Efficient Software Packet Processing on Heterogeneous Hardware Architectures

Eva Papadogiannaki

Thesis submitted in partial fulfillment of the requirements for the

Master of Science degree in Computer Science

University of Crete
School of Sciences and Engineering
Computer Science Department
University Campus, Voutes, Heraklion, GR-70013, Greece

Thesis Advisors:
Prof. Evangelos Markatos,
Dr. Sotiris Ioannidis

Heraklion, August 2017

This project has received funding from the European Union's H2020 research and innovation programme under grant agreements No 644571 and No 644312.

UNIVERSITY OF CRETE
COMPUTER SCIENCE DEPARTMENT

Efficient Software Packet Processing on Heterogeneous Hardware Architectures

Thesis submitted by
Eva Papadogiannaki
in partial fulfillment of the requirements for the
Master of Science degree in Computer Science

THESIS APPROVAL

Author: ______________________
Eva Papadogiannaki

Committee approvals: ______________________
Evangelos Markatos
Professor, Thesis Supervisor

____
Sotiris Ioannidis
Research Director, Thesis Supervisor

____
Panagiota Fatourou
Associate Professor, Committee Member

Departmental approval: ______________________
Antonios Argyros
Professor, Director of Graduate Studies

Heraklion, August 2017

Abstract

Heterogeneous and asymmetric computing systems are composed by a set of different processing units, each with its own unique performance and energy characteristics. Still, the majority of current network packet processing frameworks targets only a single device (the CPU or some accelerator), leaving the rest processing resources unused and idle. In this work, we propose an adaptive scheduling approach that supports heterogeneous and asymmetric hardware, tailored for network packet processing applications. Our scheduler is able to quickly respond to dynamic performance fluctuations that occur at real-time, such as traffic bursts, application overloads and system changes. Our experimental results show that our system is able to process data in real-time while maintaining high efficiency in terms of energy consumption. Specifically, our system is able to match the peak throughput of a diverse set of packet processing applications, adapting to real-time fluctuating incoming traffic rates, while consuming up to 3.5x less energy.

Περίληψη

Τα ετερογενή υπολογιστικά συστήματα αποτελούνται από ένα σύνολο διαφορετικού τύπου μεταξύ τους συσκευών. Κάθε τέτοιο διαφορετικό είδος συσκευής χαρακτηρίζεται συνήθως από μοναδικές επιδόσεις και ξεχωριστά χαρακτηριστικά κατανάλωσης ενέργειας. Ακόμα, η πλειοψηφία των σύγχρονων εφαρμογών και βιβλιοθηκών υλοποίησης που αφορούν κυρίως την επεξεργασία πακέτων δικτύου δεν εκμεταλλεύονται κατάλληλα όλες τις διαθέσιμες συσκευές ενός συστήματος. Αντ΄ αυτού, στοχεύουν ένα συγκεκριμένο είδος συσκευής, είτε αυτή είναι ο κύριος επεξεργαστής του συστήματος (CPU) είτε είναι κάποιο είδος επιταχυντή (accelerator) , αφήνοντας έτσι τελείως αχρησιμοποίητες και αδρανείς τις υπόλοιπες συσκευές, οι οποίες υπάρχουν και είναι διαθέσιμες σε ένα σύστημα. Σε αυτήν τη δουλειά, προτείνουμε μία διαφορετική προσέγγιση για την οργάνωση και δρομολόγηση των εργασιών επεξεργασίας πακέτων δικτύου σε συστήματα που περιέχουν ετερογενείς συσκευές. Ο αλγόριθμος που υλοποιεί αυτήν την οργάνωση και δρομολόγηση των εργασιών στον κατάλληλο συνδυασμό από συσκευές, είναι ικανός να ανταποκριθεί γρήγορα στις διακυμάνσεις που συμβαίνουν δυναμικά και σε πραγματικό χρόνο, όπως για παράδειγμα, αυξομειώσεις στην εισερχόμενη κίνηση του δικτύου, υπερφόρτωση των εφαρμογών και αλλαγές μέσα στο σύστημα. Τα αποτελέσματα που προκύπτουν από τα πειράματα που πραγματοποιήσαμε αποδεικνύουν πως όντως το σύστημά μας είναι σε θέση να επεξεργαστεί δεδομένα σε πραγματικό χρόνο διατηρώντας πάντα την καλή επίδοση του, κρατώντας την κατανάλωση της ενέργειας σε χαμηλά επίπεδα. Συγκεκριμένα, τα αποτελέσματα δείχνουν πως το σύστημα μας μπορεί να φτάσει τα μέγιστα ποσοστά απόδοσης ανάμεσα σε διαφορετικές εφαρμογές επεξεργασίας πακέτων δικτύου, έχοντας τη δυνατότητα να ανταποκρίνεται γρήγορα και σε πραγματικό χρόνο στις διακυμάνσεις της εισερχόμενης κίνησης του δικτύου. Ταυτόχρονα, διατηρεί την κατανάλωση της ενέργειας σε πολύ χαμηλά επίπεδα, μέχρι και 3.5 φορές λιγότερη κατανάλωση.

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my supervisor Dr. Sotiris Ioannidis, not only for his continious guidance and encourangement, but also for teaching me to challenge myself and my capabilities. In addition, I wish to thank the members of the committee, Prof. Evangelos Markatos and Prof. Panagiota Fatourou for their comments and insights on this thesis.

Furthermore, I would like to express my gratitude to Dr. Giorgos Vasiliadis and Lazaros Koromilas for our insightful collaboration on this work. Also, I remain indebted to both of them for helping me thrive during my early days in the DCS laboratory. Furthermore, my deepest appreciation goes to Christos Papachristos for all his support (technical or not) and companionship.

Of course, the days in the lab could not be more amusing if it were not for my pals, Kostas Solomos, Kostis Kleftogiorgos, Elias Papadopoulos, Giorgos Christou, Eirini Degleri and all the rest. I would like to explicitely thank the seniors, Antonis Papaioannou, Panagiotis Papadopoulos and Panagiotis Ilia for always offering me guidance and advice. Last but not least, I wish to express my gratitude to Dimitris Deyannis for our collaboration through all these years (especially, for being an Arch Linux enthusiast, willing to bother with all the trouble, every time my serenity was disturbed).

Finally, I would like to thank my family for selflessly supporting my choices, my best quadruped friend, Rocky, for helping me calm (playing or cuddling) and my girlfriends for always being there waiting for me, while I was away, working on my bugs. Of course, I should not omit to praise the existence of e-FOOD.gr and Netflix that made depressing days a little more tolerable.

This work has been performed at the **Distributed Computing Systems laboratory, Institute of Computer Science, Foundation of Research and Technology – Hellas (FORTH)**. In addition, this project has received funding from the European Union's H2020 research and innovation programme under grant agreements No 644571 (H2020 ICT-32-2014 Project SHARCS) and No 644312 (H2020 ICT-07-2014 Project RAPID).

Part of this work has been published in the IEEE/ACM Transactions on Networking (Volume: 25, Issue: 3, June 2017) [1].

Contents

1 Introduction **1**

2 Background **5**
2.1 Hardware Architectures 5
2.1.1 Architectural Comparison 6
2.1.2 Quantitative Comparison 7
2.2 The OpenCL Framework 7
2.3 The Netmap Framework 9

3 System Setup **13**
3.1 Hardware Setup 13
3.1.1 Power Instrumentation 14
3.2 Packet processing applications 15

4 Architecture **19**
4.1 Master-worker Model 19
4.2 Shared-nothing Model 20
4.3 Packet-processing Parallelization 21
4.3.1 Batch Processing 26
4.3.2 Performance Characterization 26
4.3.3 Energy Consumption and Efficiency 32

5 Efficiency via Scheduling **37**
5.1 Initializing the Scheduler 38
5.2 Online Adaptation Algorithm 39
5.2.1 Algorithm Analysis 41

6 Evaluation **43**
6.1 Throughput 44
6.2 Energy Efficiency 44
6.3 Latency 45
6.4 Traditional Performance Metrics 45

7 Related Work **49**
7.1 Hardware acceleration 49
7.2 Kernel concurrency 49
7.3 Load balancing and kernel distribution 50
7.4 Task offloading 50
7.5 Execution training 50
7.6 Usage predictability 51

8 Discussion **53**
8.1 Power Instrumentation 53
8.2 Application Concurrency 53

9 Conclusion **55**

List of Figures

2.1 Architectural comparison of packet processing on an (a) integrated and (b) discrete GPU. 6
2.2 Scalar vs. SIMD operations 8
2.3 OpenCL's kernel distribution among different OpenCL-compliant devices. 9
2.4 In netmap mode, the NIC rings are disconnected from the host network stack, and become able to exchange packets through the netmap API. Two additional netmap rings let the application talk to the host stack directly. . 10

3.1 Our power instrumentation scheme. We use four current sensors to monitor (real-time) the consumption of the CPU, GPU, DRAM and miscellaneous motherboard peripherals. 17

4.1 Different models for capturing the network traffic and distributing it to different computational devices for processing. 22
4.2 Network packet forwarding throughput for 60- and 1500-byte packets sustained on the (a) "master/worker" and (b) "shared-nothing" architectures . 25
4.3 Throughput, latency and power consumption of the base system for (a) IPv4 packet forwarding, (b) MD5 hashing, (c) Deep Packet Inspection, and (d) AES-CBC 128-bit encryption, when processing 1514-byte packets on a single device. 27
4.4 Throughput, latency and power consumption of the base system for (a) IPv4 packet forwarding, (b) MD5 hashing, (c) Deep Packet Inspection, and (d) AES-CBC 128-bit encryption, when processing 1514-byte packets on combinations of devices. 28
4.5 Power consumption of different GPUs on AES-CBC. 31
4.6 Performance-per-watt of different GPUs on AES-CBC (Throughput per Watt). 31
4.7 Energy efficiency of different computational devices. 32
4.8 Energy efficiency of different combinations of computational devices. . . 33

6.1 Automatic device configuration selection under different conditions for the IPv4 packet forwarding and the MD5 applications. Optimized for maximum energy efficiency. 46

6.2 Automatic device configuration selection under different conditions for the DPI and the AES-CBC applications. Optimized for maximum energy efficiency. 47

List of Tables

3.1 The hardware setup of our base system that we used for our experiments. 15
3.2 The specifications of the different GPUs that we used for our experiments. 16

Chapter 1

Introduction

The advent of commodity heterogeneous systems (i.e. systems that utilize multiple processor types, typically CPUs and GPUs) has motivated network developers and researchers to exploit alternative architectures, and utilize them in order to build high-performance and scalable network packet processing systems [2, 3, 4], as well as systems optimized for lower power envelop [5]. Unfortunately, the majority of these approaches often target a single computational *device* [1], such as a multicore CPU or a powerful GPU, leaving the other devices idle. Consequently, developing an application that can utilize *each* and *every* available device effectively and consistently, across a wide range of diverse applications, is highly challenging.

Heterogeneous, multi-device systems typically offer system designers different optimization opportunities that raise inherent trade-offs between energy consumption and various performance metrics — in our case, forwarding rate and latency. The challenge to fully tap a heterogeneous system, is to effectively map computations to processing devices — in the most automated way possible. Previous work attempted to solve this problem by developing load-balancing frameworks that automatically partition the workload across the devices [6, 7].These approaches either assume that all devices provide equal performance [6] or require a series of small execution trials to determine their relative performance [7].The disadvantage of such approaches is that they have been designed for applications that take as input constant streaming data, and as a consequence, they are

[1] Hereafter, we use the term device to refer to computational devices, such as CPUs and GPUs, unless explicitly stated otherwise.

slow to adapt when the input data stream varies. This makes them extremely difficult to apply to network infrastructure, where traffic variability [8, 9] and overloads [10] significantly affect the utilization and performance of network applications.

In this paper, we propose an adaptive scheduling approach that exploits highly heterogeneous systems and is tailored for network packet processing applications. Our proposed scheduler is designed to explicitly account for the heterogeneity of *(i)* the hardware, *(ii)* the applications and *(iii)* the incoming traffic. Moreover, the scheduler is able to quickly respond to dynamic performance fluctuations that occur at run-time, such as traffic bursts, application overloads and system changes.

The contributions of our work are:

- We characterize the performance and power consumption of several representative network applications on heterogeneous, commodity multi-device systems. We show how to combine different devices (i.e. CPUs, integrated GPUs and discrete GPUs) and quantify the problems that arise by their concurrent utilization (Chapter 4).

- We show that the performance ranking of different devices has wide variations when executing different classes of network applications. In some cases, a device can be the best fit for one application, and, at the same time, the worst for another (§ 4.3.2).

- Motivated by the previous deficiency, we propose a scheduling algorithm that, given a single application, effectively utilizes the most efficient device (or group of devices) based on the current conditions (§ 5.2). Our proposed scheduler is able to respond to dynamic performance fluctuations —such as traffic bursts, application overloads and system changes— and provide consistently good performance (Chapter 6).

The rest of the thesis is structured as follows. Chapter 2 presents the essentials for a better comprehension of our implementation. In Chapter 3 we show our hardware and power instrumentation setup, while we present the network packet processing applications that we implement and use to measure our benchmarks. In Chapter 4 we demonstrate

two different architectures for the packet capturing and we compare their performance. Also, we discuss about the parallelization of the packet processing and the performance characterization of the different combinations of devices available in our system. Then, in Chapter 5, we present our proposed scheduling approach using the *online adaptation algorithm*. In Chapter 6 we evaluate the efficiency and adaptability of our scheduler, and finally, in Chapter 7 we discuss the related work.

Chapter 2

Background

In this chapter we provide some basic information that is essential for the reader in order to cope with this thesis. Firstly, we present the different hardware architectures that offer the feature of heterogeneity in our system. Then, we compare the different architectures and their performance characteristics (§ 2.1). In the rest of the chapter, we provide some background information regarding the OpenCL (§ 2.2) and Netmap (§ 2.3) frameworks.

