UCLouvain

École polytechnique de Louvain

GPU-based Packet Processing

Authors: **Romain Van Hauwaert, Maxime Vanliefde**
Supervisor: **Tom Barbette**
Readers: **Etienne Rivière, Nikita Tyunyayev**
Academic year 2023–2024
Master [120] in Computer Science and Engineering

Abstract

For about a decade, network functions such as routers have been increasingly implemented in software rather than hardware to offer better flexibility. Small functions are combined to form the entire packet processing path. As network speeds have increased, CPU cores have struggled to keep up with such capabilities. This has prompted the search for alternative ways to process packets. GPUs have been proposed to offload computationally intensive functions from the CPU. In addition, recent SmartNICs have become capable of bypassing the CPU altogether to receive and transmit packets, using only either their own processing cores or a third-party device. In this work, three GPU-enabled network functions are provided. They can easily be placed in a processing pipeline to offload specific tasks to a GPU, while still allowing other functions to be computed on the CPU. Two CPU bypass implementations are also provided. These completely remove the CPU from the packet processing path. A comprehensive multifactor evaluation of all implementations is given for these network functions, with varying levels of computational complexity. It is shown that, depending on the configuration, GPU implementations are able to exceed the throughput of a CPU-only implementation by a factor of more than 2 and maintain satisfactory latency results, while using very few CPU cores to receive packets.

Acknowledgments

We would like to thank Prof. Tom Barbette for his constant guidance and availability throughout the year and the (bi-)weekly meetings. His ideas and feedback were precious to help us advance throughout the research and progress in the work in multiple ways. Thank you also to Nikita Tyunyayev for his participation in our meetings and for his precious technical assistance throughout the year, his responsiveness, and his reviews of this thesis.

We also thank Prof. Etienne Rivière for accepting to be a reader of this thesis and a member of our jury.

Thanks to Prof. Cristel Pelsser for helping us access routing tables, to Alexandre Vogel for the help building the Jaskier server, and to Anthony Gego for giving us access to the server room.

Finally, we are immensely grateful to our family and friends who supported us throughout our studies. Thanks also to Thomas Antoine for the proofreading of this document.

DeepL Write [18] was used in the writing of this document, and GitHub Copilot [32] in the coding of the implementations.

Contents

List of Acronyms

APU	Accelerated Processing Unit
ARM	Advanced RISC Machines
BAR	Base Address Register
BIOS	Basic Input/Output System
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
CUDA	Compute Unified Device Architecture
DMA	Direct Memory Access
DOCA	Data center-On-a-Chip Architecture
DPDK	Data Plane Development Kit
DPU	Data Processing Unit
DRAM	Dynamic Random-Access Memory
EAL	Environment Abstraction Layer
eBPF	Extended Berkeley Packet Filter
ECPF	Embedded CPU Function Ownership
FCS	Frame Check Sequence
FPGA	Field-Programmable Gate Array
GDAKIN	GPUDirect Async Kernel-Initiated Network
GDDR	Graphics Double Data Rate
GPGPU	General-Purpose computing on GPUs
GPU	Graphics Processing Unit
IDS	Intrusion Detection System
iGPU	Integrated GPU
IP	Internet Protocol
IPv4	Internet Protocol Version Four
MAC	Media Access Control
mbuf	Message Buffer

MMIO	Memory-Mapped I/O
NFV	Network Function Virtualization
NIC	Network Interface Card
NUMA	Non-Uniform Memory Access
OS	Operating System
OSI	Open Systems Interconnection
PCI	Peripheral Component Interconnect
PCIe	Peripheral Component Interconnect Express
RAM	Random Access Memory
RDMA	Remote Direct Memory Access
RIPE	Réseaux IP Européens
RIS	Routing Information Service
ROI	Region of Interest
ROM	Read-Only Memory
RSS	Receive Side Scaling
RTT	Round-Trip Time
SAM	Smart Access Memory
SDK	Software Development Kit
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Threads
SM	Streaming Multiprocessor
SoC	System on a Chip
TCP	Transmission Control Protocol
TLB	Transaction Lookaside Buffer
VGA	Video Graphics Array
VNF	Virtual Network Function
VRAM	Video Random-Access Memory
XDP	eXpress Data Path
ZLT	Zero-Loss Throughput

List of Figures

List of Tables

List of Algorithms

List of Listings

Introduction

Graphics Processing Units (GPUs) are on the rise since many years [56]. While at first they were only used for graphics processing, their parallel structure and high-bandwidth memory led to the emergence of General-Purpose computing on GPUs (GPGPU) to accelerate specific tasks that would fit this structure. Nowadays, GPUs are more and more versatile. GPGPU was made simple with the releases of NVIDIA's CUDA [60] in 2007 and Khronos Group's OpenCL [91] in 2009, providing interfaces for the developer to port code to the GPU. They power many Machine Learning and computive-intensive applications such as statistical computing [31]. GPU architecture also becomes more and more general-purpose, stepping out of its role as co-processor to become a full device in itself. As an example, NVIDIA's GPUDirect [70], allowing to access storage and network devices directly from the GPU, led to frameworks such as GPUfs [87] in 2013, which provides a POSIX-like API for programs to access the host file system.

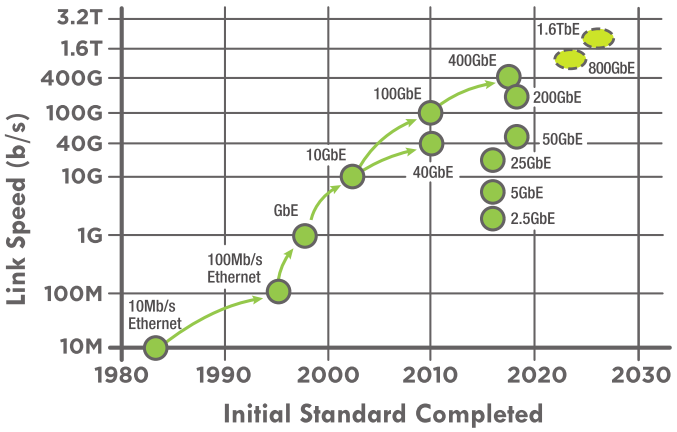


Figure 1: Ethernet standard speeds evolution throughout the last decades. A green dot means the standard is available, a dashed yellow dot that it is in development. Adapted from Ethernet Alliance [23].

GPU programming is not the only field to have significantly changed in the last decades. In the networking world, hardware is improving rapidly, offering increasingly high speeds delivered by the Network Interface Cards (NICs). Figure 1 shows the evolution of Ethernet standard speeds throughout the years, as promoted by the Ethernet

Alliance. Once a speed is standardized, it is only a matter of time before cards offering such speed arrive. Link speeds have increased from 100 Mbit/s in 2003 [30], to 10 Gbit/s in 2010 [37] and 100 Gbit/s in 2020 [100]. Cards offering up to 400 Gbit/s can today be found on the market [68]. Moreover, more than providing better speed, SmartNICs such as NVIDIA BlueField-2 [67] offer even more capacities, such as embedded ARM cores on the NIC to process packets [44], and ability to entirely bypass the CPU to receive packets in the memory space of a third device [4]. Such NICs support specific proprietary software such as DOCA [69], which allows for development on the ARM cores and use of the hardware accelerators of the BlueField.

The need of a more and more digital world, with enterprises increasingly moving their infrastructure to the cloud, and the emergence of network technologies such as 5G, increases heavily the demand for high-speed network processing. Data-centers constantly improve their hardware to reduce congestion and improve response times [83]. With such high speeds, it can be difficult to keep up with the rate at which packets are received and processed. Indeed, while Central Processing Units (CPUs) have traditionally been used to do intensive operations on packets, CPU clock speeds have not improved recently. Base clock speed reached 3 GHz in 2003, and barely increased to 4 GHz until now due to the so-called "power wall" [57]. At a speed of 100 Gbit/s, a new packet of 1024 B arrives every 83.5 ns. This considers the interpacket gap, a pause required between each emitted Ethernet frame, which leads to an overhead of 20 B per packet received (cf. Section 4.2.2). The clock cycle of a 4 GHz CPU is 250 ps. With a single processor, each packet must be processed in maximum 334 cycles to keep up. While it is feasible for lightweight applications, it can become hard to achieve with more intensive workloads. With smaller packet sizes, it becomes even harder: for example, processing a packet of 128 B should happen in 11.84 ns (i.e. 47 cycles), and for a packet of 64 B as fast as 6.72 ns (i.e. 26 cycles).

As a result of the stalling of clock frequency, CPU performance has been improved by other means, such as increasing the number of instructions issued per cycle, e.g. using hyperthreading, multicore chips, and vectorized instructions [90, 24]. This means that networking stacks have to scale well with the number of cores. Particularly, the Linux kernel networking stack inefficiencies [16] lead to the use of faster user-space networking stacks such as DPDK [19]. They generally uses Receive Side Scaling (RSS) to balance the load of packets between multiple CPU cores.

To keep up the pace, using GPUs to accelerate network packet processing has been proposed since the software router PacketShader [37] released in 2010. However, back in the day, CPU-GPU connection was done using PCIe 3.0, who provides bandwidth of an order of magnitude lower than those of GPU memories. Efforts have been made to mitigate the interconnection performance bottleneck, for example using Integrated GPUs (iGPUs) in the APUNet system [33]. Nowadays, PCIe 4.0 and 5.0 links are available and are respectively 2 and 4 times faster than the bandwidth offered by PCIe 3.0. Interconnections between NIC, CPU and GPU are thus way faster and more adapted to high-speed networking and GPU memory bandwidth, as we can see from Table 1. While NIC speeds have multiplied by 40 and PCIe speeds by 16 in about 10 years, GPU

Table 1: Speeds of recent and old NICs, PCIe interconnects and GPU memories.

Device	Year	Bandwidth
Intel X520-DA2	2009	1.25 GB/s
Mellanox ConnectX-7	2021	50 GB/s
PCIe 3 x16	2010	15.754 GB/s
PCIe 4 x16	2017	31.508 GB/s
PCIe 5 x16	2019	63.015 GB/s
NVIDIA GTX480 VRAM	2010	177.4 GB/s
NVIDIA L4 VRAM	2023	300 GB/s

memory bandwidth has doubled and is still much faster than both. With such increased speeds, the GPU is poised to become more useful than ever for network processing. Following the trend, NVIDIA recently contributed the GPUdev library [3] in DPDK, which provides APIs to enhance dialog between NIC, CPU and GPU. It also offers the GPUNetIO DOCA library [61] that adds the notion of GPU device in the DOCA ecosystem, allowing to move to a GPU-centric approach where the CPU plays no role in the control flow of received packets.

Another trend in the networking world that is worth mentioning is Network Function Virtualization (NFV). Instead of using proprietary hardware, software routers are increasingly used. They consist in separating the processing path of packets into small pieces called Virtual Network Functions (VNFs) chained together. The Click modular router [48] was made popular thanks to its extended modularity. Snap [89] in 2013 explored the difficulty to integrate GPU processing into such software routers, as packets may not always follow a specific path but can be separated and even reunited later during processing, breaking the highly parallel model needed to gain advantages of GPU processing. FastClick [12] has been built on top of Click, combining high speeds of user-level network stacks and modularity of NFV.

The goals of this thesis are twofold. First, we revisit the current state of the art for GPU-based packet processing with modern hardware to determine whether speed improvements still make using the GPU for packet processing worthwhile. We evaluate the performance of GPU-enabled elements for the FastClick modular router through both a pure CUDA implementation and the NVIDIA GPUdev library.

We also evaluate the performance of CPU bypassing by having the NIC send its packets directly to the GPU, and by having the GPU control the packet flow using the DOCA GPUNetIO library, effectively removing the CPU from the packet processing data path.

Chapter 1 introduces the concepts of networking and GPU resources that are used in this work. It also provides an overview of the state of the art in using GPUs to improve packet processing.

Chapter 2 describes in details the design of both GPU-enabled elements and DOCA CPU-bypassing application. It first explains the overall structure of elements, then dives

into explaining the factors that can influence the processing of packets, such as CPU multithreading, GPU control flow handling, and CPU-GPU communication. It finishes by explaining the implementation of the CPU bypassing model.

Chapter 3 focuses on three network applications describing typical VNFs and their implementation as elements integrating GPU acceleration.

Finally, Chapter 4 characterizes all the implementations, evaluates the test cases in terms of peak throughput, average latency, and GPU power consumption, and discusses the results. It provides a comprehensive multi-factorial analysis of performance for each implementation.

Chapter 1

Background

Before describing our implementations, we introduce some necessary concepts of network processing and GPUs architecture used in this work, and we give a survey of related works.

1.1 Networking resources

As speeds delivered by the NICs are heavily improving for the last decades, network processing paths have been improved to keep up with the rise of the rate at which packets should be processed. This section provides an overview of the networking resources currently in use and the tools available to address this challenge.

1.1.1 Kernel networking stack

Traditionally, network processing stacks are implemented in the operating system kernel, such as the Linux kernel (which will be referred to as *the kernel*). It is however inherently unsuited to the speeds offered by fast hardware. Receiving a packet from the NIC through a kernel stack to arrive at the application running in user space is a process that can be decomposed into six steps [16]. Figure 1.1a illustrates this process.

1. The NIC receives a packet. This packet is transferred into CPU main memory using a Direct Memory Access (DMA) call, and a packet descriptor is stored in a ring buffer.

2. The NIC issues a hardware interrupt.

3. The kernel receives the interrupt, wraps the packet inside a `sk_buff` structure [94].

4. Functions are called depending on the protocol of the packet. The packet is then transferred for processing to the kernel protocol stack.

5. Once the protocol processing is done, the packet is stored in a socket buffer accessible from user space.

6. The application reads the packet using a POSIX call and continues processing.

While being very general, this path is quite long, and source of several inefficiencies [16].

- Preprocessing. Each packet received induces a potential interrupt, and in all cases a `sk_buff` construction. This structure can handle every type of packet, but is thus very large even if most of its fields are unused, which leads to time spent doing unnecessary work, and an increase in memory footprint of packet processing [8].
- Data copying. The entire packet is copied from kernel to user space buffer, which has a performance cost.
- System calls. The application uses sockets to receive packets. Each POSIX call will issue a system call and a context switch from user to kernel space. A context switch can cause a performance loss, as it can pollute the cache [55] or cause a Transaction Lookaside Buffer (TLB) flush [7].

Efforts have been made to mitigate these limitations. Kernels nowadays use New API (NAPI) to mitigate interruptions, by switching to polling mode when the flow of packets is too high [93], avoiding processing an interrupt for each received packet. They also pre-allocates a number of buffers to reduce allocation overhead [95].

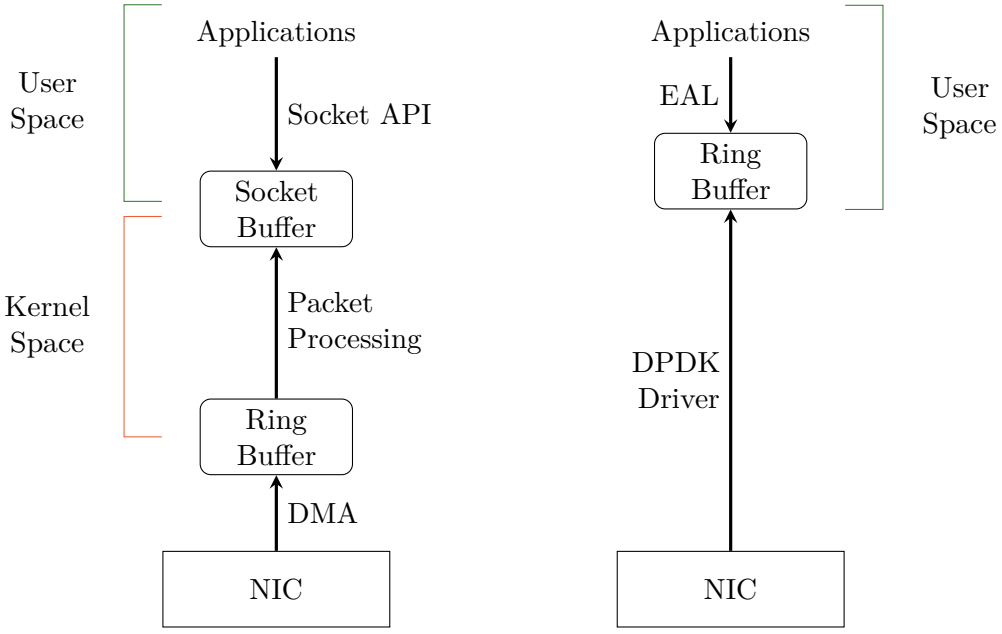
1.1.2 Bypassing the kernel networking stack

Due to these inefficiencies, network stacks bypassing the kernel stack, entirely or not, have been developed.

Extended Berkeley Packet Filter (eBPF) [22] provides an instruction set and an execution environment inside the kernel. Its goal is to make the kernel programmable at runtime to improve performance, while ensuring safety by ensuring that programs cannot crash the kernel. It includes eXpress Data Path (XDP), a kernel layer aimed at processing network packets faster and closer to the NIC by bypassing most of the kernel stack [96].

Several user space networking stacks, thus entirely working outside the kernel, have also been developed. All of these solutions move packet I/O into user space. Figure 1.1 show the processing path of a packet, either using a kernel stack or a user space stack.

Netmap [85] is the first widely-used framework addressing the kernel stack inefficiencies. While still using kernel drivers, it removes per-packet memory allocations by using pre-allocating memory, reduces system calls cost by emitting packets per batch and avoids unnecessary copies by using shared buffers between kernel and user space.



(a) Using a kernel stack. (b) Using the DPDK user space stack.

Figure 1.1: Processing path of a network packet.

Nowadays, **DPDK** [19] (described in details in Section 1.1.3) is the most widely used user space packet I/O framework.

These frameworks can be used to build even faster packet processing applications. TCP/IP stacks, such as mTCP [41] or IX [13], can further improve performance of applications using Transmission Control Protocol (TCP). They work by separating the protocol handling from the I/O control flow, dividing threads between TCP threads, handling protocol processing, and application threads. Unfortunately, they are often unmaintained: mTCP last commit was in 2019, and IX's in 2017.

Solutions to bypass the kernel networking stack can also be implemented in hardware. **Remote Direct Memory Access (RDMA)** is a form of DMA that can work across network devices [36]. It allows a device to directly access the memory of another, without involving the CPU, Operating System (OS), nor the cache of either device. Using this technology, a NIC can directly write its packets into the host memory, without involving the kernel networking stack, allowing application to directly access it without any context switch needed. This is done on Ethernet network with RoCE (RDMA over Converged Ethernet). A drawback is that a lossless network is needed for good performance. Priority Flow Control (PFC) mechanism is generally used to minimize packet loss, but it adds complexity and can lead to problems as severe as deadlocks and livelocks [35].

1.1.3 The Data Plane Development Kit

Data Plane Development Kit (DPDK) [19] is an open-source set of libraries for fast packet processing. It focuses on offloading packet processing from the kernel to user space. The polling mode drivers of DPDK allows a higher throughput than the regular interrupt based kernel packet processing. This removes the need for the kernel networking stack. The DPDK Programmer's Guide [21] describes the software and its architecture. Figure 1.1b illustrates the process of receiving a packet using DPDK.

The environment specifics are abstracted from the user under the Environment Abstraction Layer (EAL), that provides all the services the user need (e.g. hardware and memory management). Regarding memory, DPDK supports and encourages the use of hugepages, as they allow for larger contiguous chunks of memory, and cover a larger address space, reducing the number of TLB misses [53]. Shared memory pools are pre-allocated to store rings and buffers, so that the cost of creating or deleting a single entity is greatly reduced. Packet buffers are called *Message Buffers (mbufs)* and are the equivalent of the `sk_buff` struct used in the Linux kernel (see section 1.1.1). Each mbuf contains both data and metadata (e.g. control information such as length) of a packet. Communication between the NIC and the application occurs through fixed-size *transmit* and *receive rings*, allowing efficient and concurrent enqueue and dequeue operations. With such ring, the NIC can enqueue an element while the CPU is dequeuing one for the application, and vice versa.

Each DPDK process is pinned to a CPU core to avoid the overhead induced by task switching. 2 types of processing models are supported. In run-to-completion, packets are received and processed on the same logical core. In pipeline, packets are received on one core and processed on another, allowing the computation to be done in stages. This second model requires rings to transmit packets between cores. The model providing the best performance will depend on the application.

Load-balancing of packets between multiple cores is done using Receive Side Scaling [39]. For every packet that arrives, its header is hashed, and the result is used as index in an indirection table. This table maps to a receive queue, and each CPU core read from one or more queues. This process is done in hardware, directly on the NIC, to better scale with the number of cores. Note that as the mapping is static, the load is correctly balanced when flows of packets are comparable in workload and distributed across the queues, which is not always the case.

1.1.4 Network Function Virtualization and Modular Routing

While routers were first implemented using proprietary hardware, the emergence of NFV led to development of software routers, where processing paths are divided in blocks called Virtual Network Functions, chained together [99]. Many state-of-the-art frameworks support this concept, such as Click [48], FastClick [12], Vector Packet Processing [26], and Open vSwitch [27].

Click [48] was the first instance of a flexible and modular architecture for networking

systems. It proposes a new way for building modular routers that can easily be extended and are highly configurable. Its architecture is based on elements that connect to each other. Each element, as represented in Figure 1.2, represents a basic component of router processing and has some properties. The most important properties of an element are the *element class*, which determines the code to be executed upon arrival of a packet in the element, *ports*, which allow connecting to other elements, the *configuration string*, which is optional and can be used to configure the element, and the *method interfaces* supported by that element. Elements can either pull packets, by regularly polling the source element for packets, or push packets, by passing them downstream to the destination element. `Queue` elements are placed to convert from push to pull, providing a temporary packet storage. Such pipeline is called a push-to-pull path.

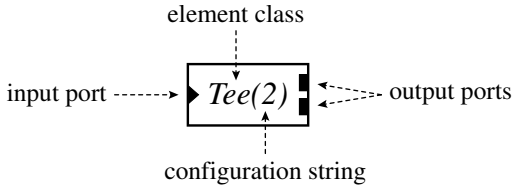


Figure 1.2: Example Click element, with properties [45].

FastClick [12] is a high speed packet I/O framework based on Click. It implements existing techniques for modern hardware and some new ideas into Click, with an improved Netmap and DPDK support. It uses some popular features common in most high performance packet I/O frameworks like zero-copy, I/O batching, multi-queueing, and kernel bypass. Custom improvements were also made to the execution model of Click. Zero-copy is used to avoid unnecessary copies of memory by pre-allocating packets in *pools* in shared regions of memory accessible to the NIC and the software. I/O batching is used by most frameworks and helps with the overhead of accessing the NIC, but can introduce latency for large batch sizes. Hardware multi-queue support utilizes the multiple queues in modern NICs in which they receive packets. Finally, while FastClick still supports push-to-pull paths, it adds the support and highly encourages for using Full Push paths. Push-to-pull paths were a source of overhead as they required a form of synchronization, and they could cause cache misses. In Full Push paths, all elements push packets and never pull. They remove the need for software `Queue` elements on the processing path. Figure 1.3 shows such a path.

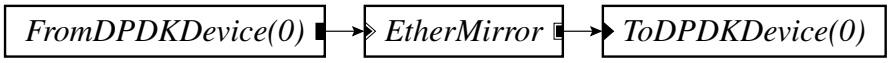


Figure 1.3: Example FastClick configuration that bounces packets from a single interface to itself, using DPDK driver.

1.1.5 Writing FastClick elements

FastClick extends Click, so that Click elements are fully compatible with FastClick, but not the opposite. FastClick elements consists of a few methods that describe the behavior of the element.

Behavioral functions are static methods, generally defined in the header file, that describe element properties, e.g. `class_name()` for the element name, `port_count()` for the description of virtual ports supported, `flow_code()` to describe how packets flow from input to output port(s), and `processing()` that specifies the processing type (such as push or pull).

The element is configured through the `configure()` method that handles the reading of the configuration string. Initialization of the element is done through the `initialize()` method, while `cleanup()` method cleans state allocated by the initialization process.

Packet and event processing methods implement the actual function of the element. Hereafter, we describe how to implement a full push path of batches, the processing encouraged by FastClick. When a batch arrives, the `push_batch()` method is called. To output a batch, the element must call `output_push_batch()`. If the processing is straightforward and can take place immediately, the element can call it from `push_batch()`. However, if the processing is longer, it is common to launch the processing for `push_batch()`, and hand over to receive more batches while processing occurs. Polling can be made using tasks that can periodically execute code, usually through the `run_task()` method, that will thus output packets in this configuration.

Finally, if the element switches threads, i.e. packets can arrive at one thread and be transmitted from another, the `get_spawning_threads()` method must explicit the possible paths.

For more information, we refer the reader to the Click [46] and FastClick [9] documentations.

1.1.6 BlueField

The NVIDIA BlueField cards [66] are a type of SmartNIC, which is an improved NIC with additional processing capabilities. SmartNICs allow the offloading of networking tasks from the host CPU, enhancing performances [44]. Most SmartNICs are programmable, meaning the developer can create custom applications to be run directly on the device. They are typically designed for data center requiring high speed networking. The BlueField-2, a NIC we use in this work, is a System on a Chip (SoC) SmartNIC that integrates a complete system. There also exists Field-Programmable Gate Array (FPGA) SmartNICs that provide more flexible and customizable features, but they come with the added complexity of FPGA devices. Examples of FPGA SmartNICs are the Intel® FPGA SmartNIC N6000-PL Platform [40] from Intel or the Xilinx Alveo U25 SmartNIC Platform [98] from AMD.

The BlueField-2's main characteristics are its multicore ARM general-purpose CPU, DDR4 Random Access Memory (RAM) and high-speed Ethernet or InfiniBand interface

[67]. The card supports networking frameworks like DPDK and Data center-On-a-Chip Architecture (DOCA), a dedicated proprietary NVIDIA software framework to create applications for the BlueField. The card can operate on three distinct mode [74].

- Data Processing Unit (DPU) Mode. In this mode, also called Embedded CPU Function Ownership (ECPF), the NIC is controlled by the Advanced RISC Machines (ARM) subsystem. The communication to the host is done through a virtual switch control plane. It is the default BlueField mode.
- Zero-Trust Mode. This mode is a variation of the DPU Mode, with an additional security layer. The DPU is not accessible through the host.
- NIC Mode. With NIC mode enabled, the BlueField behaves exactly like a classic NIC to the host.

All our measures were done using the DPU Mode.

The BlueField-2 also provides some specific hardware accelerators for computations like Regular Expressions, SHA256 encryption[67].

DOCA

NVIDIA DOCA is a software framework that aims to enhance and accelerate data center operations by utilizing NVIDIA's BlueField DPU and NVIDIA Mellanox ConnectX NICs. DOCA provides tools to develop on the ARM cores, use the hardware accelerators of BlueField devices and offers libraries that enable networking and data-processing programmability for the BlueField.

DOCA Core [77] is a fundamental component of DOCA. It provides an interface to use the different DOCA libraries. DOCA Core provides an abstraction of the hardware, creating the building blocks for any application on a BlueField. It is made to enable applications to offload network or resource intensive tasks to dedicated hardware.

In practice, DOCA Core allows to discover the hardware acceleration units from the device, query their capabilities and enable the device to allocate and share resources for hardware acceleration to the DOCA libraries. If multiple devices are present on a system, the application can choose which one to use based on the topology or the capabilities of the devices.

The tasks on the device need buffers as inputs and outputs for the data to be processed. These buffers are created and initialized directly in the application. The DOCA Software Development Kit (SDK) takes advantage of zero-copy to communicate the data with the hardware. The subsystem allows the developer to register memory and allocate buffers on that registered memory. The DOCA memory subsystem is designed to have a minimal memory footprint to improve scalability. The documentation [62] claims DOCA memory map hides a lot of details that makes it the ideal way to share memory between the host and the device, but these details are not explained in the documentation.

The buffers provided by DOCA Core reference memory accessible on the DPU. These buffers can be used on different accelerators and can reference CPU, GPU, host or RDMA memory. Each buffer contains the length of the memory region and the address pointing to it.

GPUNetIO

DOCA GPUNetIO [61] is another component of the DOCA framework that provides real-time GPU processing of network packets. The goal of GPUNetIO is to remove the CPU from the critical path of processing of packets, allowing the GPU and NIC to directly communicate between each other. This library proposes several features to allow for a GPU-centric approach of packet processing, like for instance:

- GPUDirect Async Kernel-Initiated Network (GDAKIN), which allows CUDA kernels (cf. Section 1.2.2) to interact directly with the NIC to receive or send packets.
- GPUDirect RDMA, which makes it possible to receive packets directly into the GPU memory area.
- Ethernet protocol management on GPU (DOCA Ethernet), which enables processing of Ethernet Packets on the GPU.

and other features like semaphores, smart memory allocation and RDMA protocol management on GPU. The setup of the application like device configuration, memory allocation and launch of Compute Unified Device Architecture (CUDA) kernels is done on the CPU, while the data path of network packets is exclusively on the NIC and the GPU.

