of time each, let us say 10 seconds.

During the offline analysis phase, our system collects statistics for each one of the configurations as it remains active and stores them so they can be used in the online phase. For as long as each configuration remains active (10 seconds in the previous example), the offline tool gathers the aggregated performance of the system (in terms of throughput) and stores it in a node of a data structure. Each node of this data structure represents a different system configuration, so the number of nodes is equal to the number of the configurations. In the same manner, the system creates two more instances of the data structure and stores the aggregated performance of each configuration in terms of latency and power consumption. The reason behind the decision to keep three separate instances of the same data structure and map each instance to a specific performance metric is that we implement three different performance policies (maximum performance, minimum latency and minimum power consumption) and it is more efficient to have three separate, sorted data structures to serve the requirements of each policy instead of having a single more complex data structure and sort it every time the system is ruled to change the policy at runtime.

4.2.4 Performance Policies

Since we mention the existence of performance policies, let us briefly describe each one of them before proceeding to explain how the real time scheduling works. We define and implement three different policy algorithms which we think are enough to fulfill the different needs of a potential user of our system. There is a short description of each predefined policy below.

The first policy we implement is the **maximum throughput** policy. In this scenario our scheduler should be able to activate the configuration that gives the maximum aggregated rate (in Gbits per second) at which the target devices process input data, regardless of the other performance metrics. This

is the best configuration when the user wants to maximize the throughput of the system but comes with the cost of greatly increasing the latency due to generally large-sized batches and in most of the cases the energy expenditure is also higher.

The second policy is just a contradiction of the first, as the goal is to have **minimum end-to-end latency**. In other words the user wants each packet to be processed as soon as it is captured. This kind of policy applies to latency sensitive applications that provide real-time processing as a feature. However, the main shortcoming of this policy is the inability to process large amounts of data, which means that in order to be meaningful, the traffic rate should be well below average or there should be more CPU-like devices available if the traffic rate is high.

We refer to the third and final policy as the **minimum energy consumption** policy, however its goal is not exactly to reduce the energy expenditure as much as possible. Instead, its goal is to minimize the amount of active devices in order to be able to reduce the power consumption without facing eminently performance degradation. Obviously, as a consequence, the aggregated throughput of the system is not towards the high-end. On the other hand, the latency of the system is not a major problem, when only one target device suffices and the rest are idle, especially when the active device is either the CPU or the integrated GPU.

4.2.5 Real Time Scheduling

Finally, it is time to describe the last part of our system, which is the scheduler that can detect traffic rate changes and adapt to them at real time in order to allow the system to be efficient throughout its whole execution time. At this point our system has already gone through the offline analysis phase and populated the necessary instances of data structures with the performance results of each configuration (just as described in Section 4.2.3). The user specifies the policy that best suits his needs and our system initially searches the corresponding data structure, chooses to activate the best performing configuration with respect to the user defined policy and processes incoming packets from the network. As traffic rate changes, we want our system to adapt and be able to keep processing the input data as efficient as possible. There is a dedicated **monitor thread** that periodically checks if the best configuration is still active and if it is not, it activates it. In order to achieve this, the monitor thread calls the corresponding policy function based on the current active policy. Each policy function itself, implements a scheduling algorithm that is able to detect traffic changes and force our system to adapt to continue working efficiently. To save system resources, we decide to use the main thread of our system,

which is idle after performing the offline analysis, to run the functionality of the monitor thread, instead of spawning an extra thread.

To fully understand the scheduling procedure, we present a step-by-step explanation of the scheduling algorithm below:

Step 1: Update the performance statistics of the active configuration. This update is necessary, so our system can keep training itself over time and successfully adapt to any future traffic changes regardless the size of the variance. To keep this step as efficient as possible, there is no need to perform any search operation on the data structure to update the corresponding node. Instead, we have a pointer to this node which represents the active configuration.

Step 2: Measure the current traffic rate, if no significant changes ($\leq \pm 10\%$ of previously measured traffic rate) are detected, terminate the algorithm, otherwise proceed to Step 3.

Step 3: Initiate a search operation in the corresponding sorted (by the achieved performance) data structure based on the active policy. Performance metrics are real numbers, so due to their nature, any typical search operation will not result in exact matches and will eventually fail. We modify the search operation, so it can find the configuration whose performance is as close as possible to the current traffic rate. The search operation returns a pointer to that node.

Step 4: Finally, the monitor thread is aware of the upcoming configuration change, instructs the workers to complete the active processing of the last batch, set them in an idle state and informs each one of them about the new configuration and the necessary changes that each one shall make, before setting them back in an active state again.

The generic algorithm is presented as it requires minimal changes based on each policy function. For example, when the maximum throughput policy is enabled, the algorithm will perform all the described operations on the instance of the data structure that is related with the specified policy. Furthermore, the algorithm searches for the best performing configuration based on the highest performance value in this case because we seek maximum throughput which is measured in Gigabits per second. On the other hand, if another policy was set, the algorithm would just operate on a different data structure and would choose the minimum performance value.

Although our system is highly adaptive and can detect traffic fluctuations, we do recognize that the user may need to update the way the system performs by applying a different policy under which the system will decide the workload distribution without re-initializing the whole system. We provide such a mechanism to the user by creating a custom signal handler for catching software

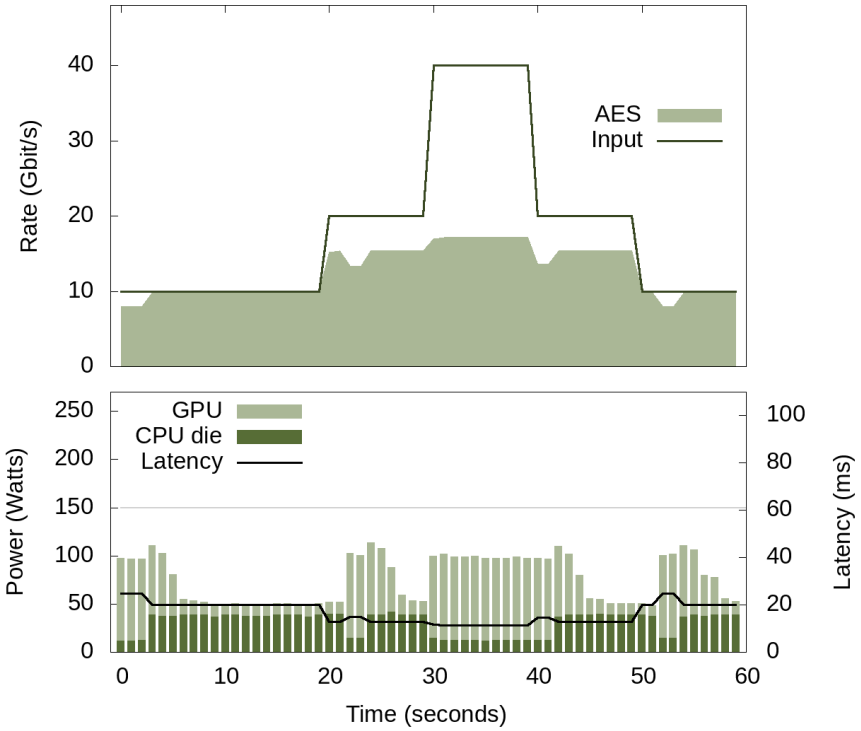


Figure 4.4: Fluctuating input rate while executing a single application (AES).

generated interrupts. The user can create an interrupt, the system will display the policy options and the user will set a new policy. When the monitor thread wakes up again, as scheduled, it will execute the appropriate policy function based on the newly set policy, so the system will forcefully change its active configuration to conform with the new scheduling instructions.

4.3 System Evaluation

In this section, we evaluate the performance of our scheduling approach by running multiple experiments in dynamic conditions. Specifically, we evaluate our scheduler using two diverse applications, AES and DPI. In Figures 4.4 – 4.7 we present the input rates, the achieved throughput, the power consumption and the latency of the device combination made by the scheduler, in dynamic conditions: i.e. (i) fluctuating incoming network traffic rate and (ii) policy change on-the-fly.

Firstly, regarding the fluctuating network traffic rate experiments, we use an energy-critical policy to handle all input traffic at maximum energy efficiency. The traffic rate is low enough for a single device to cope with it, which

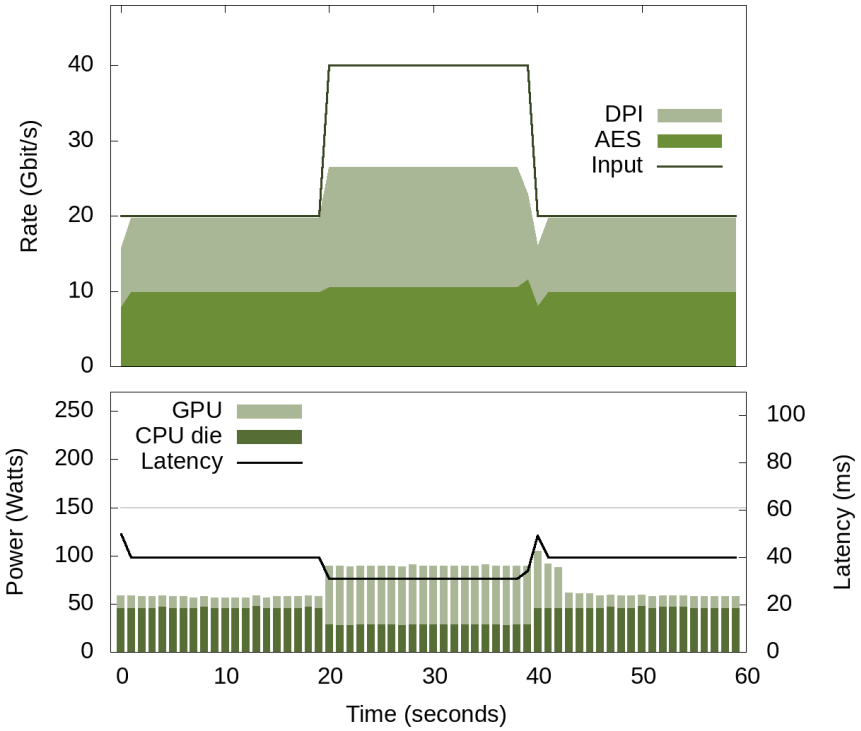


Figure 4.5: Fluctuating input rate while executing multiple applications (AES, DPI).

results to significantly low power consumption. This is not the case in the second set of experiments, where we seek the highest possible throughput before aggressively switching to an energy-efficient policy. We note that the power consumption is divided into two categories: the power consumption of the discrete GPU and that of the processor package, which also includes the power consumption of the integrated graphics card, as it is impossible to distinguish the power consumption of different parts of the same CPU die using software. For comparison, we also display with a straight gray line the maximum power consumption when both devices are exhaustively used simultaneously. Additionally, the processing latency is marked with a solid black line. The observed variability in latency is the result of dynamic scheduler decisions regarding the batching and device selection. The generated input traffic is composed by 64-Byte and 1514-Byte TCP packets at the ratio of 1 to 4.