2.1 Hardware Architectures

Typical commodity hardware architectures offer heterogeneity at three levels: *(i)* at the traditional x86 CPU architecture, *(ii)* at an integrated GPU, packed on the same processor die, and *(iii)* at a discrete high-end GPU. All three devices have unique performance and energy characteristics. Overall, the CPU cores are good at handling memory- and branch-intensive packet processing workloads, while discrete GPUs tend to operate efficiently in data-parallel workloads. Between those two, the integrated GPU features high energy efficiency without significantly compromising the processing rate or latency. Typically, the discrete GPU and the CPU communicate over the PCIe bus and they do not share the same physical address space (although this might change in the near future) [1]. The

[1] Recently, NVIDIA introduced a new GPU micro-architecture –called *Pascal*–, which has adopted the High Bandwidth Memory 2 *(HBM 2)* stacked memory and offers the NVLink technology. NVLink is a high-bandwidth bus that can connect the CPU with one or more GPUs and multiple GPUs, directly. NVLink allows higher transfer speeds, as it provides shared virtual memory across the CPUs and GPUs [11, 12]. However, using NVLink for commodity systems is not yet an option, since it is only supported by the IBM's Power8 chip [12, 13]. Consequently, the bottleneck of memory data transfers via the PCIe remains still.

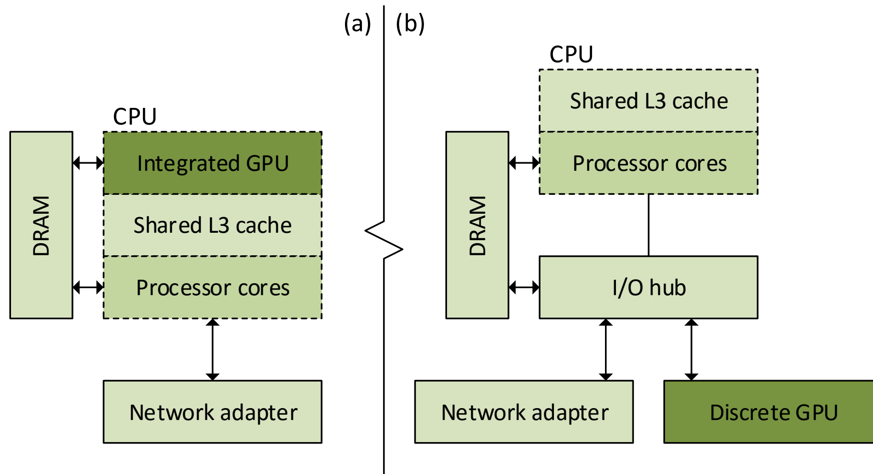


Figure 2.1: Architectural comparison of packet processing on an (a) integrated and (b) discrete GPU.

integrated GPU on the other hand, shares the LLC cache and the memory controller of the CPU.

2.1.1 Architectural Comparison

In Figure 2.1(b), we illustrate the packet processing scheme that has been used by approaches that utilize a discrete GPU [14, 3, 15, 16]. The majority of these approaches perform a total of seven steps (assuming that a packet batch is already in the NICs internal queue): the DMA transaction between the network interface and the main memory, the transfer of the packets to the I/O region, which corresponds to the discrete GPU (this operation traditionally invokes CPU caches, but the cache pollution can be minimized by using *non-temporal* data move instructions), the DMA transaction towards the memory space of the GPU, the actual computational GPU kernel itself and the transfer of the results back to the host memory. All data transfers must operate on fairly large chunks of data, due to the PCIe interconnect inability to handle small data transfers efficiently. The equivalent architecture, using an integrated GPU that is packed on the CPU die, is illustrated on the left side of Figure 2.1. The advantage of this approach is that the integrated GPU and CPU share the same physical memory address space, which allows in-place data processing. This results to fewer data transfers and hence lower processing latency. This scheme also has lower power consumption, as the absence of the I/O Hub alone saves

20W of energy, when compared to the discrete GPU setup of Figure 2.1(b).

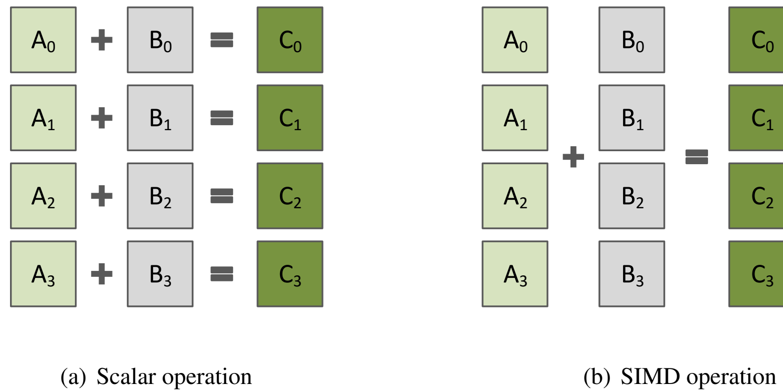
2.1.2 Quantitative Comparison

An integrated GPU (such as the HD Graphics 4000 we use in this work) has higher energy efficiency as a computational device, compared to modern processors and GPUs. The reason is threefold. First, integrated GPUs are typically implemented with low-power, 3D transistor manufacturing process. Second, they have a simple internal architecture and no dedicated main memory. Third, they match the computational requirements of applications, in which the main bottleneck is the I/O interface and thus, a discrete GPU would be under-utilized. In § 4.3.3 we show, in more detail, the energy efficiency of these devices when executing typical network packet processing applications.

2.2 The OpenCL Framework

OpenCL (Open Computing Language) is the open standard for cross-platform, parallel programming of diverse processors found in any heterogeneous system, from personal computers, servers, mobile devices to embedded platforms. A heterogeneous system mainly consists of CPUs, GPGPUs, FPGAs and other type of processors and hardware accelerators (e.g. the Intel Xeon Phi-coprocessor). OpenCL improves the performance of a wide spectrum of applications in categories, such as gaming, entertainment, scientific and medical software [17]. OpenCL provides a standard interface for parallel computing allowing task-based and data-based parallelism through the execution of the compute kernels. The major characteristics of OpenCL are (i) portability, (ii) standardized vector processing and (iii) parallel programming [18].

Portability A small sample of the OpenCL implementers consists of Intel, NVIDIA, Apple, ARM, AMD and IBM. Any vendor that provides OpenCL-compliant hardware and devices also provides the tools to compile the OpenCL code appropriately, in order to be executed on this specific hardware. Thus, the need to learn and write vendor-specific languages to program vendor-specific processors is eliminated –no separate compiler or linker is required. An OpenCL program can deploy executable code to any device as long as it is OpenCL-enabled.



(a) Scalar operation

(b) SIMD operation

Figure 2.2: Scalar vs. SIMD operations

Standardized vector processing OpenCL offers data structures that contain multiple elements of the same data type (e.g. 4 integers). As shown in Figure 2.2, during a vector operation, every element in the same data structure is operated upon in the same clock cycle –called SIMD programming (**S**ingle **I**nstruction **M**ultiple **D**ata). Nearly all modern processors can process vectors; however, vector instructions are mostly vendor-specific, since the ANSI C/C++ standard does not define and include any basic vector data types and structures. For instance, Intel processors use the SSE/AVX extensions to the x86 architecture, while NVIDIA devices use PTX instructions. Obviously, these two instruction sets do not have anything in common. However, Intel's OpenCL compiler will produce the appropriate SSE/AVX instructions, as NVIDIA's OpenCL compiler will produce the corresponding PTX instructions.

Parallel programming Besides *data* parallelism, OpenCL provides also *task* parallelism. In a *data* parallel system, one or more devices receive the same instructions and operate on different sets of data, while *task* parallelism enables the configuration of different devices to perform different tasks, and each task can operate on different data. In OpenCL, these tasks are named *kernels*. A kernel is the function that will be executed by one or more devices and is sent there by host applications. A host application is a regular C/C++ application that is responsible for dispatching kernels to the devices. Host applications manage the underlying OpenCL-enabled devices using a container that is called *context*. In order to dispatch a kernel, the host selects a function from a kernel container

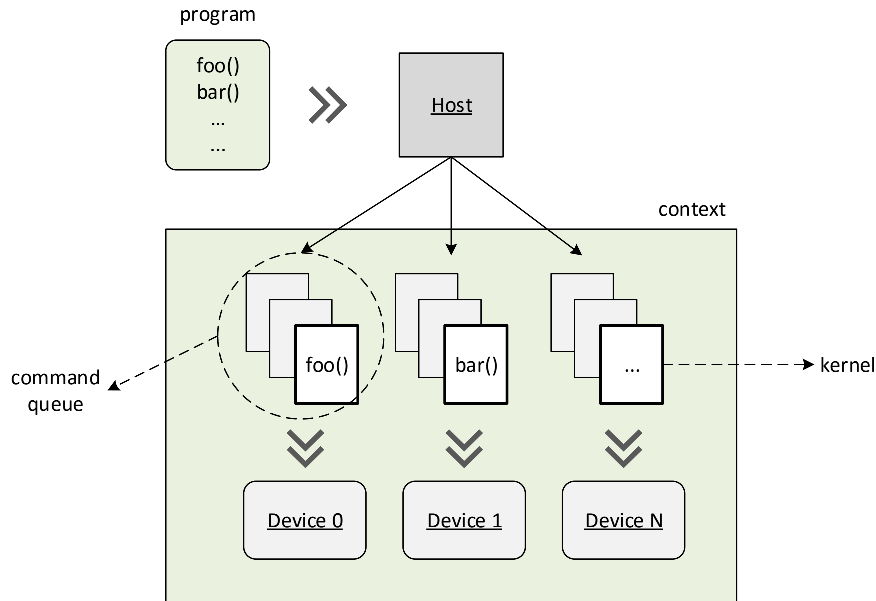


Figure 2.3: OpenCL's kernel distribution among different OpenCL-compliant devices.

called *program*. Next, the kernel is associated with the proper argument data and dispatched to a structure that is called *command queue*. Through the command queue, the host assigns work to the devices, and as soon as a kernel is enqueued, the device executes the equivalent function, as displayed in Figure 2.3.

2.3 The Netmap Framework

Netmap is a framework for high speed packet I/O, which is implemented as a kernel module for FreeBSD and Linux [19, 4]. It supports access to network cards (NICs), host stack, virtual ports, and "netmap pipes". *netmap* can easily do line rate on 10G NICs (14.88 Mpps). It can be used to build extremely fast traffic generators, monitors, software switches, network middleboxes, interconnect virtual machines or processes, do performance testing of high speed networking apps without the need for expensive hardware. In our case, we use the *netmap* framework, to exchange network traffic between two end-hosts.

Modern NICs support multiple transmit (Tx) and receive (Rx) queues, that different

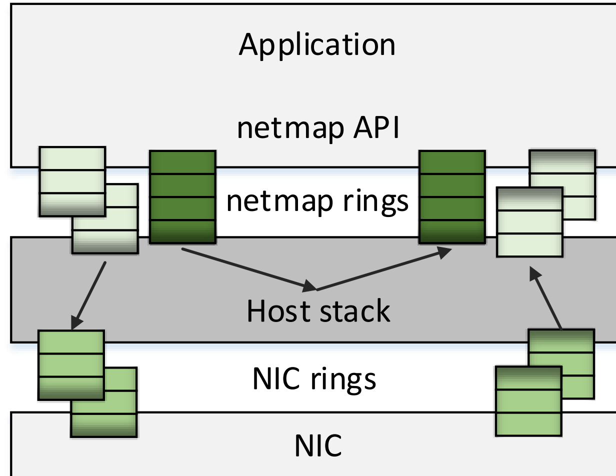


Figure 2.4: In netmap mode, the NIC rings are disconnected from the host network stack, and become able to exchange packets through the netmap API. Two additional netmap rings let the application talk to the host stack directly.

cores can use independently without need for coordination, at least in terms of accessing NIC registers and rings. Internally, the NIC schedules packets from the transmit queues into the output link and provides some form of demultiplexing so that the incoming traffic is delivered to the receive queues. Network adapters normally manage incoming and outgoing packets through circular queues (rings) of buffer descriptors. Each slot in the ring contains the length and physical address of the buffer. CPU-accessible registers in the NIC indicate the portion of the ring available for transmission or reception. On reception, incoming packets are stored in the next available buffer (possibly split in multiple fragments), and length/status information is written back to the slot to indicate the availability of new data. Interrupts notify the CPU of these events. On the transmit side, the NIC expects the OS to fill buffers with data to be sent. The request to send new packets is issued by writing into the registers of the NIC, which in turn starts sending packets marked as available in the TX ring.

This framework is a system to give user space applications very fast access to network packets, both on the receive and the transmit side, and including those from/to the host stack. *netmap* achieves its high performance through several techniques. One important

technique is the removal of data-copy costs by granting applications direct and protected access to the packet buffers. Also the same mechanism supports packet transfers between interfaces with zero-copy. Other hardware features, such as multiple hardware queues are also supported. Figure 2.4 represents a very high level view of a *netmap* application. When a program requests to put a network interface in *netmap* mode, the NIC is partially disconnected from the host protocol stack. The program gains the ability to exchange packets with the NIC and with the host stack, through circular queues of buffers –called *netmap* rings– implemented in shared memory [2]. Traditional OS primitives such as select() and poll() are used for the synchronization. Apart from the disconnection in the data path, the operating system remains unaware of the change so it still continues to use and manage the interface as during regular operation [4].

In our experiments, we use a 40-Gbps NIC, with 4 interfaces (10 Gbps each) and the *netmap* module for Linux [20].

[2] The figure is adapted from [4].

Chapter 3

System Setup

We will now describe the hardware setup, along with our power instrumentation and measurement scheme. Our scheme is capable of accurately measuring the power consumption of various hardware components, such as the CPU and GPU, in real time. We also describe the packet processing applications that we use for this work and show how we parallelize them using OpenCL, to efficiently execute in each of the three processing devices.