1.2 GPU Resources

A Graphics Processing Unit is a programmable device that typically acts as a co-processor: it receives commands, code and data from the CPU and is used to offload intensive tasks. Their inherent parallel structure makes them useful to solve very parallel problems. This section explains the architecture of modern GPUs and how to make use of them.

1.2.1 Hardware Implementation

A GPU is designed to execute thousands of *threads* in parallel to amortize the cost of a single thread and increase the overall throughput. Figure 1.4 shows a schematic of a GPU architecture. It consists of many cores and multiple levels of cache on-chip, and a separate high-bandwidth global memory. This memory is referred to as Video Random-Access Memory (VRAM).

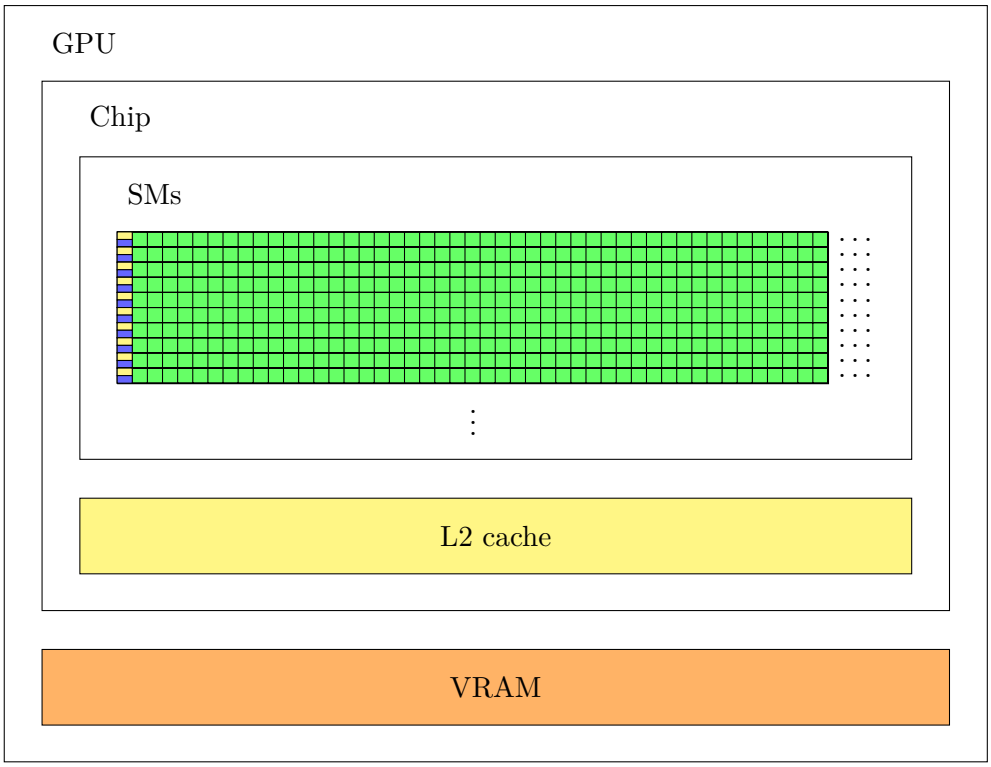


Figure 1.4: Hardware implementation of a GPU. Caches are in yellow, cores are in green, and register files are in blue.

NVIDIA GPUs are built as an array of *Streaming Multiprocessors (SMs)* that can execute in parallel. In Figure 1.4, one line of cores represents a single SM. A Streaming Multiprocessor can be seen as an independent processor and adopts the Single Instruction Multiple Threads (SIMT) model. Each one has a fixed number of cores dedicated to different operations (such as floating point arithmetic or integer arithmetic), multiple schedulers, its own L1 data and shared memory caches, and a set of 32-bit registers.

The NVIDIA RTX A2000 [73], the card we use in our experiments, has 26 SMs, each having 192 kB of unified data cache and shared memory, as well as 128 cores, for a total of 3328 cores. It also has 3 MiB of L2 cache and 6 GB of GDDR6 DRAM.

The basic execution unit is a group of 32 threads, called a *warp* or a *wave*. A warp executes one common instruction at a time. A multiprocessor distributes warps among its schedulers, and, at every instruction issue time, each scheduler issues one instruction for one of its assigned warps that are ready-to-execute. Note that switching executions has no cost because execution context of each warp is maintained on-chip during its entire lifetime.

1.2.2 Programming Model

While GPUs were created to handle computation for computer graphics, they allow accelerating general-purpose processing. This approach is known as General-Purpose computing on GPUs. To write programs executing on a GPU, a programming framework is needed.

OpenCL [91] is an open standard allowing to write programs executing in parallel on heterogeneous systems such as GPUs, but also CPUs and other types of processors. It is widely supported by a large variety of platforms such as Intel, AMD, NVIDIA, ARM and more [92]. It specifies a programming language based on C.

CUDA [60] is NVIDIA's proprietary computing platform and programming model enabling to program NVIDIA GPUs. The software environment is an extension of the C++ Language. It both supports the OpenCL programming framework and its own model. It moreover provides libraries and utilities such as a debugger and analysis tools. The CUDA C++ Programming Guide [59] explains the CUDA model and interface.

CUDA is the dominant framework for GPGPU processing. In this thesis, the choice was made to use it as the computing framework.

Functions executed on the GPU are called *kernels*. When called, a kernel is executed N times by N different threads running in parallel. Each thread that executes a kernel inherits a unique thread ID to allow distinguishing behavior.

Threads are organized in thread *blocks*, which are further organized in thread *grids*. The user can specify the number of threads per block and blocks per grid, up to 3 dimensions. Blocks and grids are also assigned to a unique block and grid ID, respectively. Threads are required to execute independently, as to be scheduled across any number of SMs. Figure 1.5 presents such a hierarchy.

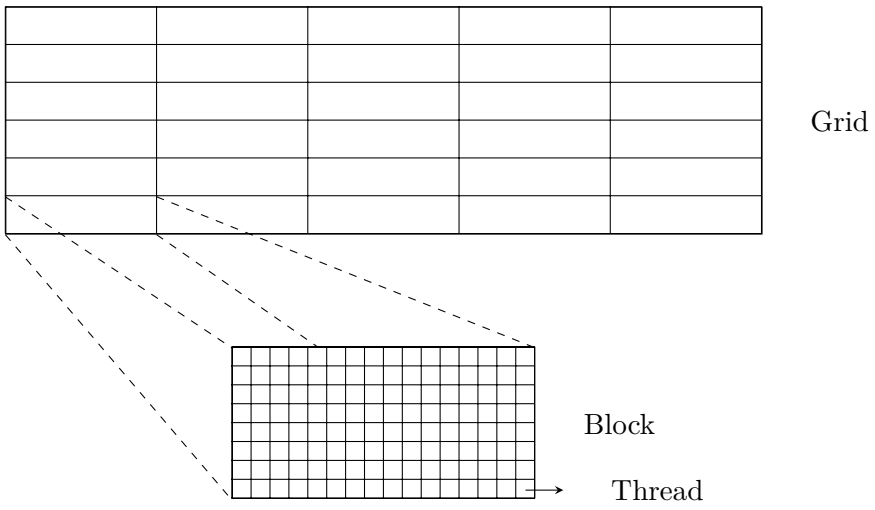


Figure 1.5: Thread hierarchy on GPUs. Here, the grid contains 6×5 blocks, and each block contains 8×16 threads.

Multiple memory spaces can be accessed by the threads during their execution. Each thread has its own private local memory and registers. Each threads of the same block can access shared memory, with the same lifetime as the block. All threads can access the same global memory. Unified Memory allows to access a memory space with a common address space from all CPUs and GPUs of the system, enabling zero-copy processing.

GPU operations, such as memory copies or kernel executions, can be issued on *streams*. Each operation associated with a stream is executed when all the previous ones are completed, providing an in-order sequence of commands. Multiple streams, on the other hand, can run on the same device simultaneously, allowing for parallel kernel executions.

Concurrency is also possible using paged-locked memory. With such host memory, recent GPUs are able to overlap kernel launch with asynchronous copies. They also support concurrent data transfers, overlapping copies to and from device. This is desirable, as, while processed data are copied back to the host, new data can simultaneously be copied to device memory.

A final type of operation that can be issued on streams is *events*. It consists of recorded timestamps on a stream. Both device and host can thus check if the event has been reached or not, and, if configured correctly, at which time it occurred, allowing for synchronization between GPU kernels or between GPU and CPU.

1.2.3 Communication with the CPU

Two types of GPUs exist, depending on their interconnection with the CPU, referred to as the *host*.

A discrete GPU is a device separated from the processor. Different types of interconnects can be used to connect it to the host [51], but PCIe is the most commonly used: the device communicates through PCIe lines. The NVIDIA RTX A2000 has 16 PCIe lines.

On the other hand, Integrated GPUs are manufactured on the same chip and shares the DRAM with the CPU. The entire device is also referred to as an Accelerated Processing Unit (APU). iGPUs usually offer lower computation capacity than discrete GPUs, but for a number of network applications they can be the most cost-effective accelerator [33], as combining the elements into a single chip reduces manufacturing costs. Sharing the DRAM memory removes the necessity to perform DMA transactions through PCIe lines, but the shared memory controller gets more contention as both device and host accesses it [54].

1.2.4 Important optimizations

Porting CPU code to run on a GPU is generally easy, thanks to the abstractions provided by CUDA. However, getting an efficiently working GPU code is not as straightforward. Some optimizations have to be applied to get the most of the GPU computation power [58].

Branching The SIMT behavior can be totally ignored by the programmer [59]. However, when not all threads in a warp agree on the execution path, both branches needs to be executed separately and then reconciled. This increases the number of instructions executed for the warp and reduces the efficiency. This approach is more flexible than the Single Instruction Multiple Data (SIMD) model used by modern CPUs, where all data elements have to be processed in the exact same way, but taking care that threads in a warp do not diverge will enable substantial performance gains.

Memory accesses Memory accesses are typically slower than computations, so programs should try to maximize bandwidth. Multiple levels of memory are provided by the GPU. The programmer can typically specify where should the data be stored, such as in constant memory for fast read-only access by all threads, shared memory for access within threads of a single block, or global memory for general accesses from all the threads. Moreover, global memory accesses should be coalesced. This is because device memory transactions are issued in 32 B, so that global memory loads and stores are coalesced into the minimal number of transactions to serve all threads of a warp. It also helps with caching, since only one transaction needs to occur on a cache hit if all accesses are coalesced and aligned on the cache line.

Host-Device Communication The peak bandwidth between the host and the device memory (32 GB/s on PCIe 4.0 x16) is much lower than the peak bandwidth between the device memory and the GPU (288 GB/s on NVIDIA RTX A2000). Such host-to-device memory transfer should thus be minimized. Moreover, there is an overhead induced for each PCIe transaction, so it performs better to batch many small transfers into a single large transfer, even if this means packing different memory regions into a buffer and unpacking it after the transfer. Also, when using zero-copy on unified memory, the GPU directly accesses the CPU memory, which issues PCIe transactions. Such data are not cached on the GPU, unified memory should thus be read or write only once.

Utilization The programmer is free to specify the number of blocks and threads per block when launching a kernel. Such parameters should be set so that the device is fully (or as much as possible) utilized, which in practice means that all SMs are as busy as possible. This can be done by launching more blocks than there are multiprocessors on the device. This allows the warp scheduler to execute other warps if one warp is stalled, due to a memory access for example. Another possibility is to execute multiple kernels concurrently thanks to streams.

1.2.5 GPUDirect RDMA

NVIDIA GPUs are capable of creating a direct path to exchange data with a third-party device on the PCIe bus that supports RDMA, such as a NIC or a storage adapter. This technology, called **GPUDirect RDMA**, uses the Peripheral Component Interconnect

(PCI) BAR registers (see Section 1.2.7). Communication is done by reading and writing to the BAR regions of the device, just as a device would read or write to the system memory.

1.2.6 GDRCopy

GPUDirect RDMA can not only map the address spaces of a GPU and of a third-party device, it can also be used to create CPU mappings of the GPU memory. This is implemented in the **GDRCopy** [65] library. The advantage is the low overhead for small data sizes. A standard CUDA memory copy is GPU-driven and performed by the GPU's DMA engine, which typically induces an overhead of 7 µs for transfers of size from 1 byte to 1 KiB in both directions (Host-to-Device and Device-to-Host). A GDRCopy memory copy is driven by the CPU using BAR mappings, which induces an overhead of less than 1 µs for Host-to-Device transfers up to 8 KiB, and of less than 3 µs for Device-to-Host transfers up to 1 KiB [63].

1.2.7 Base Address Register and Resizable BAR

Base Address Registers (BARs) are the main way for the host system to communicate to PCIe devices or for PCIe devices to communicate with each other. The BAR itself is the base address register, which is a 32-bit space in memory that contains the information for the much larger memory space that contains all the interesting data about the device. The BAR holds the pointer to this larger space, its size and a few other properties.

That region in memory permits the exchange of information with the device. One may read or write to specific addresses of the BAR, for instance, to read how the device is configured, to write a new configuration for the device, to request the execution of a task, to read the current status of the device, and so forth.

Each device on the PCIe bus is addressed by a unique identifier, known as the Bus-Device-Function (BDF) identifier. This number consists of 2 bytes with 8 bits representing the bus the device is connected to, 5 bits representing the device on the bus, and 3 bits representing the function of the device. This means that a machine can have up to 256 buses, each bus can have up to 32 devices, and each device can have up to 8 functions. These values are represented with the format **bus**:**device**.**function**, as shown in Figure 1.6. In this example, on the bus **a1**, there is a single device, the device `00`, which is the RTX A2000 GPU. The GPU has 2 functions `0` and `1`, presented as a Video Graphics Array (VGA) controller and an audio device. There is also a single device on the bus `c1`, which is the Mellanox ConnectX-6 NIC.

Each function on a device has its configuration space. There are 2 types of PCIe configuration spaces: Type 0 and Type 1. These spaces are normalized part of memory to interact with the devices. Type 0 is for endpoints, while Type 1 is for switches, root complexes, and PCI bridges.

The configuration space header of Type 0 devices, like a NIC, is organized as shown in Figure 1.7. There are in each header, 6 BAR registers, which means a device can have

```
$ lspci
...
a0:07.1 PCI bridge: Advanced Micro Devices, Inc. [AMD] Device 14a7 (rev 01)
a1:00.0 VGA compatible controller: NVIDIA Corporation GA106 [RTX A2000] (rev a1)
a1:00.1 Audio device: NVIDIA Corporation Device 228e (rev a1)
a2:00.0 Non-Essential Instrumentation [1300]: Advanced Micro Devices, Inc. [AMD] Device 14ac (rev 01)
...
c0:07.1 PCI bridge: Advanced Micro Devices, Inc. [AMD] Device 14a7 (rev 01)
c1:00.0 Ethernet controller: Mellanox Technologies MT2892 Family [ConnectX-6 Dx]
c1:00.1 Ethernet controller: Mellanox Technologies MT2892 Family [ConnectX-6 Dx]
c2:00.0 Non-Essential Instrumentation [1300]: Advanced Micro Devices, Inc. [AMD] Device 14ac (rev 01)
```

Figure 1.6: Output of the `lspci` command on the Sauron server (cf. Section 4.1).

up to 48 BAR registers (6 registers per function $*$ 8 function per device). NVIDIA GPUs use the BAR registers to expose the following spaces:

- BAR0: Memory-Mapped I/O (MMIO) registers, the main control space of the GPU.
- BAR1: VRAM aperture, which maps the GPU's VRAM. This register is very important for CPU bypassing (cf. Section 2.5) as GPUNetIO most likely requires more BAR1 than the default (in our case 256 MB).
- BAR2/3: RAMIN/VRAM aperture. This area used to be a special area after the VRAM containing control structures. At some point, NVIDIA stopped using the RAMIN area but kept the BAR2 and 3 registers having a similar use as BAR1.
- BAR4/5: Specific GPUs indirect memory access. Some older GPUs used these registers for indirect memory accesses.
- BAR6: PCI Read-Only Memory (ROM) aperture, used to expose the GPU Basic Input/Output System (BIOS).

Finally, we have memory BARs and I/O BARs. For memory BARs, the base address is a 27-bit field, referring to the corresponding BAR region's initial address in memory, which is 16-byte aligned. The maximum address of the start of a BAR region is determined by setting all these bits to 1. This means that the maximum addressable space is: $(2^{28} - 1) \cdot 16 = 4\,\text{GB}$.

Resizable BAR

There exists a few ways to improve the communication through the PCIe, like 4G Decoding or Resizable BAR [14].

Above 4G Decoding is a feature available on all motherboards supporting PCIe 3.0 [5]. This allows the BAR to use more than the 4 GB of memory, which is the default limit. With 4G Decoding enabled, 64-bit BARs are used instead of the classic 32.

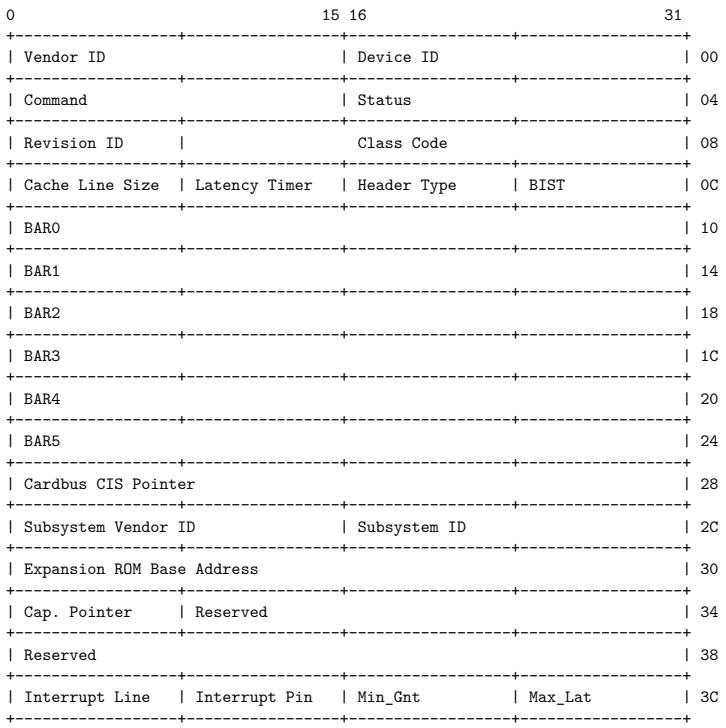


Figure 1.7: Configuration space header of Type 0 PCIe devices. [78]

There is also Resizable BAR, or Smart Access Memory (SAM) on the AMD motherboards, which is available in the BIOS of most recent motherboards. The BAR registers used to be limited to 256 MB, but the more recent GPUs have much larger frame buffers. This improvement allows the CPU to access the whole frame buffer, even if above the 4 GB default limit. Resizable BAR is required for GPUNetIO to work.

1.3 Networking using GPUs

Several configurations can be used to speed up network processing with GPUs [29]. This section presents such configurations, from those requiring the least amount of specific hardware support to those that require the most. Without loss of generality, we assume that each configuration features one CPU, one NIC and one GPU.

1.3.1 Classical CPU-driven

The most straightforward way of orchestrating the NIC and GPU together is to use the CPU as the "natural" glue between the two devices, as it would be with each device independently. The NIC does not know about the GPU, and vice versa.

Figure 1.8 shows such configuration. When the NIC receives packets, it copies them into CPU memory, for example using the DPDK library. The CPU then copies the

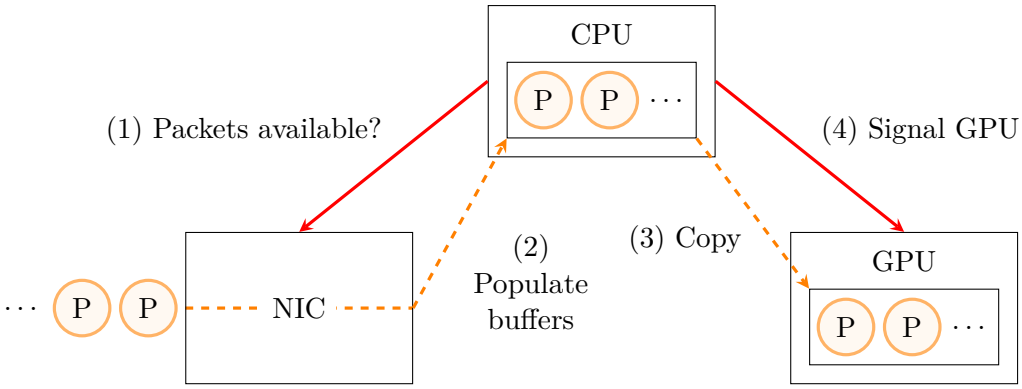


Figure 1.8: Processing path of packets following the classic CPU-driven approach.

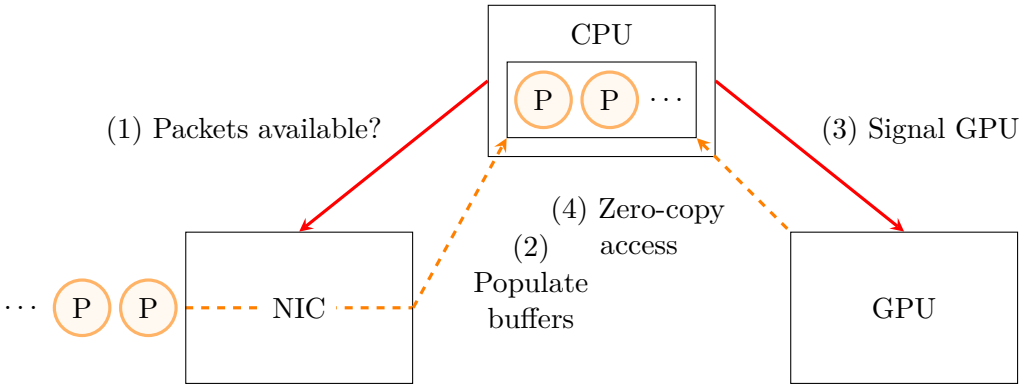


Figure 1.9: Processing path of packets following the CPU-driven approach with zero-copy.

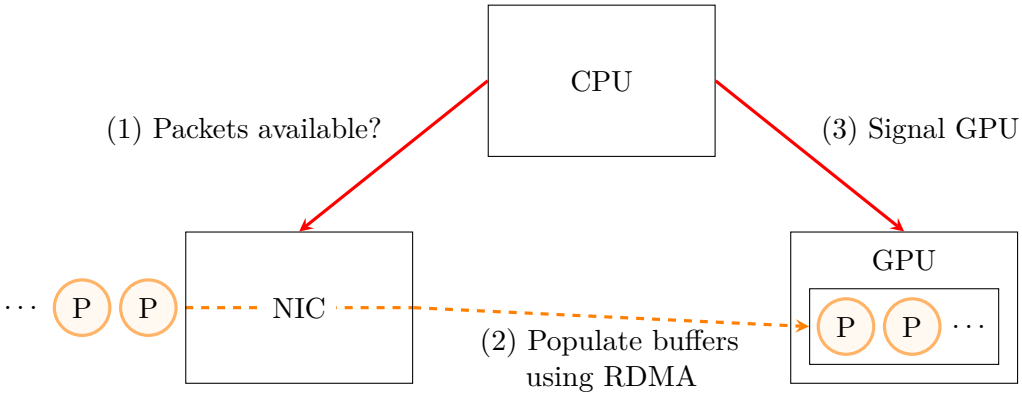


Figure 1.10: Processing path of packets following the CPU-driven approach with RDMA.

packets (or only the Regions of Interest (ROIs) of the network function) in GPU memory for it to process them.

In this pipeline, the CPU is the intermediary. However, it can become the performance bottleneck, due to all its responsibilities: it has to concurrently manage the network queues, the copies to and from GPU, and synchronize NIC and GPU tasks. This can impact the performance and latency of the application.

1.3.2 CPU-driven with Zero-Copy between GPU and CPU

A first step of improvement is to make the GPU directly access the CPU memory space where the packets reside, removing the explicit copies between CPU and GPU memories.

Figure 1.9 shows this configuration. It is mainly the same as the precedent, except that the GPU issues PCIe transactions to read the packets stored in CPU memory. As explained in section 1.2.4, using zero-copy on a discrete GPU is beneficial only if the GPU reads and writes the memory only once, as data are not GPU-side cached. Performance will thus depend on the application.

1.3.3 CPU-driven with GPUDirect RDMA between GPU and NIC

Another way of avoiding the copies of data from CPU memory is to use GPUDirect RDMA to allow the NIC to directly access GPU memory.

Figure 1.10 show such configuration. The packets do not reside anymore on the CPU memory. When the NIC receives packets, it copies them in GPU memory, using a RDMA call over PCIe. This removes the limitation of zero-copy between CPU and GPU, as the data only resides on GPU memory. Nevertheless, the role of the CPU stays important. It still manages network queues, and collects packets' info such as the addresses of payloads in the GPU memory pool and the number of packets received at each burst to notify the GPU about new received packets.

Also, since packets do not go into CPU memory, the CPU cannot access the packet data. All processing must be done on the GPU.

1.3.4 GPU- or NIC-driven

To further reduce the CPU-induced bottleneck, the final step is to simply remove the CPU from the processing path. Figures 1.11 and 1.12 show such configurations. The NIC directly accesses GPU memory to store packets. The role of coordinator, previously incumbent on the CPU, now has to be assumed by one of the two remaining devices in the processing path: either by the NIC or by the GPU.

Such configuration requires very specific hardware support to make NIC and GPU able to communicate without using a CPU. If the coordination is on the NIC side, it should include cores that allow generic processing, as a SmartNIC provides. Such device allows DPDK to run directly on it, reducing the communication overhead [52]. On an

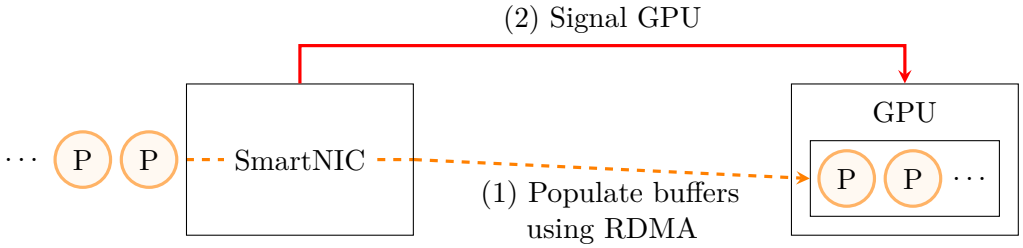


Figure 1.11: Processing path of packets following the NIC-driven approach.

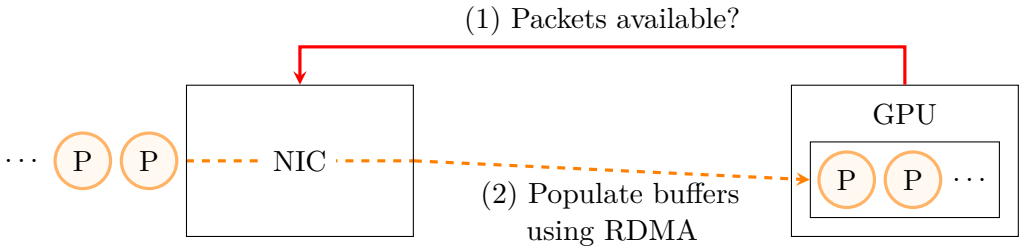


Figure 1.12: Processing path of packets following the GPU-driven approach.

NVIDIA SmartNIC such as a BlueField, DOCA would typically be used to program on the ARM cores. Nevertheless, it also needs a way to communicate with the GPU memory. With an NVIDIA GPU, the GPUDirect RDMA API makes this possible [42].

If the GPU coordinates the reception of packets, it should be able to control the NIC activity, e.g. by directly accessing the NIC registers. In this configuration, DPDK cannot be used anymore as it is a CPU framework. Another framework, such as GPUNetIO for NVIDIA NIC and GPU, must be used to handle the control path [4]. It can be considered the easiest way, as the framework directly contains all that is needed to enhance communication between the two devices.

1.4 Related Works

The idea of using GPUs for processing network packets has already been explored through multiple works [37, 89, 43, 33, 88], and summarized by surveys [15, 28]. Here, we present some frameworks that have been built following these principles. Table 1.1 summarizes the different implementations characteristics.

1.4.1 PacketShader

PacketShader [37] is one of the earliest software router frameworks using GPU acceleration. It consists of two parts: a collection of optimizations for the packet I/O engine and the GPU-accelerated framework itself. Main I/O improvements include bypassing the

Table 1.1: Comparison of related GPU network processing works. MW means One Master and Multiple Workers.