Overall, our scheduler is capable to adapt to a highly diverse computational demand among different applications, producing live decisions that aim to fulfill the performance requirements with respect to the active policy. Additionally, the scheduler avoids selecting device combinations that lead to excessive latency or power consumption unless it is absolutely necessary.

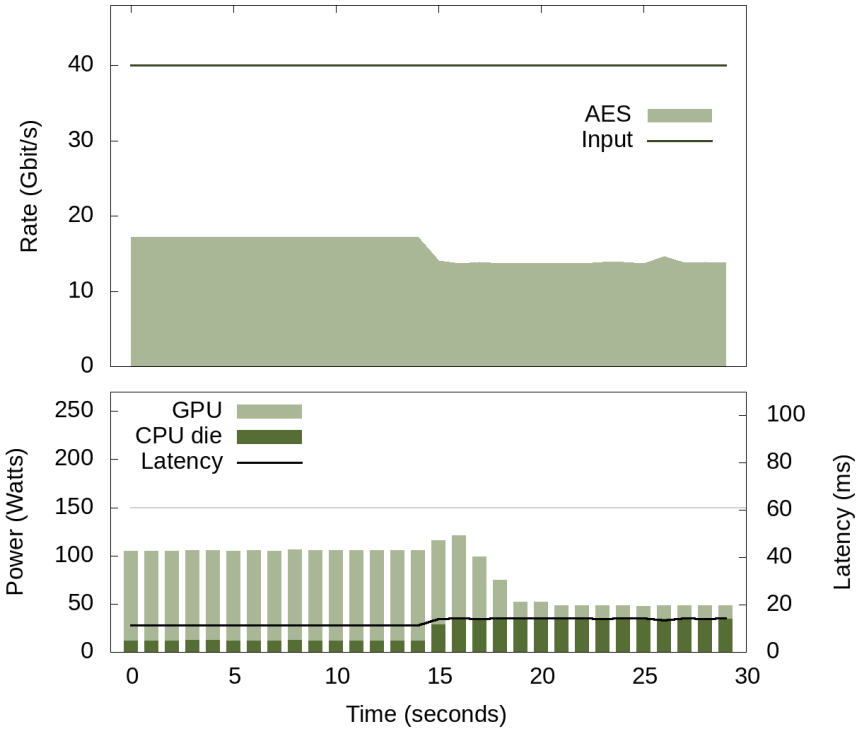


Figure 4.6: Policy change (max throughput → min power consumption) while executing a single application (AES).

4.3.1 Throughput

As we observe through our experiments, our proposed scheduler is able to choose the configuration that keeps the selected device under the processing capacity which is required to process the incoming traffic for each application. Specifically, when traffic rate is constantly high and the specified policy is to hit the maximum throughput, the system is able to process the input traffic at a rate of almost 20 Gbps, if only a single application is active and at a rate of almost 30 Gbps in the scenario of two active applications, as shown in Figures 4.6 and 4.7 respectively (between times 0 and 15 seconds). On the other hand, when the traffic rate is variable, the scheduling schema we propose manages to cope with up to 10 Gbps of input traffic rate per application by activating a single device and interleaving applications, if more that one exists, as shown in Figures 4.4 and 4.5 (at times 0 to 20). When locating changes in the traffic rate, such as the increase from 20 to 40 Gbps (Figure 4.5, a second device (in this case the GTX 1080 Ti) is enabled to increase the computational capacity of the system. An interesting area is located at times 20 to 30 of Figure 4.5 when the discrete graphics card is activated but is immediately

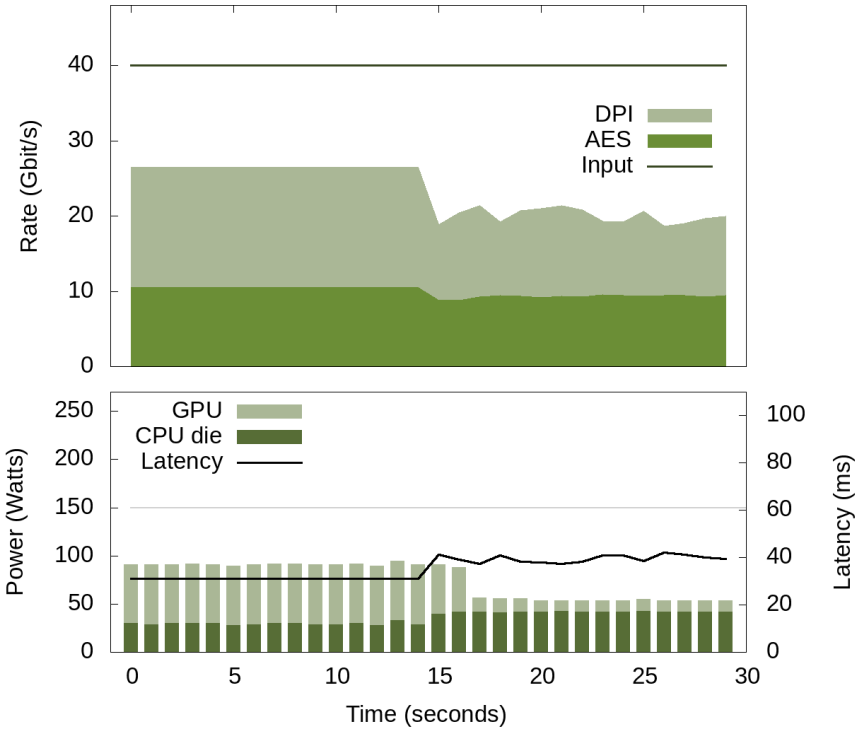


Figure 4.7: Policy change (max throughput → min power consumption) while executing multiple applications (AES, DPI).

deactivated, as the monitoring reveals that only the presence of the integrated graphics card can still cope with the incoming traffic. The GTX 1080 Ti is only re-activated when the traffic rate is doubled to 40 Gbps. Note that our system would be able to achieve higher throughput by the same active configurations if it was not for the PCIe limitation (discussed in Section 4.1.1). So from our point of view, this limitation unarguably leads to device underutilization after a fashion, but at the same time we also prove that our system is able to take the correct scheduling decision no matter of the circumstances.

4.3.2 Power Consumption

The more working devices lead to higher power consumption and vice versa. That is the reason behind the decision to design a system that creates configurations with the least possible number of devices needed to meet the requested requirements and constantly adapts by activating them as necessary. In Figures 4.6 and 4.7 at the 15-second mark the system switches from highest throughput to lowest power consumption policy, thus is shuts down the power-hungry GTX 1080 Ti, resulting in a severe drop in the overall energy usage.

It is also clear from the results of variable traffic rate experiments (Figures 4.4 and 4.5 that only when a severe increase in the traffic rate occurs and more computational capacity is needed, the system activates an extra device (the GTX 1080 Ti in this case) at the cost of greater energy expenditure.

4.3.3 Latency

As observed, increasing the batch size results to higher throughput rates; however, at the cost of increased latency, especially in the case of the discrete GPU. However, we try to minimize latency up to a point where no interference with the requested policy occurs. For example, even when the goal is to maximize the overall throughput of the system, like in Figures 4.6 and 4.7, during the first 15-seconds interval, latency remains considerably low despite the fact that the discrete GPU is active as the traffic characteristics demands so. This is mainly because the system does recognize that an even larger batch size would not result in extra performance gains. After forcefully changing the policy (same figures, second 15-seconds interval), the throughput drops and the latency is slightly increased as a natural consequence of applying a contradictory policy, but not to any point that could characterize the performance of either metric unacceptable.

4.3.4 Traditional Performance Metrics

Besides the above performance metrics, we also take account other important performance metrics in the domain of networking systems, i.e. reordering and packet loss. We describe how we prevent packet reordering (which can occur when packets that belong in a single flow get split and distributed to different devices) in Section 4.2.1.3. Furthermore, packet loss can occur in the case of slow device switches due to a scheduling decision that aims to adapt to reduced or increased traffic rate. Our scheduler though, is able to promptly adapt to changes in less than 60ms, resulting to no packet losses, even in cases where the input traffic rate increases from 20Gbps to 40Gbps. Note that we do not address the problem of packet losses because of slower than line rate packet processing here; we only claim that the required time for a configuration change is pretty small, so there are no further packet loss incidents.

4.4 System Limitations

One final comment on this system relates to its known limitations, which are (i) the limited ability to receive and process slightly more than 30Gbps due

to the PCIe interconnection problems, as described in Subsection 4.1.1 and (ii) its scalability. In the following chapter, we address the former limitation and extensively describe the hardware and design changes and a series of optimizations we use to tackle the problem. Regarding the latter, adding more network applications, network interfaces, target devices or a combination of those, results in an exponential increase in the configuration space that should be explored during the offline phase and in complicated changes in the scheduling algorithm that may lead to inefficiencies. This problem could be tackled by using a pre-trained machine learning model to predict the best performing configuration and completely detach the offline phase from the system. We plan to explore that solution in the future.

Chapter 5

Performance Optimizations

Although the previously described system is able to execute multiple applications with respect to the user-set policy and sustain high performance while taking into consideration other important metrics, we believe that there is more room for improvement. In this chapter, we address some of the previously discussed limitations by improving the underlying hardware and making the necessary software adjustments and applying some code optimizations.

We describe the necessary changes in the design of our systems, which combined with a new hardware setup completely eliminate the packet I/O limitations. After applying optimizations, our system can uninterruptedly handle 50Gbps of input traffic and offload it for further processing as needed. Furthermore, the system is no longer bound to 10Gbps network interfaces. It can receive traffic at line rate from ports with 25, 40 or even high-end 100 Gbps link speeds by just leveraging more CPU cores in a scalable way.

Finally, we present some initial benchmarking results based on the optimized system to back up our statements.

5.1 System Setup

This section outlines the new hardware setup of our system, which is completely different compared to the setup we used to develop and evaluate our scheduling approach in Chapter 4. Some key characteristics that we mention in this section are (i) the hardware advances and how they contribute to the elimination of some previously discussed limitations, (ii) the differences in the way of accessing the main memory of this system and (iii) the cache access optimizations.