3.1 Hardware Setup

As shown in Table 3.1, our base system is equipped with one Intel Core i7-3770 Ivy Bridge processor and one NVIDIA GeForce GTX 780 Ti graphics card. The processor contains four CPU cores operating at 3.4GHz, with hyper-threading support, resulting in eight hardware threads, packed with an integrated HD Graphics 4000 GPU. Overall, our system contains *three* different, highly heterogeneous, computational devices: one CPU, one integrated GPU and one discrete GPU. The system is equipped with 8GB of dual-channel DDR3-1333 DRAM with 25.6 GB/s throughput. The L3 cache (8MB) and the memory controller are shared across the CPU cores and the integrated GPU. Each CPU core is equipped with 32KB of L1 cache and 256KB of L2 cache. The GTX 780 Ti has 2880 cores in 15 multiprocessors and 3072 MB of GDDR5 memory. It is rated at 5040 GFlops, and its Thermal Design Power (TDP) [1] is 250 Watts. The HD Graphics 4000 has 16 execution units, a 64-hardware thread dispatcher and a 100 KB texture cache. The maximum estimated performance of the integrated GPU is rated at 294 GFlops

[1] The thermal design power (TDP) is the maximum power a processor can draw for a thermally significant period while running commercially useful software [21].

on the maximum operating frequency of 1150 Mhz [22]. While Intel does not provide the TDP limit of the distinct parts, we estimate that the TDP of the integrated GPU is close to 17 Watts. For the whole processor die the TDP is 77 Watts. We notice that our hardware platform exposes an interesting design trade-off: even though the integrated GPU has fewer resources (i.e. hardware threads, execution units, register file) than a high-end discrete graphics card, it is directly connected to the CPU and the main memory via a fast on-chip ring bus, and has much lower power consumption. As we will see in § 4.3.2, this design is well-suited for applications in which the overall performance is limited by the I/O subsystem, and not by the computational capacity.

In addition, to compare the performance characteristics of different modern GPUs, we also use one NVIDIA GeForce GTX 770 and one NVIDIA GeForce GTX 980, besides the already mentioned NVIDIA GeForce GTX 780 Ti graphics card. As shown in Table 3.2, the GTX 770 has 1536 cores in 8 multiprocessors and 2048 MB of GDDR5 memory. It is rated at 3213 GFlops, and its Thermal Design Power (TDP) is 230 Watts. The GTX 980 has 2048 cores in 16 multiprocessors and 4096 MB of GDDR5 memory. It is rated at 4616 GFlops, and its Thermal Design Power (TDP) is 165 Watts.

3.1.1 Power Instrumentation

To accurately measure the power consumption of our hardware system, we have designed the hardware instrumentation scheme shown in Figure 3.1. Our scheme is capable of high-rate, 1 KHz measurement, and also provides a breakdown of system consumption into four distinct components: (i) processor and network adapter, (ii) memory, (iii) discrete GPU and (iv) miscellaneous.

Specifically, we utilize four high-precision *Hall effect* current sensors [23] to constantly monitor the three ATX power-supply power lines (+12.0_a, +12.0_b +5.0, +3.3 Volts). The sensors [24], coupled with the interface kit [25], cost less than $110. The analog sensor values are converted into digital values, and transmitted over a USB interface through a dedicated data logger board to the processor. The data logger includes a high-speed analog-to-digital converter (ADC), operating at a frequency of 40 KHz. The output data produced by the ADC are continuously read by a custom firmware running on the board, which also applies a running-average filter and periodically interrupts the processor with a rate of 1 KHz to report the values. A daemon, running in our base server, periodically collects the measurements from the data logger, and makes them available for

CPU	
Model name	Intel Core i7-3770
Clock frequency (GHz)	3.4
CPUs	8
Threads per core	2
Cores per socket	4
Sockets	1
Discrete GPU	
Model name	NVIDIA GTX 780 Ti
Cores	2880
Base Clock (MHz)	875
Memory interface	GDDR5
Memory size (MB)	3072
FP performance (GFlops)	5040
TDP (Watts)	250
Integrated GPU	
Model name	HD Graphics 4000
System memory	
Size (GB)	8
Description	DDR3 @ 1600MHz

Table 3.1: The hardware setup of our base system that we used for our experiments.

monitoring and control. We take advantage of the physical layout to achieve a breakdown of the total power consumption: the $12V_a$ powers the discrete GPU, the $12V_b$ line powers the processor (along with the integrated GPU), the 5V line powers the memory, and the 3.3V line powers the rest of the peripherals on the motherboard. The $12V_b$ line also feeds the 10GbE NICs. To calculate their power consumption, we use a utilization-based model.

3.2 Packet processing applications

In this section we present the fundamentals of four different packet processing applications, implemented for the purposes of this work. These four typical packet processing applications are (i) IPv4 packet forwarding, (ii) encryption, (iii) deep packet inspection and (iv) packet hashing. The nature of these applications is diverse and requires either com-

Discrete GPUs			
Model name	GTX 770	GTX 780 Ti	GTX 980
Cores	1536	2880	2048
Base Clock (MHz)	1046	875	1126
Memory interface	GDDR5	GDDR5	GDDR5
Memory size (MB)	2048	3072	4096
FP performance (GFlops)	3213	5040	4616
TDP (Watts)	230	250	165

Table 3.2: The specifications of the different GPUs that we used for our experiments.

putational or memory resources. Thus, the performance achieved by these applications demonstrates how different kinds of applications behave on different types of hardware architectures, revealing significantly the main concepts of heterogeneity.

Packet forwarding An IPv4 packet forwarding is one of the simplest packet processing applications. Its main function is the reception and transmission of network packets from one network interface to another. Before packet transmission, the forwarder checks the integrity of the IP header, drops corrupted packets and rewrites the destination MAC address according to the specified configuration. Other functions include decrementing the Time To Live (TTL) field. If the TTL field of an incoming packet is zero, the packet is dropped and an ICMP Time Exceeded message is transmitted to the sender.

Encryption Encryption is used by protocols and services, such as SSL, VPN and IPsec, for securing communications by authenticating and encrypting the IP packets of a communication session. By employing end-to-end encryption, we can protect data flows between pairs of hosts, security gateways or both.

Deep packet inspection Deep packet inspection (DPI) is a common operation in network traffic processing applications. It is typically used in traffic classification and shaping tools, as well as in spam filters, network intrusion detection and prevention systems. For instance, network monitoring applications, such as network intrusion detection systems, are dedicated to inspecting the contents of a huge amount of network traffic against an ever increasing number of suspicious signatures. All these signatures are usually being preprocessed in a finite automaton that will be used later in order to match any suspicious

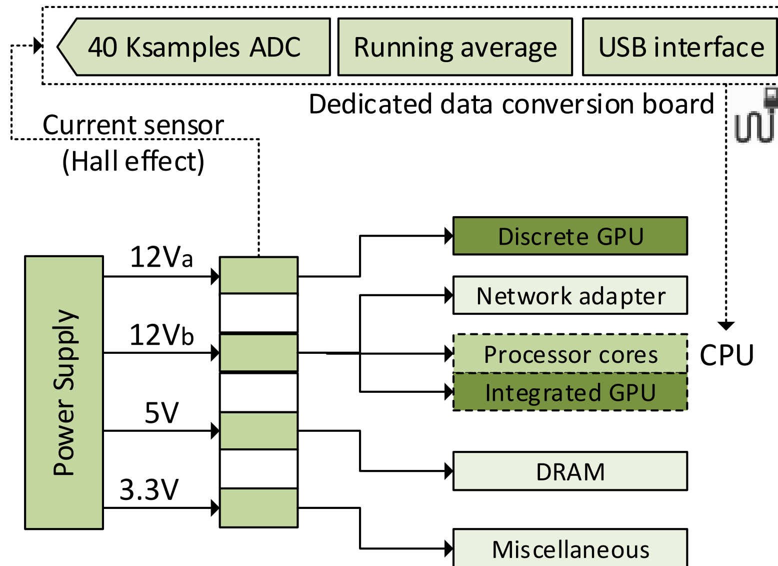


Figure 3.1: Our power instrumentation scheme. We use four current sensors to monitor (real-time) the consumption of the CPU, GPU, DRAM and miscellaneous motherboard peripherals.

incoming packets from the network.

Packet hashing Packet hashing is used in redundancy elimination and in-network caching systems [26, 27]. Redundancy elimination systems typically maintain a “packet store” and a “fingerprint table” (that maps content fingerprints to packet-store entries). On reception of a new packet, the packet store is updated, and the fingerprint table is checked to determine whether the packet includes a significant fraction of content cached in the packet store; if yes, an encoded version that eliminates this (recently observed) content is transmitted. The appliance located at the other end of the link maintains a similar packet store and thus is able to recover the original contents of the packet.

Chapter 4

Architecture

In this section we describe the architecture of our system. We describe two different models for capturing the network traffic and distributing it to different computational devices for processing. The first model, namely *master-worker*, consists of a single master thread that keeps track of the utilization and performance characterization of all heterogeneous processors, which also checks at runtime if they are active, in order to populate its buffer with new packets. In the second model, namely *shared-nothing*, the state is replicated to each worker thread, and is used at run-time to schedule the incoming flows to each active processor, without the need of synchronization. We then describe how packet processing is parallelized based on the characteristics of each computational device.

4.1 Master-worker Model

The simplest approach to capture the incoming network traffic and distribute it to different computational devices for further processing, is to utilize a separate thread. This thread, called *master-thread*, is responsible for capturing the network packets from the network interface, and offloading them to the corresponding computational devices for further processing. To avoid costly packet copies and context switches between user and kernel space, we use the netmap module [4]. The master thread is also responsible for keeping statistics, as well as the current utilization of each device. These statistics are very helpful for monitoring the load of the devices and distributing the incoming traffic based on the utilization and processing capabilities of each device accordingly. For instance, in cases of full resource utilization of a device, the master thread pauses feed-

ing it with network packets, enabling it to firstly process the already received packets. Each computational device is managed by a separate thread, called *worker-thread*, and is responsible for controlling its execution. The worker threads cooperate with the master thread, through a shared buffer, as shown in Figure 4.1(a). In particular, the master thread is in charge of filling the input buffers of each active device with packets received directly from the driver's Rx-queues (until each input buffer reaches a predefined maximum batch size). The worker threads are responsible for feeding those packets from the input buffer to the corresponding computational devices. When an input buffer of a device gets full, the master thread pauses filling the corresponding input buffer and waits the proper worker thread to consume the packets that are stored in the input buffer. Similarly, a worker remains idle while the input buffer has not reached the maximum number of packets received.

Specifically, to enable pipeline execution between the master thread and each computational device, we keep a buffer structure that consists of three buffers for each device: the input, the swap and the output buffers. As long as the input buffer becomes full, the worker switches all those packets to the swap buffer, and makes the input buffer free for new packets. Then, the buffer is copied to the device that is responsible for the processing, and spawns the device execution. After the execution, the results are transferred back from the device, while the statistics of that device are updated.

Finally, to maximize the performance, we assign NICs interrupt affinities and thread affinities, accordingly. After experimentation, we have found that the interrupt affinities of all NICs and the affinity of the master thread have to be pinned to the same CPU core. The remainder CPU-cores are used for hosting worker threads.

4.2 Shared-nothing Model

A major disadvantage of the master-worker architecture, is the excessive use of locks between the master thread and each worker thread independently. In addition, the master thread should be capable of handling the input traffic from several NICs. In our current system setup, the master thread utilizes about 60% of the CPU-core that is pinned to run. Adding more NICs, however, would require the operation of more master threads, which would lead to a demand of further synchronization between each other. This results to significant latency, and more importantly, to poor scalability.

To overcome the synchronization cost between the master and the worker threads,

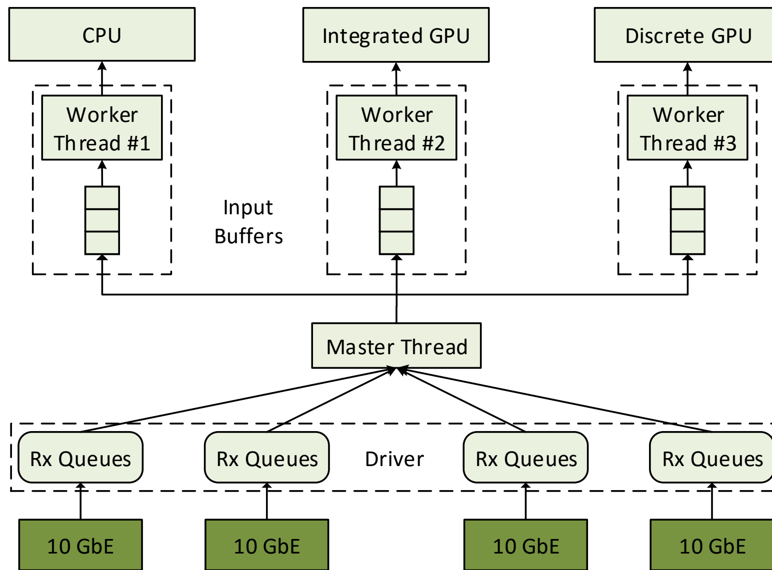
we propose a lock-free model. Instead of having a master thread to handle the incoming network traffic, we pair a single worker with a separate network interface, as shown in Figure 4.1(b). Then, the number of the worker threads is equal to the number of the network interfaces; each worker thread corresponds to a network interface. This architecture, which is mostly used on shared-memory models, splits the incoming traffic among the available CPU cores, letting each core consume and process the equivalent incoming packets.

Modern network cards partition incoming traffic into several Rx-queues [28] to avoid contention when multiple cores access the same 10GbE port. The worker threads are responsible for fetching packets from the Rx-queues and transferring them to the appropriate processing devices. Each worker thread keeps a buffer for each device existed in our system, independently. Hence, in our case, each of the four worker-threads (four interfaces) has three device-mapped-buffers (one buffer for the CPU, one for the integrated and one for the discrete GPU). A worker populates the input buffers of each device-mapped-buffer. As soon as one of those buffers reaches its maximum capacity, the worker swaps that buffer, copies it to the device, spawns the kernel and copies it back. Again, to avoid costly packet copies and context switches between user and kernel space, we use the netmap module [4]. In addition, to avoid synchronization between the worker threads, it is essential to update the statistics of each device separately.

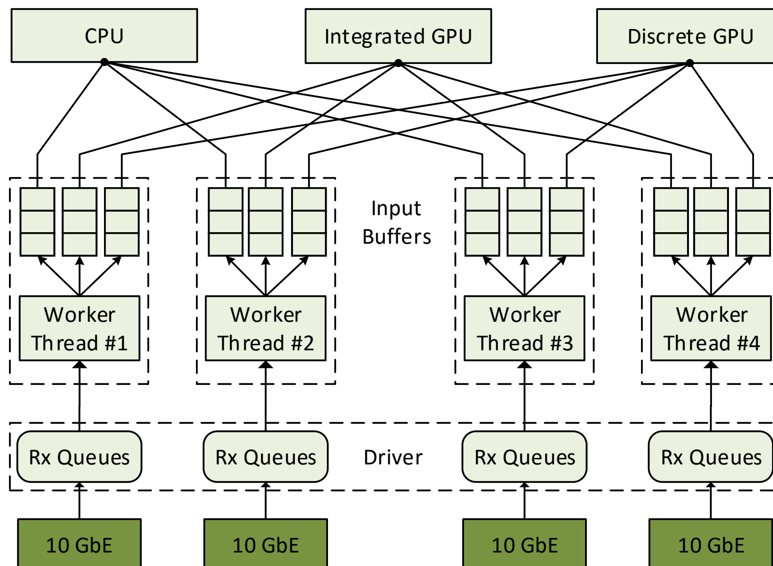
The advantages of this approach, compared to the master-worker architecture, is the alleviation of the overhead caused by the synchronization required to assure the proper execution of the worker threads. As we can see in Figure 4.2, for large-size packets, both architectures sustain similar performance. However, for small-size packets, the ”shared-nothing” architecture sustains more than three times better throughput. In the following sections, we present our work using the “shared-nothing” architecture for packet capturing.