	PacketShader	Snap	G-Opt	APUNet	GPUnet
GPU Type	Discrete	Discrete	None	Integrated	Discrete
CPU needed	Yes	Yes	Yes	Yes	Yes
Threading	MW	Pipeline	?	MW	Only GPU
Zero-Copy	No	No	No	Yes	Yes
Kernel type	Launch	Launch	None	Persistent	Persistent

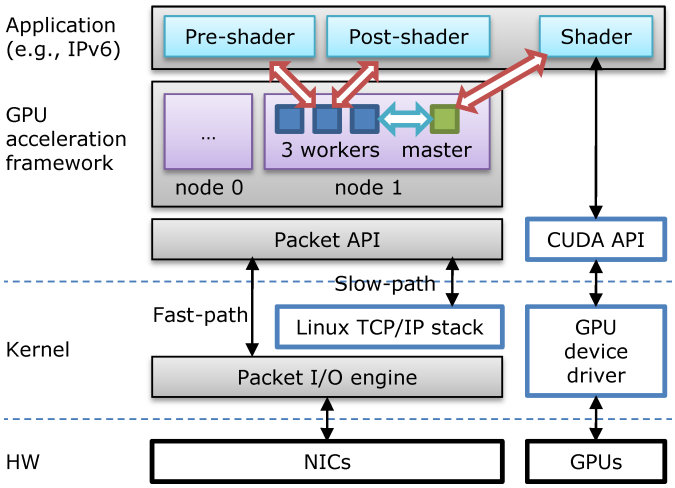


Figure 1.13: Overall PacketShader architecture [37].

Linux kernel network stack, using huge packet buffers, batch processing and Non-Uniform Memory Access (NUMA) scalability; all of these have nowadays been merged into DPDK.

Regarding GPU performance, as a motivating example, the authors show that, at peak performance to perform IPv6 lookup, the NVIDIA GTX480 GPU throughput is comparable to 10 times the throughput of an Intel X5550 processor. However, this requires many packets per batch sent to the GPU. With a small number of packets per batch, the CPU clearly outperforms the GPU. It is because GPU hide memory latency by using numerous threads. With a small amount of threads used, the GPU is underutilized.

The overall architecture of PacketShader is shown in Figure 1.13. The workflow of the framework follows a single master - multiple workers (MW) strategy, to avoid frequent context switching overheads because of multiple CPU threads accessing the same GPU, which was the case in the version of CUDA used. Workers are responsible for the packet I/O while the master is the only thread communicating with the GPU. The master and workers are each associated to one CPU thread running on its own CPU core. Packet processing is made of three steps:

- *Pre-shading.* Worker threads receive batch of packets on their RX queues. They

perform pre-processing, such as dropping malformed packets, and classify the packets needing GPU processing. They enqueue those packets in the master queue.

- *Shading.* The master transfers the data from host memory to GPU memory, launches a CUDA kernel, and waits for the result. One GPU thread is generally dedicated per packet. For more intensive operations such as encryption, more than one thread can be dedicated per packet. Once finished, it copies the data back from device memory to host memory, and places the results back in the output queue of the worker thread that received the packet.
- *Post-shading.* The worker thread gets the results from its output queue and performs final modifications before transmitting the packets to the right port.

The system is partitioned into NUMA nodes, each one processing packets independently. Each node follows the master - workers architecture; workers thus only communicate with their local master, i.e. the one on the same NUMA node. This avoids expensive cross-nodes communication. The GPU has to be connected to the same NUMA node to benefit from the best performance, possibly providing a GPU per NUMA node.

1.4.2 Snap

Snap [89] is based on the Click modular router and aims to enhance the Click architecture and to provide GPU-accelerated Click elements. Integrating the speedups into a modular software router is more flexible as parts of the pipeline can continue to be handled by the CPU while using the GPU only for elements that will benefit from it, i.e. heavy computation processes. The fast path constitutes the processing performed on GPU, while the slow path is the processing done on CPU. A pipeline can be composed of fast and slow paths, combined thanks to adapters elements.

As explained in section 1.1.4, Click only allows elements to flow one at a time. As this is not enough to benefit from GPU offloading, Snap adds a new batch mode to Click; a similar implementation has been nowadays implemented in FastClick. A `Batcher` and a `Debatcher` elements are provided to collect packets in batch and separate batches into packets.

The workflow of a batch processing path with one GPU element is as follows. Packets enter the pipeline one at a time, either coming from a NIC or from another slow path Click element. They are first batched thanks to a `Batcher`, then copied from host memory to device memory thanks to a `HostToDeviceMemcpy` element. A GPU element then launches a GPU kernel for the batch and yields to a `DeviceToHostMemcpy` element, that will automatically copy back data from device to host memory when kernel execution finishes. At this point in the pipeline, as the GPU induces parallel processing, batches can arrive in disorder regarding the reception order. However, this can hurt TCP performance [50]. To prevent this, a `GPUCompletionQueue` element waits for preceding GPU operations to complete, by maintaining a FIFO queue of current operations. It only releases batches when all previous batches have been received. A `Debatcher` finally passes the packets

one at a time to connect the pipeline to a slow path element. All GPU interactions are managed by a `GPURuntime` element that communicates with the CUDA driver.

A difficulty faced by Snap is that all packets in Click do not follow the same path element-wise. Batches may need to be split while on the GPU, or before they reach the GPU. When packets diverge before reaching the GPU, it causes memory fragmentation, as batches can contain packets on non-contiguous memory regions. To avoid copying unnecessary packets, elements are first copied in a contiguous memory region in RAM then transmitted with a single transfer to the GPU. To overcome the need for classification on the GPU, predicates bits are attached to each packet. The GPU sets bits and path divergence happen on CPU, after the GPU processing, by reading the bits to indicate where each packet should be redirected.

Moreover, to further reduce unnecessary copies, Snap allows slicing packets by specifying Regions of Interest. A ROI is a contiguous range of data on which the GPU will operate. For example, an IPv4 lookup only uses 4 B of each packet, namely the destination IP address. Copying the entirety of the packet to device memory would be a huge waste of bandwidth.

Snap is designed to be multi-path, and such pipelines can occur in different cores of the CPU to accelerate the processing. The whole path is completely duplicated on each CPU core.

Thanks to its I/O optimizations and GPU offloading, the authors measured Snap performance to be up to 4 times better than the standard Click implementation for a fully functional IP router combined with an Intrusion Detection System (IDS).

1.4.3 APUNet

Kalia and al. have claimed [43] that the benefits of using GPUs arrive only thanks to latency hiding property of these devices and not thanks to the GPU hardware itself. This has led to the development of G-Opt [43], a CPU-only framework that accelerates packet processing by hiding Dynamic Random-Access Memory (DRAM) access latency, making extensive use of group prefetching and fast context switching.

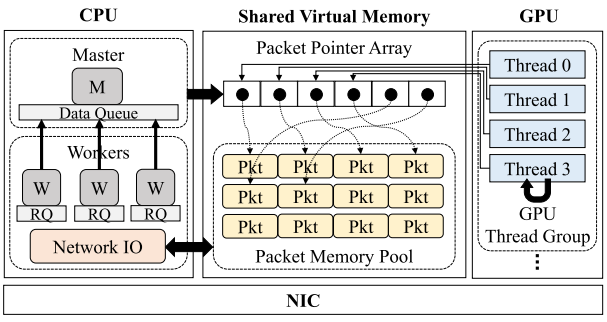


Figure 1.14: Overall APUNet architecture [33].

These claims have since been revisited by Go and al. [33] to find not only that GPU's computational power can well increase the performance of computationally intensive applications such as packet encryption, but also that the advantage of CPU over GPU lies in the data transfer bottleneck in PCIe communication. Indeed, PCIe bandwidth is way smaller than VRAM bandwidth. To overcome this, they propose to use an iGPU in an APU platform, removing the PCIe communication. *APUNet* [33] is an APU-accelerated network packet framework exploiting iGPUs that has precisely been developed to avoid using these costly PCIe communication.

However, as discussed in section 1.2.3, using an iGPU comes with the drawback that it does not include Graphics Double Data Rate (GDDR) memory, such that the GPU has to share the DRAM with the CPU. This can induce contention on the shared memory bus and degrade memory bandwidth. For that reason, APUNet applies zero-copy technique to all packet buffers, in all stages of packet processing, from reception to transmission and including GPU and CPU processing. Only pointers to data are passed between CPU and GPU during execution. This avoids doing unnecessary copies as the memory is shared anyway.

To reduce latency, APUNet uses persistent GPU threads execution. GPU threads are continuously running for a continuous input of packets. This completely removes the kernel launch overhead. A drawback of using persistent GPU kernels is that it holds GPU resources such as SMs, which is not the best option if the GPU has to be used for other tasks [3].

The overall architecture of APUNet is show in Figure 1.14. It uses a single master - multiple workers framework, just as PacketShader. A memory pool stores all packet payload and is shared between CPU and GPU. Each warp of 16 threads is initialized idle. DPDK and RSS is used to receive packets on worker threads. Packets are fed to the GPU in batch of 32 packets so that all threads in a warp will execute concurrently. Each packet is processed by exactly one thread. Each GPU group has a state shared with the CPU, indicating if the group is idle or active. Initially, all GPU thread groups are initialized idle and are polling on the state. When packets arrive, the master thread activates an idle group by setting its state to active. When it encounters it, the group processes the given packets and then switches the state back to idle. The master thread, also periodically polling on the state, when it detects the change of state back to idle, retrieves packets and pass them to workers.

APUNet is shown to outperform CPU baseline and G-Opt performance for memory intensive applications, such as IPv6 forwarding, IPsec gateway or a IDS. However, it performs worse for IPv4 forwarding. One core communicating with the GPU seems to be insufficient to match the performance of four CPU cores on this relatively lightweight operation.

1.4.4 GPUnet

GPUnet [88] features another approach to GPU programming. The authors note that GPUs lack abstractions allowing to control the data flow of a network system. Due

to this, a CPU is generally required to manage buffers or optimize memory transfers between NIC and GPU. This requires to deal with rather low-level and machine-specific details, such as memory transfers via DMA and network sockets.

GPUnet provides a socket abstraction and a high-level networking API directly to GPU code. It supports sockets for both network communications and inter-process communication on a local machine. Such abstractions remove the CPU from the interactions between NIC and GPU. It also unites packet I/O and application computation both on the same device. It leads to increased performance and reduces code writing for developers.

GPU sockets are very similar to existing CPU sockets. All GPU sockets are shared across all GPU threads and support the main calls in the standard network API. This allows CPU applications to also access remote GPU sockets. A main difference is that GPU sockets must be called at the granularity of a thread block, while CPU socket are single-threaded. This sticks to the groups of threads model used by GPUs, where divergence between warps reduce performance. A call returns as many elements as they are threads in the block, and each thread in the block treats one element received from the call.

To remove memory copies of packets between CPU and GPU memory, GPUnet uses a RDMA-capable NIC. It allows the NIC itself to store the messages in application memory, concurrently between both CPU and GPU memory. This allows processes running on the CPU and using sockets to still access messages. At the same time, new GPU-sockets enabled applications will get their own messages without having to coordinate their access to the hardware for each memory access. GPUnet still uses both CPU and GPU to interact with the NIC: while the GPU handles buffers for GPU applications, the CPU controls the NIC using the standard host drive to let it available to all processors in the system. The CPU is also used at initialization to allocate GPU memory buffers and register the memory with the NIC.

The authors evaluated GPUnet performance and found that it exceeds the throughput of a hexa-core CPU by 1.5 times using a single GPU for a face verification server, while offering 3 times slower latency than the CPU.

Chapter 2

Design

This chapter describes the design of multiple GPU-enabled solutions processing network packets at high-speed. We explore both VNF elements that can be placed in a processing pipeline, and CPU-bypassing.

Specific FastClick implementation

With the exception of DOCA, all implementations are based on and implemented using FastClick. We provide specific explanations in green boxes, but they are not essential to understand the design and functionality of the elements.

Implementations on CPU and DPU consists of FastClick pipelines using already existing elements. GPU-enabled FastClick elements that take advantage of GPU computation power are explained in detail in this chapter.

We forked FastClick from commit `5e29d77`[a] to create FastClick-GPU and FastClick-DPU. FastClick-GPU adds CUDA support to the FastClick toolchain, while FastClick-DPU allows for compilation on the BlueField-2 ARM cores.

[a] `https://github.com/tbarbette/fastclick/commit/5e29d77e9bad2906265e5794881ece3496c02bd9`

The GPU elements and the DPU implementation can be found on GitHub, respectively at `https://github.com/maxvanliefde/fastclick-gpu` and `https://github.com/OnyxDurga/fastclick_dpu`. The DOCA application is available on GitHub at `https://github.com/OnyxDurga/doca-app`.

2.1 Overall structure of GPU Elements

GPU-enabled elements repose on best practices for multithreading as described in [12] and follows a full-push multithreaded processing path. We provide a base GPU element skeleton, so that the user only needs to modify the GPU processing part and the needed

CPU parts. Figure 2.1 depicts the overall working of a GPU element in a pipeline, on a single CPU core.

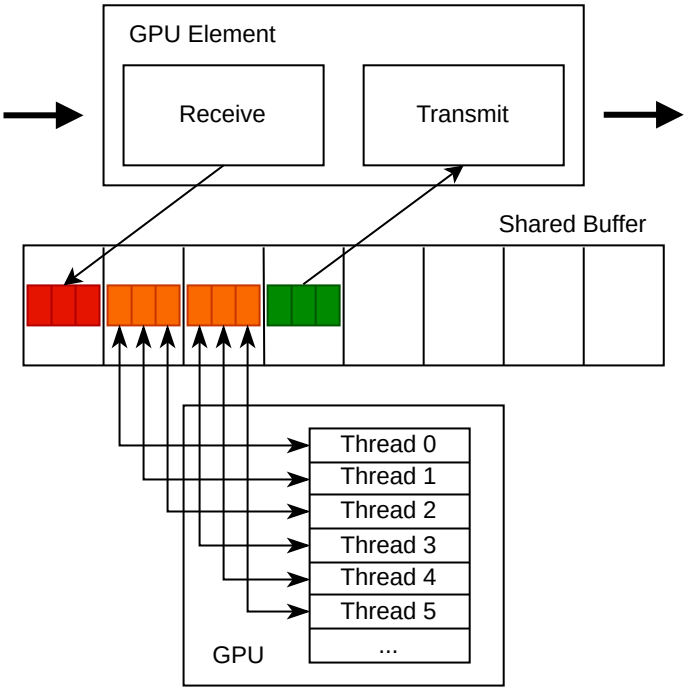


Figure 2.1: Simplified processing path of a GPU Element on a single CPU core. Red batch is waiting to be processed, orange ones are being processed, and the green one is processed.

Note that packets arrive in form of FastClick batches through the `push_batch` method. A FastClick batch is a linked list of Click elements, but elements expect packets to be pointers to a DPDK mbuf. FastClick must be configured accordingly with the `--enable-dpdk-packet` option. This way, a Click packet is simply a pointer to an mbuf in the DPDK memory pool.

Batches of packets are enqueued to a shared buffer allowing concurrent GPU and CPU access. The nature of this buffer differs following the memory access mode (cf. Section 2.4). Once a batch has been enqueued, the GPU is either awakened or realizes itself that it has to process packets (cf. Section 2.3).

The GPU workload is then performed in parallel on all packets of the batch. One packet is usually processed with one GPU thread (cf. Section 2.3). When the processing on all packets is done, a flag is set. During the GPU processing, the CPU is polling on the flag. When it finds that the batch is processed, it simply outputs it to the next element in the pipeline.

The polling is done using Click's task facilities. FastClick automatically alternates between polling on the GPU and on the NIC, so packets can still arrive even while batches are being processed.

Note that batches in a shared buffer are processed sequentially on the GPU. Two techniques can increase parallelism, resulting in higher GPU utilization. First, when the element receives larger batches, the GPU performs its computation on more packets at the same time. The element does not perform any further batching, it simply uses the batches as received on its receive port. Second, multiple shared buffers can be used in parallel. Elements can be configured in that way. The CPU fills the buffers in a round-robin fashion. The GPU, however, processes the current batch of packets in all buffers in parallel, using different CUDA streams, one per shared buffer. This uses more of the GPU's resources, and aims to reduce the packet processing latency: if the application is computationally intensive, new batches can arrive in less time than it takes to process a single batch.

The algorithms executed on receive and transmit sides can be found in Algorithms 2.1 and 2.2. A number of parameters can be specified by the user and are common to all GPU elements: `MAX_BATCH`, the maximum number of packets in a batch the element can handle; `CAPACITY`, the number of places in each shared buffer; `LISTS_PER_CORE` or `QUEUES_PER_CORE`, the number of shared buffers per core; and `BLOCKING`, specifying if the element should drop or not the packets when a buffer is full.

The algorithm on the receive side works as follows. When a batch is received, it is first checked if the batch is not too big. It only keeps the `MAX_BATCH` first packets, and drops the following ones. The batch is then prepared for processing, in a way that differs based on the memory access mode (detailed in Section 2.4). The current shared buffer is chosen and it is verified if the element at the current put-index is free. If it is not, it means that the GPU has not yet finished the processing of the item, but that the rest of the list has been filled: the buffer is full. In that case, if the element is in blocking mode, the processing is paused until the item has been processed; otherwise, the packet is dropped, and the processing is interrupted until another batch arrives and re-runs the algorithm. If the place is free, the batch is enqueued in the buffer, and the flag is set to *ready*. After that, the GPU is awakened, in a way that differs depending on the control flow (detailed in Section 2.3). Once the processing is started, the indexes of the next free space and of the next buffer to use are incremented, wrapping around respectively the capacity of the buffer and the number of buffers. This specifies a round-robin strategy to select the buffer to use. Finally, the transmit-side task is scheduled.

The algorithm on the transmit side works as follows, and is run periodically. The current shared buffer is chosen, and it is verified if the element at the current get-index has been processed. If it is not the case, the task is rescheduled and returns false. If the item has been processed, it is post-processed in a way that differs based on the memory access mode (detailed in Section 2.4). The element outputs the batch to the next element. Finally, the flag of the item slot is set to 'free' to specify that a new batch can use the

Algorithm 2.1 Simplified algorithm of a GPU-enabled element on CPU receive side.

Input: *batch*, a batch of received packets
Input: *putBufId*, the ID of the current buffer from which enqueue
Input: *task*, the task executing the transmit side when fired
Output: true if the batch has been enqueued, false otherwise

```
if COUNT(batch) > MAX_BATCH then
    a, b ← SPLIT(batch, MAX_BATCH)
    batch ← a                          ▷ Only first MAX_BATCH packets are processed...
    DROPBATCH(b)                       ▷ ...remaining packets are dropped
end if
batch ← PREPARE(batch)                 ▷ cf. Section 2.4
buf ← GETSHAREDBUFFER(putBufId)
putId ← GETPUTID(putBufId)
while GETSTATUS(putId) is not FREE do
    if BLOCKING is false then
        DROP(batch)
        return false
    end if
end while
PUT(buf, batch, putId)
SETSTATUS(getId, READY)
AWAKEGPU(putId)                        ▷ cf. Section 2.3
putBufId ← putBufId + 1                ▷ Wraps around the number of buffers
SETPUTID(putBufId, putId + 1)          ▷ Wraps around the capacity of the buffer
RESCHEDULE(task)
return true
```

Algorithm 2.2 Simplified algorithm of a GPU-enabled element on CPU transmit side.

Input: *getBufId*, the ID of the current buffer from which dequeue
Input: *task*, the task executing the algorithm
Output: true if the batch has been transmitted, false otherwise

```
buf ← GetSharedBuffer(getBufId)
getId ← GetGetId(getBufId)
if GetStatus(getId) is not DONE then
    Reschedule(task)
    return false
end if
batch ← Get(buf, getId)
batch ← Process(batch)                          ▷ cf. Section 2.4
OutputBatch(batch)
SetStatus(getId, FREE)
getBufId ← getBufId + 1                         ▷ Wraps around the number of buffers
SetGetId(getBufId, getId + 1)                   ▷ Wraps around the capacity of the buffer
return true
```

slot. The indexes of the next dequeue space and of the next buffer to use are incremented, wrapping around respectively the capacity of the buffer and the number of buffers, to respect the same round-robin strategy as on the received side.

The algorithm on the receive side is implemented by the `push_batch` method, and on the transmit side by the `run_task` method. Click uses the boolean return value of this last method for its internal scheduling of tasks.

Note that depending on the network speed, multiple batches can be pushed to the element before the task is actually fired. This is because FastClick only allows a task to be scheduled or unscheduled and does not count the number of times it has been scheduled. This would result in packets from the last batch queued not being processed. To fix that, a per-thread timer is also run for each thread. Its only utility is to continuously reschedule the task. Since timers are of lower priority that tasks, it will run only when no task performs any useful work anymore. It induces a small CPU overhead as it continuously reschedules the task, only when the element is not overloaded. If the packet flow is high, the task from `FromDPDKDevice` is always performing useful work, since there are always new packets from the NIC. It thus does not affect the maximum throughput.

2.2 CPU Multithreading

Previous section presented packet flow within the GPU element on a single CPU core. However, RSS balances the load between different CPU cores. Multiple CPU cores can

thus receive packets, leading to two possible multithreading schemes.

2.2.1 Full Path

Full Path means that each packet is processed by the CPU core that received it, from reception to transmission. A CPU thread is associated to each CPU core. To handle multithreading, each thread is assigned one shared buffer, and the pipeline is duplicated as many times as there are threads. As each buffer is associated with its own CUDA stream, the GPU can process concurrently all buffers of all cores. Figure 2.2 shows how packets flow through a classic pipeline using 2 CPU cores, hence 2 threads.

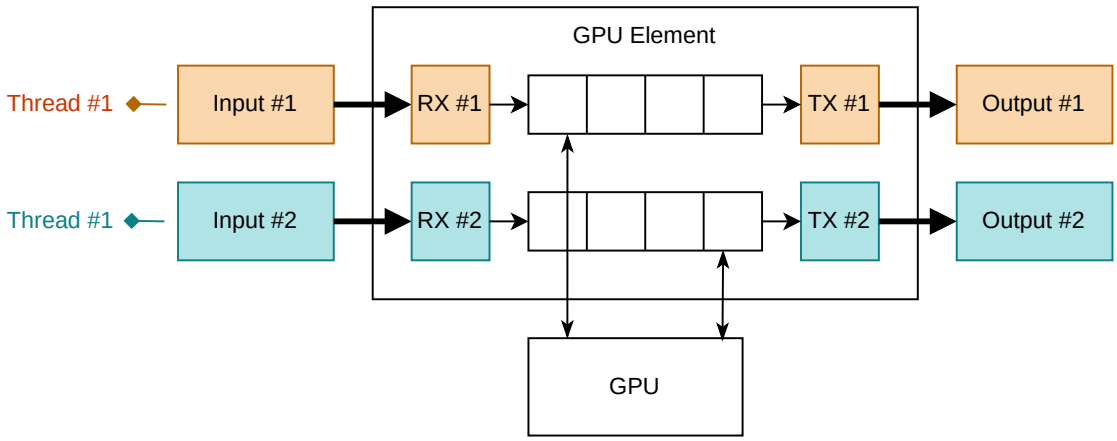


Figure 2.2: Simplified multithreaded full processing path on 2 cores. GPU threads are omitted for legibility.

With such configuration, increasing the number of cores used to retrieve packets not only spreads the load across the cores, but can also increase the utilization of the GPU by providing at least one shared buffer per core.

FastClick determines which threads may follow a path through each element on the pipeline. GPU Elements are designed to be thread-safe, using FastClick capabilities such as state de-duplication. Data can be specified as thread-specific and are then duplicated for all thread, e.g. for shared queues. Shared non-mutable data are stored once and can be accessed by all thread, e.g. parameters of the element.

2.2.2 Master-Workers

As a matter of comparison, we also implemented the Master-Workers model of execution that was used in previous works [37, 33].

In this model, only one CPU core handles communication with the GPU. GPU utilization is thus independent of the number of CPU cores used. Using more cores only

helps to spread the load of received packets using RSS, and thus receive more packets, but every packet will in the end transit on the same core.

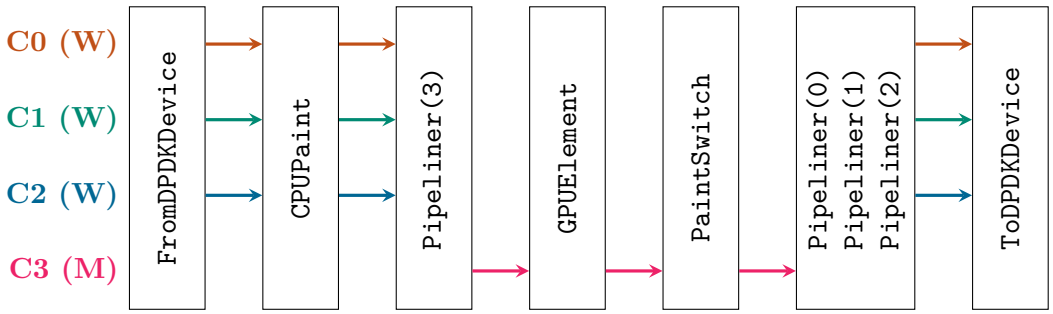


Figure 2.3: Click pipeline simulating a 1 Master - 3 Workers threading strategy. Packets issued on a thread are indicated by arrows. Worker threads run on cores 0 to 2, and the master thread runs on core 3.

We simply leverage Click capabilities to simulate such a model. We use a Full Path GPU element with only one thread pushing elements to it, which is the master thread. Packets are directed from worker threads to master and from master to workers threads using FastClick elements. Multiple worker threads receive packets and redirect all to the master thread who shares them to the GPU.

Such a pipeline can be found in Figure 2.3. Packet flows are indicated by arrays, which also indicate on which core the flow takes place. Workers receive packets, balanced between cores using RSS, through the `FromDPDKDevice` element. Each packet is then annotated with the CPU core that received it using a `CPUPaint` element. A `Pipeliner` (standard FastClick element) then redirects all traffic from worker threads to the master thread. Such an element behaves like a full-push queue: it stores incoming packets in a FIFO queue and periodically issues them all to a single thread [11]. The next element in the pipeline is the GPU Element. When FastClick detects that a single thread can push packets to it, it will initialize the element as monothreaded. This GPU element will thus handle all communication with the GPU from this master thread, and output each packet to the same thread. We then use a `PaintSwitch` element that reads the previously set annotation and then outputs each packet on the Click virtual port corresponding to the annotation. We place a `Pipeliner` on each output port i to redirect the traffic from the master thread to the worker thread i. finally, we connect all these `Pipeliners` to the `ToDPDKDevice` element, which will detect that multiple threads can push to it and use multiple queues and cores to handle the traffic. In the end, all packets will be output on the core that received them, while all GPU communication will be done on the master thread.

2.3 GPU Workload and Control Flow

Two methods are proposed to handle the control flow on GPU side: single kernel launches or persistent kernels. The GPU needs to be awakened by the CPU on receive of a new batch, and notifies the CPU back when all packets in the batch have been processed.

2.3.1 Single Kernel Launch

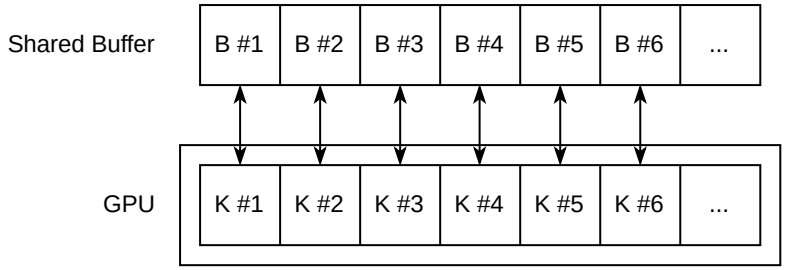


Figure 2.4: Control Flow when launching a kernel for each packet batch. B designates a batch, and K a GPU kernel.

The simplest method is to launch a new CUDA kernel to process each batch of packets. The kernel simply accesses the packets' data, processes it, and finishes. Figure 2.4 illustrates this control flow. Each batch in a shared buffer is processed by a separate kernel that is launched only for that batch and is never reused.

```
__global__ void kernel_single_launch(
    char **data, size_t *sizes, uint32_t num_pkts, ...
) {
    int pkt_id = blockIdx.x * blockDim.x + threadIdx.x;
    if (pkt_id < num_pkts) {
        size_t pkt_size = sizes[pkt_id];
        char *pkt_data = data[pkt_id];

        /* Perform workload on pkt_data */
        /* ... */
    }
}
```

Listing 2.1: Pseudo-implementation of a kernel launched for each batch.

A pseudo-code of how such kernel can be implemented can be found in Listing 2.1. In this example, each packet is processed by a single thread. The packet in the batch to process is computed based on the thread and block IDs; this allows to launch multiples blocks per batch. For example, to process a batch of 4096 packets, one could launch a kernel of 4 blocks each having 1024 threads.