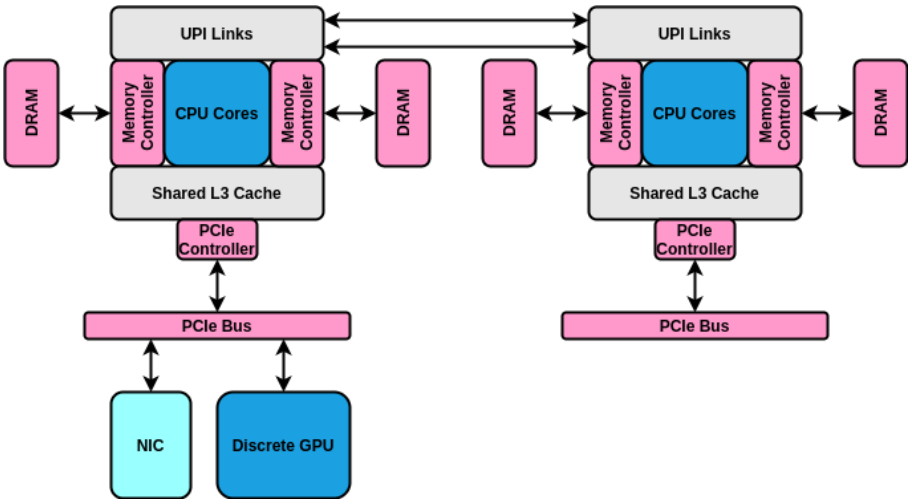


Figure 5.1: High level overview of a dual processor (NUMA) system setup.

5.1.1 Hardware Setup

This new setup also consists of commercial products (just like the first), yet many of them are mostly intended for server use, so the architecture of those products is slightly changed to target the performance needs of server-like applications. In detail, this system consists of two Intel Xeon Gold 5218 processors running at a base frequency of 2.3GHz (or at a higher frequency, up to 3.9GHz, if needed, using Intel Turbo Boost technology); each one packs 16 physical cores and 32 threads, when Intel Hyper-Threading technology is enabled, which all share a huge 22MB last level cache. Each processor chip is also equipped with two memory controllers and has 16GB DDR4-2666 DRAM installed per controller, which results in 64GB of total main memory, with an estimated peak throughput of almost 80GB/s. On top of that, each CPU die has two UPI links and the corresponding controllers, which provide the necessary intercommunication between the two processors. In addition, each processor is connected to 48 PCIe lanes via the PCIe controller, which can potentially be exploited to massively increase either raw I/O throughput, or raw computational capacity by installing powerful accelerators or both. However, those processors lack the computational capacity that an integrated GPU offers and the TDP is also higher (at 125 Watt) compared to a desktop processor. Furthermore, we replace the discrete graphics card by a higher processing capacity NVIDIA GeForce RTX 2080 Ti which has 4352 CUDA cores, 11GB of GDDR6 global memory and is rated at 13.5 TFlops, a 20% increase compared to the GTX 1080 Ti, while also having the same TDP (at 250 Watt). Lastly, we use the same network adapter, just like we described in our first approach,

to connect our system to the network. However, this time we use the DPDK instead of the Netmap framework and our system has no hardware limitations regarding the PCIe lanes. It is extremely important to mention that both the network adapter and the discrete GPU are connected to the same CPU die, as shown in Figure 5.1; the reason of importance is explained in the next subsection. It is obvious that there is enough room for at least one more network adapter and one more accelerator. We discuss in detail later on how the availability in expansion slots and PCIe lanes in conjunction with the presence of multiple multicore processors can lead in predictable performance and almost linear increase of the aggregated throughput.

The same typical network applications (Hashing, Encryption/Decryption and Deep Packet Inspection) are being used in order to test and conduct the benchmarking procedure; a description of each workload can be found in Subsection 4.1.4. We also note that the only target device is the high-end discrete GPU, so kernel multiplexing is also achieved as described earlier in Subsection 4.1.3 and power consumption measurements are being performed by exploiting our profiling tool (see Subsection 4.1.2).

5.1.2 Non Uniform Memory Access

In modern, server computer architectures, two or more processor chips can be placed on the same board, acting as a single unit for the end user, as shown in Figure 5.1. This setup is commonly known as NUMA, or Non-Uniform Memory Access. NUMA is a shared memory architecture that describes the placement of main memory modules with respect to the processor chips that are installed in the system. In the NUMA shared memory architecture, each processor has its own memory controllers, which are embedded in the same package as the CPU and its own local memory module that can access directly with a distinctive performance advantage. At the same time, it can also access any memory module belonging to another processor using a shared bus (or some other type of interconnect), the UPI links in our case, which are part of the Intel QuickPath Interconnect (QPI) bus [6]. What gives NUMA its name is that memory access time varies with the location of the data to be accessed. If data resides in local memory, access is fast but if it resides in remote memory, access is slower. The advantage of the NUMA architecture as a hierarchical shared memory scheme is its potential to improve average case access time through the introduction of fast, local memory, while increasing the aggregated memory bandwidth and the available memory of a system. However, it is critically important to allocate and access local memory and avoid costly remote accesses, in order to benefit the most from this memory architecture [11, 15].

On that note, it should be clear why it is also important to place the network I/O device and the target accelerator (i.e. the discrete GPU) on the same CPU node, as shown in Figure 5.1. When receiving network packets, data is transferred from the NIC's memory to the host's memory via DMA transactions. Then the host processor places them into batches, which also reside in the main memory, in order to later offload any packet processing to the GPU or any other target device. If the network card is placed on a different node than the GPU, then in order to offload the computation, the two CPU nodes will inevitably communicate and exchange data, which results in many, unnecessary remote memory accesses and significant performance penalties, such as saturation of the Intel QPI bus, pollution of caches between CPUs, extra latency to the memory fetch, increase in CPU resource utilization and increase in power consumption. On top of that, the performance will neither be predictable due to the CPU intercommunication interference, nor should we expect linear throughput increase when the second CPU socket is populated with more devices.

Finally, it is equally important to adjust the way we use CPU cores to poll the network interfaces for ingress packets. If the mapping of CPU cores to the network ports is done in a NUMA-aware way, none of the aforementioned problems will emerge and the performance will be optimal. On the other way, if the application developer randomly assigns cores to poll the network interfaces or even worse, if the underlying OS is allowed to take that decision, the application will suffer from sub-par performance. Before we benchmark the performance of our new setup, we put the needed effort to tackle all of those NUMA-related problems.

5.1.3 Direct Data Input/Output (DDIO)

Up to this point, the main memory was the primary source and destination of the network packets instead of the scarce resource of cache, which resulted in an excessive number of trips to the main memory subsystem for data consumed and/or delivered by I/O devices (i.e. the network interfaces in our case). Those trips loaded the memory subsystem up to five times the link speeds, forcing the CPU and the I/O subsystem to run slower and also consume more power. With the arrival of new generation Intel server processors, the environment has changed, as those processors support up to 20MB of last level cache (LLC), maybe even more in some cases, so LLC resources are no longer scarce. Intel has updated the architecture of the Intel Xeon family of processors to remove the inefficiencies of the classic model by enabling direct communication between its own Ethernet controllers and adapters and the host processor cache. The elimination of frequent visits to the main memory subsystem results in reduced

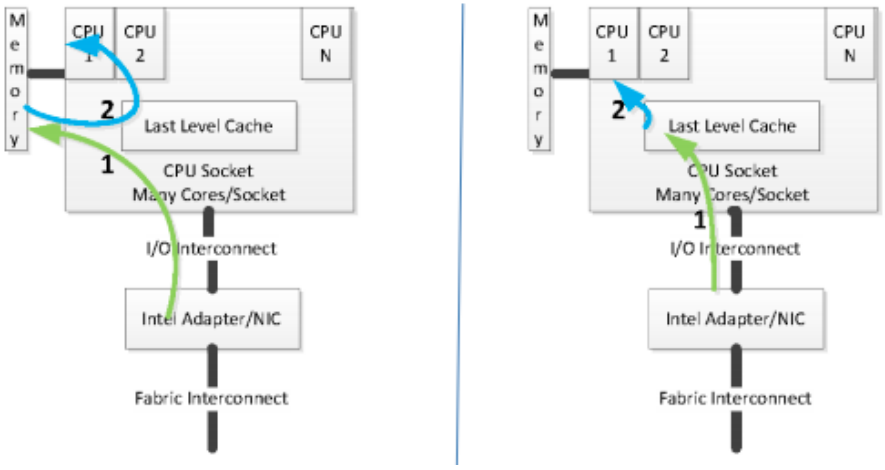


Figure 5.2: Differences in steps taken when a packet is received in the NIC using common hardware (left) versus Intel DDIO enabled hardware (right).

power consumption, greater I/O bandwidth scalability and also lower latency. Intel introduced this new and efficient platform technology under the name DDIO [3], which stands for Direct Data Input/Output.

Intel DDIO functionality is best summarized by studying data operations (either read or write) from a network device. Specifically, an I/O read operation is initiated when a NIC performs a transmit operation and a write operation is initiated when a network packet is received and has to be transferred along with its control structures to the host memory for further processing. Figure 5.2 illustrates the differences in the sequence of steps that occur for I/O write operations on systems with and without Intel direct data I/O technology. In the left, data is delivered without Intel DDIO technology and the required steps are the following:

Step 1: Data delivery operations have the NIC transferring data (packets or control) to host memory. If the data being delivered happens to be in the CPU's caching hierarchy, it is invalidated.

Step 2: Software running on the CPU reads the data to perform processing tasks. These data access operations misses cache and causes data to be read in from memory, into the CPU's caching hierarchy.

In a related way, the same operation on a setup with Intel DDIO technology (right) requires the following set of steps:

Step 1: I/O data delivery uses Write Update or Write Allocate operations (depending on cache residency of the memory addresses to which data is being

delivered), which causes data to be delivered to the cache, without going to memory.

Step 2: The subsequent read operations initiated by software are satisfied by data from the cache, without causing any expensive cache misses.

Thus, I/O device data delivery operations with Intel DDIO technology are achieved with fewer (and ideally zero) trips to the main memory of the system. Also, from a CPU caching perspective, the data in cache is not disturbed by virtue of an I/O data delivery operation, which creates opportunities for intelligent hardware/software design optimizations. Both the host processors and the network card we use are DDIO enabled devices, so we take advantage of this technology to further decrease the number of copies from/to the main memory, as well as the latency and also increase the throughput and power efficiency of our system.