4.3 Packet-processing Parallelization

To execute the packet processing applications uniformly across the different devices of our base system, we implement them on top of OpenCL 1.1. Our aim is to develop a portable implementation of each application, that can also run efficiently on each device. Our system runs Linux 3.19.3 with the in-tree i915 driver for the integrated graphics, and nvidia 352.30 driver for the discrete graphics. We use the Intel OpenCL Runtime 2014



(a) Master-worker Model



(b) Shared-nothing Model

Figure 4.1: Different models for capturing the network traffic and distributing it to different computational devices for processing.

R2 for the Core processor family, Beignet 1.0.2 for OpenCL on HD Graphics, as well as the OpenCL implementation that comes with NVIDIA CUDA Toolkit 7.0.

Each of our representative applications is implemented as a different compute kernel. In OpenCL, an instance of a compute kernel is called a *work-item*; multiple work-items are grouped together and form *work-groups*. We follow a thread-per-packet approach, similar to [29, 2, 3], and assign each work-item to handle (at least) one packet; each work-item reads the packet from the device memory and performs the processing. As different work-groups can be scheduled to run concurrently on different hardware cores, the choice of work-groups number raises an interesting trade-off: a large number of work-groups provides more flexibility in scheduling, but also increases the switching overhead. GPUs contain a significantly faster thread scheduler, thus it is better to spawn a large number of work-groups to hide memory latencies: while a group of threads waits for data fetching, another group can be scheduled for execution. CPUs, on the other hand, perform better when the number of different work-groups is equal to the number of the underlying hardware cores. Indeed, after experimentation on each of our applications, we have found that regarding GPUs it is better to spawn a large number of work-groups, while on CPUs it seems to be optimal to keep the number of work-groups close to the number of cores.

When executing compute kernels on the discrete GPU, the most crucial thing to consider is how to transfer the packets to and from the device. Discrete GPUs have a memory space that is physically independent from the host. To execute a task, explicit data transfers between the host and the device are required. The transfers are performed via DMA, hence the host memory region should be page-locked to prevent page swapping during the transfers. A data buffer required for the execution of a computing kernel has to be created and associated to a specific *context*; devices from different *platforms* (i.e. heterogeneous) cannot belong to the same context in the general case, and thus, cannot share data directly [1].

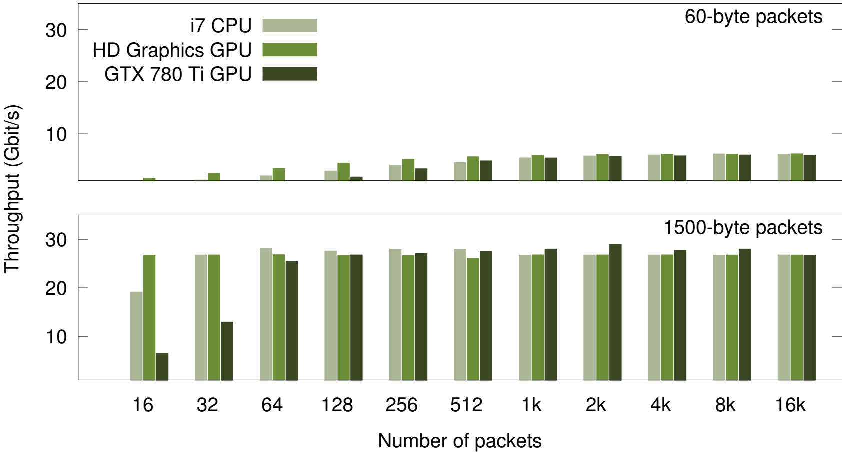
To overcome this, we explicitly copy received packets to a separate, page-locked buffer that has been allocated from the context of the discrete GPU and can be transferred safely via DMA. The data transfers and the GPU execution are performed asynchronously, to allow overlap of computation and communication and improve parallelism. Whenever a batch of packets is transferred or processed by the GPU, new packets are copied to another batch in a pipeline fashion. We notice that different applications require different

[1] Context sharing is available in the Intel OpenCL implementation[30]. However, we do not use it in this work, because it does not include our discrete NVIDIA GTX 780 Ti GPU.

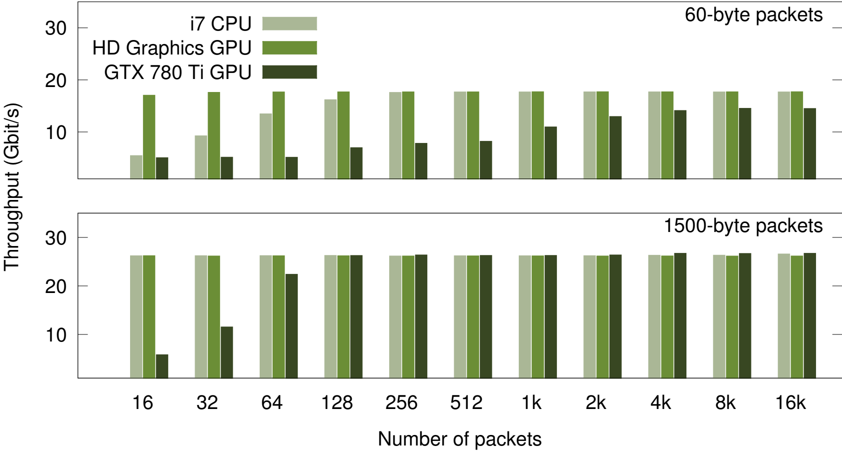
data transfers across the discrete GPU. For instance, DPI and MD5 do not alter the contents of the packets, hence it is not needed to transfer them back; they are already stored in the host memory. Packets have to be transferred back, when processed by the AES and the IP forwarding applications, as both applications alter their contents. Still, the IP packet forwarder processes and modifies only the packet headers. In order to prevent redundant data transfers, we only transfer the headers of each packet to and from the GTX 780 Ti, for the IP packet forwarding case; the packet headers are stored separately in sequential header descriptors (128 bytes each), a technique already supported by modern NICs [31]. Nevertheless, the data transfers are unnecessary when the processing is performed by the CPU or the integrated GPU, as both access directly the host memory. To avoid the extra copies, we explicitly map the corresponding memory buffers directly to the CPU and the integrated GPU, using the `clEnqueueMapBuffer()` function.

One important optimization is related to the way the input data are loaded from the device global memory. Accessing data in the global memory is critical to the performance of all of our representative applications. GPUs require column-major order to enable memory loads to be more effective, the so-called memory coalescing [32]. CPUs require row-major order to preserve the cache locality within each thread. As the impacts of the two patterns are contradictory, we firstly try to transpose the whole packets in GPU memory to column-major order, and benefit from memory coalescing. However, the overall costs of the corresponding data movements, pay off only when accessing the memory with small vector types (i.e. `char4`); when using the `int4` type though, the overhead is not amortized by the resulting memory coalescing gains, in none of our representative applications. Besides the GPU gains, the CPU enables the use of SIMD units when using the `int4` type, because the vectorized code is translated to SIMD instructions [33]. To that end, we re-design the input process and access the packets using `int4` vector types in a row-major order, for both the CPU and the GPU.

Finally, OpenCL provides a memory region, called *local memory*, that is shared by all work-items of a work-group. The local memory is implemented as an on-chip memory on GPUs, which is much faster than the off-chip global memory. Therefore, GPUs take advantage of local memory to improve performance. By contrast, CPUs do not have a special physical memory designed as local memory. As a result, all memory objects in local memory are mapped into sections of global memory, and will have a negative impact on performance. To overcome this, we explicitly stage data to local memory only when performing computations on the discrete GPU.



(a) Architecture "master-worker" for 60- and 1500-byte packets



(b) Architecture "shared-nothing" for 60- and 1500-byte packets

Figure 4.2: Network packet forwarding throughput for 60- and 1500-byte packets sustained on the (a) "master/worker" and (b) "shared-nothing" architectures

4.3.1 Batch Processing

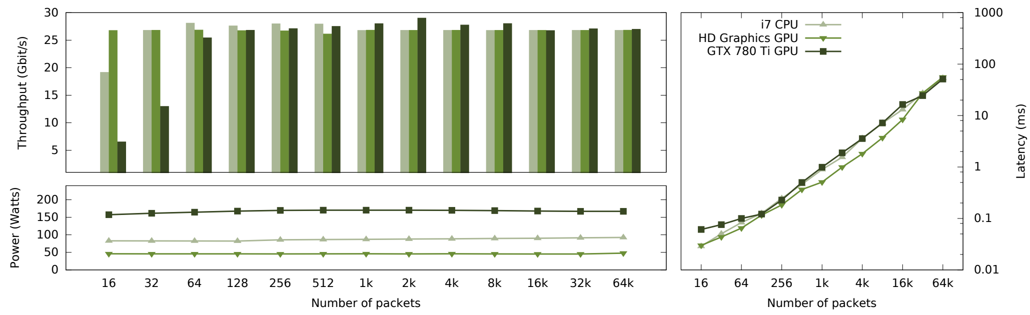
Network packets are placed into batches in the same order they are received. In case multiple devices are used simultaneously though, it is possible to be reordered. One solution to prevent packet reordering is to synchronize the devices using a barrier. By doing so, we enforce all the involved devices to execute in a lockstep fashion. Unfortunately, this would reduce the overall performance, as the fast devices will always wait for the slow ones. This can be a major disadvantage in setups where the devices have large computational capacity discrepancies. To overcome this, we firstly classify incoming packets to flows before creating the batches (by hashing the 5-tuple of each packet), and then, ensure that packets that belong to the same flow will never be placed in batches that will execute simultaneously to different devices. The batches are delivered to the corresponding devices, by the CPU core that is responsible to handle the traffic of each network interface. Each device has a different queue —that is allocated within the device's context— where newly arrived batches of packets are inserted.

4.3.2 Performance Characterization

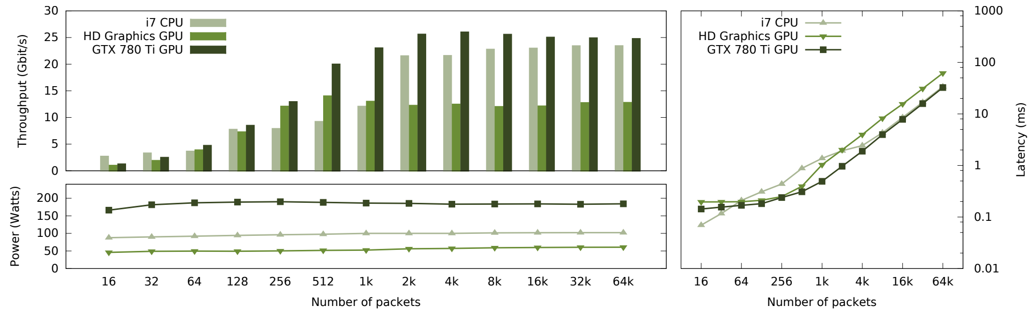
In this section we present the performance achieved by our applications. Specifically, we measure the sustained throughput, latency and power consumption for each of the devices that are available in our base system. Using the *netmap packet generator tool* we feed our base system with network packets originated from an end-to-end connected host (using four interfaces of 10GbE) [2].The reported performance measurements are presented using a different packet batch size each time.

To accurately measure the power spent for each device to process the corresponding batch, we measure the power consumption of *all* the components that are required for the execution. The reason behind this is that different devices require different components during their execution. For instance, when we use the GPU for packet processing, the CPU has to collect the necessary packets, transfer them to the device (via DMA), spawn a GPU kernel execution, and transfer the results back to the main memory. Instead, when we use the CPU (or the integrated GPU), we power-off the discrete GPU, as it is not needed. By measuring the power consumption of the right components each time, we can accurately compare different devices.

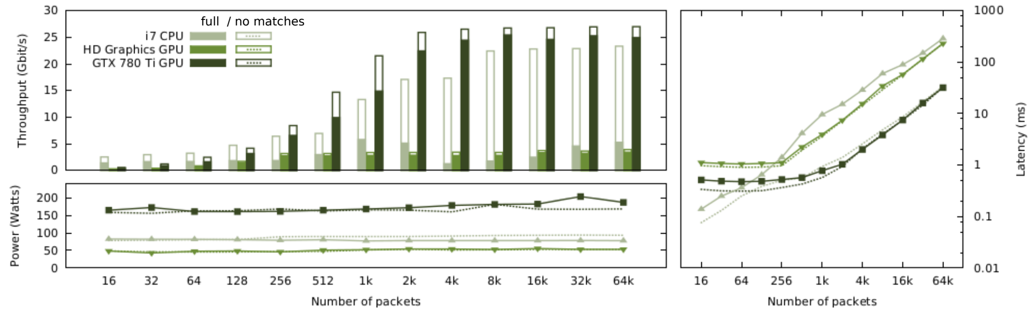
[2] For the DPI application, we alter each packet payload, accordingly, in order to provide performance measurements when our pattern matching engine reports 0% and 100% hits.