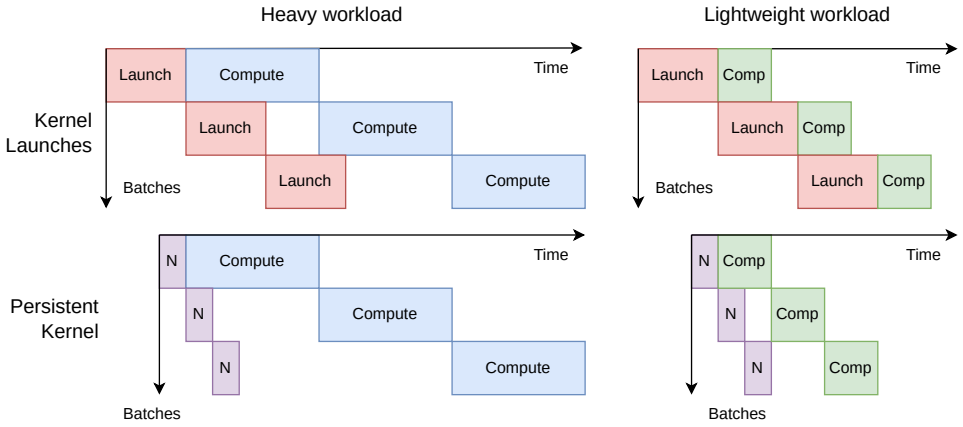


Figure 2.5: Timeline of batches processing with heavy and lightweight workload, using kernel launches and persistent kernels. N stands for Notify GPU, and Comp for Computation.

It is important to verify that the batch contains at least `pkt_id` packets before processing. If there are more threads than packets in the batch, some threads will have nothing to do. Since the "if" statement has no "else" part, the divergence in threads does not affect the execution time. It may be more efficient to launch more threads than necessary, especially so that the number of threads is a multiple of 32, as this facilitates coalescing [58]. When all threads have completed their workload, the kernel ends.

A drawback of this approach is that launching a kernel introduces a latency for every received batch. This overhead increases with the total number of threads in the kernel [6]. By profiling our application, we measure this latency to be about 5 µs with 1024 threads. This however can be mitigated if the workload executed by the kernel is heavy enough and the traffic is sufficient. GPUs are able to prepare the launch of a kernel while the previous one in the stream is running. If the kernel execution time is longer than the launching latency, the launch latency of the latest batch can overlap with the processing of the previous batch in the shared buffer. Figure 2.5 illustrates this behavior. Recall that the GPU processes the batches in the list sequentially. For a lightweight workload, time is lost when the computation on a batch is finished and the next kernel has not yet started.

An advantage of this method is that it is easy to synchronize. The CPU wakes up the GPU to work on a batch by running a kernel. The processing is done on all packets when this kernel finishes, which can be detected by the CPU, e.g. by a CUDA event issued in the stream after the kernel launch (cf. Section 2.4). A consequence of this is that any number of threads can easily be dedicated to each packet, as there is no need to synchronize threads on the GPU side: they all have to arrive at the end of the kernel.

On the CPU side, the AwakeGPU() procedure call from Algorithm 2.1 simply consists of a kernel launch, possibly followed by a CUDA event registration (cf. 2.4).

2.3.2 Persistent Kernel

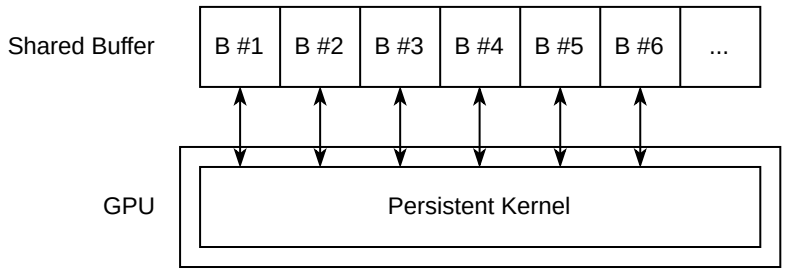


Figure 2.6: Control Flow when using a persistent kernel for each shared buffer. B designated a batch.

Another approach is to pre-launch a persistent CUDA kernel for each shared buffer during the element initialization phase. Each kernel loops through its own shared buffer. It waits until the first batch becomes ready, processes the packets, notifies the CPU, and moves on to the next batch in the list. Figure 2.6 illustrates this control flow. Each batch in a shared buffer is processed by the same persistent kernel. Each shared buffer is associated with a single persistent kernel, which processed the batches sequentially as they are placed in the buffer.

The easiest way to implement it is to associate a flag with each batch. The flag resides in either GPU or CPU memory and is shared between both the host and the device.

The advantage of this approach is that it removes the kernel launch overheads. The overhead of setting a flag can be lowered if it resides in GPU memory by using the GDRCopy library. As discussed in Section 1.2.6, a standard CUDA memory copy takes about 7 µs, which would not be beneficial since it is relatively the same latency as a kernel launch. However, a CPU-driven GDRCopy takes only 1 to 3 µs, depending on the copy direction. The flag could also reside in CPU memory visible by the GPU, but then the GPU polling on the flag would pollute the PCIe connection and needlessly consume bandwidth [76].

As seen in Figure 2.5, such improvement will not be visible if the workload is heavy. If it is lightweight however, using persistent kernels can help reduce the time before the computation of the next batch begins, reducing latency and improving throughput.

One drawback of this approach is the synchronization required between threads needed on the GPU side. Indeed, all threads working on packets in a batch must complete their processing before the flag is set to done. This requires thread synchronization of on GPU-side. Unfortunately, CUDA only provides group-wide synchronization through the `__syncthreads()` function[1]. Because of this, all threads working on a batch need to be part of the same group, which limits the number of threads per batch to 1024.

[1]Starting with CUDA 9, Cooperative Groups [59] allow threads from different blocks to synchronize with each other using grid synchronization. However, current GPUs only support parallel execution of a single Cooperative Group kernel, which prevents execution of more than one persistent kernel, which is not acceptable with our implementation.

```
__global__ void kernel_persistent(
    char ***data, size_t **sizes, uint32_t *num_pkts,
    uint32_t buffer_size, uint32_t *flags, ...
) {
    int pkt_id = threadIdx.x;
    uint32_t item_id = blockIdx.x;
    uint32_t flag;
    __shared__ uint32_t shared_flag;
    while (1) {
        /* Polling on the flag */
        if (pkt_id == 0) {
            while(1) {
                flag = RTE_GPU_VOLATILE(*flags[item_id]);
                if (flag != FREE) {
                    flag_shared = flag;
                    __threadfence_block();
                    break;
                }
            }
        }
        __syncthreads();
        __threadfence_system();
        if (wait_status_shared != READY) break;

        /* Perform workload on pkt_data */
        if (pkt_id < num_pkts) {
            size_t pkt_size = sizes[item_id][pkt_id];
            char *pkt_data = data[item_id][pkt_id];
            /* ... */
        }
        __threadfence();
        __syncthreads();

        /* Notify CPU */
        if (pkt_id == 0) {
            RTE_GPU_VOLATILE(*flags[item_id]) = DONE;
            __threadfence_system();
        }
        item_id = (item_id + gridDim.x) % comm_list_size;
    }
}
```

Listing 2.2: Pseudo-implementation of a persistent kernel running through a shared buffer.

Typically, since only threads of the same group will handle a batch, only one thread of the group needs to poll on the flag, while the others wait on a `__syncthreads()` barrier. This eliminates unnecessary memory accesses, as a flag is dedicated to an entire batch and not per packet. Of course, the same goes for the writing of the flag after all threads have finished processing.

Another drawback is that persistent kernels hold GPU resources even when network traffic is low, increasing power consumption. However, if the GPU is only or mainly used for this purpose, this is not a problem. It can also cause packet losses if there are more kernels launched than there are SMs on the device. Since such kernels cannot be preempted, some kernels will never be able to run.

A final point to consider is memory consistency. NVIDIA GPU architectures only guarantees PCIe read-after-write ordering for the BAR [2]. However, we need write-after-write ordering to ensure that when the flag is written, data written to packets is also written. This is done using memory barriers. On the GPU, `__threadfence_block()` ensures write-after-write ordering on all threads in the block, `__threadfence()` on all threads in the device, and `__threadfence_system()` on all threads in both the device itself and peer devices. On the CPU, the memory barriers provided by DPDK are used, as `rte_wmb()` for a write barrier and `rte_rmb()` for a read barrier. On the receive side, a write barrier is used after the packet information is written to the shared buffer and before the flag is set to wake up the GPU. On the transmit side, a read barrier is issued between accessing the flag and accessing the packet data.

On GPU side, Figure 2.2 shows the CUDA implementation of such a persistent kernel. On CPU side, the AwakeGPU() procedure call from Algorithm 2.1 only consists of a write-barrier to ensure memory coherency, as detailed before.

2.4 CPU-GPU Communication

To simplify communication between the CPU and the GPU, DPDK offers, thanks to a contribution for NVIDIA, the General-Purpose Graphics Processing Unit Library [76] – abbreviated as GPUdev – since version 21.11. This library only supports NVIDIA GPUs. While it enhances dialog between the CPU and the GPU, it may not always be the best choice for high-speed processing. This section discusses first the usage of the DPDK framework, and then a pure CUDA implementation based on coalescent ROIs.

> Due to the very different nature of these two implementations, each implemented VNF is available as two different elements: `GPUElementCommList` and `GPUElementCoalescent`.

2.4.1 Communication List

The GPUdev library is threefold: it allows detection of GPUs in DPDK as PCIe devices; it allows packets to be sent and received directly to and from GPU memory using

GPUDirect RDMA, removing copies from and to CPU memory; and finally, it provides abstractions to communicate between the CPU and GPU [49].

To use the abstractions offered, the DPDK integration in FastClick had to be modified regarding memory pool allocation. The `DPDKInfo` element allows users to specify whether memory should be allocated on the CPU or GPU. If the pools reside in CPU memory, the CPU memory is registered and made available to the GPU device. This allows classic CPU FastClick elements to modify the packets, and GPU elements to access and modify them using PCIe requests. If pools are in GPU memory, the CPU cannot access the memory. The advantage is that GPU accesses are much faster, since the bandwidth of the VRAM is much higher than that of PCIe. The drawback is that all processing on packets must be performed by GPU kernels, as the CPU does not have access to them: existing FastClick elements cannot be used in this mode.

Such GPUdev library abstractions include *communication lists*. Their structure is shown in Listing 2.3.

An instance of the communication list is an array of n `rte_gpu_comm_list` items. The entire list is pre-allocated in CPU memory visible from the GPU, with a given capacity n, to avoid memory (de)allocation on the critical path. Each element in the list is associated with a GPU (designated by `dev_id`) and can contain up to 1024 packets[2]. `mbufs` is an array of all mbufs in the list item, while `pkt_list` is an array of `rte_gpu_comm_pkt`. While they both represent the same packets, the mbufs are used by DPDK on CPU side, while the GPU only needs the packets list. Each packet is represented by the address of the beginning of its payload, `addr`, and its size, `size`. Finally, a `rte_gpu_comm_list_status` status flag allows synchronization between CPU and GPU through polling on it. It specifies the item processing status: it can be `FREE`, i.e. free to be filled with new mbufs as nobody is currently using it; `READY`, i.e. filled and ready to be processed by the GPU; or `DONE`, i.e. already processed and ready to be freed. The erroneous state allows to specify that the GPU has to exit processing of new packets. This flag is allocated initially free on GPU memory using the GDRCopy library, because it reduces the latency to retrieve value when polling on it both from the GPU and from the CPU (cf. Section 1.2.6).

Finally, each communication list is associated with a `comm_list_state` that specifies the capacity of the list, the current put and get indexes, and a CUDA stream where kernel operations working on that list will be issued.

The status flag is the critical piece that allows for control flow between the CPU and the GPU. When the CPU is working on populating a batch, the GPU polls on the flag and waits for it to become `READY`. When the GPU is processing a batch, the CPU

[2]This limitation is due to the fact that, as explained in Section 2.3, all threads processing a batch must be part of the same thread block to allow processing from a persistent kernel on the GPU. Since the maximum number of threads in a kernel block is 1024, the maximum number of packets in a batch is also 1024.

```c
/* From DPDK gpudev library */
struct rte_gpu_comm_pkt {
    uintptr_t addr;
    size_t size;
};
enum rte_gpu_comm_list_status {
    RTE_GPU_COMM_LIST_FREE = 0,
    RTE_GPU_COMM_LIST_READY,
    RTE_GPU_COMM_LIST_DONE,
    RTE_GPU_COMM_LIST_ERROR,
};
struct rte_gpu_comm_list {
    uint16_t dev_id;
    struct rte_mbuf **mbufs;
    struct rte_gpu_comm_pkt *pkt_list;
    uint32_t num_pkts;
    enum rte_gpu_comm_list_status *status_h;
    enum rte_gpu_comm_list_status *status_d;
};

/* Own implementation */
struct comm_list_state {
    struct rte_gpu_comm_list *comm_list;
    uint32_t comm_list_size;
    uint32_t comm_list_put_index, comm_list_get_index;
    cudaStream_t cuda_stream;
};
```

Listing 2.3: Structures and enumeration implementing the communication list [76], and its use in the implementation of GPU elements.

polls the flag and waits for it to become `DONE`. This allows for low latency, since a flag change is detected almost immediately on both sides, but comes with an overhead in data transfers on the PCIe link.

The functions of the GPUdev library used for our implementation are listed in Table 2.1. This allows us to specify the function calls used in Algorithm 2.1. The shared buffers used are communication lists that are pre-allocated during element initialization.

- PREPARE(*batch*). It converts the batch into an array of `rte_mbuf` pointers.
- GETSTATUS(*id*). It calls `rte_gpu_comm_get_status` on the item of index *id*.
- PUT(*buf,batch,putId*). It calls `rte_gpu_comm_populate_list_pkts` with the communication list item, and the batch as a `rte_mbuf` pointer array, with its size as an extra argument. This sets the status flag to `READY`.

Table 2.1: GPUdev functions used on CPU side in our implementation, and their utility [76].

Name	Utility
`rte_gpu_comm_create_list`	Allocate list, on initialization
`rte_gpu_comm_populate_list_pkts`	Populate item, on batch receive
`rte_gpu_comm_get_status`	Polling, during item processing
`rte_gpu_comm_cleanup_list`	Reset item state, on batch transmit
`rte_gpu_comm_set_status`	Terminate item processing, on cleanup
`rte_gpu_comm_destroy_list`	Free list, on cleanup

In the process of Algorithm 2.2, similar processes occur.

- GetStatus(*getId*). It calls `rte_gpu_comm_get_status` to retrieve the status of the *getId* item.
- Get(*buf,getId*). It simply accesses the `mbufs` field of the list item of ID *getId* to receive an array of `rte_mbuf` pointers.
- Process(*batch*). This call scans the array to recreate a proper batch.
- SetStatus(*getId, FREE*). It calls `rte_gpu_comm_cleanup_list` with the communication list item, that will reset the item and set its status flag to `FREE`.

Prepare(*batch*) and Process(*batch*) are needed because FastClick batches are implemented as linked lists, while the communication lists use arrays. With FastClick configured as specified before, packets are just `rte_mbuf` pointers, as used by the communication list. However, it is still necessary to iterate through the FastClick batch to copy each packet's `rte_mbuf` pointer into an array, and vice versa on the transmit side.

As indicated by the presence of status flags, communication lists are designed to be compatible with persistent kernels (cf. Section 2.3). However, the use of standard kernel launches is also supported. On the GPU side, VNFs kernels only need access to the communication list items that they need to process, depending on the chosen control flow: the whole list for a persistent kernel to loop through, or just one item for a single kernel. While the GPUdev library functions are not designed to run on the GPU, the device knows the list structure and can directly access the status flag and packet payloads.

While using the communication list offers a desirable abstraction to enhance communication between the CPU and the GPU, it may not always be the perfect implementation. Drawbacks include the need to transform batches from linked-list to array and vice versa, and limiting the size of a batch to 1024 packets. Kernel code also has to be specialized to handle the communication correctly.

Moreover, all packet payloads reside in memory pools managed by DPDK, which does not guarantee any order [81]. Each thread reading a different packet will issue a different read request, needing one memory transaction for each thread. Depending on the VNF, it may be useful to extract the payload of each packet to a contiguous memory location.

2.4.2 Coalescent Regions of Interest

A number of VNFs only use a portion of the packet to process it. In particular, many only need a protocol header. For example, IP lookups only need the destination IP address, and Ethernet mirroring only needs the source and destination MAC addresses. The contiguous byte range of the packet that is used is called a Region of Interest. Recall also from Section 1.2.4 that GPUs read from global memory by section of 32 B. It is interesting to coalesce ROIs of packets if this ROI is smaller than the size of a single read. This way, multiple threads of the GPU can be served with a single memory access.

Shared buffers are simply implemented as a large contiguous memory area, called a *queue* in our implementation. This allows to specialize this area depending on the VNF implemented. Just as with the communication list, the user has to specify the capacity c of each queue, and the maximum number of packets per batch p. If n bytes are required per packet, $c \cdot p \cdot n$ bytes are allocated per queue. The structure of this memory area differs for each VNF and is explained in Chapter 3. For example, if the ROI is n bytes in size, then n bytes need to be allocated per packet. We say that the *stride* is n bytes long. It is up to the user to specify the region of bytes needed, and to take into account padding to align packet data on the cache line if necessary.

```
/* Own implementation */
struct queue_state {
    char *h_memory, *d_memory;
    uint32_t put_index, get_index;
    cudaStream_t cuda_stream;
    cudaEvent_t *events;
    PacketBatch **batches;
};
```

Listing 2.4: Structure implementing the queue.

The structure of the shared buffer in this implementation is shown in Listing 2.4. `h_memory` and `d_memory` hold a pointer to the allocated memory area, accessible from the host and device, respectively. Each queue is associated with the current put and get indexes, and with a CUDA stream where operations working on that queue are issued. Regarding communication, while we could have used a mechanism like the communication list provides by polling a flag, we preferred to use events, CUDA's way of synchronizing within streams. `events` is an array of c events. Finally, `batches` is an array of all batches associated with currently processed elements. Each element in the queue is thus

composed of a memory area of $p \cdot n$ bytes containing the required payload bytes, a CUDA event used to record when a processing is finished, and a pointer to the associated batch.

Table 2.2: CUDA Event Management functions used on CPU side in our implementation, and their utility [59].

Name	Utility
`cudaEventCreateWithFlags`	Allocate event, on initialization
`cudaEventRecord`	Record event, on batch receive
`cudaEventQuery`	Polling, during item processing
`cudaEventDestroy`	Free event, on cleanup

The element on the fast processing path that coordinates the CPU and GPU is a CUDA event. Useful CUDA functions operating on this structure are listed in Table 2.2. Note that events are created during element initialization with disabled timing and no blocking synchronization. They can also be reused without cleanup: each record call overwrites the previously captured state. After a kernel launch, an event is recorded on the stream using `cudaEventRecord`. When the kernel finishes, the CUDA library automatically updates the flag: the programmer does not need to do anything in the kernel code. Using the `cudaEventQuery` function, the CPU can query the event status to detect when this happens. This function returns `cudaSuccess` if the captured work has been performed, i.e. if the batch processing is finished, or `cudaErrorNotReady` if it is not the case.

With only two states for this event flag, it is not possible to distinguish whether the item slot is free or if it is ready but not yet transmitted. To prevent the CPU from overriding a non-free element, which would result in a leak of packets, the batch pointer is set to `nullptr` after the element outputs the batch to indicate that the queue item is now free. Otherwise, it contains the pointer to the batch to be processed.

Another important consideration is how the memory is accessed. Memory areas are pinned, just as what the communication list does. The GPU can directly access this memory by issuing PCIe transactions during kernel execution. By using this zero-copy technique, the GPU cannot use its cache, so the data has to be read or written only once. Zero-copy can therefore lead to performance degradation if further processing is required on the GPU side. In this case, it may be more efficient to first explicitly copy the data to GPU memory first, process it, and then to copy it back to the host. The `ZEROCOPY` configuration parameter allows the user to choose whether to use it or not.

From the GPU side, the processing must take into account that the given pointer points to the memory area reserved for the batch, and contains only ROIs. It thus does not point to the beginning of the payloads.

The function calls used on received side in Algorithm 2.1 are specialized in the following way.

- PREPARE(*batch*). For each packet in the batch, the ROI is copied in the correct memory area of the queue. These ROI are made contiguous in memory.

- GetStatus(*putId*). It verifies if the batch of the item is `nullptr`. If so, the item is free.
- Put(*buf,batch,putId*). It sets the batch of the item to the current batch. This allows to remember the packets pointers after the processing is done.

Similarly, the function calls on transmit side of Algorithm 2.2 are also specialized.

- GetStatus(*getId*). It queries the event and verifies that both the batch of the item is `nullptr` and the event query returned `cudaSuccess`. If so, the processing is done.
- Get(*buf,getId*). It simply retrieves the item of ID *getId* in the `batches` array.
- Process(*batch*). This call scans the memory of the item to copy each modified ROI back into the original packet memory in the DPDK memory pool.
- SetStatus(*getId, FREE*). It sets the batch of the item to `nullptr`.

A drawback of this event-based approach is that persistent kernels cannot be used. Indeed, events record when a kernel finishes its processing, but persistent kernels would loop on each item in the queue without finishing.[3]

Moreover, while the implementation allows to specify a beginning and end of a ROI per packet, this is not always sufficient, for example if the processing needs access to two ROIs. The user can fully customize the memory area, but in this case they have to specify the structure of the memory and follow it carefully on CPU and GPU side. They must also consider adding padding bytes, as performance is increased when each ROI of a packet resides in only one cache line.

This approach can lead to better performance when multiple ROIs can be contiguous in one area of 32 B, as it is how reads and writes are issued to global memory. In this way, one memory transaction may serve multiple threads. For example, for an IPv4 lookup whose ROI size is 4 B, one memory transaction would serve $32/4 = 8$ threads all in one. When explicit copies are used, the full power of the GPU memory can be used because the data is stored in global memory and is automatically cached at various levels by the GPU.

2.5 CPU bypass

The code for the CPU bypass application using DOCA is heavily based on the GPUNetIO Ethernet Simple Receive [64]. The modified code with explanations on how to use it are available at `https://github.com/OnyxDurga/doca-app`.

[3] An alternative could be to use a *CUDA graph*. Such structure records operations issued on a CUDA stream and can be replayed multiple times. It helps reduce launch kernel times [59]. However, based on tests, while kernel launch times are indeed reduced by a few microseconds, this solution introduces a latency on the order of milliseconds each time the graph has to be replayed, which is not acceptable in fast packet processing.

2.5.1 Main Architecture

DOCA uses receive and send queues to transfer packets from the NIC to the GPU and the other way around. The buffers of these queues are mapped to the NIC using the `nvidia-peermem` kernel module. For each buffer mapped to the NIC, some space of the BAR1 is used. As these buffers require more than the default 256 MB of memory, resizable BAR is required for this application.

The application first initializes what it needs on the CPU. This includes devices configuration, memory allocation, memory mapping, and launch of the CUDA kernels. Once everything is initialized and started correctly, the GPU can communicate directly with the NIC, and the CPU is out of the data path and thus not needed anymore for packet processing.

2.5.2 Multithreading

Algorithm 2.3 Simplified algorithm of a CUDA kernel for CPU bypass.

Input: rxq, a receive queue
Input: txq, a transmit queue
Input: $batch$, the maximum amount of packets to receive per block
  $num \leftarrow$ RECEIVE($rxq, batch, 0$) ▷ Stores the number of received packets in num.
  $id \leftarrow threadIdx.x$
  while $id < num$ **do** ▷ Ensures all packets of the batch are processed.
    $ptr \leftarrow$ GETBUF($rxq, num + id$)
    $addr \leftarrow$ GETADDR(ptr)
    PROCESSPKT($addr$) ▷ Implements the network function.
    ENQUEUE(txq, ptr)
    $id \leftarrow id + blockDim.x$ ▷ Allows each thread to process different packets.
  end while
  if $threadIdx.x = 0$ **then** ▷ Only a single thread sends the packets.
    COMMIT(txq)
    PUSH(txq)
  end if

We modified the code to handle up to 8 queues with a capacity of 16384 packets each. Each queue has a dedicated block on the GPU to process its packets. Each block has 32 threads processing packets from the same queue in parallel. The simplified logic of a CUDA kernel processing packets using DOCA for one block is shown in Algorithm 2.3. The `Receive` function takes the queue from which to read packets and allows to specify a maximum amount of packets to receive in one time and a timeout delay. The maximum number of packet is used to receive packets by batches of specified sizes. The batching size is specified as an argument at launch, while timeout is always set to 0. Next, each thread gets a packet's buffer address and processes it. The threads process packets in parallel. If the batching size is greater than the number of threads per block, each thread

iterates over the packets by jumps of `blockDim.x`, the number of threads spawned in the block, so that all the packets are processes and each thread processes different packets.

When a thread finishes processing a packet, it enqueues it in the send queue. Finally, when all the packets of the batch are processed, one thread commits and pushes the send queue, effectively telling the NIC to send the packets in the queue.

There are two ways of enqueuing packets in a queue: Strong Mode and Weak Mode. Strong Mode enqueues the packet in the next available spot in the queue. This is simpler than Weak Mode, but can cause more latency. Weak Mode, on the other hand, lets the developer calculate the next available position and enqueue the packet wherever it wants in the queue. This is more challenging to handle, but might lead to better performance and may allow the developer to improve the memory locality. We chose to implement our application using Strong Mode for simplicity.

Chapter 3

Applications

Three applications describing typical VNFs have been implemented and evaluated using each type of GPU element as described in Chapter 2: Ethernet Mirroring, IP Lookup, and CRC Computation. This chapter describes these functions and their implementation.

FastClick configuration

These VNFs are already implemented as FastClick elements, which allows for easy comparison between existing CPU and new DPU and GPU implementations. FastClick configurations are provided in green boxes.

The FastClick CPU implementation is provided as a baseline, where the CPU cores handle both the I/O and the packets. The DPU implementation uses the same pipeline as the CPU, but runs on the SmartNIC cores using FastClick-DPU.

CUDA kernel code that implements the functions for both element versions can be found in the FastClick-GPU repository, under the `lib/cuda` subfolder.

3.1 Ethernet Mirror

Ethernet Mirroring is a network function that operates on Ethernet frames, whose format is shown in Figure 3.1. Such a function operates in the Data Link Layer, the second layer (L2) of the Open Systems Interconnection (OSI) model. The workload consists of swapping the frame's source and destination Media Access Control (MAC) addresses.

A typical use case is to form an L2 forwarder: each packet received on a port is forwarded back to its sender. It is a classic example application with a light workload, as only 12 B are read and modified for each received packet, regardless of its size – Destination MAC Address and Source MAC Address fields, both of 6 B. Such application is available as a DPDK sample [20] and as an NVIDIA sample using the GPUdev library [2].

The two GPU elements that implements this network function, using the GPUdev

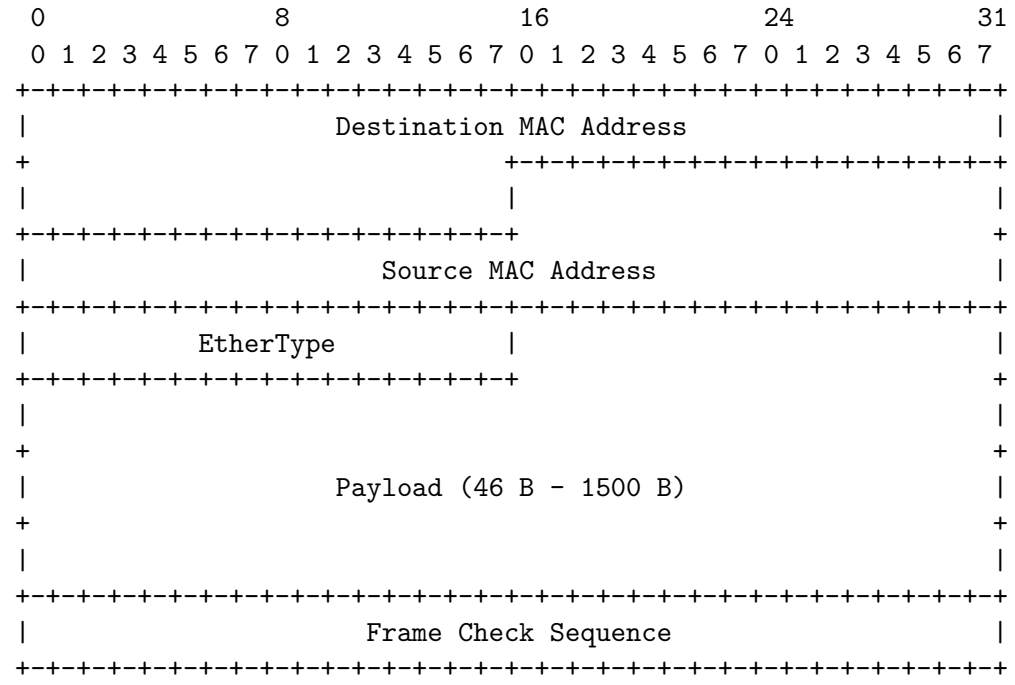


Figure 3.1: Ethernet frame format [38]. Each tick mark represents one bit position.