5.2 System Design

In this subsection, we present the changes applied to our previous approach and the problems that we eliminate by applying the proposed optimizations. We also discuss how those optimizations lead to predictable system performance, which enhances its scalability.

5.2.1 Architecture

Every optimization we propose is based on the previously discussed "Lock-Free" architectural model, which means that each worker can run independently of the others, without the need for any kind of "puppeteer" thread to synchronize or feed them with ingress network traffic. Figure 5.3 illustrates the architectural changes we propose, which mainly differ in the way the CPU cores poll the network interfaces and offload the packet processing computation to the GPU. For simplicity, we only show two network interfaces and how they are mapped to the CPU cores but the concept remains the same when more ports are active; the only difference is in the number of active CPU cores.

Firstly, we leverage the DPDK framework to tackle the Netmap limitations, such as the maximum number of ports and the maximum port speed we could have, which used to cap the aggregated input rate at the first place. With DPDK we are able to instantiate a variable number of hardware receive rings (Receive Queues or RxQs) and also parameterize their size (how many packets can be stored), during the port initialization phase. Those features can be exploited when a faster network card is installed and a single CPU core could

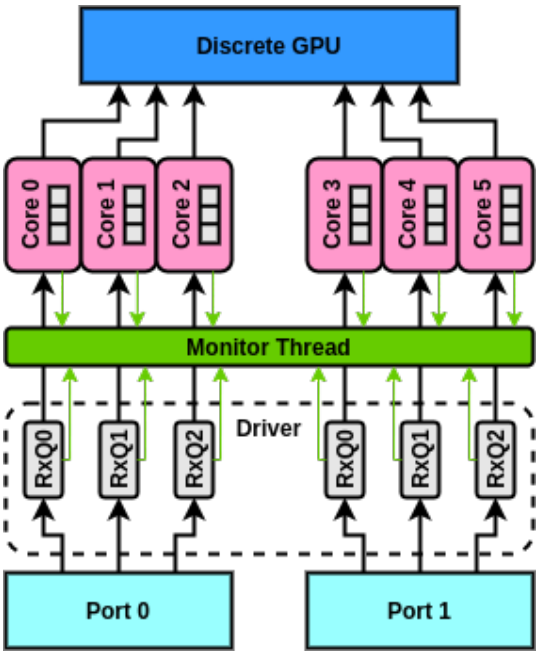


Figure 5.3: High level overview of the scalable architecture.

not handle the increased amounts of traffic. By initializing more Rx rings, we can have a number of CPU cores dedicated to the receiving process, without any need to apply any coarse-grain port locking mechanism. The system spawns and maps a hardware thread to each distinct receive ring. This is an one to one mapping, so the number of active CPU threads is equal to the number of available network ports times the number of Rx queues per port. Obviously, in our case with that mapping policy, we can have up to 32 active workers per processor package due to the use of Intel Hyper-Threading technology. The choice of CPU threads is neither serial nor random, our system chooses to assign the worker's job to a thread or completely ignore it, based on its PCIe locality, or put it in other words, if a thread can directly access both the network interface and the target device, it suits for the worker's job, but if intercommunication is required between the two processor packages (via the UPI links) in order to access any of the PCIe devices, this specific thread is listed as inactive. By applying the aforementioned, simple yet effective, policy, our system can decide the optimal job to device placement and experience all the benefits of fast, local memory and direct data accesses (as explained in Subsections 5.1.2 and 5.1.3 respectively).

From that point on, each active thread has the same functionality, which is described by the following set of actions: (i) allocate memory and initialize OpenCL buffers with the context of the assigned target device, (ii) capture

network traffic from the assigned Rx queue and fill the input buffers with the ingress packets, (iii) when input packet buffers are full, transfer them to the global memory of the assigned GPU and spawn OpenCL kernel execution (only if the previous kernel execution, that has been initiated by the same worker, is completed) and (iv) collect performance statistics, return to step (ii) and repeat the same procedure.

Additionally, the monitor thread (see Figure 5.3) runs once every second to collect statistics from the active network interfaces and workers, in order to analyze them and present them in a meaningful way. Note that the monitor thread only reads the necessary data structures or hardware registers of the worker threads in order to collect the performance or power statistics and aggregates the collected data using its own data structures, so it introduces no synchronization burden or congestion for resources to the rest of the workers.

Finally, we do not tackle packet reordering using software solutions on purpose, mainly for scalability and secondarily for performance optimization reasons; this decision is further explained in the following subsection.

5.2.2 Receive Side Scaling

Over the last few years, contemporary NICs support multiple receive and transmit descriptor queues, a technique widely known as Receive Side Scaling, or RSS [13]. On reception, a NIC can send different packets to different queues to distribute processing among CPUs. The NIC distributes packets by applying a filter to each packet that assigns it to one of a small number of logical flows. Packets for each flow are steered to a separate receive queue, which in turn can be processed by separate CPUs. The goal of RSS is to increase performance uniformly. Multi-queue distribution can also be used for traffic prioritization, but that is neither the main focus of this technique, nor the reason we decided to use it in this work.

The filter used in RSS is typically a hash function over the network and/or transport layer headers. For instance, it could be a 4-tuple hash over IP addresses and TCP ports of a packet. The most common hardware implementation of RSS uses a 128-entry indirection table where each entry stores a queue number. The receive queue for a packet is determined by masking out the low order seven bits of the computed hash for the packet, taking this number as a key into the indirection table and reading the corresponding value.

RSS should be enabled when latency is a concern or whenever the receive rate of a single core is unable to match the ingress traffic rate. Spreading load between CPUs decreases queue length. For low latency networking, the optimal setting is to allocate as many queues as there are CPUs in the system (or the NIC maximum, if lower). The most efficient high-rate configuration

is likely the one with the smallest number of receive queues where no receive queue overflows due to a saturated CPU.

The driver for a multi-queue capable NIC typically provides a kernel module parameter for specifying the number of hardware queues to configure. DPDK provides API calls that communicate with the network card drivers, thus the programmer is allowed to configure the RSS parameters at runtime. We use those API calls to enable RSS and configure it as per our needs in order to be able to keep our architectural model free from locking or other synchronization mechanisms.

A limitation of the RSS mechanism is its asymmetric nature which contradicts the needs of many typical network applications. In general, it is important to have the same CPU core handle both sides of a connection or packet flow and not letting two different CPU cores share information, as the latter negatively affects the performance. RSS algorithm is usually using the Toeplitz hash function, which takes two inputs: the static hash key and the tuples which are extracted from the packet. The problem is that the default hash key that is used in DPDK (and is also the recommended key from Microsoft) does not distribute symmetrical flows to the same CPU. In many cases, one can achieve symmetric RSS by changing the hash key such that the 32 most significant bits of the key are identical to the next 32 bits, and the 16 bits afterwards should be identical to its 16 least significant bits. By applying those modifications on the key, we achieve symmetrical RSS, but the problem is that key changes lead to bad distribution of the traffic between the different cores. Obviously, load balancing the packets between the cores is a very challenging problem and we do not want to deal with it. Luckily, a specific hash key which gives us both symmetrical and uniform flow distribution has been previously proposed [69] and works really well, so we configure DPDK to use it in its internal RSS advance configuration structures.

By enabling RSS and changing the hash key to match our needs, we address the problem of pre-classification of the input traffic before filling the batches and processing them, which leads to higher CPU utilization. This classification is performed in the network card from the hardware itself upon packet arrival, which is fast and further offloads the CPU, leaving more room for performance improvements. As a consequence, the packets of each flow are being placed in batches and processed in the exact same order they are captured, so any performance overhead related to the flow reconstruction is simply eliminated.

5.2.3 Performance Predictability

The proposed changes in the model allow our system to scale better in every logical level between the arrival and the departure of the network packets. In

detail, we use the best proposed solution to bypass the kernel network stack (i.e. DPDK) when receiving ingress traffic, which introduces no limitations regarding the number of network interfaces, so we could potentially add more ports to further stress our system and identify its maximum capacity. Next, we fully exploit the hardware capabilities by using techniques like RSS to (i) tackle as early as possible the flow classification problem, which otherwise can cause significant performance overhead, and eliminate the need for packet re-ordering and flow reconstruction and (ii) ensure that our system is able to receive the ingress traffic at line rate, regardless of the maximum data transfer rate of the underlying network interfaces, by leveraging the required amount of CPU cores or threads that are available in modern processors. Additionally, we have completely eradicate the intercommunication between the CPU cores which can lead to a number of performance related problems, such as congestion in locking mechanisms, excessive data transfers and cache pollution. We manage to architecturally build our system in a way that could satisfy those properties by design, so scaling it up later would be just a matter of changing the underlying hardware to provide more computational resources, instead of putting the effort in making structural design changes.

On that note, if we add more or faster network interfaces, or introduce a new network function that is more demanding computationally-wise, a single target device may not be able to sustain line rate processing. In this case, we can add extra target devices as long as there are still available PCIe lanes in the CPU, so by just populating the expansion slots and changing the underlying hardware, the performance of the system scales up proportionally. Furthermore, our system is fully aware of the locality of the PCIe devices (whether it is the NICs or the target accelerators) relatively to the available CPU and memory nodes, so it is able to utilize its resources as effectively as possible in order to meet its performance goals. Additionally, if the traffic rate does not saturate the capacity of the network ports, we could offload the processing to a less powerful but more power-efficient device and the performance will still be predictable.

On the same note, we could replicate the installed PCIe devices across the second CPU node, which is completely idle. By doing so, we could double the overall performance of our system in two different ways. The first one is to just double the input traffic rate of the system and let the replicated devices to run identically to the devices of the first CPU node, forming a doubled capacity middlebox, while the second approach would be to reproduce the same traffic and feed the replicated CPU node and corresponding devices, in order to run a different network function in parallel with the first. We claim that the overall performance of our system will double, as the two CPU nodes neither interfere, nor is there any other hardware, software or architectural burden to hinder the performance growth. However, in order to be completely

confident, we shall conduct the necessary experiments and the corresponding quantitative measurements.

5.3 System Benchmarks

During our experiments we use a variable number of ports to determine the maximum performance capabilities of the system, after applying all of the previously mentioned hardware changes and software optimizations. We repeat the execution of the benchmarks multiple times, each one using a different network traffic trace with discrete characteristics in order to find the hyperparameters that need to be tuned and their corresponding optimal values. We present the results of all those experiments using plots, so the interpretation is more engaging and we also discuss the conclusions that are drawn.