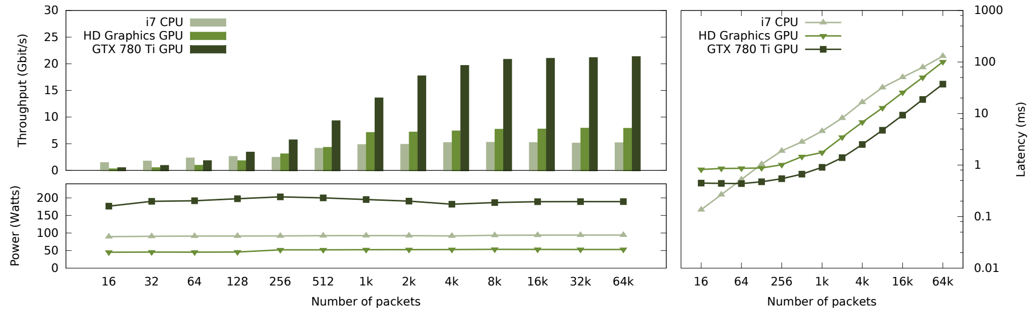
(a) IPv4 Packet Forwarding



(b) MD5

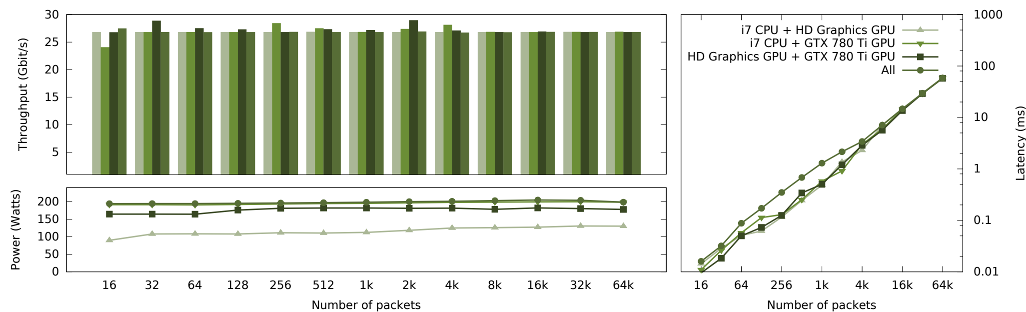


(c) DPI

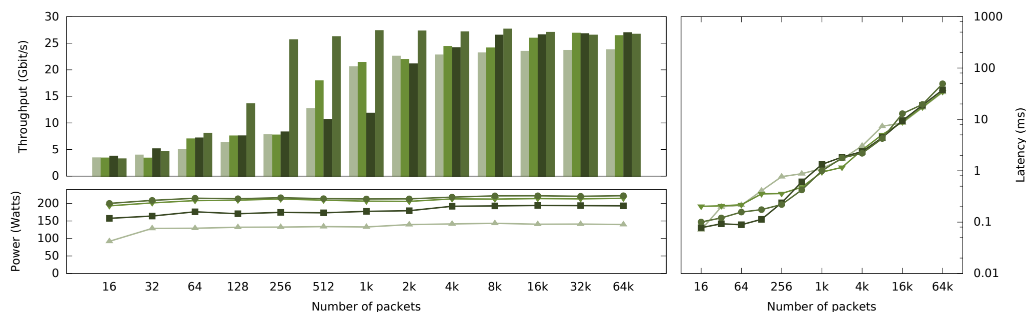


(d) AES-CBC

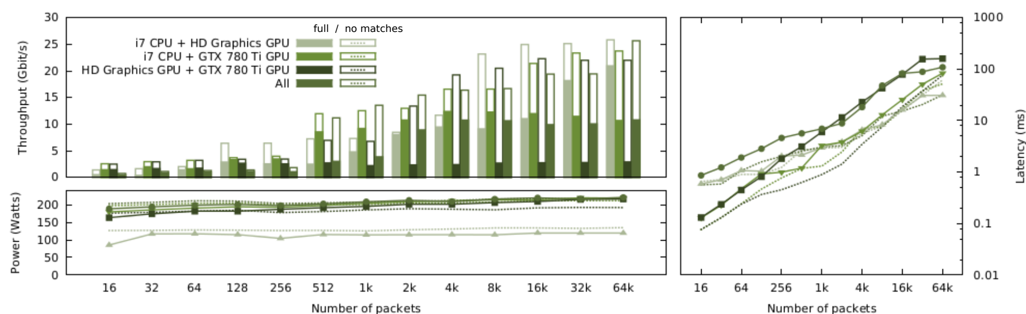
Figure 4.3: Throughput, latency and power consumption of the base system for (a) IPv4 packet forwarding, (b) MD5 hashing, (c) Deep Packet Inspection, and (d) AES-CBC 128-bit encryption, when processing 1514-byte packets on a single device.



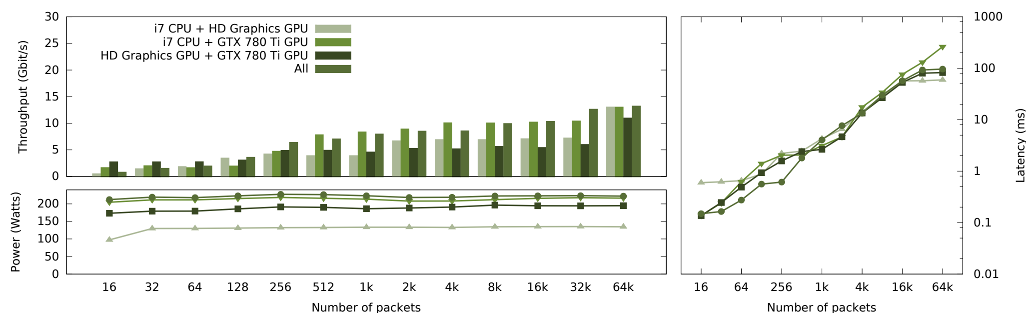
(a) IPv4 Packet Forwarding



(b) MD5



(c) DPI



(d) AES-CBC

Figure 4.4: Throughput, latency and power consumption of the base system for (a) IPv4 packet forwarding, (b) MD5 hashing, (c) Deep Packet Inspection, and (d) AES-CBC 128-bit encryption, when processing 1514-byte packets on combinations of devices.

Figures 4.3 and 4.4 summarize the characteristics of each of these types of packet processing during a “solo” run (one device runs the packet-processing application, while all the other devices are idle) and a “combo” run (more devices contribute to the packet-processing), respectively. In the combo run, the batch of packets needs to be further split into sub-batches of different size that will be offloaded to the corresponding devices. We have exhaustively benchmarked all possible combinations of sub-batches for each packet batch and pair of devices. Due to space constraints though, we plot only the best achieved performance for each case. In the case of the i7 processor, we include the results when using all four cores in parallel (“i7 CPU”). Note that in the IPv4 forwarding application, the reported throughput corresponds to the size of full packet data, even though only their headers are processed in separate header buffers, as we described in § 4.3. Concerning the IPv4 forwarding application, we observe that our performance –that reaches almost 30 Gbps– is inadequate, when compared with other similarly published performances. Other works tend to use the Xeon processor, configured with two NUMA nodes. Instead, we use the single-node i7 CPU, in order to take advantage of the integrated GPU. However, we justify that our system will reach a doubled performance, in terms of throughput, with a dual-node configuration.

We observe that the throughput is always improved when we increase the batch size. However, different applications (as well as the same application on different devices) require a different batch size to reach their maximum throughput. Computationally intensive applications (i.e. AES) benefit more from large batch sizes, while memory intensive applications (such as the IPv4 forwarding) require smaller batch sizes to reach the peak throughput. This is mainly the effect of cache sizes in the memory hierarchy of the specific device. For example, for the DPI in Figure 4.3(c) we see that a working set larger than 256 packets results in lower overall throughput for the CPU. Increasing the batch size, after the maximum throughput has been reached, results to linear increases in latency (as expected). As such, there is a not a single batch size that provides the best throughput for all applications and devices. Furthermore, we can see that the sustained throughput is not consistent across different devices. For example, the discrete GPU seems to be a good choice when performing DPI and AES on large batches of packets. The integrated GPU provides the most energy-efficient solution for all applications, however it cannot surpass the throughput of other devices. The CPU is the best option for latency-aware setups, as when using a small batch size (i.e., 16 packets), it can sustain processing times below 0.1–0.2 ms for all applications. In general though, there is not a clear ranking between

the devices, not even a single winner. As a matter of fact, some devices can be the best fit for some applications, and at the same time the worst option for another (as observed in the case of DPI and IPv4 forwarding when executing on the HD Graphics). Besides, we can see that the traffic characteristics can affect the performance of an application significantly. As we can see in Figure 4.3(c), the performance of DPI has large fluctuations; when there is no match in the input traffic the throughput achieved by all devices is much higher over the case where the matches overwhelm the traffic. The reason behind this is that as the number of pattern matches decreases, the DFA algorithm needs to access only a few different states. These states are stored in the cache memory, hence the overall throughput increases due to the increased cache hit ratio.

When pairing different compute devices, the resulting performance does not yield the aggregate throughput of the individual devices. For example, when executing MD5, the CPU yields 23 Gbit/s and the integrated GPU yields 12 Gbit/s, while when paired together they achieve only 24 Gbit/s. The reason behind this deviation is two-fold. First, when using devices that are packed in the same processor (i.e. the CPU and the integrated GPU), their computational capacity is capped by the internal power control unit, as they exceed the thermal constraints (TDP) of the processor. Second, they encounter resource interference, as they share the same last level cache. Actually, this is the case for all the pairs of devices, except in the IP forwarding case, where the CPU alone reaches the physical limits of the memory bandwidth, hence any extra device does not help to increase the overall throughput. When using all three devices, we can see that the overall throughput is always lesser than the throughput of the individual devices, as a result of high memory congestion.

Similarly, we compare the performance characteristics of three different modern high-end discrete GPUs; (i) one NVIDIA GeForce GTX 770, (ii) the NVIDIA GeForce GTX 780 Ti that is part of our base system and (iii) one NVIDIA GeForce GTX 980 [3]. For the comparison of the three GPUs, we choose to measure the performance of AES-CBC, the most power hungry application among the four.

In Figure 4.5, we see that GTX 770 is more power efficient than GTX 780 Ti for batch sizes smaller than 4K. Increasing the packet batch size while executing AES-CBC on GTX 770 results in the increment of the power consumption, making the GTX 780 Ti a better choice. However, considering a single performance metric – here, the power effi-

[3] The specifications of these GPUs are presented in Table 3.2

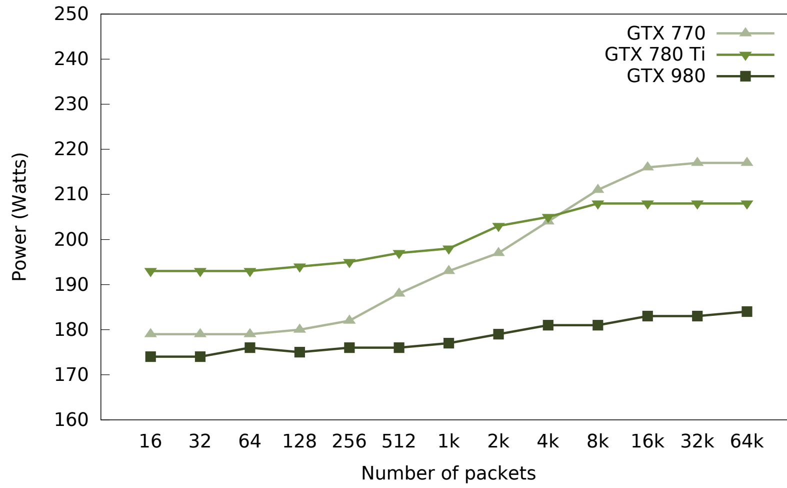


Figure 4.5: Power consumption of different GPUs on AES-CBC.

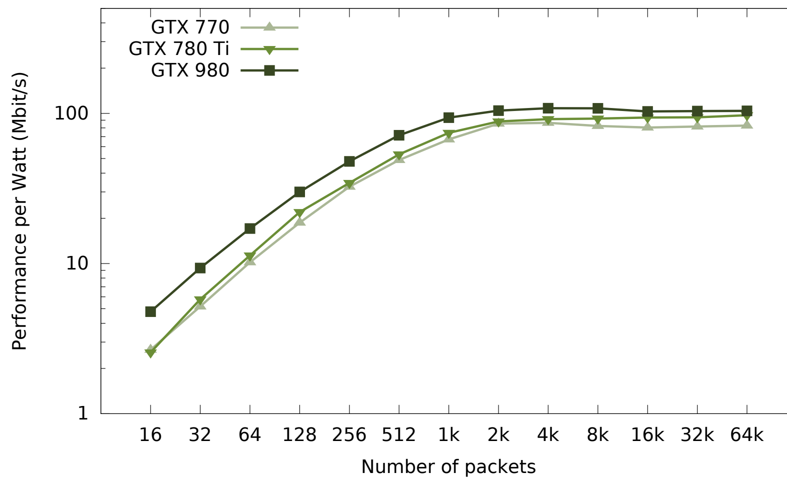


Figure 4.6: Performance-per-watt of different GPUs on AES-CBC (Throughput per Watt).

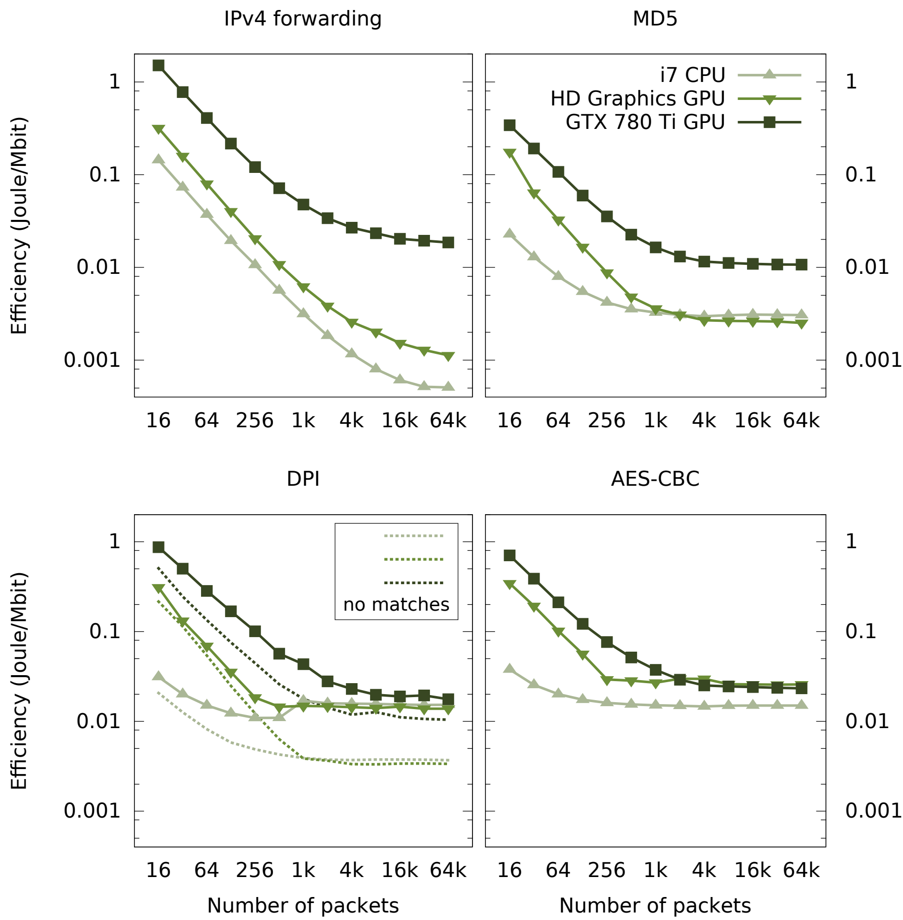


Figure 4.7: Energy efficiency of different computational devices.

ciency – does not always demonstrate the big picture. As we see in Figure 4.6, GTX 770, despite being more power efficient for smaller batch sizes (< 4K packets), GTX 780 Ti performs better in any batch size giving higher throughput per watt consumed. Obviously, GTX 980 is the true winner, performing the maximum throughput with the minimum power.