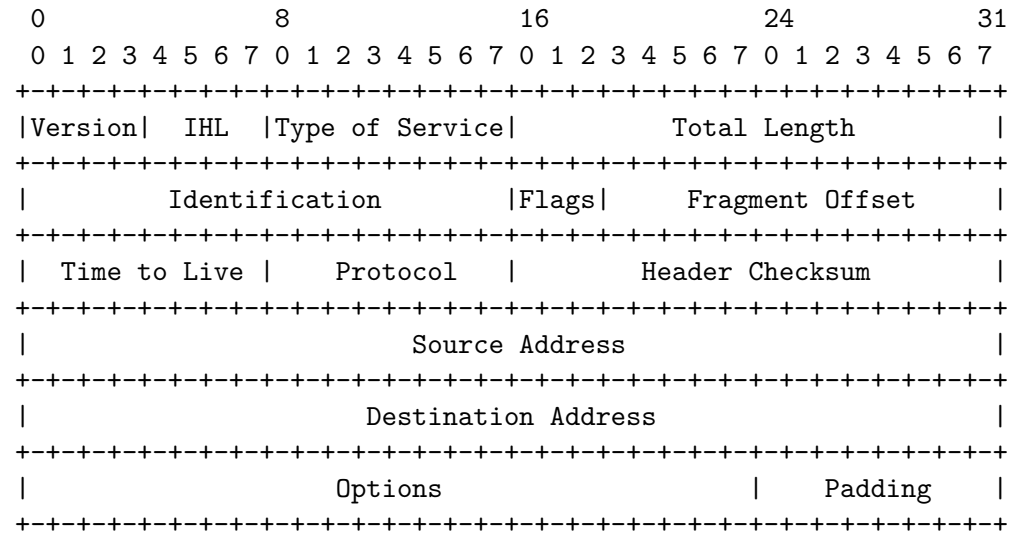


Figure 3.2: IPv4 packet header format [82]. Each tick mark represents one bit position.

communication lists and the coalescent ROIs described in Section 2.4 respectively, are `GPUEtherMirrorCommList` and `GPUEtherMirrorCoalescent`.

The following standard FastClick configuration implements an Ethernet Mirroring application [25].

```
FromDPDKDevice(0) -> EtherMirror -> ToDPDDevice(0)
```

`FromDPDKDevice` issues packets in bursts of 32. Packets are generally further batched before being sent to the GPU elements, with a batch size of $n > 32$. The `MinBatch` element batches packets up to the given parameter plus the burst size. To ensure that the maximum number of elements batches is n, we set the `MinBatch` parameter to $n - burst = n - 32$.

```
FromDPDKDevice(0)
    -> MinBatch(n − 32)
    -> GPUEtherMirrorCommList(MAX_BATCH n)
    -> ToDPDDevice(0)
```

`GPUEtherMirrorCommList` can also be replaced by `GPUEtherMirrorCoalescent`. These three pipelines are functionally equivalent.

With this network function, the GPU version using coalescent ROIs can be expected to outperform the version using communication lists, as only a portion of each packet of less than 32 B is only needed (cf. Section 2.4). The stride for the coalescent version is thus simply $n = 12$ B.

3.2 Internet Protocol Lookup

IP Lookups are network operations that consist of routing packets and determining their next hops. IP lookups operate on Internet Protocol (IP) frames, whose format is shown in Figure 3.2. These IP frames reside in the Networking Layer, which is the third layer (L3) of the OSI model. The main work of this function consists of looking up the IP destination address of the packet in a routing table to find out which interface the packet has to be sent on to eventually reach its destination. There exists plenty of different ways to iterate through a routing table. We used a linear search for its simplicity and ease of implementation on a GPU. We also used exclusively Internet Protocol Version Four (IPv4) packets.

This workload is usually done by routers managing traffic through several interfaces. The workload is highly dependent on the routing table's size, but is usually much heavier than for an Ethernet Mirror. Only 4 B of information are needed for each packet, as this is the size of the destination IP addresses of IPv4 packets. This is an advantage for GPU

processing, as the information that has to be sent to the GPU is very small, 4 B per packet, and the bulk of the work is done on the GPU.

The routing table used comes from the Réseaux IP Européens (RIPE) Routing Information Service (RIS) database [84]. We obtained the data from the collector 01 on 8 April 2024 thanks to a script provided by Pr. Cristel Pelsser [79]. We have formatted the data and removed any duplicate entry. Appendix B provides the scripts we used.

The linear IP lookup element provided by Click can be used as shown in the following pipeline adapted from [47], which performs an IP lookup with a table of 4 entries.

```
rt::LinearIPLookup(
    1.2.3.4/16 0,
    5.6.7.8/16 1,
    9.10.11.12/16 2,
    13.14.15.16/16 3);

FromDPDKDevice(0)
-> Strip(14)
-> GetIPAddress(16)
-> rt;

rt[0] -> Unstrip(14) -> ToDPDKDevice(0);
rt[1] -> Unstrip(14) -> ToDPDKDevice(1);
rt[2] -> Unstrip(14) -> ToDPDKDevice(2);
rt[3] -> Unstrip(14) -> ToDPDKDevice(3);
```

Each route has to be specified in the configuration of the element. The Linear IP Lookup from Click uses a Vector of structures as shown in Figure 3.1 to store the routing table. This structure stores the entry's IP address, mask, gateway, interface port, and an extra field. This extra field is used to optimize the lookup process. When the vector is initialized, if a subnet is contained in an already present entry, its extra field is updated to contain the index of the new entry. This reduces the total number of checks needed to find a match.

The two corresponding GPU elements are `GPUIPLookup` and `GPUIPLookupWithCopy`, respectively using the GPUdev's communication lists and the coalescent ROIs. The use of these elements is slightly different from their CPU version. An example for the `GPUIPLookup` can be found below, but `GPUIPLookupWithCopy` works the same way. Packets are batched up to n packets per batch. These elements implement exactly the same algorithm as the `LinearIPLookup` element, but the whole Vector is stored as an Array on the GPU itself.

```
FromDPDKDevice(0)
```

```
-> MinBatch(n − 32)
-> GPUIPLookup(TABLE table)
-> ToDPDDevice(0)
```

Note that the element now takes a table number as entry. We stored the content of the RIPE RIS routing table in a file and modified the elements to read the list from this file instead of computing it on every initialization. This allowed to save some time, as the time to create this list could take several minutes for larger routing tables due to the aforementioned optimization.

```
struct IPRoute {
    IPAddress addr;
    IPAddress mask;
    IPAddress gw;
    int32_t port;
    int32_t extra;
}
```

Listing 3.1: Structure used to store entries of the routing table.

3.3 Cyclic Redundancy Check Computation

A Cyclic Redundancy Check (CRC) is an error-detecting code used mainly in digital networks. Each Ethernet frame (see Figure 3.1) ends with a Frame Check Sequence (FCS), which consists of a 32-bit CRC that allows the detection of corrupted data on receiver side.

CRC Computation of the process of computing the CRC-32 FCS value of a given Ethernet frame. This value is computed as a function of the contents of all fields of the frame, except the FCS field. It consists of polynomial division of the data by a 32nd degree polynomial, as described in detail in section 3.2.9 of the IEEE 802.3 specification [38].

Two VNFs take advantage of this computation. *SetCRC32* computes the CRC-32 of the frame, appends it to the frame and transmits it. *CheckCRC32* computes the CRC-32 of the frame, checks if the FCS field matches the computed value. It drops the frame if it does not, otherwise it removes the CRC field and transmits the frame.

These two functions are usually implemented in hardware on the NICs, as they can incrementally compute the CRC on the bits while they receive or transmit a frame [17]. CRC Computation is still interesting to compare between CPU and GPU because it is a computation-intensive application. Moreover, the amount of data processed depends on the size of the frame, as it is a function of all the contents of a frame. Also, the computation cannot be easily parallelized because the code is cyclic. Indeed, the

calculation on one byte requires the result of the calculation on the previous byte. The computational power of the GPU can thus be exacerbated, while a CPU with n cores will struggle to compute more than n CRCs in parallel.

FastClick offers two elements, `SetCRC32` and `CheckCRC32`, that implement these two functions in software, operating on bytes. It is based on a pre-computed table of size 256 – called the *CRC table* – and has been shown to be nearly five times faster than a software implementation based on bits [80]. Appendix A.1 provides more details on this implementation. The following FastClick pipeline implements the *SetCRC32* VNF. The `Strip` element removes the just appended bytes to avoid returning more bytes back to the generator than were sent.

```
FromDPDKDevice(0) -> SetCRC32 -> Strip(4) -> ToDPDDevice(0)
```

We decided to implement only the GPU equivalent of the `SetCRC32` element, namely `GPUSetCRC32CommList` and `GPUSetCRC32Coalescent`, because both VNFs have very similar workload. CUDA implementations are an exact port of the FastClick algorithm on GPU. The following pipeline is the adaptation with a GPU element, with packets batched up to n packets per batch.

```
FromDPDKDevice(0)
    -> MinBatch(n − 32)
    -> GPUSetCRC32CommList(MAX_BATCH n)
    -> ToDPDDevice(0)
```

For GPU versions, the CRC table is pre-computed on CPU when the element is initialized and copied to GPU memory allocated by `cudaMalloc` using a standard `cudaMemcpy`. The CUDA kernels then get access to the memory area where the table resides by passing it as an argument at kernel launch.

With this network function, the GPU version using coalescent ROIs can be expected to perform worse than the version using communication lists, as the entire packet is always read. Thus, the additional copy of data to CPU and explicit copies from CPU to GPU do not allow faster memory access on GPU than a zero-copy access directly from the DPDK memory pool memory that resides in GPU memory.

Regarding the GPU coalescent ROIs version, the GPU kernel must have access not only to all bytes of the frame, but also to the size of each packet and to a memory space to write back the computed values. Moreover, the stride needs to be of fixed size throughout the execution of the element, but the packet sizes can vary. Therefore, it is mandatory to specify the maximum packet size that the element can handle using `MAX_PKT_SIZE` argument in the Click configuration string. If the maximum size is s, then the stride for each packet has to be $s + 4 + 4$, because the CRC is 4 B long and the size is encoded using a `uint32_t` type. It also means that $s + 8$ bytes are copied for each received packet, regardless of the size of the received packet. Many bytes may thus be

copied unnecessarily, and if a packet longer than s is encountered, it will be dropped.

The memory layout allocated per queue in shown in Figure 3.3. First, all packet sizes in the batch are coalesced. Then, all payloads are coalesced. Finally, all computed CRC values are coalesced. This format allows only sizes and payloads to be copied from host to device memory, and then only CRC values to be copied from device to host memory.

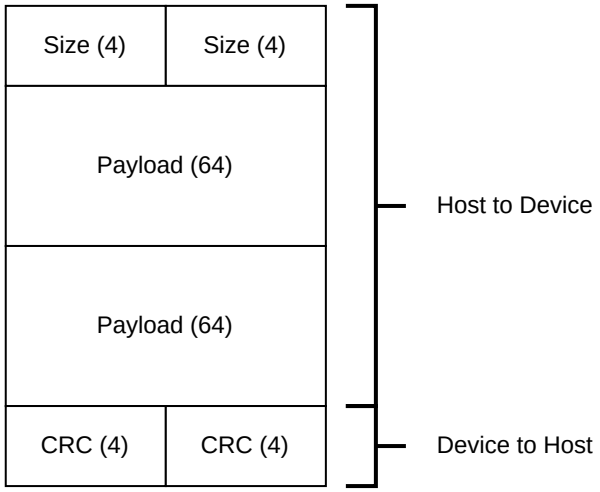


Figure 3.3: Coalescent memory layout used for a batch of 2 packets, with a maximum packet size of 64 B. Sizes in bytes are indicated in brackets.

Chapter 4

Evaluation

This section characterizes the three applications using CPU, DPU and all GPU implementations, and discusses the results. The performance of our models is mainly evaluated using two important metrics: throughput and latency, but we also consider GPU power consumption. We start by explaining the testbed and methodology used for the tests and measuring the baseline results of our servers. We then compare implementations for each type of application. Finally, we take a closer look at how factors in each implementation can affect the performance.

4.1 Testbed

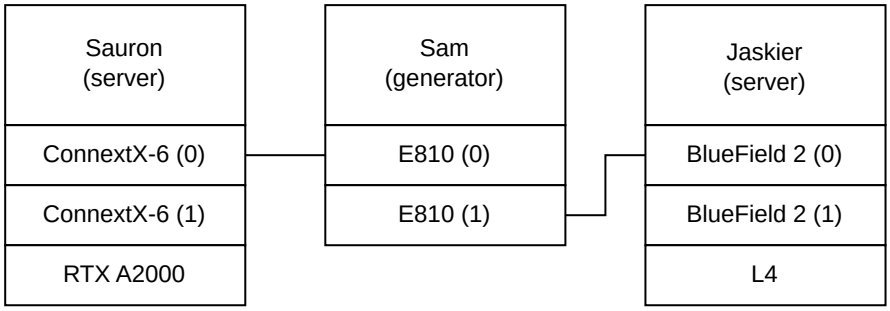


Figure 4.1: Experimental setup and connections between servers. Port numbers are between brackets.

Figure 4.1 shows the experimental setup used to run the experiments. 3 machines are used. All NICs and links support 100 Gbit/s. The generator used for packets is Pktgen-DPDK [97].

- Sam. The generator server is equipped with an Intel® Xeon® E-2378 octa-core Processor @ 2.60 GHz, with 32 GB of RAM, and a single NUMA node. The NIC is

an Intel® Ethernet Controller E810-CAM1/2 100GbE, connected to the machine on a PCIe 4.0 x16 slot. The server runs the Linux kernel 5.15.0-113-generic, and Pktgen 23.06.1 is used with DPDK 23.03.0 to generate packets.

- Sauron. This server is equipped with an AMD EPYC™ 9124 16-core Processor @ 3 GHz, with 256 GB of RAM. It has 2 NUMA nodes, each with 8 physical cores and 128 GB of RAM. A NVIDIA RTX A2000 GPU and a Mellanox Technologies MT2892 Family ConnectX-6 100GbE NIC are connected on the same NUMA node, both on a PCIe 4.0 x16 slot. The server runs the Linux kernel 5.15.0-113-generic. DPDK 23.11.0 is used along with CUDA 12.4 and the NVIDIA driver 550.54.15. The SmartNIC Mellanox firmware version is 22.39.3004, the Mellanox OFED driver version is 23.10-2.1.3, and DOCA version is 2.5.1007.

- Jaskier. This server is equipped with an Intel® Core™ i9-12900KS 16-cores Processor, with 32 GB of RAM, and a single NUMA node. Only the 8 performance cores @ 3.40 GHz are used. The NIC is a Mellanox Technologies MT42822 BlueField-2 integrated ConnectX-6 100GbE, and it has an NVIDIA L4 Tensor Core GPU, both devices being connected through a PCIe 4.0 x8 slot. It runs the Linux kernel 6.5.0-44-generic. DPDK 23.11.0 is used along with CUDA 12.5 and the NVIDIA driver 555.42.02. The SmartNIC Mellanox firmware version is 24.41.1000, the Mellanox OFED driver version is 24.01-0.3.3, and DOCA version is 2.6.0058-1.

For system profiling, NVIDIA Nsight Systems [72] 2024.3.1.75 is used, while for GPU kernel profiling, NVIDIA Nsight Compute [71] 2024.1.1.0 is used.

Since currently GPUdev does not support the L4 GPU, we made a fix to DPDK. This fix is available on GitHub[1]. This is the DPDK version used to compile FastClick-GPU.

On both servers, FastClick-GPU is checked-out from commit `c6c4b2d`[2], configured and then compiled as described in the README file.

DOCA application is checked-out from commit `5570d6f`[3] and compiled as described in the README.

On the SmartNIC, FastClick-DPU is checked-out from commit `7ec3d8f`[4], also configured and compiled accordingly to the README file.

4.2 Methodology

For each application, measures have been performed to compare existing and new implementations, on both CPU, and DPU, and multiple implementations using GPU.

[1] https://github.com/maxvanliefde/dpdk/commit/13e12d243dfd13e8adba11e58aa13c630fdeeb27

[2] https://github.com/maxvanliefde/fastclick-gpu/commit/c6c4b2d725d2b6f45ac5d2599feb201a6fceb34a

[3] https://github.com/OnyxDurga/doca-app/commit/5570d6f67daaf6b497a79fd8d47943c658675701

[4] https://github.com/OnyxDurga/fastclick_dpu/commit/7ec3d8fbd48ef7a68d312957d837d8b5c81560ae

An experiment consists in measuring the response variables (cf. Section 4.2.2) for a range of different factors (cf. Section 4.2.1 during 10 seconds. The application is first initialized on a server, waiting for requests to process them. The generator is then launched. During 10 seconds, it sends requests to the server and waits for the answer. Response variables are measured on the generator server by Pktgen. Each experiment is run three times, which allows for analysis of variability across tests. NPF [10] is used to perform the tests and output the majority of the graphs.

All our experiments are reproducible using the scripts available on GitHub at `https://github.com/OnyxDurga/gpu-npf`. The repository also contains all the results and graphs presented in this section.

4.2.1 Factors

Multiple factors are considered, depending on the implementation and application being tested. They have been presented in Chapters 2 and 3, and are summarized in Table 4.1.

Table 4.1: Summary of factors based on implementation. CL and ROI refers to the GPU implementation based on Communication Lists and on Coalescent ROIs, respectively.

Factor	CPU	DPU	CL	ROI	DOCA
Packet Size	Y	Y	Y	Y	Y
Number of Cores	Y	Y	Y	Y	
GPU Queues					Y
GPU Batching Size			Y	Y	
Number of Shared Buffers per Core			Y	Y	
Zero-Copy				Y	
Memory Pool Location			Y		
Persistent Kernel Use			Y		
Lookup Table Size (IP Lookup)	Y	Y	Y	Y	Y

Packet size is a common factor for each implementation, and the number of cores used to receive packets is only absent from the DOCA implementation, as processing happens entirely on GPU. DOCA performance rather depends on the number of GPU queues used to receive packets.

The performance of the two last GPU implementations also depends on the size of the GPU batching performed and the number of shared buffers (either communication lists or queues) used per CPU core to communicate with the GPU.

Furthermore, the GPU implementation based on communication lists will depend on the memory pool location (either on CPU or GPU memory) and the use or not of persistent kernels. The implementation based on coalescent ROIs will be affected by the use or not of zero-copy.

Finally, workloads may add others factors. In our analysis, only the IP lookup VNF includes one, the size of the lookup table, which will affect all implementations.

4.2.2 Response Variables

Throughput

The *throughput* is defined as the rate at which the application answers the requests of the generator. It can be expressed in bits per second (bit/s or bps) or in packets per second (p/s or pps). If the packets are of fixed size, the rate in bits per second is given by the rate in packets per second multiplied by the size of the packets in bits.

Pktgen provides us the number of packets received during the time of the test, n_{rx}. Throughput in pps is given by the total number of packets received by the generator during the test, divided by the duration of the test d in seconds:

$$\text{Throughput [pps]} = \frac{n_{rx}}{d}$$

Pktgen also provides us with the total amount of bytes received during the test, B_{rx}. This is the number of bytes received in the L2 level where Pktgen resides, but for each packet additional bytes are transmitted on the network, i.e. in the L1 Physical Layer. At the start of the frame, it consists of 7 B of preambule and a Start Frame Delimiter of 1 B. At the end of the frame, the CRC of 4 B is also checked and removed by the NIC in L1 layer. Finally, 12 B of interpacket gap (not part of the frame) are transmitted after each frame. The total overhead is thus of 24 B per frame received [38]. Throughput in bps is computed taking into account this overhead, that is:

$$\text{Throughput [bps]} = \frac{8\,\text{bit}}{1\,\text{B}} \cdot \frac{n_{rx} \cdot 24\,\text{B} + B_{rx}}{d}$$

To measure the throughput of an application, Pktgen is set to send packets at its maximum possible rate. If the reception rate is the same as the transmission rate, it means that the NIC saturates. In such cases, the application can process requests as fast as they arrive, without dropping any.

Loss Rate

The *loss rate* is the percentage of packets lost during the time of the test. Pktgen provides the number of packets received n_{rx} and transmitted n_{tx}, which easily allows to compute the loss rate:

$$\text{Loss rate} = 1 - \frac{n_{rx}}{n_{tx}}$$

If the loss rate is null, then the application answers all requests and the NIC is saturated, which is highly beneficial as it usually means there is still room to further process packets in real time.

Average Latency and Zero-Loss Throughput Latency

The *average latency* is the average time from sending a packet to receiving the answer, expressed in seconds. It is processed by Pktgen, by adding a timestamp in the packet and checking the elapsed time when it is received back. As it is measured on the generator server, it includes both the average Round-Trip Time (RTT) of packets during the experiment, i.e. the time to send back and forth to the server, and the average time to respond the request on the application server.

To get reliable latency results, the loss rate has to be zero. Otherwise, the time spent to waiting in the server queue would be incorrectly reported. There are two ways to achieve this. The first is to simply set the rate of the generator to a very low value, such as 0.1 % = 100 Mbit/s, making sure that every application is able to respond to such a rate. With this solution, GPU batching dramatically increases latency. In fact, GPU implementations wait for many packets to arrive, say 1024, before sending them to the GPU. Since only a few packets arrive per second, the time it takes to get 1024 packets is large and causes high latency for the first packets in the batch. For example, at 100 Mbit/s, a new packet of 64 B arrives every 6.72 µs, so a batch of 1024 packets will take 6.881 ms before even starting to process the packets. For this reason, when we set the generator rate to 100 Mbit/s, we make sure to disable additional batching on the GPU side, leaving only the I/O batching, where packets are emitted from DPDK in batches of 32 packets.

To account for the batching of GPU implementations and get reliable latency results, it is necessary to have a higher rate. For each test, we set the rate to the highest throughput that results in zero loss. This is done automatically by NPF using the zero-loss throughput research functionality[5]. The latency at such a rate is called *Zero-Loss Throughput (ZLT) latency.*

GPU Power Consumption

The *GPU power consumption* refers to the amount of electrical energy consumed per second to run the GPU. It is measured in watts (W). It is reported by the NVIDIA System Management Interface (`nvidia-smi`) tool. The documentation claims that the reading of the last measured power draw for the entire board is "accurate to within +/- 5 watts" [75].

The measurement is only done for GPU implementations, as the CPU and DPU could work without GPU connected to the system. It consists of the average of 10 measured power readings of the board while the application is running, separated by one second.

The total board power of the RTX A2000 GPU is 70 W [73], and without any workload it consumes around 20 W, as informed by `nvidia-smi`.

[5]At the time of writing, it is available in preview through the commit `https://github.com/tbarbette/npf/commit/dfaa8b9eedf8c5b85352978a500ed9e96c07ba63`.

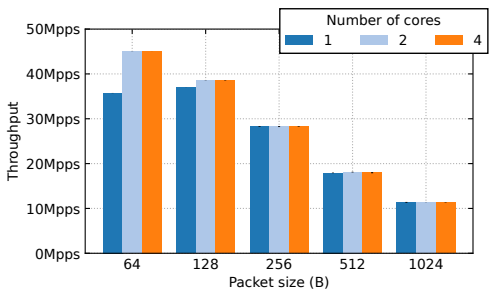
4.3 Baseline

As baseline results, we ran a CPU-only application that does nothing but forward immediately the packets it receives, using 1 to 4 CPU cores to receive packets, and for packet sizes from 64 B to 1024 B. Since adding a VNF to the pipeline increases the workload, no other result could be better than these. Results for throughput and loss rate are compiled in Figure 4.2. As latency computation in Pktgen requires an Ethernet mirror applied to the packets to work, no latency analysis is done here.

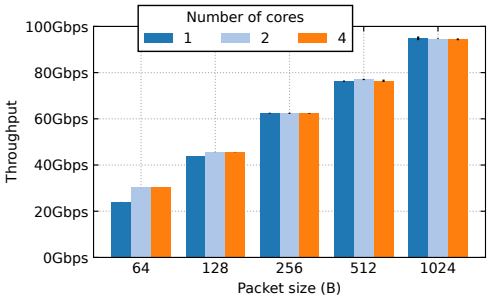
The configuration used is the following, with a variable number of cores *cores*. Each configuration used for testing contains at least these two elements.

```
FromDPDKDevice(0, NDESC 2048, MAXTHREADS cores)
    -> ToDPDKDevice(0, NDESC 2048, BLOCKING false)
```

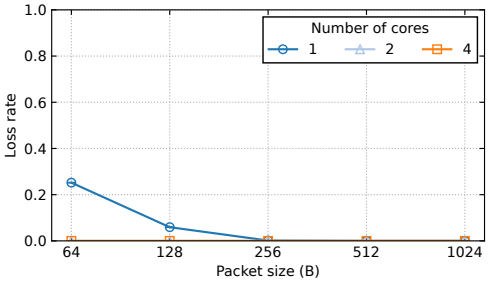
In blocking mode, `ToDPDKDevice` drops packets when the output ring is full. As in all experiments, the rate is fixed, full rings mean the server cannot handle the flow, but since it will not decrease, it is useless to wait for the rate to drop.



(a) Throughput in Mpps.



(b) Throughput in Gbps.



(c) Loss rate.

Figure 4.2: Baseline CPU results. Received packets are simply retransmitted to the generator.

For all packet sizes, all packets are retransmitted with at least 2 cores, as we can see

from the null loss rate in Figure 4.2c. Increasing the number of cores does not change this result, as RSS distributes packets more evenly across the cores. However, with a single core, only flows of packets with sizes of at least 256 B are fully processed. Looking at Figure 4.2a, we deduce that the maximum number of packets processed on a single core is in between 30 Mp/s and 40 Mp/s. We also deduce the two limits of the generator server from Figures 4.2a and 4.2b. It cannot generate more than 45 Mp/s, as we see with packets of 64 B. For larger sizes, the limit is rather the number of bits per second: it approaches 100 Gbit/s with packets of 1024 B, as the NICs allows.

In future explorations, we will only consider throughput in Gbps, as each experiment is performed with a fixed packet size.

4.4 Comparison of Implementations

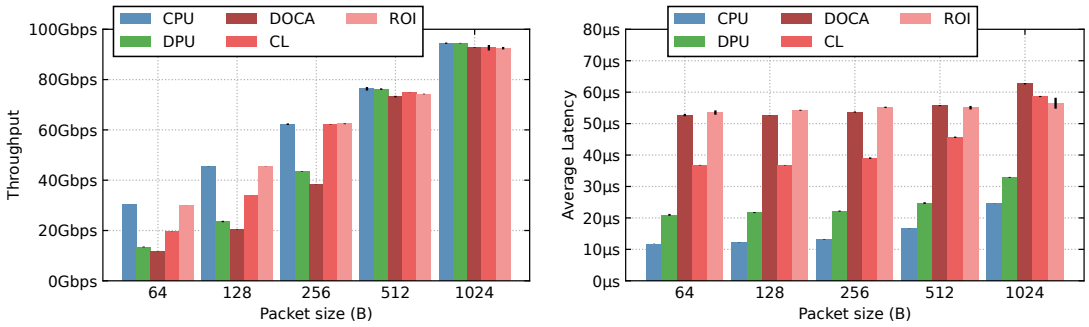
This section discusses relative performance of each implementation for three workloads. In all discussions and legends, CL and ROI designates the GPU implementations based on communication lists and coalescent ROIs, respectively.

Factors have been set to compare implementations. 4 CPU cores are used to receive packets. GPU-elements implementations feature one shared buffer of 256 elements per core. They are batched with 1024 packets to measure the throughput and ZLT latency, and 32 packets to measure the fixed-rate latency. Elements are multithreaded following the full-path strategy. The ROI implementation does not use zero-copy, and the CL implementation uses persistent kernels and memory pool on the GPU. DOCA implementation uses 8 receive queues, and batches packets by 2048 to measure throughput, and 32 to measure latency. The IP lookup application is done on a routing table of 100 elements.

Unless explicitly stated otherwise, these factors are set to the same values in the following sections.