5.3.1 Throughput

Figures 5.4 – 5.12, illustrate the achieved end-to-end throughput of our system using only a fraction of all the CPU cores from the first node and a single target device. We created every possible configuration combination using all of the available Rx ring sizes (8 options), four different RSS configurations (i.e. 3, 4, 5 and 6 Rx Rings per port) and 8 different batch sizes, resulting in 256 different tested system configurations. For the sake of clarity and simplicity, we only plot the results of 64 different configurations each time (combinations of discrete ring and batch sizes). The RSS configuration is the one which leverages the minimum number of Rx rings in order to meet the performance goal of the system (i.e. line rate processing, which is the optimal scenario, or as close to it as possible). In the experiments that are depicted by Figures 5.4 – 5.9, we apply the DPI function on the incoming traffic from every available network port, as it is memory and compute intensive and it also exhibits high performance fluctuations depending on the input data characteristics. For the rest of the experiments (Figures 5.10 – 5.12), the traffic from the first set of ports is filtered through the DPI engine, while we apply the encryption algorithm on the traffic from the second set of ports. We choose to measure the achieved throughput in million packets per second as it is more representative because of the variability of our experiments regarding the traffic rate characteristics. Each measurement that is presented in the following figures is the average of a 15-seconds window, during which the system was active and processing the incoming traffic, while the monitor collected statistics every second.

Specifically, we conducted experiments using fixed-sized packets using both the minimum payload of 64 Bytes and the maximum payload of 1500 Bytes.

Figures 5.4 and 5.5 depict the performance results of each configuration using minimum packet payload while executing the DPI application and Figure 5.10 reveals the performance under the same traffic conditions while executing both AES and DPI. For all cases of running applications, regardless if the system polls four or five network ports for an aggregated input rate of 40 and 50Gbps respectively, the traffic of each port should be split in four Rx queues, given that there is a torrential amount of very small packets and there is a need for many CPU cores in order to handle them as efficient as possible. We observe that the system reaches top performance when each Rx ring is configured to be maximum-sized and is combined with large batches (10K, 12K or 14K packets), regardless of the executing applications or the number of ports. However, we notice that for minimum-sized packets, our system cannot sustain line rate performance when it is requested to handle more than 40Gbps of traffic and the deep packet inspection function is applied. From Figure 5.5 it is clear that the best configuration still results in $\approx 15\%$ packet loss, or put it in another way, the best configuration can sustain a performance of almost 43Gbps.

Contrary to the first set of experiments, when a single application is active (DPI) and the traffic rate consists of packets of maximum size, the best configuration implies by the combination of small-sized batches (2K or 4K packets), three rings per port and minimum-sized Rx rings, if the system polls four network interfaces (see Figure 5.6) or four minimum-sized Rx queues, if ingress traffic is captured on five ports (see Figure 5.7). In both cases, our system meets its performance requirements and is able to achieve line rate processing at 40Gbps without packet loss and at 50 Gbps with less than 5% packet loss respectively. Surprisingly, when both DPI and AES are being executed and the system faces the same traffic conditions, if we setup 4 Rx rings per interface, we notice that the system achieves line rate processing regardless of the ring and the batch size. We repeated and manually inspected this set of experiments to rule out the possibility of outlier results and also observed very limited GPU utilization and decreased power consumption.

The results so far reveal that the application with the most unstable behavior is the DPI, but still the performance of the system is predictable. If we multiplex two applications, not only we do not experience interference effects, but the GPU resources are shared surprisingly well, which leads to increased performance stability.

Lastly, our previous benchmarking methodologies both rely on conditions with constant packet sizes in order to reveal the capabilities of our system and the optimal hyperparameter values. However, testing and operational deployment conditions differ significantly, so we also conducted experiments with a mixture of packet sizes. In Figures 5.8, 5.9 and 5.12 we demonstrate the achieved throughput of the system using the most realistic and typical

traffic scenario, which is none other than Internet mix traffic, as defined in IMIX RFC document [48]. The PCAP file that is used to conduct this set of experiments contains 100 packets with sizes and ratio according to the IMIX specs (7:4:1 distribution of Ethernet-encapsulated packets of sizes 64, 570 and 1518 Bytes respectively, with an average packet size of 353 Bytes). For 40Gpbs of input traffic (Figure 5.8) filtered by the DPI, three Rx rings per port is the optimal RSS configuration, while we also observe that as long as the batch size is relatively small (4K or 6K packets), we can choose any small Rx ring size (between 1K and 3K descriptors per ring) without hurting the performance. Similarly, if the system polls five network ports with the same application being active, four medium-sized receive rings per port (1K to 2K packet descriptors) are needed combined with 4K to 6K packets per batch, in order to achieve almost 50Gbps of processing throughput. When the traffic of the first two ports is processed using DPI and the traffic of the last two ports is being encrypted, we can choose the ring size to be anywhere from minimum-sized to mid-sized as long as we instantiate 4 Rx rings and be sure that the batch size is relatively small.

It is clear that there is a huge pool of different combinations, yet only a very small fraction of them results in optimal performance and efficient system usage. The vast majority of those combinations score not only sub-optimal performance, in terms of throughput, but also contribute to increased system latency (unnecessarily large batch size) and inefficient utilization of the devices. For those reasons, it is vital to fine-tune the most influential hyperparameters, such as the number of Rx queues, their size and the size of the processing batches in the most optimal way, otherwise the performance of the system will definitely suffer.

5.3.2 Power Consumption

Although the benchmark the optimized system to define its maximum performance, we shall not try to achieve it by any means and definitely shall not neglect the power consumption, which is an important metric when designing and implementing networked systems. In Subsection 4.3.2 we note that more active devices lead to higher power consumption and vice versa. For that reason, we choose to fully utilize a single power-hungry target device to prove that it can achieve very high processing performance without excessively increasing the total energy expenditure of the system, or without the need to add any extra devices for supplementary purposes. Figure 5.13 demonstrates the power consumption of our system for each one of the nine discrete traffic experiments we presented in the previous subsection. Note that for the sake of simplicity, we only present the measured power consumption of the best performing

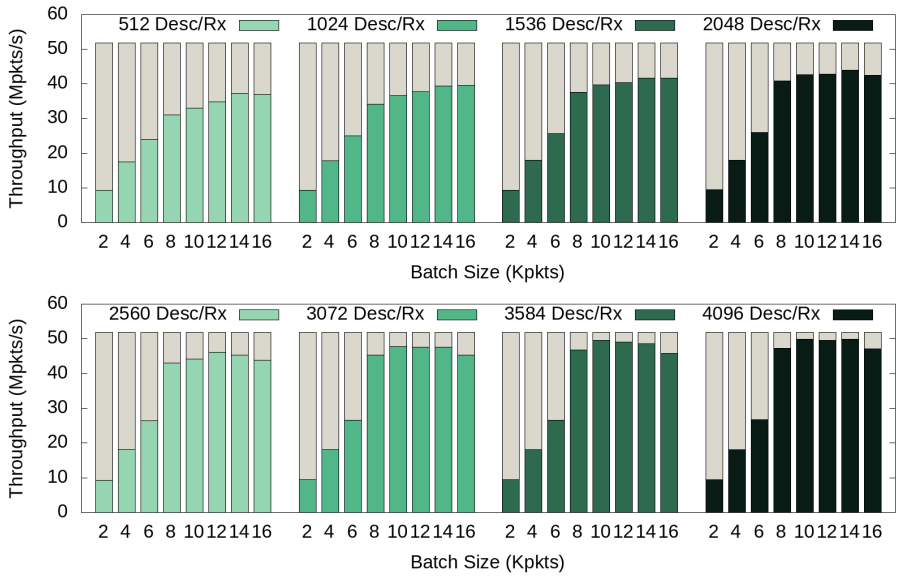


Figure 5.4: Performance results of different batch and Rx ring size configurations while executing DPI with 4 10G ports and 4 Rx rings per port using 64-Bytes packets.

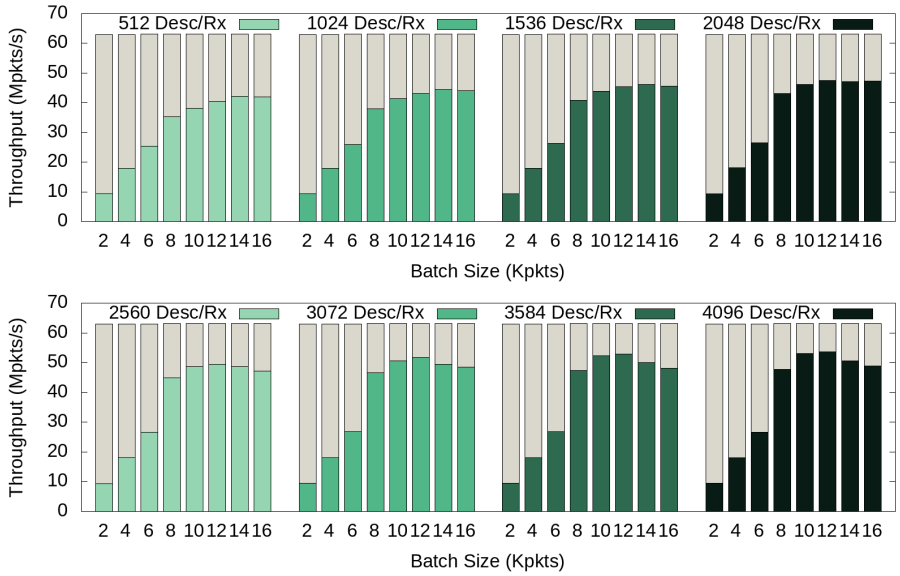


Figure 5.5: Performance results of different batch and Rx ring size configurations while executing DPI with 5 10G ports and 4 Rx rings per port using 64-Bytes packets.