4.3.3 Energy Consumption and Efficiency

Figure 4.7 shows the energy efficiency of each packet processing application, on each computational device. The lines show the Joules that are needed to process one Mbit of data (the lower the better), under different batch size configurations (x-axis). We observe that IPv4 forwarding ends up (for the larger batch sizes) to be the most efficient applica-

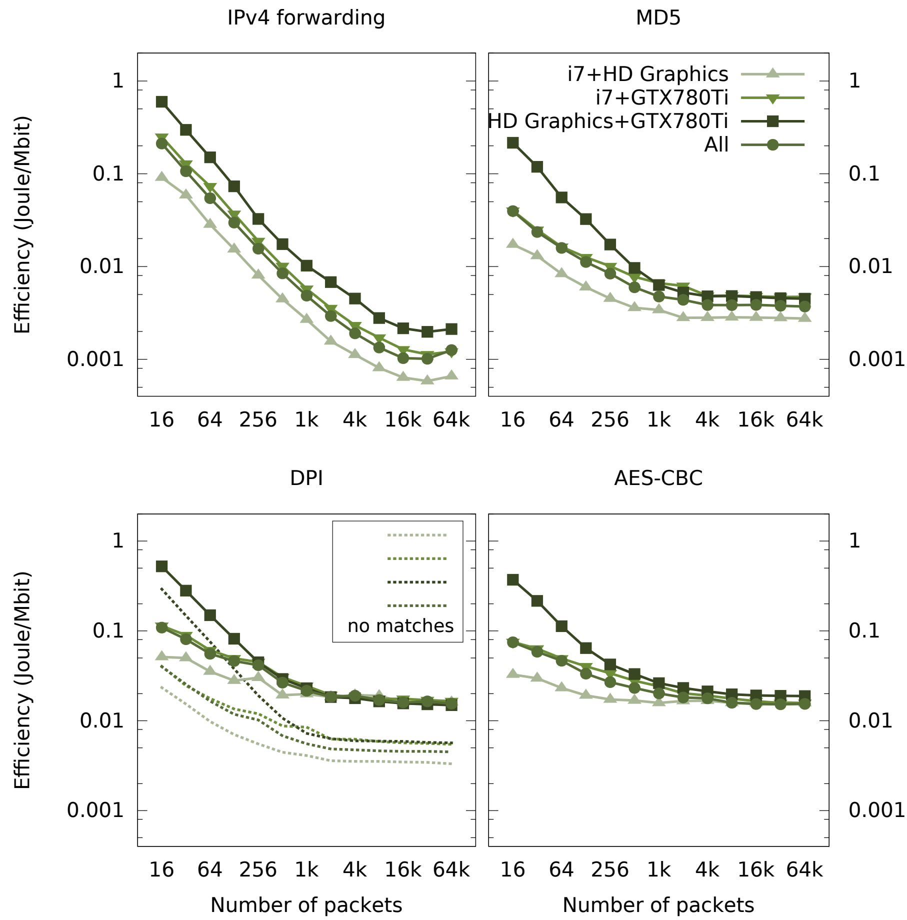


Figure 4.8: Energy efficiency of different combinations of computational devices.

tion when using the i7 CPU or the integrated HD-Graphics GPU, and at the same time the worst when utilizing the GTX 780 Ti discrete GPU. MD5 follows the same pattern, with the gap between the integrated and discrete GPU closing in. For the case of DPI (Deep Packet Inspection) and AES encryption in CBC mode, we can see that all devices converge to about the same efficiency; such large batch sizes, however, negatively affect latency and may be impractical to use for certain scenarios. Smaller batch sizes almost always have a clear winner. The dotted lines in DPI, show what happens when no pattern matches any of the input data, and thus the state transition table used in the matching engine is not really expertized; it serves as the best case and may be the common case for certain applications where matches are infrequent. In Figure 4.8 we see how different combinations of devices perform with respect to *energy efficiency*. Compared to single-device configurations, the combinations always perform worse, even though they deliver higher aggregate throughput. Among all, the least efficient device combination is the pair of the two GPUs (HD Graphics and GTX 780 Ti), especially for small batch sizes, where they remain under-utilized. On the other hand, the most efficient combination is the i7 CPU paired with its integrated HD-Graphics, which deliver both low consumption and acceptable throughput.

Ground truth measurements The reported power measurements have been conducted via our custom power instrumentation scheme. In order to check the validity of the measurements, we use equivalent tools, available for Linux systems –as in our base system. The NVIDIA System Management Interface (`nvidia-smi`) is a command line utility, intended to aid in the management and monitoring of NVIDIA GPU devices [34]. This utility allows users to query GPU device state and with the appropriate privileges, permits the modification of the GPU device state. Since the debut of the NVIDIA's Kepler architecture, `nvidia-smi` supports the power management utility, which is able to report the board's power draw in a live manner. This reported power draw conforms with the results of our power instrumentation scheme within an acceptable average error of 3 Watts [4].

Except for the `nvidia-smi` utility, which is intended specifically for NVIDIA's graphics processors, there is a number of other power management utilities, specialized for diverse hardware architectures and processor types. For instance, Intel's Processor Counter Monitor (PCM) [35] is an application programming interface, coming with a set of tools based on the API, which monitor performance and energy metrics of Intel pro-

[4]The `nvidia-smi` manual mentions that the power draw reading is accurate to within +/- 5 Watts.

cessors. Repeating the execution of the packet processing applications – this time using the Intel's PCM utility – led to the verification of our power measurements. Thus, we remain confident of the accuracy of the reports originated from our power instrumentation scheme.

Chapter 5

Efficiency via Scheduling

The performance characterization in Figures 4.3 and 4.4 indicates that there is not a clear ranking between the benchmarked computational devices. As a consequence of their architectural characteristics, some devices perform better under different metrics, while these metrics may also deviate significantly among different applications. As an example, the GTX 780 Ti achieves the best performance for the AES encryption but not the best performance for the IP forwarding. Additionally, the traffic characteristics can affect the performance achieved by a device. For example, DPI achieves the second best performance (almost 24 Gbit/s) on the i7 CPU while there are no matches on the input traffic. On the contrary, the rate falls significantly (at 6 Gbit/s), when the matches overwhelm the traffic (which is the one fifth of the performance sustained by a GTX 780 Ti).

It is obvious that our system faces heterogeneity in three levels: *(i)* the different processors, *(ii)* the diverse applications and *(iii)* the unstable incoming traffic rates. With these observations in mind, we propose an online scheduler, which is able to successfully adapt to any real-traffic state of a network, in order to maximize the performance. In the section below, we explain how we accomplish it.

Our scheduler explores the parameter space and selects a subset of the available computational devices to handle the incoming traffic for a given kernel. The goal of our method is to minimize: *(i)* energy consumption, and *(ii)* latency, or maximize throughput. Our scheduling consists of two phases. The first phase performs an initial, coarse profiling of each new application. In this phase, the scheduler learns the performance, latency and energy response of each device, in respect to the packet batching as well as the partitioning of each batch on every device. In the second phase, the scheduler decides

the best combination of available devices that meet a desired *target policy* (e.g. maximized processing throughput) and continually keeps track of the incoming traffic in order to adapt the batching and the batch partitioning.

5.1 Initializing the Scheduler

We first discover the best-performing configuration for each device; we then use these per-device configurations to also benchmark the remaining configurations comprised by combinations of devices. For each combination of our parameter space, we measure the sustained throughput, latency and power, and store them to a dictionary; the dictionary will be used at runtime in order to acquire the most suitable configuration. The time needed to compute the whole table requires 90–360 seconds, using a time quantum of 40 ms or a minimum of two samples, whichever comes last, for each configuration.

We use a different red black tree to store each achieved outcome (i.e. throughput, latency, and power) for each configuration. The motivation behind this is to allow throughput-, latency- and energy-aware applications, to find quickly the most appropriate configuration accordingly. At runtime, the corresponding metric (i.e. throughput, latency, and power) is used to acquire the most suitable configuration. The reason that we use a red black tree is to allow fast insertions/updates and (more importantly) support both exact and nearest-neighbor searches. Each node in the tree holds *all* the configurations that correspond to the requested result. In addition, since we have chosen to use the shared-nothing (*lock-free*) model for our implementation, it is essential to store the performances that each worker achieves, separately, in order to avoid using other synchronization techniques that lead to an overhead on latency. Therefore, we replicate the red black tree, so as to have *one* for *each* worker. A periodic signal, then, is in charge of selecting the one configuration that conduces toward the best performance among the trees.

We also reverse the value of power and latency, before normalizing them, as they represent less-is-better metrics. The motivation of using three indices is to allow an application to select *(i)* either the configuration that is best for a single requirement, or *(ii)* the configuration that achieves the best outcome for multiple requirements. As indices, we use priority queues as they can return the element with the maximum weight in $O(1)$. However, requirements (i.e. throughput, latency, and power) are measured with float precision. Therefore, exact matches will be very rare, at runtime. To overcome this, we can either round to the smallest integral value (e.g. to the nearest multiple of 100 Mbit/s),

or implement support for nearest neighbor search queries. By rounding to the smallest integral value we do not guarantee that a given value should be present in the tree; we have to explicitly fulfill the values for all missing neighbors. Since updating the values of all missing neighbors can be quite costly – especially in sparsely populated cases – we implement the latter solution. Therefore, if we do not have an exact match, we select the immediately nearest (either smaller or greater) match. Since we utilize a red black tree, the selection of the immediately nearest value can be obtained at the same cost. Moreover, in order to prevent from overloading the tree, before inserting a new node, we check if it differs with its parent by a threshold δ. If not, we merge them in order to save space. The δ threshold is also used as a parameter of our adaptation algorithm that is described in the next section.

5.2 Online Adaptation Algorithm

The goal of the online adaptation algorithm is to determine quickly and accurately which device or combination of devices is more suitable to sustain the current input traffic rate, and to be able to adapt to changes in traffic characteristics, with the minimum overhead possible. Moreover, it allows an application to get the most suitable configuration based on its own needs (i.e. throughput-, latency-, or energy-critical). For instance, it would be better for a latency critical application to submit the incoming traffic to more than one devices, while in an energy-critical setup it would be better to use only the most energy efficient device. Our scheduling algorithm is laid out as follows. We create a queue for each device, and place packet batches in those queues, according to the following iterative algorithm:

1. Measure the current traffic rate. Get the best configuration from the red black tree using the desired requirement (i.e. latency-, throughput-, or energy-aware). Change to this configuration only if it was measured better than the current one by a factor of λ. Initialize variables α and β.

2. Start creating batches of the specified size. If more than one devices are required, create batches for each device accordingly. The batches are inserted into the queue of the corresponding device(s).

3. Measure the performance [1] achieved by each of the devices for the submitted batch(es). If the sustained performance is similar to the one requested from the red black tree (up to a threshold δ), return to Step 1; otherwise, update the tree accordingly, and:
   - If the performance achieved by each device is worse, increase the batch size by a factor of α; set $\beta = \alpha/2$, and go to Step 3.
   - If the performance achieved by each device is better, decrease the batch size by a factor of β; set $\alpha = \beta/2$, and go to Step 3.

The scheduler gets continually cross-trained and improves as more network traffic is processed across different devices. Moreover, the scheduler can easily adapt to traffic-, system-, or application-changes. Traffic changes (such as traffic bursts) can easily be tolerated by our scheduler by quickly switching to the appropriate configuration (Step 1), without requiring to update the scheduler. In contrast, system- and application-changes should update the scheduler, accordingly; the loop, that starts at the Step 3 of the adaptation algorithm above, finds the best configuration of the given device for the current conditions. After that, the scheduler returns to Step 1, as more appropriate devices might exist to handle the current conditions. The purpose of the λ factor is to avoid alternating among competing configurations and just maintain a “good enough” state.

Thus, our scheduler can tackle system changes, such as throttling and contention, that may occur more frequently in the i7 Ivy Bridge processor, where multiple computational devices are integrated into a single package and sharing a single memory system and power budget. Application changes, such as in the case of the DPI which has large performance fluctuations according to the current traffic characteristics are also confronted. To prevent temporal packet loss, in the inter-time that our scheduler needs to adapt to the new conditions, we maintain queues of sufficient size for each device. In Chapter 6 we show that a few hundred MBs are sufficient to guarantee that no packet loss will occur, during any traffic-, system-, or application-changes.

For the experiments presented in this paper, we set the difference threshold, δ, between the expected and the measured performance to 10%, and the growth and decrease rate (i.e. α and β variables) to 2x; we found that these values provide the best average performance across the set of applications we studied.

[1] We chose to represent the three non-proportional metrics (latency, power consumption and throughput) with the term *performance*. Regarding latency and power, better performance means lower latency/power. The opposite stands for throughput, in which better performance means higher throughput.

5.2.1 Algorithm Analysis

The complexity of the algorithm, when searching for the configuration of a specific requirement, is $O(logN)$, where N is the total number of configurations. Indeed, the configurations are stored in a red black tree, hence the searching cost is $O(logN)$. Hence, the overall cost to acquire the most efficient configuration for a given requirement, is $O(logN)$. However, our adaptive algorithm requires that the given configuration should be updated, in case the sustained performance differs by a threshold δ. The update cost is equal to the cost required to find the node in the red black tree ($O(logN)$). After the update, the algorithm converges to the batch size that achieves the requested performance, if any (Step 3). This can take up to $log_{\alpha}M$ steps (or $log_{\beta}M$ equivalently), where M is the maximum batch size.