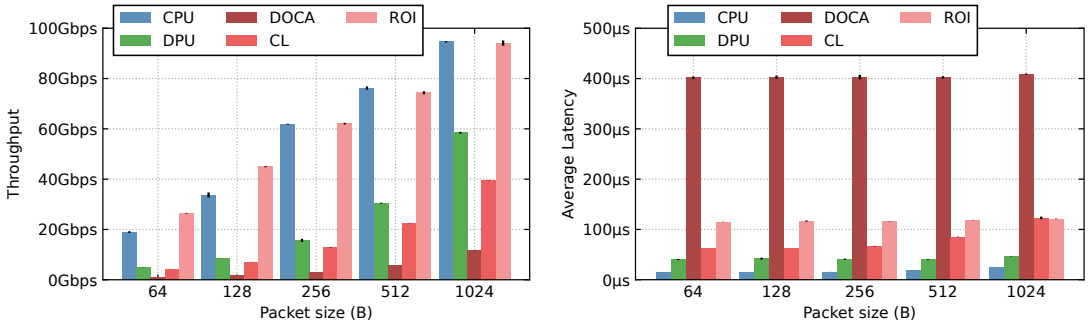
Ethernet Mirroring Ethernet Mirroring peak throughput and fixed-rate latency for each implementation are shown in Figure 4.3. The standard deviation is virtually nonexistent because the workload is deterministic and very fast to execute. The results for the GPU are disappointing. Indeed, for both the throughput and the latency, the CPU implementation has the best results for all packet sizes.

Recall that for this application, only 12 B are read and written per packet, independently of packet size. As the packet size increases, we receive fewer packets, but for a higher total bandwidth. This means that the actual number of bytes read/written per second decreases, making computation easier. For all implementations, we see that throughput increases logarithmically as packet size increases. On the other hand, at fixed rate, latency stays more or less the same and only increases slightly when packet size becomes very large, since larger packets do not change the time it takes to process a packet.



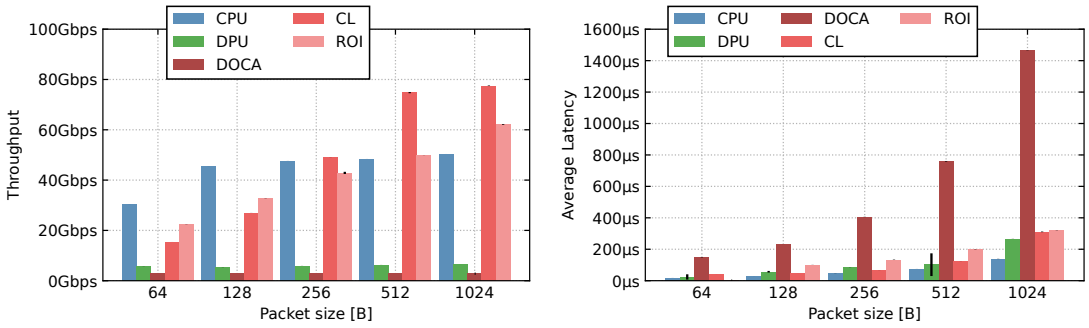
(a) Ethernet Mirror Throughput. (b) Ethernet Mirror fixed-rate latency.

Figure 4.3: Ethernet Mirror performance depending on implementation.



(a) IP Lookup Throughput. (b) IP Lookup fixed-rate latency.

Figure 4.4: IP Lookup performance depending on implementation.



(a) CRC Computation Throughput. (b) CRC Computation fixed-rate latency.

Figure 4.5: CRC Computation performance depending on implementation.

For large packet sizes, all implementations are able to saturate the 100 Gbit/s link. For smaller packet sizes, only the ROI implementation is able to maintain the same throughput as the CPU. As predicted, the CL implementation performs worse because the coalescent version is favored by the fact that the ROI is less than 32 B per packet. Finally, DOCA provides about the same throughput as the DPU implementation.

This can be explained by the fact that the workload is very lightweight and very fast to execute on a CPU. The potential gains of performance offered by GPU parallelism and memory bandwidth are not useful for such a light workload, since only a few bytes are read and written per packet.

Latency results are close between CPU and DPU versions. The CPU has a latency of the order of 10 µs with small packets. The SmartNIC performs worse, around 20 µs, which is not surprising, as the cores it contains are much less powerful than those of the Sauron server. GPU elements introduce more latency due to GPU transactions overhead and queuing in the shared buffer, but the range remains reasonable. The ROI version has a bit more latency (around 50 µs) because it does more copying of data before sending it to the GPU: it copies the ROIs in a coalescent memory region, explicitly copies it to the GPU, and then does the same in the opposite order when the processing is finished. On the other hand, the CL version allows the GPU to access the packet data that resides in the GPU's memory, providing a lower latency (around 35 µs). Unfortunately, DOCA offers about the same latency as the ROI implementation. It is surprising, because the data path is greatly reduced, as packets do not pass through the CPU. It is however difficult to fully analyze this result as many DOCA functions are closed-source. This result could be improved by adding more receive queues.

IP Lookup IP Lookup peak throughput and fixed-rate average latency for each implementation are shown in Figure 4.4. It uses a small lookup table of 100 entries. These measures were made using small lookup tables as the `LinearIPLookup` algorithm struggled with large table sizes, though a discussion about varying table sizes can be found in Section 4.11.

As reminder, this workload requires 4 B per packets to be checked linearly against all elements of a routing table. Like with Ethernet Mirroring, the throughput increases with the packet size. As the packets get larger, the amount of packets received per second decreases, and this means fewer lookups to perform as well as less data read/written to the device for the GPU implementations. The throughput is increasing logarithmically for the CPU and ROI implementations, while it increases more linearly for the CL and the DPU implementations. It is hard to determine the tendency for the DOCA implementation as its throughput is extremely low.

Both the CPU and ROI implementations follow the same trend as with Ethernet Mirroring, showing a logarithmic increase. For the implementation with the communication list, the fact that the whole packet has to be copied to the GPU seems to be dragging down the throughput. That throughput increases more with the packets getting larger, resulting in a more linear progression. The same tendency can be observed with the

DPU, but it is slow simply due to its weaker cores seemingly struggling with the lookups. Finally, DOCA performs much worse than all the other implementations as soon as there is more computing than for Ethernet Mirroring. We unfortunately have not been able to figure out why.

The latency measures for IP Lookup follow the ones with Ethernet Mirroring closely. The only exception is the latency of the DOCA implementation which is about 4 times worse than the other implementations, reaching 400 µs. The other implementations have a bit more latency than for Ethernet Mirroring, reaching 100 µs for the ROI implementation. The main observations are similar though, with the ROI implementation showing the most latency, followed by the CL one, the DPU and the CPU having the lowest latency. For large packets, the ROI and the CL implementations are almost identical.

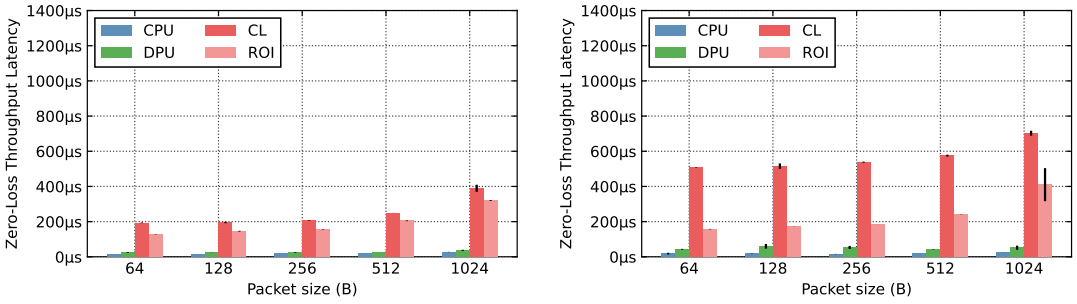
CRC Computation Finally, Figure 4.5 compares CRC computation peak throughput and fixed-rate average latency for each implementation. Unfortunately, for unknown reason, using packets of 64 B makes it impossible for Pktgen to measure the latency using the GPU coalescent ROIs version, which is missing from the graph.

For this application, the workload depends on the packet size: the larger the packets, the more reads and computations there are, as the CRC is computed byte per byte. As a result, for CPU and DPU implementations, increasing packet size does not significantly change throughput, as fewer packets are received per core, but each packet takes more time to process. For GPU implementations however, the throughput still increases with packet size. It is because of the offload of the computation to the GPU. Indeed, the CPU cores are not doing any computation, they just communicate with the GPU, and with fewer packets coming in, they have less work to do. The GPU, however, as explained in Section 4.6, is able to keep up with the computation, increasing throughput for larger packets. Inevitably, for all implementations, average latency at fixed rate increases, as time taken to process a single packet increases.

The performance of the DPU is poor, again because ARM cores are much less powerful than the cores on the server and than the GPU for this heavy workload. It is also noticeable that the DOCA implementation is completely unable to compete with other implementations. As for the IP lookup application, we were not able to find the bottleneck with this implementation.

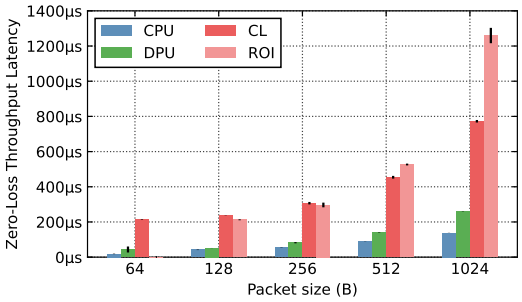
Regarding other GPU implementations, the CL implementation performs slightly worse than ROI with small packets, but much better with larger packets. Since the entire packet must be read, copying the entire packet to coalescent memory has no advantage, as the payload of the packet is already coalesced in DPDK memory pool and already reside in GPU memory. For smaller packets, ROI does fewer copies, while CL struggles because of the batching size (cf. Section 4.7).

For a fixed throughput, average latency across implementations is similar to other applications. The CPU and the DPU offer the best latency results, then comes the CL due to its slower data path, followed by ROI due to additional memory copies, and finally DOCA with extremely poor results.



(a) Ethernet Mirror.

(b) IP Lookup.



(c) CRC Computation.

Figure 4.6: Zero-loss throughput average latencies of each implementation, excluding DOCA. Batching to GPU is enabled to 1024 packets.

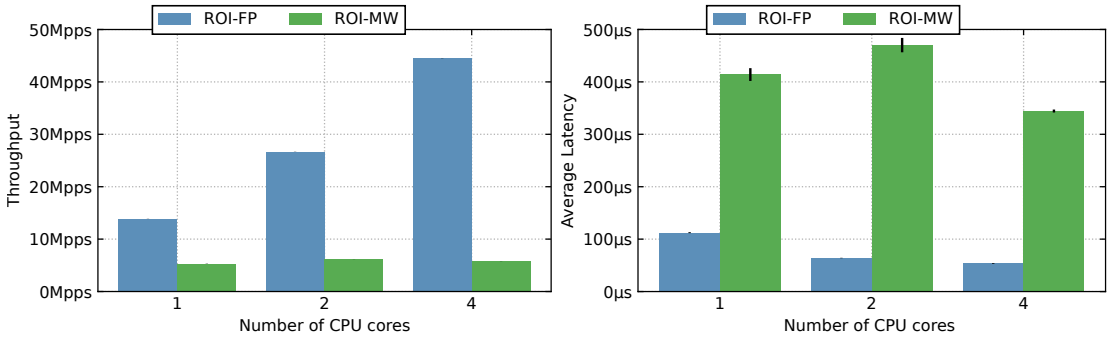
Zero-Loss Throughput Latency Finally, we can take a look at zero-loss throughput latency. It uses batching for GPU implementations, as it is done to measure the throughput. It gives a better idea of what latency would be at high speeds. Figure 4.6 shows these latencies, measured for each test for the best throughput shown in previous figures. For readability, DOCA implementation is not included in the figures due to its very high latency.

The results follow the same trend as the average latency at fixed rate and with batching disabled. We still see an increase in latency for GPU implementations. For lightweight applications, results remain in an acceptable range. Interestingly, the ROI implementation now outperforms the CL implementation for Ethernet Mirroring and IP Lookup. Because it provides more throughput, more packets are received per second, and the latency caused by GPU batching is reduced, resulting in lowere ZLT latency. It is a real time saver to merge ROIs before sending them to GPU, as more threads in a warp are served with a single read/write request to data.

For heavy computations, ZLT latency becomes very high. While offering the best throughput, the CL implementation introduces a latency of around 800 µs to compute the CRC for 1024 B packets, which can be problematic for low-latency targeted applications. Reducing the batch size can help reduce latency (cf. Section 4.7).

4.5 Element Multithreading Strategy

The performance of the GPU Coalescent ROIs element on Ethernet Mirroring for the two available multithreading strategies are shown in Figure 4.7. Recall that in the Master-Worker strategy, explored in previous works, increasing the number of cores does not increase the utilization of the GPU, while the full path strategy, following FastClick best practices, scales the GPU utilization with the number of CPU cores receiving packets.



(a) Throughput.

(b) Fixed-Rate Average Latency.

Figure 4.7: Ethernet Mirror performance based on the multithreading strategy, with variable number of CPU cores. Packets are of size 64 B, and the coalescent ROIs implementation is used. FP means Full-Path and MW Master-Workers.

We notice the extremely poor performance of the Master-Workers threading strategy.

Indeed, it saturates around 5 Gbit/s, regardless of the number of cores receiving packets. It is clear that this threading strategy is no longer viable at today's network speeds, since a single core can already handle 70 % of the traffic (cf. Section 4.3). As only one CPU thread handles all communication with the GPU, all traffic is effectively redirected to a single one core, which struggles to process it all. On the other hand, the full-path strategy scales well with the number of cores as GPU utilization increases. This strategy is better suited to high-speeds networking because it allows better throughput.

The Master-Workers strategy also performs worse in terms of latency. This was expected, as while the latency of the full-path strategy comes from shared buffers, the latency of the Master-Workers strategy comes both from both the shared buffer and the switching between worker and master threads. When the number of CPU cores increases, we see that latency decreases for full-path multithreading, because more GPU computational power is used, but remains relatively the same for the Master-Worker strategy, because the GPU is not used more.

The master-worker strategy performs worse here than what was found in previous works such as PacketShader [37] and APUNet [33]. The FastClick configuration relies on `Pipeliner` elements to switch packets between cores (cf. Section 2.2). Such elements may induce overhead compared to what was done in PacketShader, as they were not designed with this kind of use in mind. However, the fixed-rate latency is in the range of what was found with PacketShader, as more time is given to process each packet. APUNet significantly reduces latency thanks to using iGPU capabilities, which is not done in this work.

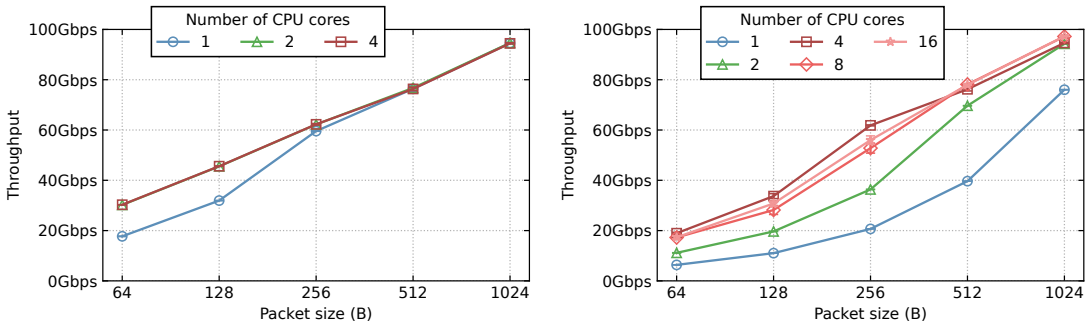
Given the poor performance of the Master-Workers strategy for the simplest application, we stick to the full-path multithreading strategy in the following test cases.

4.6 Load Balancing

For the CPU implementation, increasing the number of cores leads to a better load balancing between cores, but each core also has to run the VNF on packets. Obviously, increasing the number of cores will help with the throughput by giving each core more time to process each packet.

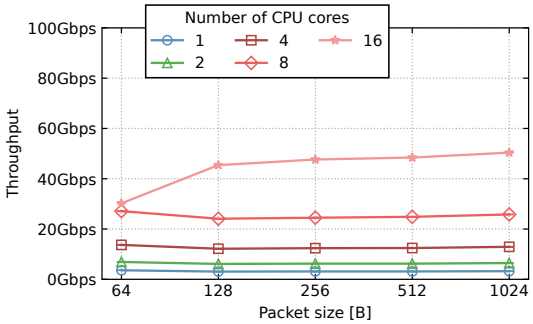
For GPU elements implementation, CPU cores do not perform the workload, but need to handle communication with the GPU. Thus, increasing the number of cores is also a good way of reducing losses – as long as a full-path threading strategy is followed. If the bottleneck is not the reception of the packets, but the computation on the GPU, another factor can help: increasing the number of shared buffers per CPU core. Recall that 2 cores can receive all the traffic for packets as small as 64 B. The load can be balanced across communication buffers instead of CPU cores, reducing the number of CPU cores needed.

The throughput performance of the CPU implementation as the number of CPU



(a) Ethernet Mirror.

(b) IP Lookup.



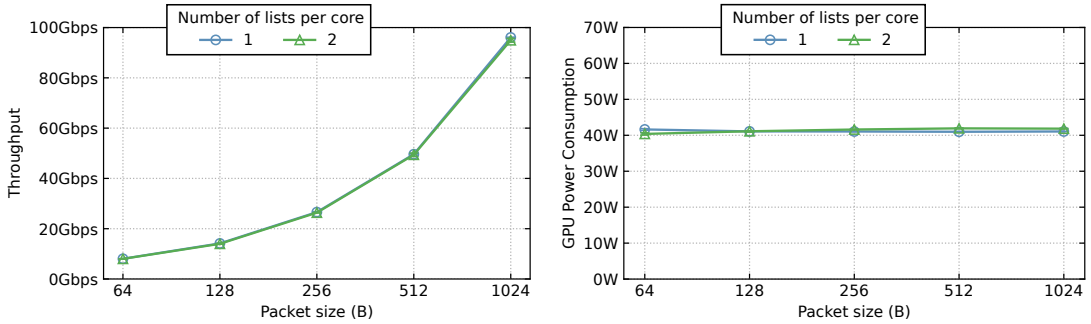
(c) CRC Computation.

Figure 4.8: Throughput of the CPU implementation with variable number of CPU cores.

cores increases is shown in Figure 4.8. The number of cores required to saturate the link depends on the evaluated application. 2 CPU cores are sufficient to handle Ethernet Mirroring on all packet sizes. 4 cores are needed to achieve this with an IP Lookup, a heavier workload. For CRC computation, whose complexity is a function of the size of the packets, 16 CPU cores saturate the link only for packets of 64 B. Recall that CPU cores handle both receiving and sending packets, as well as performing the workload: if it is heavy, there are less CPU cycles available to receive packets, leading to greater losses that are mitigated by spreading the load across more CPU cores.

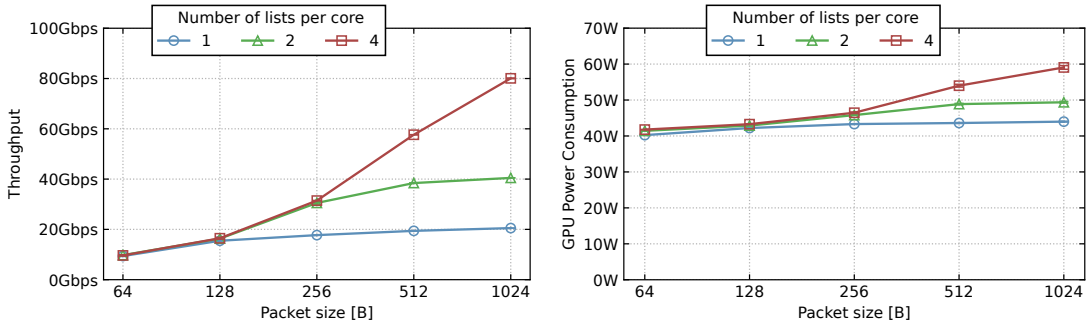
Throughput performance as well as GPU power consumption of the CL implementation are shown in Figure 4.9, with only one CPU core receiving packets, and increasing number of communication lists used to communicate with the GPU. IP Lookup is omitted, because its results follow the exact same trend as Ethernet Mirroring.

For Ethernet Mirroring, we see that increasing the number of communication queues does not increase the throughput. This implies that the bottleneck is not the GPU, since increasing the number of queues increases GPU usage, but rather the link between the CPU and the GPU. Spreading the load across multiple queues has no effect, so it means that the CPU cannot keep up with the speed at which the GPU is processing packets,



(a) Ethernet Mirroring Throughput.

(b) Ethernet Mirroring GPU Consumption.



(c) CRC Computation Throughput.

(d) CRC Computation GPU Consumption.

Figure 4.9: Performance of the CL implementation with one CPU core and variable number of lists.

and that they remain in the communication list the same amount of time. Since the GPU is not processing more packets, its power consumption stays the same. In this case, increasing the number of cores would be beneficial because the as load would be spread across cores, reducing the load on the CPU. Recall from Figure 4.3a that with 4 cores and only 1 communication queue, throughput was around 2 times higher.

For CRC computation, however, the results vary depending on packet size. For small packets, the same thing happens: too many packets arrive at the CPU core, the bottleneck is the CPU-GPU communication. But, for larger packets, increasing the number of queues turns out to be extremely beneficial. In this case, with few communication lists, the GPU is the bottleneck: packets are added to the list faster than the GPU can process them, because it iterates through the list sequentially. Spreading packets across multiple lists that are processed simultaneously increases GPU utilization, allowing it to process more packets per second. Ii is clear from the GPU power consumption graph that its workload increases, as its consumption increases by around 5 to 10 W per list added. Such result is encouraging, as we achieve 80 Gbit/s for heavy workload using a single CPU core, for a 60 W additional GPU consumption, while a 16-core CPU can only process 50 Gbit/s for the same workload.

We can very roughly estimate the number of 1024 B packets processed per watt consumed. The Thermal Design Point (TDP) of the Sauron CPU is 200 W [1]. This TDP indicates the amount of power the CPU consumes when processing a worst-case application [34]. Since CRC computation is heavy, we consider the CPU consumption to be the TDP. In practice, the effective power consumption could be higher or lower, e.g. if CPU boosting is enabled. The CPU implementation processes 50 Gbit/s which is about 6 Mp/s. The energy efficiency of this implementation is:

$$T_{\text{CPU}} = \frac{6\,\text{Mp/s}}{200\,\text{W}} \approx 30\,\text{kp/J}$$

Considering that power consumption is linear in the number of cores, a CPU core at full load consumes about $200/16 = 12.5\,\text{W}$. Again, this is a rough estimate: the effective power consumption of a single core would certainly be a bit higher. For the GPU implementation, the CPU core is always polling to receive packets, so we consider the same amount of power. The GPU implementation processes 80 Gbit/s which is about 10 Mp/s. The energy efficiency of this implementation is:

$$T_{\text{GPU-CL}} = \frac{10\,\text{Mp/s}}{12.5 + 60\,\text{W}} \approx 138\,\text{kp/J}$$

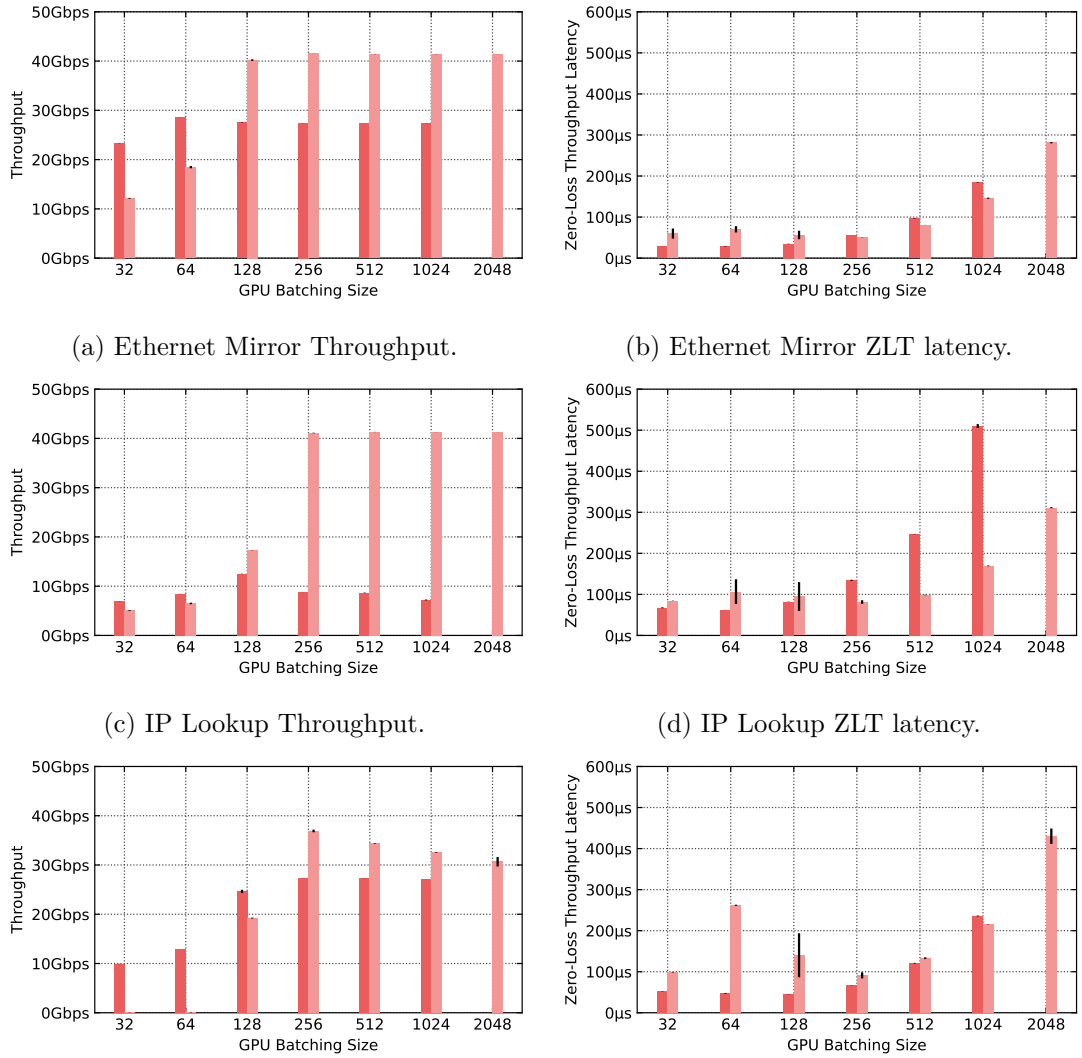
Under these assumptions, the GPU implementation is more than 4 times more energy efficient to process CRCs on 1024 B packets.

In summary, while the CPU can only spread the load across its cores to balance it, the GPU implementations can also use multiple communication queues that are processed in parallel by the GPU. By using few cores to receive packets, but multiple queues to control the GPU, we can achieve much better results for heavy workloads while consuming less power, leading to a more energy-efficient way to process compute-intensive network functions.

4.7 Batching

All implementations use I/O batching, with DPDK retrieving packets from the NIC in bursts of up to 32 packets. GPU implementations further batches packets before sending them to the GPU. It improves throughput performance, by making better use of GPU parallelism and hiding the GPU data transfer delay. But at the same time, it increases average latency, as computation can start only once a batch is full: the RTT of the first packet in the batch includes the time taken just to receive as many packets as the batch needs. Finding the optimal batching size is thus a difficult trade-off, and depends on multiple factors as packets size, network bandwidth and GPU processing power.

Peak throughput and ZLT latency with variable batching sizes are shown in Figure 4.10. GPUdev communication list API only supports batches of size up to 1024 packets, as it is the maximum number of GPU threads per GPU block, and one block performs



(a) Ethernet Mirror Throughput.

(b) Ethernet Mirror ZLT latency.

(c) IP Lookup Throughput.

(d) IP Lookup ZLT latency.

(e) CRC Computation Throughput.

(f) CRC Computation ZLT latency.

Figure 4.10: Performance of the CL and ROI implementations with variable GPU batching size. Packets size is 128 B, 4 CPU cores retrieve packets, and there is 1 shared buffer per core. CL uses persistent kernels and GPU memory pool, and ROI uses explicit-copies.

workload on each batch. For readability, DOCA results are on separate graphs that can be found in Figure 4.11.