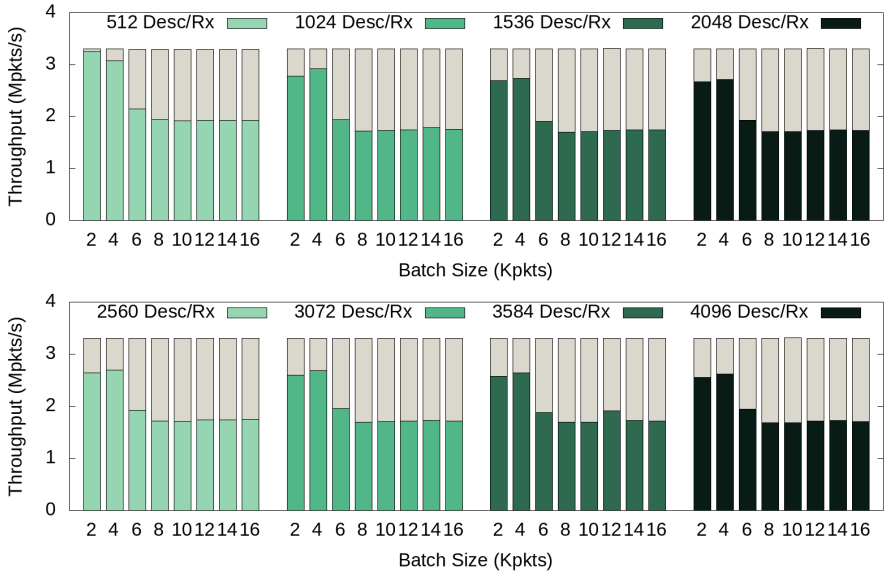


Figure 5.6: Performance results of different batch and Rx ring size configurations while executing DPI with 4 10G ports and 3 Rx rings per port using 1500-Bytes packets.

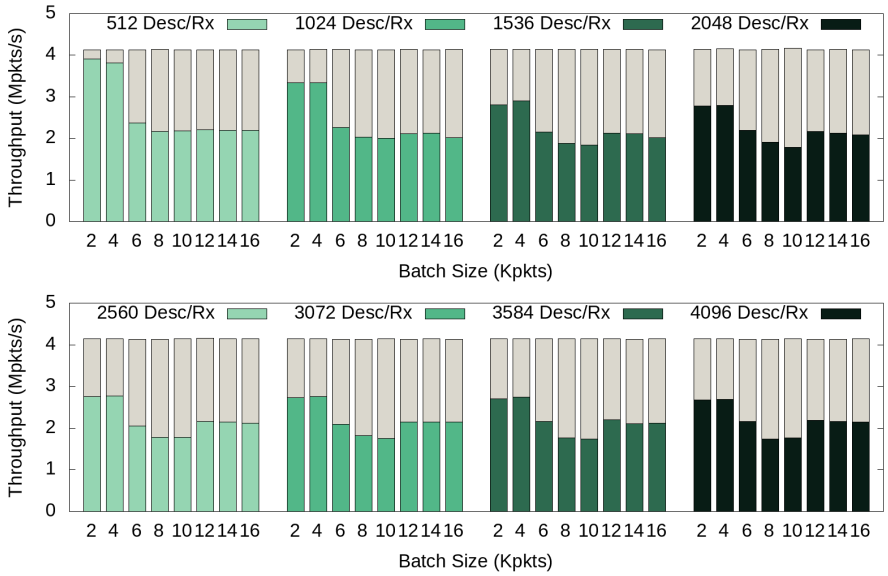


Figure 5.7: Performance results of different batch and Rx ring size configurations while executing DPI with 5 10G ports and 4 Rx rings per port using 1500-Bytes packets.

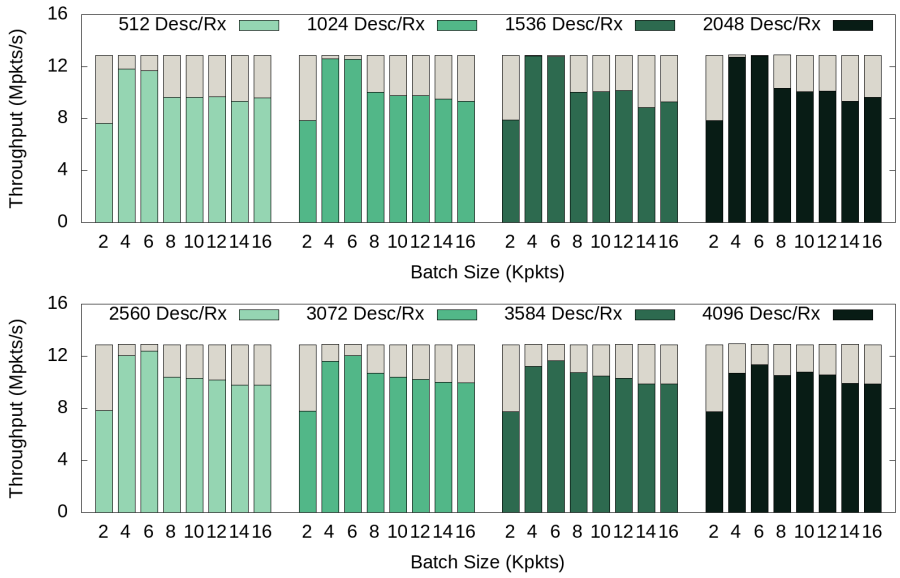


Figure 5.8: Performance results of different batch and Rx ring size configurations while executing DPI with 4 10G ports and 3 Rx rings per port using IMIX traffic.

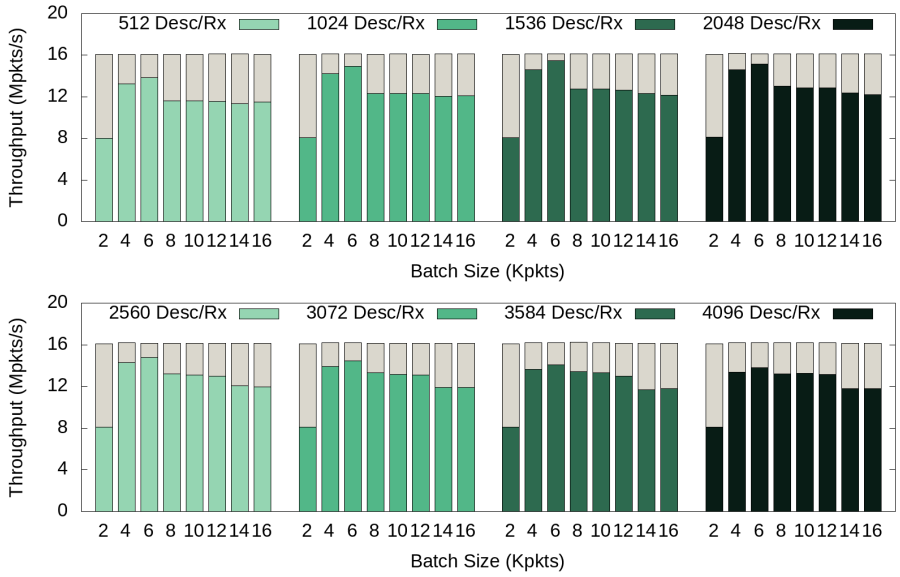


Figure 5.9: Performance results of different batch and Rx ring size configurations while executing DPI with 5 10G ports and 4 Rx rings per port using IMIX traffic.

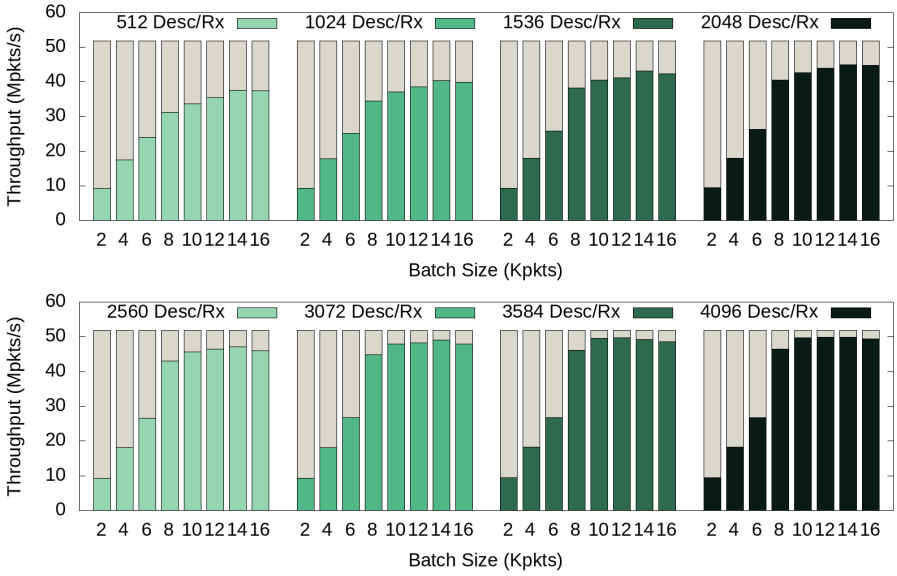


Figure 5.10: Performance results of different batch and Rx ring size configurations while concurrently executing DPI and AES with 4 10G ports and 4 Rx rings per port using 64-Bytes packets.

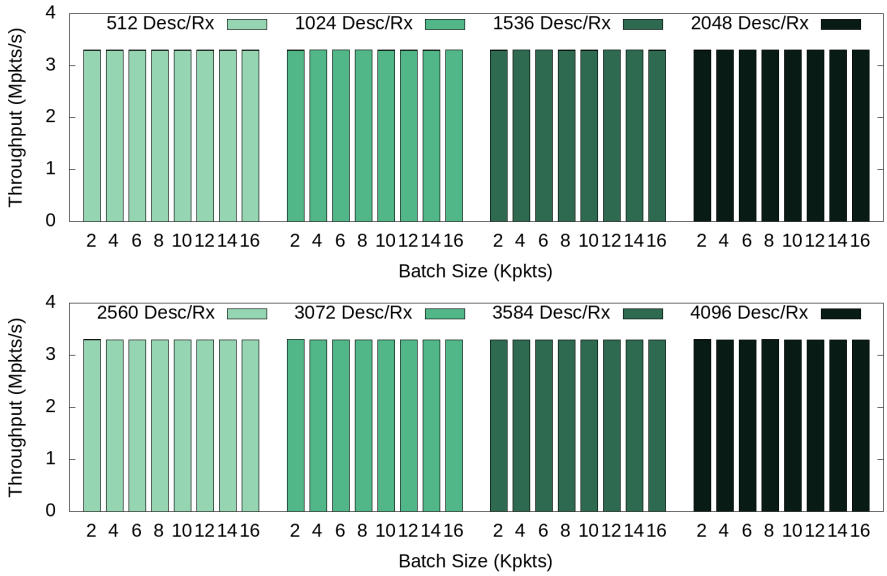


Figure 5.11: Performance results of different batch and Rx ring size configurations while concurrently executing DPI and AES with 4 10G ports and 4 Rx rings per port using 1500-Bytes packets.