Chapter 6

Evaluation

We now evaluate the performance of our scheduling algorithm, using the packet processing applications described in § 3.2. We use an energy-critical policy, i.e. handle all input traffic at the maximum energy efficiency. In Figure 6.2(a)–(d) we present the applied and achieved throughput, the power consumption and the device selection made by the scheduler, for the four applications under study. We note that the power consumption is divided into three categories: the power consumption of the (i) GTX 780 Ti, (ii) Intel i7 die, and (iii) DRAM and other miscellaneous motherboard peripherals. For comparison, we also illustrate with a dashed line the power consumption when all three devices are used simultaneously. Additionally, we provide the experienced latency with a solid line. Latency variability is a result of dynamic scheduler decisions for the batching and computational device selection. The input traffic has the same profile for all applications and is comprised of 25% 60-byte TCP packets and 75% 1514-byte TCP packets. Overall, our scheduler adapts to the highly diverse computational demand among the selected applications, producing dynamic decisions that maintain the maximum energy efficiency during all times. Additionally, it sustains high throughput and avoids excessive latency when possible. Furthermore, our scheduler is able to respond to application specific performance characteristics. For example during DPI (Figure 6.2(c)), our algorithm detects the requirement for a different configuration at times (x axis) 10, 40 and 70. These times introduce packets with a high match rate (in contrast to the low previous match rate), where the target cannot be satisfied without the use of the energy-hungry GTX 780 Ti.

6.1 Throughput

We observe that our proposed scheduler is able to switch to the configuration that keeps the selected target, under the required computational capacity which is required to process the incoming traffic for each application. However, there are some cases, in which our architecture does not sustain the input traffic rate: *(i)* in the IPv4 Packet Forwarding, *(ii)* in the MD5 application, *(iii)* in the DPI application, and *(iv)* in the AES application, between the times 30 and 80. The reason is that there is not a device, or combination of devices, to handle these cases, as we have already seen in Figures 4.3 and 4.4. More specifically, the DFA used by the Aho-Corasick algorithm exhibits strong locality of reference when the traffic does not contain any pattern matches; however, when the traffic consists of many different patterns, it forces the DFA to follow different states, which subsequently decreases the spatial locality. The HD Graphics does not handle more than 5 Gbit/s.

6.2 Energy Efficiency

Our proposed scheduler consistently switches to the most energy efficient configuration at all rates for each application. The advantage of our approach is more noticeable when the load is fairly low (10 Gbit/s) as it switches to the energy-efficient integrated GPU. Especially for the IP forwarding, the integrated GPU is able to cope with the input traffic at all rates, providing a constant 50 W consumption, which is *two* times better over the CPU-only and *more than three* times over the discrete GPU only. Packet hashing switches to the CPU when the rate reaches 30 Gbit/s and then switches to the CPU-HD Graphics pair (at time 40) to handle the 40 Gbit/s input traffic rate. DPI follows a more composite behavior, as it is affected by both the traffic rate and characteristics (i.e. number of matches). Overall, DPI ends up utilizing the two GPU devices when processing full-matches traffic at rates of 20 Gbit/s or higher. Nevertheless, when the matches drop to zero, the CPU is able to cope the input traffic; at rates of 30 Gbit/s or higher the system employs the i7 CPU together with the HD Graphics. For all other input rates the CPU or HD Graphics alone can sustain the traffic. At time 50, we synthetically raise the number of matches that results to a temporal fall to 19 Gbit/s, before our scheduler considers to also utilize GTX 780 Ti, too. With increased rate, while keeping the number of matches at full ratio, we observe that there is no increase in the sustained rate because there is no better configuration available. AES, which is the most computationally intensive application in our

set, ends up using all three devices when the traffic rate exceeds 10 Gbit/s (time 0 to 110), and are able to handle up to 19 Gbit/s rate.

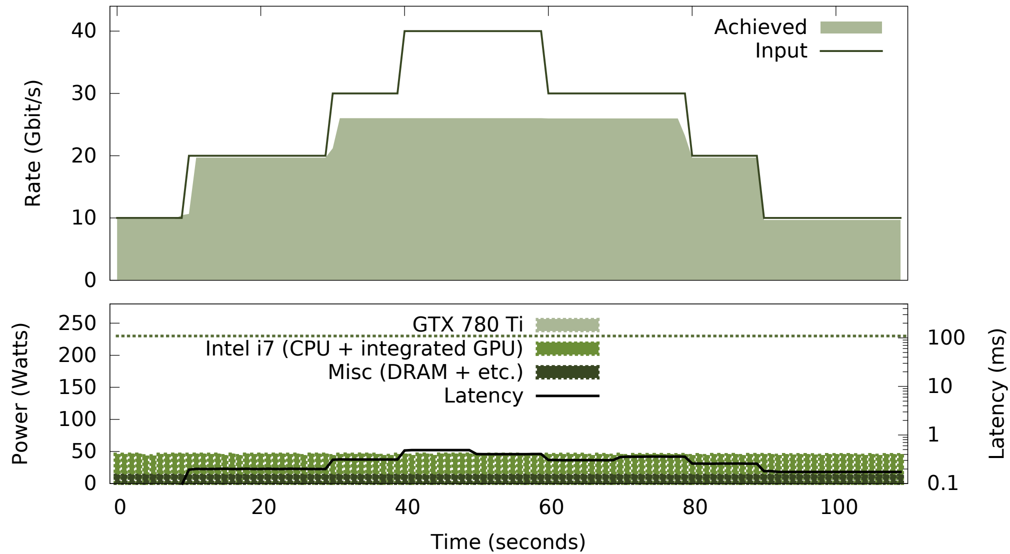
Overall, our scheduler reaches the maximum consumption in the following cases only: *(i)* when the traffic rate exceeds 10 Gbit/s for AES, and *(ii)* when the rate exceeds 20 Gbit/s for DPI, and is overwhelmed with matches. In DPI, interestingly enough, the HD Graphics plus GTX 780 Ti pair is the winner. Overall, our architecture yields an overall energy saving between 3.5 times (IPv4 forwarding) and 30% (AES-CBC) compared to the energy spent when using all three devices.

6.3 Latency

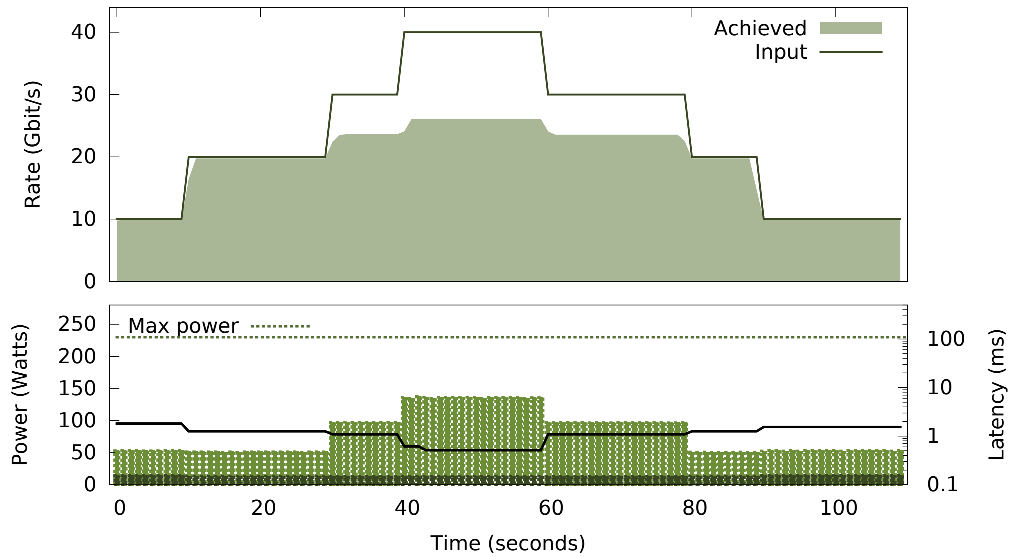
Increasing the batch size results in better sustainable rate at the cost of increased latency, especially for the GTX 780 Ti. IPv4 forwarding – executed solely on the HD Graphics – provides a latency that increases linearly with the batch size. However, this is not always the case. For example, in the case of the MD5 application, latency drops significantly in the time range: 30–80. The reason behind this is that the scheduler switches from the HD Graphics to the i7 CPU, in order to handle the increasing traffic rate. Given that the CPU is able to handle the requested traffic using a much smaller batch size, results to an extensive latency drop. Similar transitions occur in other applications as well, e.g. at times 20 and 100 for DPI. We note, though, that in our experiments we focus primarily on providing a minimum power utilization setup. By using a latency-aware policy, we can obtain much better latency, at the cost of increased power consumption.

6.4 Traditional Performance Metrics

In addition to the previous studied metrics, we measure other significant metrics which are present in the software packet processing domain, namely: packet loss, and reordering. Our algorithm may introduce packet loss by switching to a device too slowly in the face of varying traffic demands. Reordering may be introduced when packets belonging to the same flow are redirected to a different device. Regarding packet loss, our experiments show that our algorithm can react quickly enough to avoid packet drops. We observe that in all cases our proposed scheduler can adapt to changes in less than 300 ms – which is the case where we use the GTX 780 Ti with a batch size of 64K. This roughly results to 1.46 GB of received data (in a 40 GbE network, for a MTU of 1500 bytes), hence a buffer

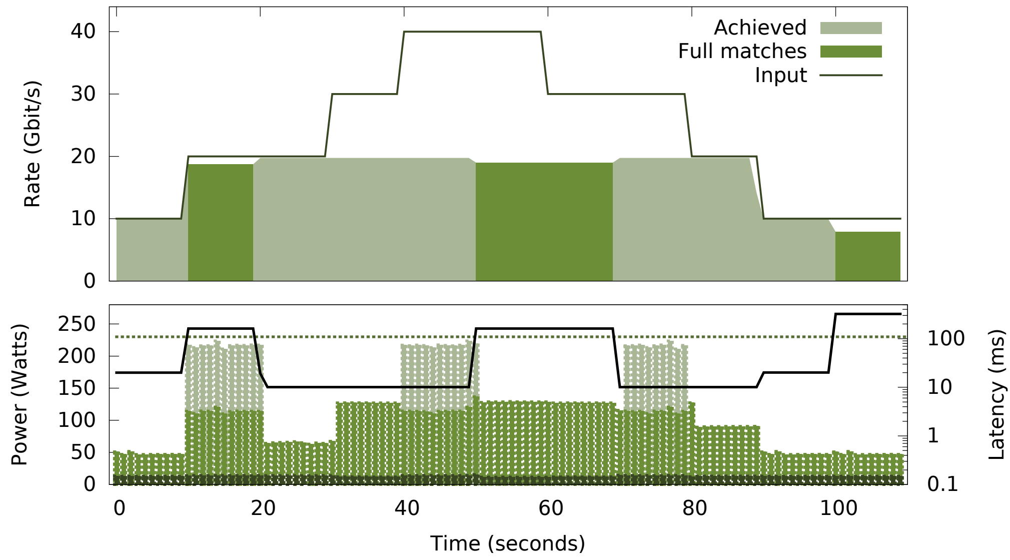


(a) IPv4 Packet Forwarding.

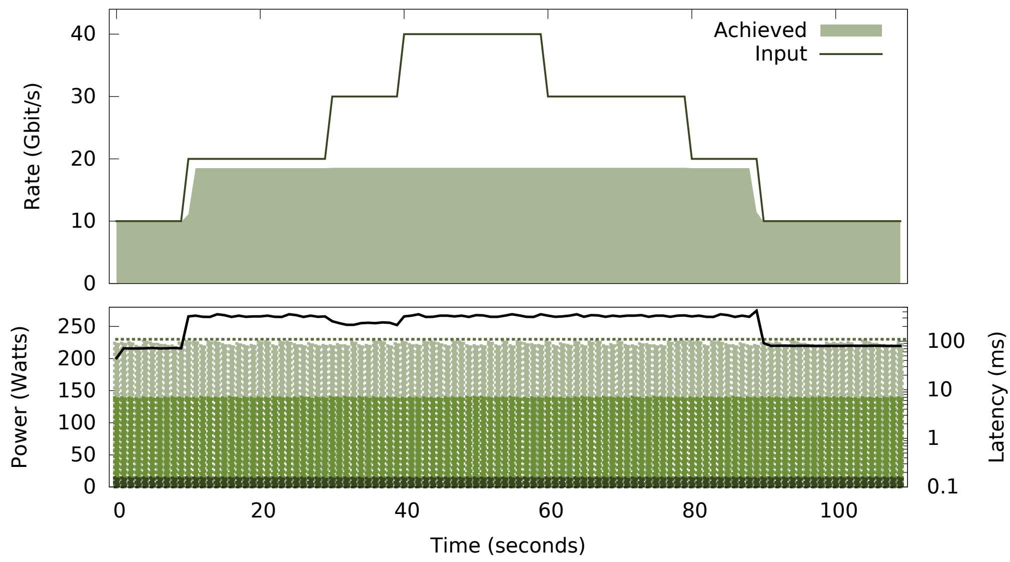


(b) MD5.

Figure 6.1: Automatic device configuration selection under different conditions for the IPv4 packet forwarding and the MD5 applications. Optimized for maximum energy efficiency.



(a) DPI.



(b) AES-CBC.

Figure 6.2: Automatic device configuration selection under different conditions for the DPI and the AES-CBC applications. Optimized for maximum energy efficiency.

of this size is sufficient to guarantee that no packet loss will occur in the inter-time that our scheduler needs to adapt to the new conditions. We notice however that this is the worst case, in which the input rate goes from zero to 40 Gbit/s and at the same time the algorithm pushes the system to a configuration with the worst latency (300 ms). In our experiments, using a 500 MB buffer was enough. Finally, we measure packet reordering. In our system, reordering can only occur when traffic is diverted to a new queue. However, as we have described in § 4.3.1 we ensure that packets with the same 5-tuple will never be placed in batches that will execute simultaneously to different devices, guaranteeing that the packets of the same flow will always processed by the same device.