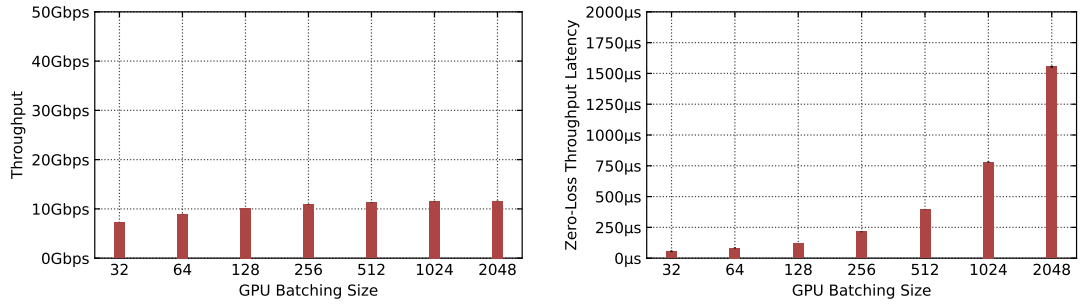
Finding the best batching size is challenging. The CL implementation provides the best throughput for relatively small batching sizes for Ethernet Mirroring and IP Lookup (resp. 64 and 128). For CRC computation however, the best result are obtained with batching sizes from 256 to 1024 packets. For lightweight applications, small batches perform better: the parallelism offered by the GPU does not improve the computation to the point where waiting for larger batches would be useful, as the computation is very fast. Simply waiting for a larger batch takes more time than the time saved in the computation. At the same time, increasing the batching size always increases the latency with this implementation, because we have to wait for a larger batch to be filled before starting the computation. For Ethernet Mirroring (resp. IP Lookup), using batches of 64 packets (resp. 128) is optimal, as it leads to the best throughput while ensuring a low ZLT latency of $\sim 30\,\mu s$, which is only 2 times longer than what the CPU computation offers (resp. $\sim 80\,\mu s$, 4 times slower than CPU). For CRC computation, a batching size of 256 packets allows for the best throughput for the implementation, while providing a ZLT latency of $\sim 65\,\mu s$ (only 1.5 times slower than CPU implementation).

Regarding the ROI implementation, results are more volatile. All implementations perform poorly with batching sizes of 32 to 64 packets. With such batching sizes, the CRC computation does not even reach a throughput of 1 Gbit/s. They also have slightly higher latencies than larger batching sizes. The lighter the application, the smaller the batch size: Ethernet Mirroring throughput saturates at 128 packets, IP Lookup at 256, while CRC application performs best with 256 packets per batch. Here, the same batching sizes always provide the best latency results for the implementation.

In conclusion, the batching size affects both latency and throughput, and choosing the best will depend on both the application and the objective. It seems difficult to provide a static best batching size, as if the traffic rate decreases, the latency automatically increases with high batching sizes, but so does the throughput. It is up to the user to choose the best batching size depending on the application and the specifications required.

DOCA Peak throughput and ZLT latency of the Ethernet Mirroring application with variable batching size for the DOCA implementation are shown in Figure 4.11. The measures were not made for the IP lookup and CRC workload as the performance of DOCA was too low to evaluate batching (cf Figure 4.4 and 4.5).

Batching barely impacts the throughput, going from about 8 Gbit/s with a batching of 32 packets to about 11 Gbit/s with a batching of 2048. These results are odd because the DOCA API behaved peculiarly. Indeed, if the timeout to receive the batch was set to anything other than 0, the maximum number of packets requested for batching would simply not be met. The timeout seems to have a higher priority than the batching size, even if the timeout occurred after receiving more packets than the requested amount. The measures were thus done with a timeout of 0 s.



(a) Ethernet Mirror Throughput.

(b) Ethernet Mirror ZLT latency.

Figure 4.11: DOCA implementation Ethernet Mirroring performance with variable GPU batching size. Packets size is 128 B, and 8 GPU queues are used.

Regarding the latency, it grows extremely fast with the batching size. The increase is expected as the larger the batching size is, the more we have to wait for a packet, and thus the higher the latency is. But the speed at which it grows is very surprising, especially compared to the other GPU implementations. The latency is at a reasonable 100 µs with a batching size of 32 packets and grows exponentially to an extremely high 1.5 ms with a batching size of 2048 packets. These results are also very puzzling as they are the zero-loss throughput latency, meaning the latency observed with a rate of packets low enough to not produce any loss. As the throughput seems to increase slightly with the batching size, the latency should not increase. Unfortunately, this is not the case, but as before, the API behaved strangely.

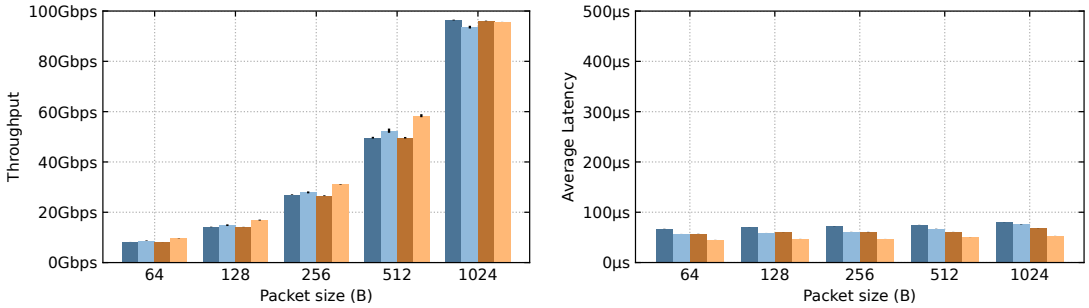
4.8 Persistent Kernels and Memory Pool Location in Communication List implementation

Recall the CL implementation has two exclusive features: the location of the DPDK memory pool, either in CPU or in GPU memory, and the control of the GPU control flow, either using kernel launches for each batch or a persistent kernel per communication list.

Peak throughput and average fixed-rate latency achieved for each application are shown in Figure 4.12. Other factors set as follows: 1 CPU core is used to receive packets, 1 communication list to communicate with the GPU, and GPU batches packets by 1024 for throughput measurements and by 32 for latency measurements. Figure 4.13 shows the GPU power consumption for each application with the same factors.

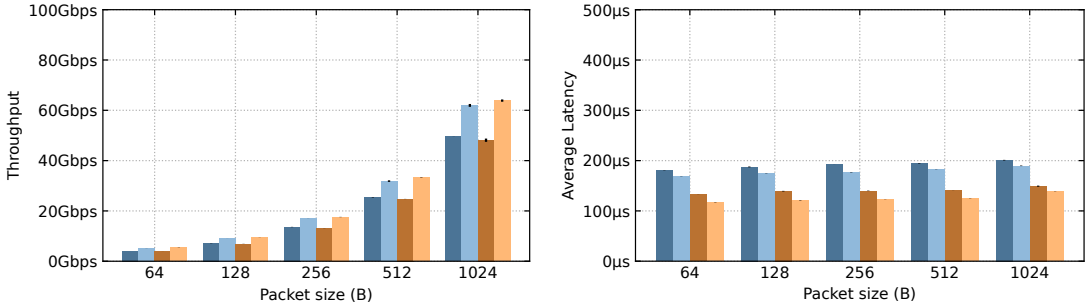
We observe that these two factors are related and affect performance based on the application being evaluated.

Using a GPU memory pool always increases throughput and reduces latency compared to a CPU memory pool. Data is written by the NIC directly in GPU memory. GPU



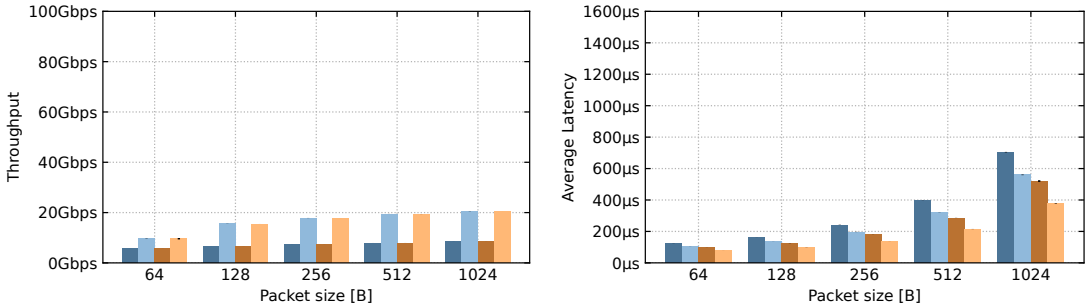
(a) Ethernet Mirroring Throughput.

(b) Ethernet Mirroring fixed-rate latency.



(c) IP Lookup Throughput.

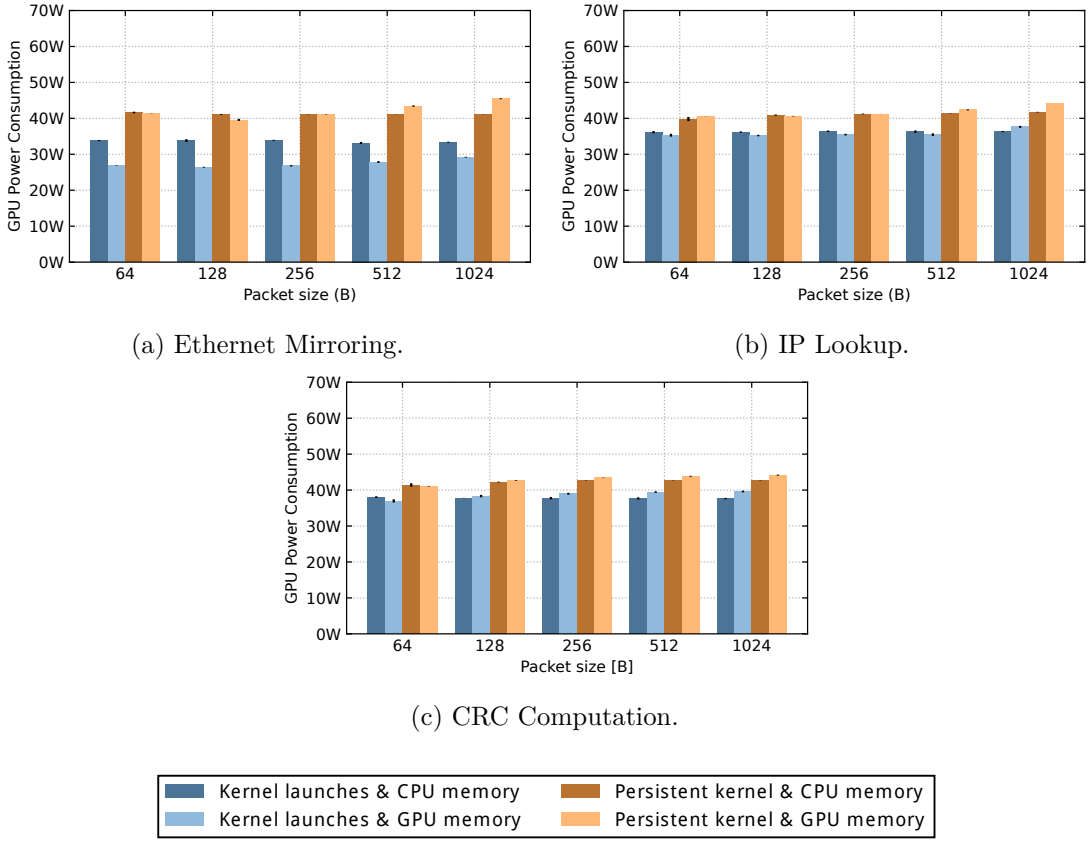
(d) IP Lookup fixed-rate latency.



(e) CRC Computation Throughput.

(f) CRC Computation fixed-rate latency.

Figure 4.12: Performance of the CL implementation depending on memory pool location and GPU control flow.



(a) Ethernet Mirroring.

(b) IP Lookup.

(c) CRC Computation.

Figure 4.13: GPU power consumption of the CL implementation depending on memory pool location and GPU control flow.

benefits of much higher bandwidth than when it has to issue PCIe requests to read and write data from CPU memory, leading to a better throughput. The heavier the workload, the greater the improvement: when for Ethernet Mirroring it slightly increases performance, for CRC computation it can more than double the bandwidth for large packets. As data only travels from the NIC to the GPU, latency decreases. Also, the improvement is better when the workload is heavier.

On the other hand, using persistent kernels instead of multiple kernel launches is especially beneficial for throughput performance when the workload performed is light and the data resides in GPU memory. For CRC computation, for example, there is almost no improvement. For such heavy workloads, kernel launch latency is hidden by the computation time (cf. Section 2.3.1). Persistent kernels always reduce the average latency at a fixed rate, as the flow of packets is too low to hide kernel launch times with computation times.

It is worth noting that these improvements increase power consumption. Persistent kernels, by keeping all GPU resources busy, polling on flags, induce a large increase in

consumption if the workload is somewhat lightweight, up to 15 W. However, for CRC computation, it only increases by around 5 W, as resources are already more used. On the other hand, depending on the workload and packet size, using a GPU memory pool can reduce power consumption, but for heavy workloads it slightly increases it, the majority of power being drawn to perform computations.

Despite being attractive, these improvements should not always be used. Indeed, with persistent kernels, each list has its own kernel processing it. If the number of kernels exceeds the number of SMs in the GPU, it will block because kernels cannot be preempted. At least one kernel will never run. There also seems to be a bug in the GPUdev library. In our setup, using more than 4 CPU cores causes a segmentation fault when DPDK tries to transmit packets through the `rte_eth_tx_burst()` function. We were not able to find the cause of the bug, but as a result, using of GPU memory pools is only possible with up to 4 CPU cores to receive packets.

4.9 Zero-Copy usage in Coalescent ROIs implementation

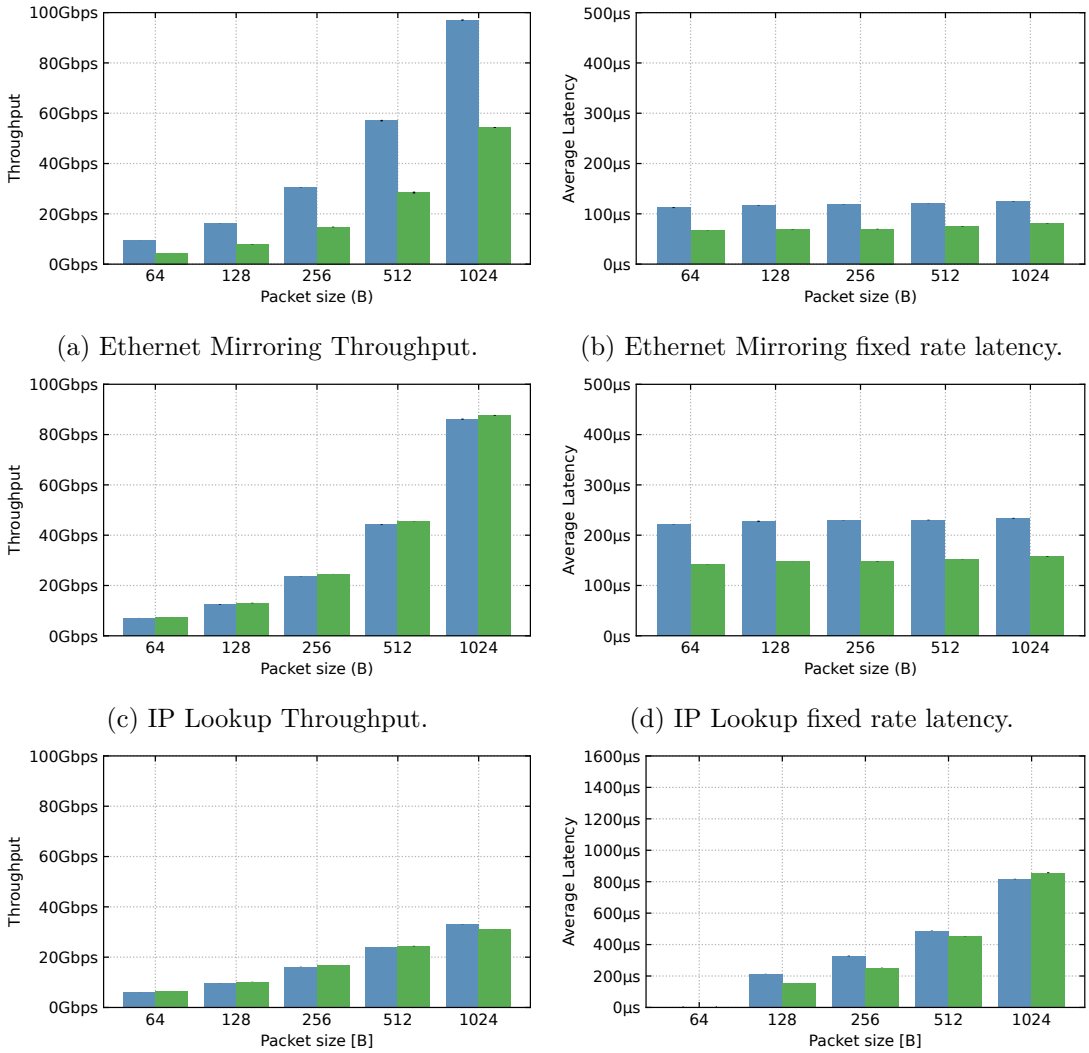
The ROI implementation performance is affected by whether zero-copy is used to access packet data in the coalesced memory zone of the CPU. Disabling zero-copy results in explicit data copies, back and forth if needed, of the memory. Enabling it allows the GPU to directly issue PCIe requests to read data from CPU during the kernel execution.

Peak throughput and average fixed-rate latency achieved for each application is shown in Figure 4.12. Other factors are set as follows: 1 CPU core is used to receive packets, 1 queue is used to communicate with the GPU, and GPU batches packets by 1024. Figure 4.15 shows the GPU power consumption of each application with the same factors.

We see that the use of zero-copy does not greatly improves throughput; in fact, it can dramatically reduce it for lightweight applications. Using zero-copy, the GPU reads/writes only the bytes it needs, avoiding unnecessary copies that would lower the bandwidth. However, in this implementation, ROIs are first coalesced in CPU memory: all the bytes are useful for the GPU. As all bytes copied are used, using zero-copy cannot improve throughput.

We see that it even decreases performance of the Ethernet Mirroring application. When profiling the Ethernet Mirroring application, we see that the bandwidth to read the bytes with zero-copy drop to 10 MB/s between the device memory and L2 cache, leading to these poor performances. Without zero-copy, such bandwidth is around 1 GB/s. This only happens at full rate with this workload, and we were not able to explain why.

However, at fixed rate and without batching, using zero-copy reduces latency, as data is directly accessible from the GPU. The latency caused by the copies of the data from host to device, and possibly from device to host, increases the average latency of the packets. With zero-copy, while the GPU is waiting for data to arrive, it can already process other data from other PCIe requests that have already been served. However, when data is explicitly copied, the GPU does not execute the corresponding kernel until



(a) Ethernet Mirroring Throughput.

(b) Ethernet Mirroring fixed rate latency.

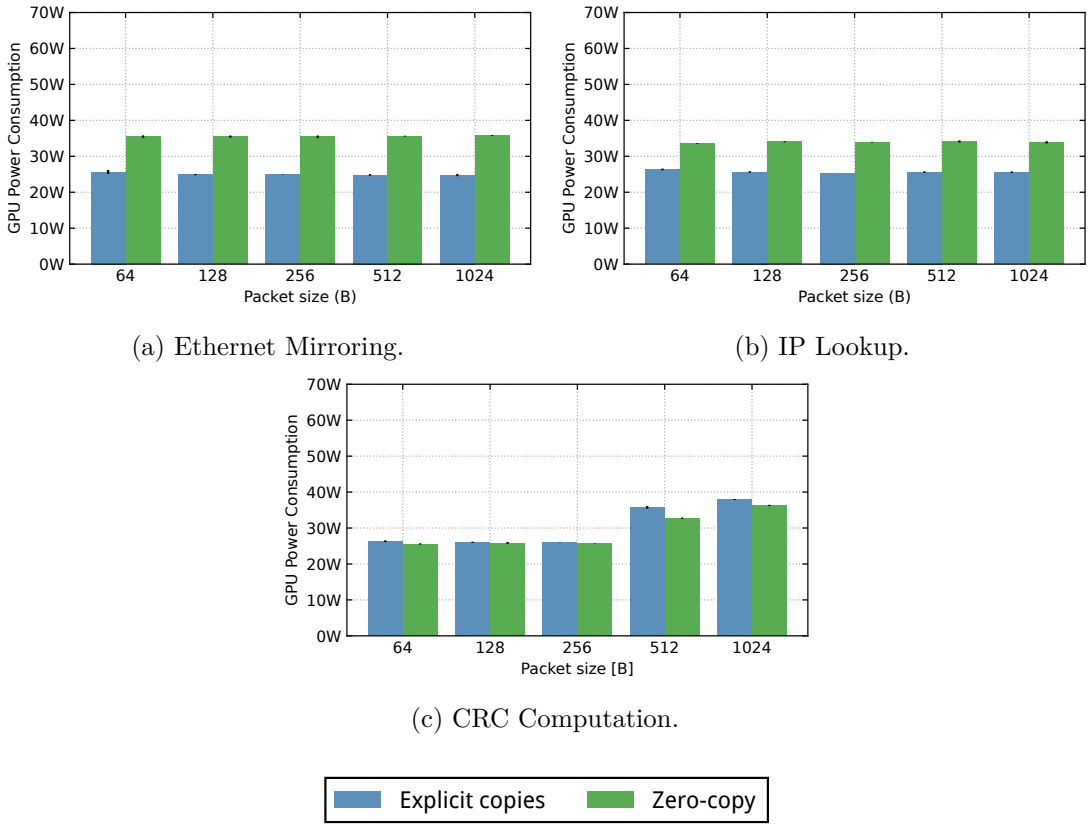
(c) IP Lookup Throughput.

(d) IP Lookup fixed rate latency.

(e) CRC Computation Throughput.

(f) CRC Computation fixed rate latency.

Figure 4.14: Performance of the ROI implementation depending on the use of zero-copy.



(a) Ethernet Mirroring.

(b) IP Lookup.

(c) CRC Computation.

Figure 4.15: GPU power consumption of the ROI implementation depending on the use of zero-copy.

the copy is complete, and the rate is too low to allow overlapping copies and kernel executions.

This feature also comes at the cost of an increased GPU power consumption. Multiple PCIe requests issued during the kernel execution consume more power than an explicit copy. In fact, memory copies performed with `cudaMemcpy` are optimized because they only happen once per batch and are performed by the GPU's DMA engine. For light applications, the increase is about 10 W. For heavy applications such as CRC computation, the difference in consumption is negligible, because the most of the GPU consumption comes from the computation.

In summary, while zero-copy does not affect throughput, or even affects it negatively, it always provides a decrease in latency, which can be desirable for some applications. It also comes with a slight increase in power consumption for light applications.

4.10 Number of GPU Queues in DOCA Implementation

The DOCA implementation supports up to 8 GPU queues that receive packets. Each queue is processed by one GPU block. Increasing the number of queues thus not only increases the number of packets received, but also increases the GPU utilization.

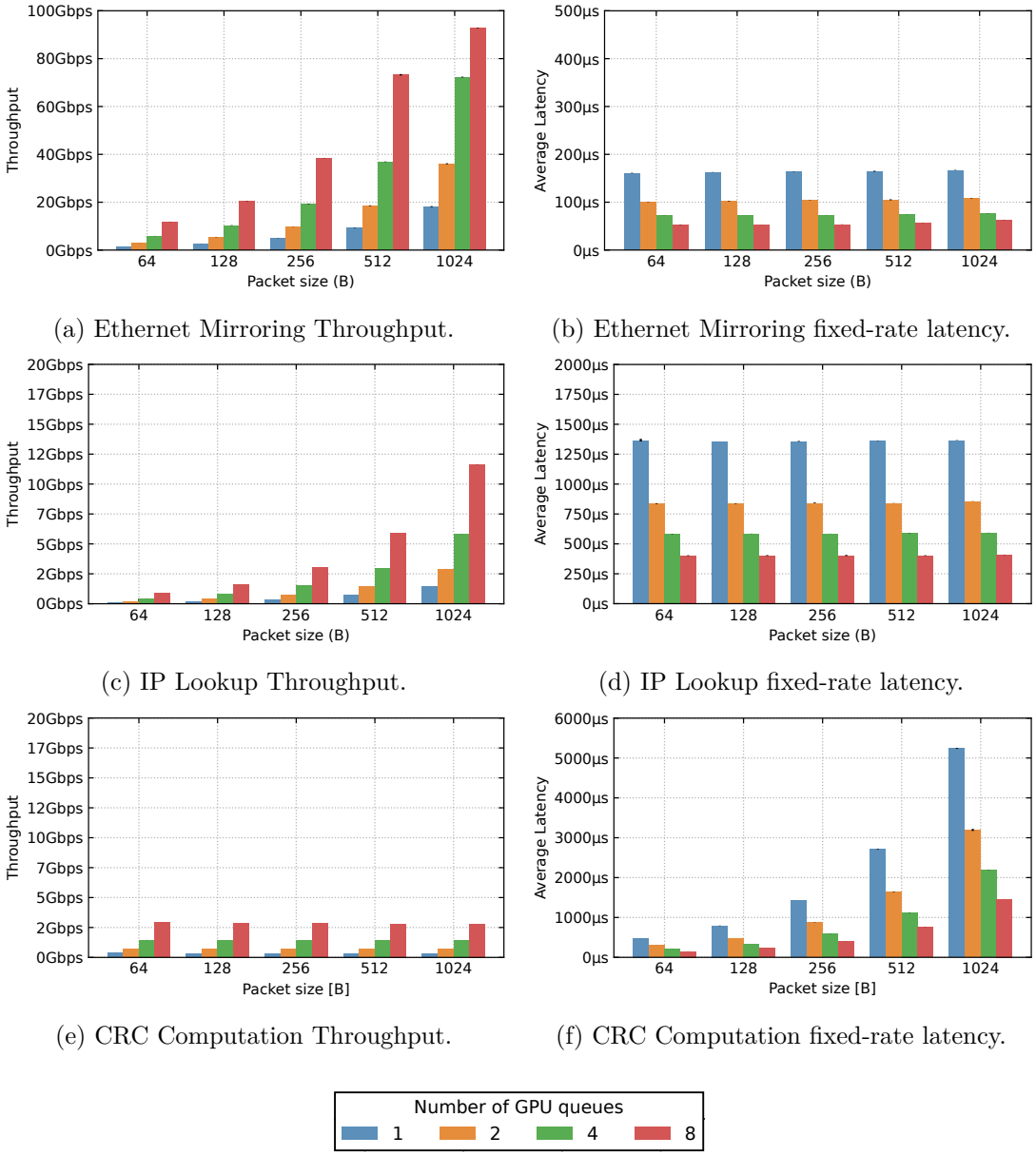
Throughput and fixed-rate average latency achieved with variable number of GPU queues is shown in Figure 4.16.

For the Ethernet Mirror and the IP Lookup, the throughput increases with the size of the packets for the same reason as in Section 4.4: the larger the packets, the smaller the number of packets, and thus the lower the amount of operations to perform. The throughput is directly proportional to the amount of DOCA queues, until we reach saturation. The throughput when performing IP lookups is much lower with this implementation than with others, reaching only 12 Gbit/s with 8 queues and not even 2 Gbit/s with 1. These measurements were made using the same routing table as for the other implementations, consisting of only 100 elements. The throughput does not saturate with 8 queues and with packets of 1024 B, so adding queues might improve the throughput. For the CRC workload, it makes sense that the throughput does not depend on the packet size since processing is done on every byte. As for IP lookups, DOCA is struggling with CRCs, only being able to deliver 2 Gbit/s with 8 queues. Again, the throughput is directly proportional to the amount of queues, so adding more than 8 queues will likely improve the throughput.

Each DOCA queue is assigned a block on the GPU, and each block has 32 threads, as in the sample provided by NVIDIA [64]. We tried to run DOCA with more threads per block, but we always got the exact same performance as with 32 threads. These low performances are surprising because CPU bypass seemed promising in theory as the data path is shorter than the other implementations, while still leveraging the GPU's capabilities.

Regarding latency, it is once again much higher than the other implementations in Section 4.4. For the Ethernet Mirroring and IP Lookup, the latency does not depend on the packet size at all and decreases linearly with the number of queues. For the CRC, the latency does increase with the size of the packets like with the other implementations. It also decreases linearly with the amount of queues. DOCA achieves as low as 50 µs latency for the Ethernet Mirror, about 400 µs for the IP Lookup and a few hundreds µs for the CRC.

Figure 4.17 shows the GPU power consumption for each application with the same factors. The DOCA application consumes a consistent amount of power for all workloads and packet sizes. With almost 40 W with only 1 queue and going slightly above 40 W with 8 queues, the difference is minimal for adding queues. The Ethernet Mirroring workload does consume more with larger packets reaching almost 50 W with 1024 byte packets and using 8 queues. This more pronounced increase is, as in other implementations, due to the fact that the power is not mainly drawn for computations, which is very low, but for packet handling, whose load increases with the number of queues.



(a) Ethernet Mirroring Throughput.

(b) Ethernet Mirroring fixed-rate latency.