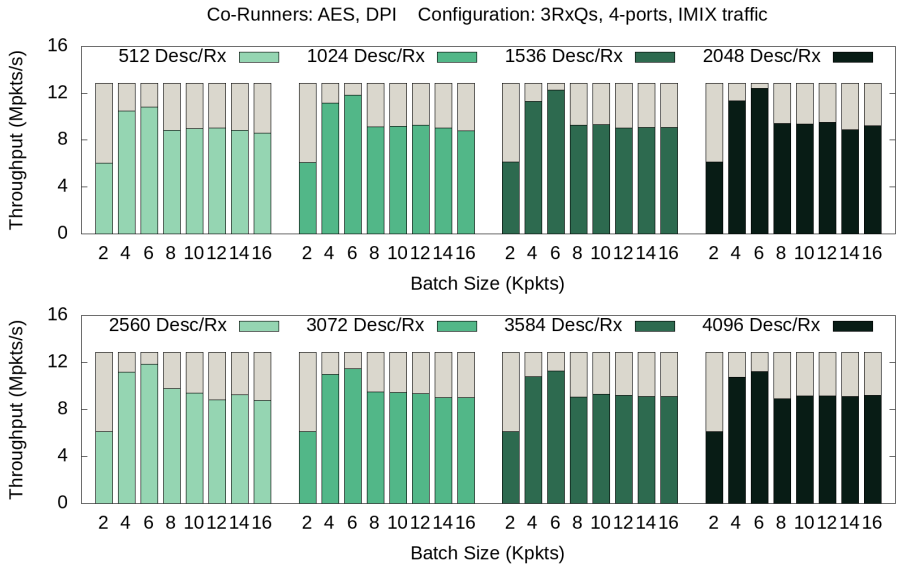


Figure 5.12: Performance results of different batch and Rx ring size configurations while concurrently executing DPI and AES with 4 10G ports and 3 Rx rings per port using IMIX traffic.

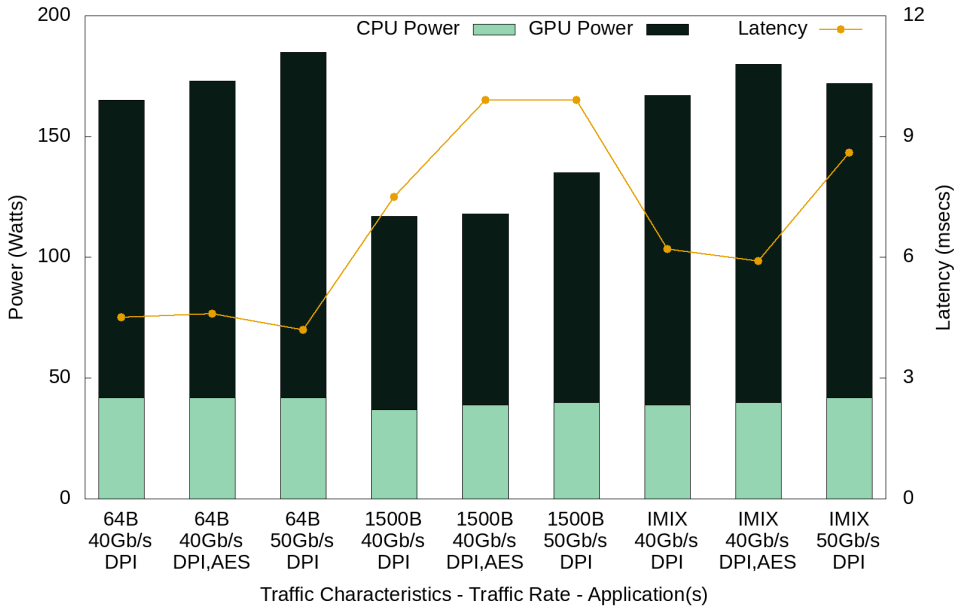


Figure 5.13: Power consumption and per-batch latency characterization of the best performing configuration for every different traffic experiment.

configuration in each case, while neglecting the sub-optimal configuration measurements. As it can be seen from the figure, the energy expenditure of the target device (i.e. the discrete GPU) increases as the packet sizes decrease; this is a direct result of higher utilization of the GPU resources. On the other hand, the consumption of the host processor is roughly the same in every case (around 40 Watts) and slightly higher than its idle value (17 Watts). When we experiment with maximum-sized packets, we achieve optimal performance to power consumption ratio (341, 370 Mbps/Watt for processing 40 and 50 Gbps respectively using a single application and 339 Mbps/Watt for processing 40 Gbps while executing two applications). However, optimal from realistic cases differ; even in the worst cases, when receiving 64-Byte and IMIX traffic from 4 network ports, the system achieves optimal performance in terms of throughput, without facing energy overconsumption; the resulted performance per power unit in each case is 242 and 239 Mbps/Watt, which is reasonable if we consider the traffic characteristics.

5.3.3 Latency

Figure 5.13 also depicts the end-to-end batch latency of the same best performing configurations, as defined in the previous subsection. The reason we decided to measure the latency per batch instead of the end-to-end packet latency is simple; from the moment of its arrival, a packet is placed in a packet buffer. It shall not be forwarded into the pipeline for processing until the buffer is full and shall not be transmitted until the processing of the buffer is completed, so per-packet latency measurements are meaningless. A general rule of thumb is that the bigger the payload of the packet, the higher the latency, which at first seems counter-intuitive; one could expect that if ingress traffic consisted of fewer, big-sized packets, the latency would be low. However, due to batching and the way we measure latency, this is a normal consequence. For instance, to achieve line rate performance when dealing with minimum-sized traffic, the system generally uses large batches. The amount of data per batch is equal to the packet size (64 Bytes) times the number of packets per batch (at least 10K packets). On the other hand, when 1500-Byte packets dominate the network traffic, the desired performance is achievable using small batches (usually less than 4K packets per batch). Those calculations reveal that an interpretation that seemed wrong, is indeed correct. Regardless of the traffic and configuration characteristics and the way the latency is affected, we observe that if correctly configured, our system is able to keep the overall latency really low, between 4 and 10 milliseconds per batch, which is surprisingly low not only for a throughput-oriented system, but under certain circumstances, even for a latency-aware environment.

5.4 Future Work

Even though the optimizations we apply to our system's design address some of its limitations, especially those regarding the raw I/O and the overall processing performance, it still has some limitations that cannot be neglected. Firstly, we have not yet tested its scalability in practice due to the lack of extra network interfaces, so we are only able to predict the expected theoretical performance and not prove it. This is not a direct limitation, but it has to be mentioned. We do plan to mirror our device setup and leverage the second CPU socket as well to support our predictions.

So far, we have just benchmark results of the maximum performance of the optimized system; we are only able to achieve it by manually selecting the best configuration. We plan to install a second, less powerful, less energy-hungry accelerator and apply the scheduling schema we have already proposed and evaluate the efficiency of our new system. Some changes may be required to the scheduler, mainly changes that are related to the features that affect the scheduling decisions. We even think to leverage the appropriate machine learning algorithms, feed them with the large amount of data we have collected during our experimentation and benchmarking phase and train a model that will act as a scheduler and be able to suggest an optimal or close to optimal configuration based on the real-time characteristics of the ingress traffic.

Chapter 6

Conclusions

In this work we address the problem of efficiently improving the performance of network middleboxes using commercial heterogeneous hardware and optimized software techniques. Statically mapping applications to devices can lead to inefficiencies due to wrong placement, network traffic fluctuations or interference effects. We propose an adaptive and highly dynamic scheduling solution that enables real-time application multiplexing across heterogeneous and asymmetric architectures that can be found on commodity, off-the-shelf hardware setups. We manage to improve the overall efficiency of the tested applications, since our scheduler is able to choose the configuration that results to the optimal performance each time relatively to the current state, responding quickly either to network fluctuations or system changes. Furthermore, we address some of the hardware and software limitations that prevent our system from reaching top performance. Specifically, we change some of the hardware components to eliminate the PCIe related problems and apply software optimizations to prove that if fully and properly utilized, discrete GPUs can provide top performance without sabotaging the end-to-end latency or unnecessarily increasing the power consumption. Through benchmarks, we experienced line rate packet processing performance up to 50Gbps using only one target device, up to 2 times higher power efficiency for some cases and up to 5 times lower end-to-end latency.

Bibliography

[1] Home - dpdk. https://www.dpdk.org/. Accessed: 2020-09-11.

[2] Intel core i7-8700k processor. https://ark.intel.com/content/www/us/en/ark/products/126684/intel-core-i7-8700k-processor-12m-cache-up-to-4-70-ghz.html. Accessed: 2020-09-28.

[3] Intel data direct i/o technology (intel ddio): A primer. https://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/data-direct-i-o-technology-brief.pdf. Accessed: 2020-10-6.

[4] Intel hyper-threading technology. https://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html. Accessed: 2020-09-14.

[5] Intel performance counter monitor - a better way to measure cpu utilization. https://software.intel.com/content/www/us/en/develop/articles/intel-performance-counter-monitor.html. Accessed: 2020-09-28.

[6] An introduction to the intel quickpath interconnect. https://www.intel.com/content/www/us/en/io/quickpath-technology/quick-path-interconnect-introduction-paper.html. Accessed: 2020-10-7.

[7] Nvidia management library (nvml). https://developer.nvidia.com/nvidia-management-library-nvml. Accessed: 2020-09-28.

[8] Opencl device fission extension - the khronos group inc. https://www.khronos.org/registry/OpenCL/extensions/ext/cl_ext_device_fission.txt. Accessed: 2020-09-24.

[9] Opencl overview - the khronos group inc. https://www.khronos.org/opencl/. Accessed: 2020-09-22.

[10] Openssl project. http://www.openssl.org/. Accessed: 2020-09-24.

[11] Optimizing applications for numa. https://software.intel.com/content/dam/develop/external/us/en/documents/3-5-memmgt-optimizing-applications-for-numa-184398.pdf. Accessed: 2020-10-7.

[12] Pci express base specification revision 2.1. https://www.intel.com/content/dam/altera-www/global/en_US/uploads/e/e2/PCI_Express_Base_r2.1.pdf. Accessed: 2020-09-28.

[13] Scaling in the linux networking stack. https://www.kernel.org/doc/Documentation/networking/scaling.txt. Accessed: 2020-10-8.

[14] The snort ids/ips. http://www.snort.org/. Accessed: 2020-09-24.

[15] White paper: Design considerations for efficient network applications with intel multi-core processor-based systems on linux*. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/multi-core-processor-based-linux-paper.pdf. Accessed: 2020-10-7.