Chapter 7

Related Work

7.1 Hardware acceleration

Recently, GPUs have provided a substantial performance boost to many individual network-related applications, including intrusion detection [14, 36, 3, 37, 3], cryptography [38], and IP routing [2]. In addition, several programmable network traffic processing frameworks have been proposed, such as Snap [39] and GASPP [16], that manage to simplify the development of GPU-accelerated network traffic processing applications. Knapp [40] is a packet processing framework destined for manycore accelerators, such as the Intel Phi coprocessor. Then, APUNet is an APU-accelerated network packet processing system, which exploits the integrated GPU for parallel packet processing while it utilizes the CPU for scalable packet I/O [41]. PIPSEA is an IPsec DPDK-based gateway tailored specifically for the architecture of an APU [41]. The main difference with these works is that we focus on building a software network packet processing framework that combines different, heterogeneous, processing devices and quantify the problems that arise with their concurrent utilization. By effectively mapping computations to heterogeneous devices, in an automated way, we provide more efficient execution in terms of throughput, latency and power consumption.

7.2 Kernel concurrency

Recently proposed load-balancing systems support applications with multiple concurrent kernels [42, 43, 44, 45]. FLEP [46] enables kernel preemption on GPUs, using a com-

pilation engine that is able to transform the GPU program into the proper preemptable form, which can be interrupted during execution and yield the streaming multi-processors in the GPU. Wang et al. propose a fine-grained GPU sharing mechanism, which supports control over the progress of kernels and the amount of thread-level parallelism of each kernel [47].

7.3 Load balancing and kernel distribution

Approaches to load-balance a single computational kernel include [6, 7, 48, 49]. The simplest approach [48] target homogeneous GPUs and require no training as they use a fixed work partition [6]. Wang and Ren propose a distribution method on a CPU-GPU heterogeneous system that tries a large number of different work distributions to find the most efficient [50]. Other approaches rely heavily on manual intervention by the programmer [51, 52].

7.4 Task offloading

Another work proposes treating the CPU as the primary processor, since it offers low latency, offloading processing tasks to accelerators only when they result in throughput benefit [53]. Unlike this work, we target more heterogeneous processors (including integrated GPUs that have high on-chip dependencies with the CPU), and also we take into consideration the power characteristics of each computational device.

7.5 Execution training

Other approaches require a series of small execution trials to determine the relative performance [7, 49]. The disadvantage of these approaches is that they have been designed for applications that take as input constant streaming data and, thus, they adapt very slowly when the input data stream varies. That makes them tough to be applied to network processing applications in which the heterogeneity of *(i)* the hardware, *(ii)* the applications, and *(iii)* the traffic vastly affect the overall efficiency in terms of performance and power consumption. To that end, our proposed scheduling algorithm has been designed to explicitly account for this.

7.6 Usage predictability

Ongoing work provides performance predictability [54] and fair queueing [55] when running a diverse set of applications that contend for shared hardware resources. There is also work on packet routing [5] that draws power proportional to the traffic load. The main difference with our work, is that they focus solely on homogeneous processing cores; instead we present a system that utilizes efficiently a diverse set of devices.

Chapter 8

Discussion

In the following paragraphs we mention the limitations of our work. These limitations arise from features that our system assumes and utilizes but are not available in current consumer products, as well as features that our system lacks and we would like to add in the future.

8.1 Power Instrumentation

Our scheduler requires live power consumption feedback for each of the available computational devices in the system. Even though such schemes have currently become common in commodity processors (e.g. the *Running Average Power Limit interface* present on latest Intel processor series), they are still in a preliminary stage in current graphics hardware architectures (although this is something that is expected to change in the near future). To overcome the lack of such power estimation in current GPU models, we propose the use of a power model as a substance of real instrumentation [56].

8.2 Application Concurrency

Another limitation of our architecture is the lack of optimization capabilities for concurrent running applications. The optimal parallel scheduling of an arbitrary application mixture is a highly challenging problem, mainly due to the unknown interference effects. These effects include but are not limited to: contention for hardware recourses (e.g. shared caches, I/O interconnects, etc.), software resources, and false sharing of cache blocks.

Moreover, the scheduler complexity grows exponentially with the introduction of multiple applications, as the parameter space should be explored for all possible application combinations. In this work we solely focus on optimizing the performance of a single active application that executes on a set of computing devices. As part of our future work we plan to experiment with application multiplexing and investigate the feasibility of a more generic energy-aware scheduler.

Chapter 9

Conclusion

In this work we address the problem of improving the efficiency of network packet processing applications on commodity, off-the-self, heterogeneous architectures. Heterogeneous systems can provide substantial performance improvements, but only with appropriately chosen partitioning. Using a static approach can lead to suboptimal performance when the state of traffic, system or application changes. To avoid this, we propose an online adaptive scheduling algorithm, tailored for network packet processing applications, that can *(i)* respond effectively to relative performance changes, and *(ii)* significantly improve the energy efficiency of packet processing applications. Our system is able to efficiently utilize the computational capacity of its resources on demand, resulting in energy savings ranging from 30% on heavy workload, up to 3.5 times for lighter loads. As part of our future work, we plan to extend our architecture in order to support the concurrent execution of various heterogeneous packet processing applications.

Bibliography

[1] E. Papadogiannaki, L. Koromilas, G. Vasiliadis, and S. Ioannidis, "Efficient software packet processing on heterogeneous and asymmetric hardware architectures," *IEEE/ACM Transactions on Networking*, 2017.

[2] S. Han, K. Jang, K. Park, and S. Moon, "PacketShader: a GPU-accelerated software router," in *Proceedings of SIGCOMM*, 2010.

[3] G. Vasiliadis, M. Polychronakis, and S. Ioannidis, "MIDeA: A Multi-Parallel Intrusion Detection Architecture," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, 2011.

[4] L. Rizzo, "netmap: A Novel Framework for Fast Packet I/O," in *Proceedings of the 2012 USENIX Annual Technical Conference*, 2012.

[5] L. Niccolini, G. Iannaccone, S. Ratnasamy, J. Chandrashekar, and L. Rizzo, "Building a Power-Proportional Software Router," in *Proceedings of the 2012 USENIX Annual Technical Conference*, 2012.

[6] J. Kim, H. Kim, J. H. Lee, and J. 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*, 2011.

[7] M. Boyer, K. Skadron, S. Che, and N. Jayasena, "Load Balancing in a Changing World: Dealing with Heterogeneity and Performance Variability," in *Proceedings of the ACM International Conference on Computing Frontiers*, 2013.

[8] G. Maier, A. Feldmann, V. Paxson, and M. Allman, "On dominant characteristics of residential broadband internet traffic," in *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement Conference*, 2009.

[9] T. Benson, A. Anand, A. Akella, and M. Zhang, "Understanding Data Center Traffic Characteristics," *SIGCOMM CCR*, vol. 40, no. 1, January 2010.

[10] S. A. Crosby and D. S. Wallach, "Denial of service via algorithmic complexity attacks," in *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, 2003.

[11] "Inside Pascal: NVIDIA's Newest Computing Platform," Available: https://devblogs.nvidia.com/parallelforall/inside-pascal/, Accessed on Dec. 28, 2016.

[12] "NVLink Takes GPU Acceleration To The Next Level," Available: https://www.nextplatform.com/2016/05/04/nvlink-takes-gpu-acceleration-next-level/, Accessed on Dec. 28, 2016.

[13] "IBM's new Power8 server packs in Nvidia's speedy NVLink interconnect," Available: http://www.pcworld.com/article/3117718/ibms-new-power8-server-packs-in-nvidias-speedy-nvlink-interconnect.html, Accessed on Dec. 28, 2016.

[14] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, and S. Ioannidis, "Gnort: High Performance Network Intrusion Detection Using Graphics Processors," in *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection*, 2008.

[15] M. Jamshed, J. Lee, S. Moon, I. Yun, D. Kim, S. Lee, Y. Yi, and K. Park, "Kargus: a Highly-scalable Software-based Intrusion Detection System," in *Proceedings of the 19th ACM Conference on Computer and Communications Security*, 2012.

[16] G. Vasiliadis, L. Koromilas, M. Polychronakis, and S. Ioannidis, "GASPP: A GPU-

Accelerated Stateful Packet Processing Framework," in *Proceedings of the 2014 USENIX Annual Technical Conference*, 2014.

[17] "OpenCL," Available: http://www.khronos.org/opencl/, Accessed on Dec. 7, 2016.

[18] M. Scarpino, "Foundations of opencl programming," *OpenCL in Action, second edition, Shelter Island NY, USA: Manning*, 2012.

[19] "The netmap project," Available: http://info.iet.unipi.it/~luigi/netmap/, Accessed on Feb. 28, 2017.

[20] "Netmap github repository," Available: https://github.com/luigirizzo/netmap, Accessed on Feb. 26, 2017.

[21] "Intel's whitepaper on measuring processor power," Available: http://www.intel.com/content/dam/doc/white-paper/resources-xeon-measuring-processor-power-paper.pdf, Accessed on May 4, 2017.

[22] *Intel HD Graphics DirectX Developer's Guide*, 2010.

[23] N. Gudino, M. J. Riffe, J. A. Heilman, and M. A. Griswold, "Hall effect current sensor," Feb. 19 2013, uS Patent 8,378,683.

[24] "1122_0 - 30 Amp Current Sensor AC/DC," Available: http://www.phidgets.com/, Accessed on Dec. 7, 2016.

[25] "1018_2 - PhidgetInterfaceKit 8/8/8," Available: http://www.phidgets.com/, Accessed on Dec. 7, 2016.

[26] A. Anand, A. Gupta, A. Akella, S. Seshan, and S. 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.

[27] B. Aggarwal, A. Akella, A. Anand, A. Balachandran, P. Chitnis, C. Muthukrishnan, R. Ramjee, and G. Varghese, "EndRE: an end-system redundancy elimination service for enterprises," in *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, 2010.

[28] Microsoft Corporation, *Scalable Networking: Eliminating the Receive Processing Bottleneck - Introducing RSS*, 2005.

[29] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy, "RouteBricks: Exploiting Parallelism to Scale Software Routers," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, 2009.

[30] *Intel SDK for OpenCL Applications 2013: Optimization Guide*, 2013.

[31] *Intel 82599 10 GbE Controller Datasheet, Revision 2.0*, July 2009.

[32] "CUDA C Programming Guide, Version 8.0," Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/, accessed: March 14, 2017.

[33] J. Shen, J. Fang, H. Sips, and A. L. Varbanescu, "Performance Traps in OpenCL for CPUs," in *Proceedings of the 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 2013.

[34] "Nvidia system management interface program," [Accessed: 4-May-2017]. [Online]. Available: http://developer.download.nvidia.com/compute/DCGM/docs/nvidia-smi-367.38.pdf

[35] "Intel's processor counter monitor," [Accessed: 17-May-2017]. [Online]. Available: https://software.intel.com/en-us/articles/intel-performance-counter-monitor

[36] G. Vasiliadis, M. Polychronakis, S. Antonatos, E. P. Markatos, and S. Ioannidis, "Regular Expression Matching on Graphics Hardware for Intrusion Detection," in *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection*, 2009.

[37] R. Smith, N. Goyal, J. Ormont, K. Sankaralingam, and C. Estan, "Evaluating GPUs for Network Packet Signature Matching," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, 2009.

[38] O. Harrison and J. Waldron, "Practical Symmetric Key Cryptography on Modern Graphics Hardware," in *Proceedings of the 17th USENIX Security Symposium*, 2008.

[39] W. Sun and R. 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.

[40] J. Shim, J. Kim, K. Lee, and S. Moon, "Knapp: A packet processing framework for manycore accelerators," in *High-Performance Interconnection Networks in the Exascale and Big-Data Era (HiPINEB), 2017 IEEE 3rd International Workshop on*. IEEE, 2017, pp. 57–64.

[41] Y. Go, M. A. Jamshed, Y. Moon, C. H̃wang, and K. Park, "Apunet: Revitalizing gpu as packet processing accelerator," 2017.

[42] G. F. Diamos and S. Yalamanchili, "Harmony: An Execution Model and Runtime for Heterogeneous Many Core Systems," in *Proceedings of the 17th International Dymposium on High Performance Distributed Computing*, 2008.

[43] E. Sun, D. Schaa, R. Bagley, N. Rubin, and D. Kaeli, "Enabling Task-Level Scheduling on Heterogeneous Platforms," in *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, 2012.

[44] C. Gregg, M. Boyer, K. Hazelwood, and K. Skadron, "Dynamic Heterogeneous Scheduling Decisions Using Historical Runtime Data," in *Proceedings of the Workshop on Applications for Multi- and Many-Core Processors*, June 2011.

[45] C. Margiolas and M. F. O'Boyle, "Portable and transparent host-device communication optimization for gpgpu environments," in *Proceedings of Annual IEEE/ACM*

International Symposium on Code Generation and Optimization. ACM, 2014, p. 55.

[46] B. Wu, X. Liu, X. Zhou, and C. Jiang, "Flep: Enabling flexible and efficient preemption on gpus," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2017, pp. 483–496.

[47] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo, "Quality of service support for fine-grained sharing on gpus," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 2017, pp. 269–281.

[48] A. Moerschell and J. D. Owens, "Distributed texture memory in a multi-GPU environment," in *Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, 2006.

[49] C.-K. Luk, S. Hong, and H. Kim, "Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009.

[50] G. Wang and X. Ren, "Power-efficient work distribution method for cpu-gpu heterogeneous system," in *Proceedings of the 2010 International Symposium onParallel and Distributed Processing with Applications*, 2010.

[51] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng, "Merge: A Programming Model for Heterogeneous Multi-core Systems," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[52] C. Müller, S. Frey, M. Strengert, C. Dachsbacher, and T. Ertl, "A Compute Unified System Architecture for Graphics Clusters Incorporating Data Locality," *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, no. 4, Jul. 2009.

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

[54] M. Dobrescu, K. Argyraki, and S. Ratnasamy, "Toward Predictable Performance in Software Packet-Processing Platforms," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, 2012.

[55] A. Ghodsi, V. Sekar, M. Zaharia, and I. Stoica, "Multi-Resource Fair Queueing for Packet Processing," in *Proceedings of SIGCOMM*, 2012.

[56] S. Hong and H. Kim, "An integrated gpu power and performance model," in *ACM SIGARCH Computer Architecture News*, 2010.