[16] White paper: Introduction to intel architecture. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-introduction-basics-paper.pdf. Accessed: 2020-09-16.

[17] White paper: Nvidia turing gpu architecture. https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf. Accessed: 2020-09-17.

[18] White paper: Nvidia's next generation cuda compute architecture: Fermi. https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf. Accessed: 2020-09-21.

[19] Bhavish Agarwal, Aditya Akella, Ashok Anand, Athula Balachandran, Pushkar Chitnis, Chitra Muthukrishnan, Ramachandran Ramjee, and George Varghese. Endre: An end-system redundancy elimination service for enterprises. In *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation*. NSDI, San Jose, CA, USA, April 2010.

[20] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[21] Ashok Anand, Archit Gupta, Aditya Akella, Srinivasan Seshan, and Scott Shenker. Packet caches on routers: the implications of universal redundant traffic elimination. In *Proceedings of the ACM SIGCOMM 2008 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, 2008.

[22] Tom Barbette, Cyril Soldani, and Laurent Mathy. Fast userspace packet processing. In *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 5–16. IEEE, 2015.

[23] Pavel Benáček, Viktor Puš, Jan Kořenek, and Michal Kekely. Line rate programmable packet processing in 100gb networks. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–1. IEEE, 2017.

[24] Michael Boyer, Kevin Skadron, Shuai Che, and Nuwan Jayasena. Load balancing in a changing world: Dealing with heterogeneity and performance variability. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF '13, New York, NY, USA, 2013. Association for Computing Machinery.

[25] B. Carpenter and S. Brim. Rfc3234: Middleboxes: Taxonomy and issues, 2002.

[26] Ruining Chen and Guoao Sun. A survey of kernel-bypass techniques in network stack. In *Proceedings of the 2018 2nd International Conference on Computer Science and Artificial Intelligence*, CSAI '18, pages 474–477, New York, NY, USA, 2018. Association for Computing Machinery.

[27] David Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

[28] Gregory F. Diamos and Sudhakar Yalamanchili. Harmony: An execution model and runtime for heterogeneous many core systems. In *Proceedings of the 17th International Symposium on High Performance Distributed Computing*, HPDC '08, pages 197–200, New York, NY, USA, 2008. Association for Computing Machinery.

[29] Mihai Dobrescu, Katerina Argyraki, and Sylvia Ratnasamy. Toward predictable performance in software packet-processing platforms. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 141–154, San Jose, CA, April 2012. USENIX Association.

[30] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. Routebricks: Exploiting parallelism to scale software routers. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, 2009.

[31] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 365–376, New York, NY, USA, 2011. Association for Computing Machinery.

[32] Sebastian Gallenmüller, Paul Emmerich, Florian Wohlfart, Daniel Raumer, and Georg Carle. Comparison of frameworks for high-performance packet io. In *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '15, pages 29–38, USA, 2015. IEEE Computer Society.

[33] Ali Ghodsi, Vyas Sekar, Matei Zaharia, and Ion Stoica. Multi-resource fair queueing for packet processing. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 1–12, 2012.

[34] Giannis Giakoumakis, Eva Papadogiannaki, Giorgos Vasiliadis, and Sotiris Ioannidis. Pythia: Scheduling of concurrent network packet processing applications on heterogeneous devices. In *2020 6th IEEE Conference on Network Softwarization (NetSoft)*, pages 145–149. IEEE, 2020.

[35] 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, 2017.

[36] Chris Gregg, Michael Boyer, Kim Hazelwood, and Kevin Skadron. Dynamic heterogeneous scheduling decisions using historical runtime

data. In *Workshop on Applications for Multi-and Many-Core Processors (A4MMC)*, pages 1–12, 2011.

[37] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. Building a single-box 100 gbps software router. In *2010 17th IEEE Workshop on Local & Metropolitan Area Networks (LANMAN)*, pages 1–4. IEEE, 2010.

[38] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. Packetshader: a gpu-accelerated software router. In *Proceedings of SIGCOMM*, 2010.

[39] O. Harrison and J. Waldron. Practical symmetric key cryptography on modern graphics hardware. In *Proceedings of the 17th USENIX Security Symposium*, 2008.

[40] Joongi Kim, Keon Jang, Keunhong Lee, Sangwook Ma, Junhyun Shim, and Sue Moon. Nba (network balancing act) a high-performance packet processing framework for heterogeneous processors. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–14, 2015.

[41] Jungwon Kim, Honggyu Kim, Joo Hwan Lee, and Jaejin Lee. Achieving a single compute device image in opencl for multiple gpus. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 277–288, New York, NY, USA, 2011. Association for Computing Machinery.

[42] Purushottam Kulkarni, Fred Douglis, Jason LaVoie, and John M. Tracey. Redundancy elimination within large collections of files. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, 2004.

[43] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100x gpu vs. cpu myth: An evaluation of throughput computing on cpu and gpu. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 451–460, New York, NY, USA, 2010. Association for Computing Machinery.

[44] W. Lee, X. Wong, B. Goi, and R. C. . Phan. Cuda-ssl: Ssl/tls accelerated by gpu. In *2017 International Carnahan Conference on Security Technology (ICCST)*, pages 1–6, 2017.

[45] Y. Li and X. Qiao. A parallel packet processing method on multi-core systems. In *2011 10th International Symposium on Distributed Computing and Applications to Business, Engineering and Science*, pages 78–81, 2011.

[46] Michael D Linderman, Jamison D Collins, Hong Wang, and Teresa H Meng. Merge: a programming model for heterogeneous multi-core systems. *ACM SIGOPS operating systems review*, 42(2):287–296, 2008.

[47] C. Luk, S. Hong, and H. Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 45–55, 2009.

[48] A. Morton. Rfc6985: Imix genome: Specification of variable packet sizes for additional testing, 2013.

[49] Luca Niccolini, Gianluca Iannaccone, Sylvia Ratnasamy, Jaideep Chandrashekar, and Luigi Rizzo. Building a power-proportional software router. In *Presented as part of the 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*, pages 89–100, 2012.

[50] NVIDIA. Cuda c++ programming guide. Technical report, August 2020.

[51] Yasuhiro Ohara, Yudai Yamagishi, Satoshi Sakai, Abhik Datta Banik, and Shin Miyakawa. Revealing the necessary conditions to achieve 80gbps high-speed pc router. In *Proceedings of the Asian Internet Engineering Conference*, pages 25–31, 2015.

[52] Eva Papadogiannaki, Lazaros Koromilas, Giorgos Vasiliadis, and Sotiris Ioannidis. Efficient software packet processing on heterogeneous and asymmetric hardware architectures. *IEEE/ACM Transactions on Networking*, 25(3):1593–1606, 2017.

[53] M. P. Pineda Vargas, R. A. A. Rodriguez, and O. J. Salcedo Parra. Algorithm for the optimization of rsa based on parallelization over gpu ssl/tls protocol. In *2017 IEEE International Conference on Smart Cloud (SmartCloud)*, pages 294–297, Nov 2017.

[54] G. Regnier, D. Minturn, G. McAlpine, V. A. Saletore, and A. Foong. Eta: experience with an intel xeon processor as a packet processing engine. *IEEE Micro*, 24(1):24–31, 2004.

[55] Luigi Rizzo. netmap: A novel framework for fast packet i/o. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 101–112, Boston, MA, June 2012. USENIX Association.

[56] Jie Shen, Jianbin Fang, Henk Sips, and Ana Lucia Varbanescu. Performance traps in opencl for cpus. In *Proceedings of the 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 2013.

[57] Randy Smith, Neelam Goyal, Justin Ormont, Karthikeyan Sankaralingam, and Cristian Estan. Evaluating GPUs for network packet signature matching. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, 2009.

[58] Enqiang Sun, Dana Schaa, Richard Bagley, Norman Rubin, and David Kaeli. Enabling task-level scheduling on heterogeneous platforms. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, GPGPU-5, pages 84–93, New York, NY, USA, 2012. Association for Computing Machinery.

[59] Weibin Sun and Robert Ricci. Fast and flexible: Parallel packet processing with gpus and click. In *Proceedings of the 9th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 2013.

[60] Jitendra V Tembhurne and SR Sathe. Rsa public key acceleration on cuda gpu. In *Artificial Intelligence and Evolutionary Computations in Engineering Systems*, pages 365–375. Springer, 2016.

[61] Janet Tseng, Ren Wang, James Tsai, Saikrishna Edupuganti, Alexander W Min, Shinae Woo, Stephen Junkins, and Tsung-Yuan Charlie Tai. Exploiting integrated gpus for network packet processing workloads. In *2016 IEEE NetSoft Conference and Workshops (NetSoft)*, pages 161–165. IEEE, 2016.

[62] Giorgos Vasiliadis, Spiros Antonatos, Michalis Polychronakis, Evangelos P. Markatos, and Sotiris Ioannidis. Gnort: High performance network intrusion detection using graphics processors. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection*, 2008.

[63] Giorgos Vasiliadis, Lazaros Koromilas, Michalis Polychronakis, and Sotiris Ioannidis. Gaspp: A gpu-accelerated stateful packet processing framework. In *Proceedings of the 2014 USENIX Annual Technical Conference*, 2014.

[64] Giorgos Vasiliadis, Michalis Polychronakis, Spiros Antonatos, Evangelos P. Markatos, and Sotiris Ioannidis. Regular expression matching on

graphics hardware for intrusion detection. In *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection*, 2009.

[65] 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*, 2011.

[66] Guibin Wang and Xiaoguang Ren. Power-efficient work distribution method for cpu-gpu heterogeneous system. In *Proceedings of the 2010 International Symposium on Parallel and Distributed Processing with Applications*, 2010.

[67] Yi Wang, Yuan Zu, Ting Zhang, Kunyang Peng, Qunfeng Dong, Bin Liu, Wei Meng, Huicheng Dai, Xin Tian, Zhonghu Xu, Hao Wu, and Di Yang. Wire speed name lookup: A gpu-based approach. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 199–212, Lombard, IL, April 2013. USENIX Association.

[68] C. M. Wittenbrink, E. Kilgariff, and A. Prabhu. Fermi gf100 gpu architecture. *IEEE Micro*, 31(2):50–59, 2011.

[69] Shinae Woo and KyoungSoo Park. Scalable tcp session monitoring with symmetric receive-side scaling.

[70] Noa Zilberman, Yury Audzevich, G Adam Covington, and Andrew W Moore. Netfpga sume: Toward 100 gbps as research commodity. *IEEE micro*, 34(5):32–41, 2014.