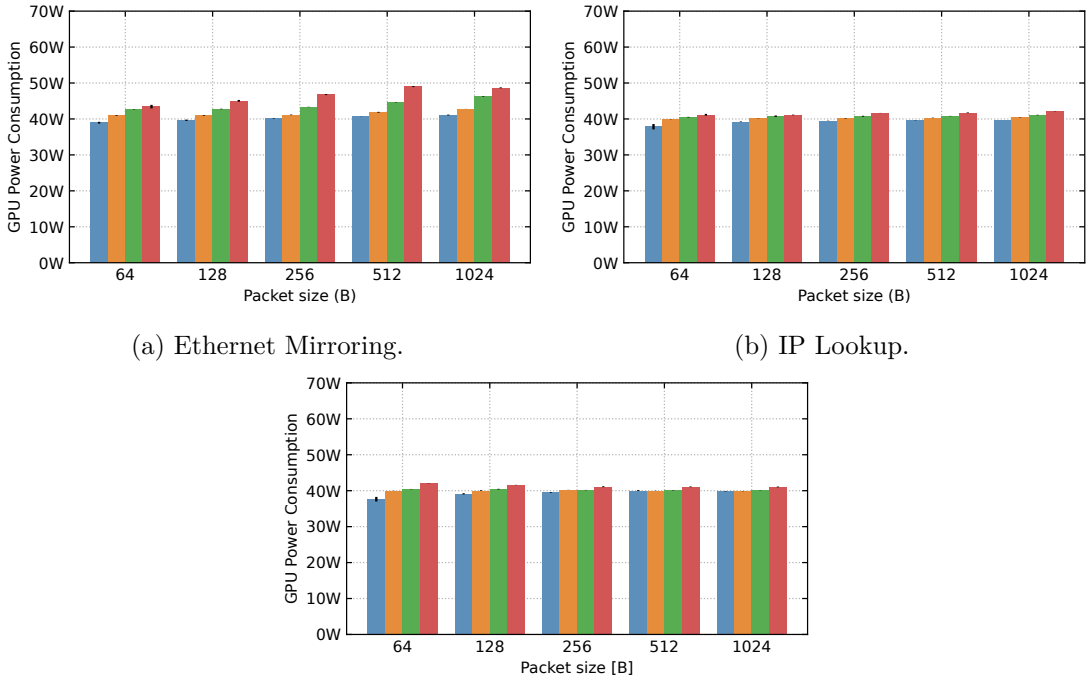
(c) IP Lookup Throughput.

(d) IP Lookup fixed-rate latency.

(e) CRC Computation Throughput.

(f) CRC Computation fixed-rate latency.

Figure 4.16: Performance of the DOCA implementation with variable number of GPU queues.



(a) Ethernet Mirroring.

(b) IP Lookup.

(c) CRC Computation.

Figure 4.17: GPU power consumption of the DOCA implementation with variable number of GPU queues.

In conclusion, DOCA performs much worse than other implementations, even for light workloads such as IP lookup on a small table. The lack of detailed documentation and the closed-source implementation leave little room for analysis of results. Since the throughput is linear in the number of queues, one could add more and more queues to increase throughput, but this would certainly reach a maximum at some point. Moreover, given current power consumption and throughput, it would certainly not be the most energy-efficient way to handle large amount of packets, even for computationally intensive network functions. In the end, using the CPU might be the better alternative as it seems like the GPU is not able to pull packets as fast as the CPU.

4.11 Size of the Routing Table for IP Lookup Application

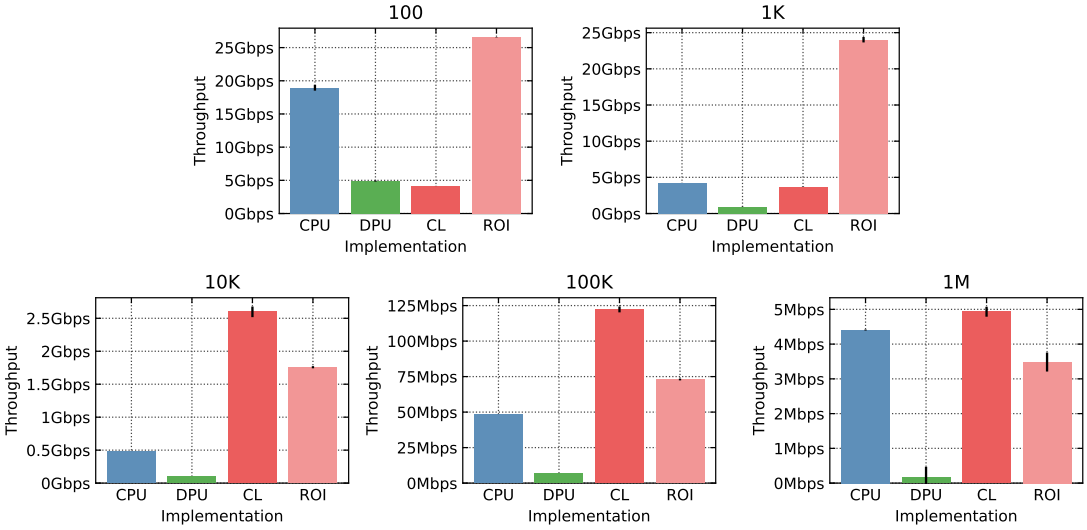


Figure 4.18: IP Lookup Throughput with variable routing table sizes, using 4 CPU cores to receive packets, and one shared buffer per core.

The throughput of the IP lookup workload with different routing table sizes for all implementations is shown in Figure 4.18. The DOCA implementation is omitted due to its poor results. The measurements are made with packets of 64 byte, batching of 1024 packets and 4 CPU cores. The number at the top of each graph represents the number of elements in the routing table.

With very few elements, the CPU and ROI implementations work best, with their throughput reaching around 20 Gbit/s. The CL implementation is much worse, only able to deliver 5 Gbit/s. A better batching size would increase the throughput (cf. Figure 4.10c). The DPU is much slower than the other implementations due to its fewer and weaker cores. This trend holds valid for all table sizes, and is even more noticeable with larger tables.

With the table size growing to 1000 elements, the impact of the GPU is noticeable with the ROI implementation, still able to deliver the same throughput, while the CPU implementation is struggling at 5 Gbps. The CL implementation has the same throughput as with 100 elements, being slowed down by not accessing packet ROIs in a coalescent manner.

With more elements, the lookups start to be the main bottleneck for all implementations. With 10.000 elements, all GPU implementations are able to transmit about 2 Gbit/s, while the CPU only delivers 0.5 Gbps. With 100.000 elements, the GPU implementations only deliver about 100 Mbit/s, and the CPU, 50 Mbit/s. Finally, with 1.000.000 elements, all implementations except the DPU provide only around 4 Mbit/s. Now, the CL implementation performs better, because at low throughput, shared buffers are full. Explicit copies of the ROI implementation have to wait for an empty space in the buffer before starting.

We see that as the table size increases, the speedup offered by the GPU over the CPU increases. However, it starts to decrease when the table reaches 100.000 elements. Adding more queues to communicate between the CPU and the GPU would lead to better performances, as the buffer is full for such a low throughput.

The rates were too low to get a meaningful latency measurement for large tables. The latency analysis with 100 elements can be found in Section 4.4.

In summary, we see that as the table size increases, the speedup offered by the GPU over the CPU increases. However, it starts to decrease with tables of 100.000 or more elements. For these large tables, a more adapted algorithm will likely produce better results.

Conclusion and future work

In this thesis, GPUs are used to accelerate the processing of network packets. We have implemented GPU elements that can be placed in a packet processing pipeline, using the CPU to coordinate communication between host and device. These elements are based on either a pure CUDA implementation, or on a communication list offered by the GPUdev DPDK library. We also explored CPU bypassing to remove the CPU from the packet processing path with 2 implementations. In the DPU implementation, a SmartNIC is used to process packets directly on its ARM cores. In the GPU-based application, the NIC passes its packets directly to the GPU, which takes over the control flow.

Elements are configurable at many levels, such as multithreading, control flow from the GPU, and CPU-GPU communication. They also provide many parameters that can influence their performance, such as the GPU batching size or the number of shared buffers used per CPU core to communicate with the GPU.

The implementations are evaluated using on three applications with varying degrees of workload: Ethernet Mirroring, which is very fast to compute; IP Lookup, whose complexity varies with the size of the lookup table; and CRC computation, which becomes harder to compute as packet size increases.

Evaluation shows that GPU elements outperform CPU-only implementations in multiple configurations. Performance depends on workload, packet flow rate, and packet sizes. Parameters must be carefully chosen to get the best performance from the GPU. Ethernet Mirroring shows little improvement, as the computation is very simple. IP Lookup shows better GPU performance when the lookup table contains many elements. GPUs are much better suited for heavy workloads such as CRC computation, even when using a single CPU core to receive packets. For such applications, allocating more shared buffers per core allows more of the GPU computational power to be used, resulting in a much lower overall power consumption than the CPU implementation. Using GPU memory pools so that the NIC writes its packets directly to the GPU's internal memory also shows great improvements by eliminating the need to pass data through CPU memory.

The Master-Workers model featured in previous works shows poor performance at today's network speeds. Using a full-path model, where each CPU core communicates with the GPU to process the packets it receives, shows a greater improvement in both throughput and latency.

CPU bypassing applications generally perform poorly. The DPU implementation is always slower than the CPU implementation, because the ARM cores are much less powerful than the CPU cores. The DOCA GPU-based implementation shows poor performance across all workloads. Increasing the number of queues improves performance by increasing GPU utilization, but it is still very low. With current implementations, keeping the CPU in the packet processing path to perform I/O provides better performance.

There are a number of directions in which these implementations could be improved. While VNF elements offload their computations to the GPU, they encapsulate the GPU processing. Adding multiple GPU elements in a chain leads to inefficiencies, as packets transit back and forth between CPU and GPU at each element. It would be beneficial to separate the GPU communication logic and the GPU computation into different elements, so that a pipeline could contain a point where all subsequent elements work directly on the GPU, avoiding unnecessary copies or redundant creation of CPU batches to communicate between elements.

Another interesting path to explore would be to improve the performance of the CPU bypassing applications. First, for specific applications, the DPU implementation could take advantage of the hardware accelerators present in current SmartNICs, e.g. IPsec encryption for IPsec elements. Second, tweaking the DOCA implementation to achieve peak performance is difficult, as DOCA functions are closed-source and documentation is lacking. The results seem off with this implementation and could certainly be improved, e.g. by exploring the number of threads receiving packets per GPU block.

Finally, more GPU-specific structures and algorithms could be used to improve GPU computational throughput. Algorithms performed on packets are the same on CPU and GPU. While this allows for easy comparison, specializing GPU algorithms using state-of-the-art GPU implementations could lead to large improvements in GPU computation time per packet. For example, multi-bit tries would be used for storing the lookup table of the IP lookup algorithm [86].

Still, this work shows that even at current GPU bandwidth, NIC, and PCIe interconnect speeds, GPUs still lead to great performance improvements over current CPU implementations for packet processing, providing high throughput and acceptable latencies for computational-intensive applications, while reducing overall power consumption.

Appendix A

CRC Computation

A.1 CRC32 Implementation

Click includes a CRC-32 Computation byte implementation based on a lookup table, as presented by Aram Perez [80].

The table is pre-generated using the polynomial to use for the computation. As per the IEEE 802.3 specification, this polynom is

$$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x^1 + x^0$$

which is represented by the hexadecimal representation `0x04C11DB7`.

Listing A.1 shows how to compute the table of type `uint32_t crc_table[256]`.

Listing A.2 shows how the table is used to compute a CRC-32 on a given message. The initial `crc_accum` given is `0xffffffff`.

```
#define POLYNOMIAL 0x04c11db7L

void gen_crc_table(uint32_t *crc_table)
 /* generate the table of CRC remainders for all possible bytes */
 { register int i, j;  register uint32_t crc_accum;
   for ( i = 0;  i < 256;  i++ )
       { crc_accum = ( (uint32_t) i << 24 );
         for ( j = 0;  j < 8;  j++ )
              { if ( crc_accum & 0x80000000L )
                   crc_accum =
                     ( crc_accum << 1 ) ^ POLYNOMIAL;
                else
                   crc_accum =
                     ( crc_accum << 1 ); }
         crc_table[i] = crc_accum; }
   return; }
```

Listing A.1: CRC-32 lookup table computation.

```
uint32_t update_crc(uint32_t crc_accum,
           const char *data_blk_ptr,
           int data_blk_size)
{
  int i, j;
  static int initialized = 0;

  if(initialized == 0){
    initialized = 1;
    gen_crc_table(crc_table);
  }

  for ( j = 0;  j < data_blk_size;  j++ ){
    i = ( (uint32_t) ( crc_accum >> 24) ^ *data_blk_ptr++ ) & 0xff;
    crc_accum = ( crc_accum << 8 ) ^ crc_table[i];
  }
  return crc_accum;
}
```

Listing A.2: CRC-32 computation, consisting of updating the CRC on the data block one byte at a time.

Appendix B

Routing Table Generation

B.1 Obtaining the data

Data is obtained from the RIPE RIS database. The script we used was written by Pr. Cristel Pelsser and is available on GitHub [79].

B.2 Formatting the data

Bash script used to extract prefixes and next hops from the table is provided in Listing B.1.

B.3 Removing duplicates

Python script used to remove duplicates from the list is available in Listing B.2.

```bash
#!/bin/bash

if [ $# -ne 2 ]; then
    echo "Usage: $0 <file>"
    exit 1
fi

if [ ! -f "$1" ]; then
    echo "File '$1' not found."
    exit 1
fi

awk '/PREFIX:/ {prefix=$2; count++} /NEXT_HOP:/ {next_hop=$2; print
↪  "\t\twrite r.add", prefix, "", next_hop, "0,"; if (count=='$2')
↪  exit}' "$1"
```

Listing B.1: Shell script to extract prefixes and next hops from the data.

```python
input_file = "input_file.txt"
output_file = "output_file.txt"

prefixes = set()

with open(input_file, 'r') as infile, open(output_file, 'w') as outfile:
    for line in infile:
        prefix = line.split()[2]

        if prefix not in prefixes:
            outfile.write(line)
            prefixes.add(prefix)
```

Listing B.2: Python script to remove duplicates from the list.

Bibliography

[1] Advanced Micro Devices, Inc. *AMD EPYC™ 9124*. 2022. URL: https://www.amd.com/en/products/processors/server/epyc/4th-generation-9004-and-8004-series/amd-epyc-9124.html (cit. on p. 72).

[2] Elena Agostini. *Accelerate DPDK Packet Processing Using GPU*. NVIDIA Corporation. Apr. 2021. URL: https://www.nvidia.com/en-us/on-demand/session/gtcspring21-s31972/ (cit. on pp. 40, 49).

[3] Elena Agostini. *Boosting Inline Packet Processing Using DPDK and GPUdev with GPUs*. Apr. 2022. URL: https://developer.nvidia.com/blog/optimizing-inline-packet-processing-using-dpdk-and-gpudev-with-gpus/ (cit. on pp. 3, 26).

[4] Elena Agostini. *Inline GPU Packet Processing with NVIDIA DOCA GPUNetIO*. Dec. 2022. URL: https://developer.nvidia.com/blog/inline-gpu-packet-processing-with-nvidia-doca-gpunetio/ (cit. on pp. 2, 22).

[5] Jasmin Ajanovic. "Pci express*(pcie*) 3.0 accelerator features". In: *Intel Corporation* 10 (2008), p. 6 (cit. on p. 18).

[6] Gargi Alavani and Santonu Sarkar. "Inspect-GPU: A Software to Evaluate Performance Characteristics of CUDA Kernels Using Microbenchmarks and Regression Models." In: *ICSOFT*. 2023, pp. 59–70 (cit. on p. 37).

[7] Nadav Amit, Amy Tai, and Michael Wei. "Don't shoot down TLB shootdowns!" In: *Proceedings of the Fifteenth European Conference on Computer Systems*. 2020, pp. 1–14 (cit. on p. 6).

[8] Gabriele Ara et al. "On the use of kernel bypass mechanisms for high-performance inter-container communications". In: *International Conference on High Performance Computing*. Springer. 2019, pp. 1–12 (cit. on p. 6).

[9] Tom Barbette. *FastClick wiki*. 2024. URL: https://github.com/tbarbette/fastclick/wiki (cit. on p. 10).

[10] Tom Barbette. *Network Performance Framework*. URL: https://github.com/tbarbette/npf (cit. on p. 59).

[11] Tom Barbette. *Pipeliner Element Documentation*. 2020. URL: https://github.com/tbarbette/fastclick/wiki/Pipeliner (cit. on p. 35).

[12] Tom Barbette, Cyril Soldani, and Laurent Mathy. “Fast userspace packet processing”. In: *The Eleventh 2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. 2016 (cit. on pp. 3, 8, 9, 29).

[13] Adam Belay et al. “{IX}: a protected dataplane operating system for high throughput and low latency”. In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 2014, pp. 49–65 (cit. on p. 7).

[14] Andrew Burnes. *Resizable BAR*. 2021. URL: `https://www.nvidia.com/en-us/geforce/news/geforce-rtx-30-series-resizable-bar-support/` (cit. on p. 18).

[15] Danilo Cerović et al. “Fast packet processing: A survey”. In: *IEEE Communications Surveys & Tutorials* 20.4 (2018), pp. 3645–3676 (cit. on p. 22).

[16] Ruining Chen and Guoao Sun. “A survey of kernel-bypass techniques in network stack”. In: *Proceedings of the 2018 2nd International Conference on Computer Science and Artificial Intelligence*. 2018, pp. 474–477 (cit. on pp. 2, 5, 6).

[17] Edward Cree. *Checksum Offloads*. 2016. URL: `https://www.kernel.org/doc/html/latest/networking/checksum-offloads.html` (cit. on p. 53).

[18] DeepL SE. *DeepL Write: AI-powered writing companion*. 2024. URL: `https://www.deepl.com/en/write/` (cit. on p.).

[19] *DPDK*. The Linux Foundation Projects. URL: `https://www.dpdk.org/` (cit. on pp. 2, 7, 8).

[20] DPDK. *L2 Forwarding Sample Application (in Real and Virtualized Environments)*. 2014. URL: `https://doc.dpdk.org/guides/sample_app_ug/l2_forward_real_virtual.html/` (cit. on p. 49).

[21] DPDK. *Programmer’s Guide*. URL: `https://doc.dpdk.org/guides/prog_guide/` (cit. on p. 8).

[22] eBPF.io authors. *eBPF*. URL: `https://ebpf.io/` (cit. on p. 6).

[23] Ethernet Alliance. *2024 Ethernet Roadmap*. 2024. URL: `https://ethernetalliance.org/technology/ethernet-roadmap/` (cit. on p. 1).

[24] Daniel Etiemble. “45-year CPU evolution: one law and two equations”. In: *arXiv preprint arXiv:1803.00254* (2018) (cit. on p. 2).

[25] Alireza Farshin et al. “PacketMill: toward per-Core 100-Gbps networking”. In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 2021, pp. 1–17 (cit. on p. 51).

[26] fd.io. *VPP*. URL: `https://wiki.fd.io/view/VPP/` (cit. on p. 8).

[27] The Linux Foundation. *Open Virtual Switch*. URL: `https://www.openvswitch.org/` (cit. on p. 8).

[28] Eduardo Freitas et al. “A survey on accelerating technologies for fast network packet processing in Linux environments”. In: *Computer Communications* 196 (2022), pp. 148–166 (cit. on p. 22).

[29] Yuchen Gao. *High-Performance Network Data Transfers to GPU: A Study of Nvidia GPU Direct RDMA and GPUNetIO*. 2023 (cit. on p. 19).

[30] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. "The Google file system". In: *Proceedings of the nineteenth ACM symposium on Operating systems principles*. 2003, pp. 29–43 (cit. on p. 2).

[31] Jayshree Ghorpade et al. "GPGPU processing in CUDA architecture". In: *arXiv preprint arXiv:1202.4347* (2012) (cit. on p. 1).

[32] GitHub, Inc. *GitHub Copilot - You AI pair programmer*. 2024. URL: `https://github.com/features/copilot/` (cit. on p.).

[33] Younghwan Go et al. "APUNet: Revitalizing GPU as Packet Processing Accelerator". In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 2017, pp. 83–96 (cit. on pp. 2, 15, 22, 25, 26, 34, 69).

[34] Corey Gough et al. "Cpu power management". In: *Energy Efficient Servers: Blueprints for Data Center Optimization* (2015), pp. 21–70 (cit. on p. 72).

[35] Chuanxiong Guo et al. "RDMA over commodity ethernet at scale". In: *Proceedings of the 2016 ACM SIGCOMM Conference*. 2016, pp. 202–215 (cit. on p. 7).

[36] Zehua Guo, Sen Liu, and Zhi-Li Zhang. "Traffic control for RDMA-enabled data center networks: A survey". In: *IEEE Systems Journal* 14.1 (2019), pp. 677–688 (cit. on p. 7).

[37] Sangjin Han et al. "PacketShader: A GPU-Accelerated Software Router". In: *ACM SIGCOMM Computer Communication Review* 40.4 (2010), pp. 195–206 (cit. on pp. 2, 22, 23, 34, 69).

[38] "IEEE Standard for Ethernet". In: *IEEE Std 802.3-2022 (Revision of IEEE Std 802.3-2018)* (2022), pp. 1–7025. DOI: `10.1109/IEEESTD.2022.9844436` (cit. on pp. 50, 53, 60).

[39] Intel Corporation. "Improving Network Performance in Multi-Core Systems". In: (2007). URL: `https://www.intel.com/content/dam/support/us/en/documents/network/sb/318483001us2.pdf` (cit. on p. 8).

[40] Intel Corporation. *Intel FPGA SmartNIC N6000-PL Platform*. URL: `https://www.intel.com/content/www/us/en/products/details/fpga/platforms/smartnic/n6000-pl-platform.html` (cit. on p. 10).

[41] EunYoung Jeong et al. "{mTCP}: a highly scalable user-level {TCP} stack for multicore systems". In: *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. 2014, pp. 489–502 (cit. on p. 7).

[42] Changue Jung et al. "GPU-Ether: GPU-native Packet I/O for GPU Applications on Commodity Ethernet". In: *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*. IEEE. 2021, pp. 1–10 (cit. on p. 22).

[43] Anuj Kalia et al. "Raising the Bar for Using GPUs in Software Packet Processing". In: *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. 2015, pp. 409–423 (cit. on pp. 22, 25).

[44] Georgios P Katsikas et al. “What you need to know about (smart) network interface cards”. In: *International Conference on Passive and Active Network Measurement*. Springer. 2021, pp. 319–336 (cit. on pp. 2, 10).

[45] Eddie Kohler. *Click Diagrams*. Available under the Click license: `https://github.com/kohler/click/raw/master/LICENSE`. 2000. URL: `https://github.com/kohler/click/tree/master/etc/diagrams/` (cit. on p. 9).

[46] Eddie Kohler. *Click Wiki*. 2017. URL: `https://github.com/kohler/click/wiki` (cit. on p. 10).

[47] Eddie Kohler. *LinearIPLookup Element Documentation*. 2017. URL: `https://github.com/kohler/click/wiki/LinearIPLookup` (cit. on p. 52).

[48] Eddie Kohler et al. “The Click Modular Router”. In: *ACM Transactions on Computer Systems* 18.3 (2000), pp. 263–297 (cit. on pp. 3, 8).

[49] Lopamudra Kundu et al. “Hardware acceleration for open radio access networks: A contemporary overview”. In: *IEEE Communications Magazine* (2023) (cit. on p. 41).

[50] Michael Laor and Lior Gendel. “The effect of packet reordering in a backbone link on application throughput”. In: *IEEE network* 16.5 (2002), pp. 28–36 (cit. on p. 24).

[51] Ang Li et al. “Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect”. In: *IEEE Transactions on Parallel and Distributed Systems* 31.1 (2019), pp. 94–110 (cit. on p. 15).

[52] Jianshen Liu et al. “Performance characteristics of the bluefield-2 smartnic”. In: *arXiv preprint arXiv:2105.06619* (2021) (cit. on p. 21).

[53] Taowei Luo et al. “Improving TLB performance by increasing hugepage ratio”. In: *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE. 2015, pp. 1139–1142 (cit. on p. 8).

[54] Sparsh Mittal and Jeffrey S Vetter. “A survey of CPU-GPU heterogeneous computing techniques”. In: *ACM Computing Surveys (CSUR)* 47.4 (2015), pp. 1–35 (cit. on p. 15).

[55] Jeffrey C Mogul and Anita Borg. “The effect of context switches on cache performance”. In: *ACM SIGPLAN Notices* 26.4 (1991), pp. 75–84 (cit. on p. 6).

[56] B Neelima and Prakash S Raghavendra. “Recent trends in software and hardware for GPGPU computing: a comprehensive survey”. In: *2010 5th International Conference on Industrial and Information Systems*. IEEE. 2010, pp. 319–324 (cit. on p. 1).

[57] Tatjana R Nikolić et al. “From single cpu to multicore systems”. In: *2022 57th International Scientific Conference on Information, Communication and Energy Systems and Technologies (ICEST)*. IEEE. 2022, pp. 1–8 (cit. on p. 2).

[58] NVIDIA Corporation. *CUDA C++ Best Practices Guide*. URL: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/ (cit. on pp. 15, 37).

[59] NVIDIA Corporation. *CUDA C++ Programming Guide*. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/ (cit. on pp. 14, 16, 38, 45, 46).

[60] NVIDIA Corporation. *CUDA Zone*. 2024. URL: https://developer.nvidia.com/cuda-zone/ (cit. on pp. 1, 14).

[61] NVIDIA Corporation. "DOCA GPUNetIO". In: (2024). URL: https://docs.nvidia.com/doca/sdk/doca+gpunetio/index.html (cit. on pp. 3, 12).

[62] NVIDIA Corporation. *doca_mmap*. 2023. URL: https://docs.nvidia.com/doca/archive/doca-v1.5.2/doca-core-programming-guide/index.html#doca-mmap (cit. on p. 11).

[63] NVIDIA Corporation. *gdrcopy*. 2024. URL: https://github.com/NVIDIA/gdrcopy (cit. on p. 17).

[64] NVIDIA Corporation. *GPUNetIO Simple Receive Sample*. URL: https://docs.nvidia.com/doca/archive/2-5-2/doca+gpunetio/index.html#src-2448900781_id-.DOCAGPUNetIOv2.5.0-SimpleReceive (cit. on pp. 46, 81).

[65] NVIDIA Corporation. *Magnum IO GDRCopy*. 2024. URL: https://developer.nvidia.com/gdrcopy/ (cit. on p. 17).

[66] NVIDIA Corporation. *NVIDIA BlueField Networking Platform*. URL: https://www.nvidia.com/en-us/networking/products/data-processing-unit/ (cit. on p. 10).

[67] NVIDIA Corporation. *NVIDIA BlueField-2 DPU*. URL: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-2-dpu.pdf (cit. on pp. 2, 11).

[68] NVIDIA Corporation. *NVIDIA ConnectX-7 400G Adapters*. 2024. URL: https://www.nvidia.com/content/dam/en-zz/Solutions/networking/infiniband/connectx-7-datasheet.pdf (cit. on p. 2).

[69] NVIDIA Corporation. *NVIDIA DOCA Software Framework*. URL: https://developer.nvidia.com/networking/doca (cit. on p. 2).

[70] NVIDIA Corporation. *NVIDIA GPUDirect*. 2024. URL: https://developer.nvidia.com/gpudirect/ (cit. on p. 1).

[71] NVIDIA Corporation. *NVIDIA Nsight Compute*. 2024. URL: https://developer.nvidia.com/nsight-compute (cit. on p. 58).

[72] NVIDIA Corporation. *NVIDIA Nsight Systems*. 2024. URL: https://developer.nvidia.com/nsight-systems (cit. on p. 58).

[73] NVIDIA Corporation. *NVIDIA RTX A2000*. 2021. URL: https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/rtx-a2000/nvidia-rtx-a2000-datasheet.pdf (cit. on pp. 13, 61).

[74] NVIDIA Corporation. *NVIDIA® BlueField® DPU Modes of Operation*. URL: https://docs.nvidia.com/networking/display/bluefielddpubspv422/modes+of+operation (cit. on p. 11).

[75] NVIDIA Corporation. *System Management Interface SMI*. URL: https://developer.nvidia.com/system-management-interface (cit. on p. 61).

[76] NVIDIA Corporation & Affiliates. *General-Purpose Graphics Processing Unit Library*. 2021. URL: https://doc.dpdk.org/guides/prog_guide/gpudev.html (cit. on pp. 38, 40, 42, 43).

[77] NVIDIA Corporation and Affiliates. *DOCA Core*. 2023. URL: https://docs.nvidia.com/doca/archive/doca-v1.5.2/doca-core-programming-guide/index.html (cit. on p. 11).

[78] PCI-SIG. *PCI Express Base Specification Revision 3.0*. Available at: https://qjdxmy.com/download/pcie-base-spec.pdf, page 595. PCI-SIG. 2010 (cit. on p. 19).

[79] Cristel Pelsser. *ripe-ris-download*. 2020. URL: https://github.com/cpelsser/ripe-ris-download/ (cit. on pp. 52, 91).

[80] Aram Perez. "Byte-wise CRC calculations". In: *IEEE Micro* 3.03 (1983), pp. 40–50 (cit. on pp. 54, 89).

[81] Julien Plante et al. "A high-performance data acquisition on COTS hardware for astronomical instrumentation". In: *Software and Cyberinfrastructure for Astronomy VII*. Vol. 12189. SPIE. 2022, pp. 328–337 (cit. on p. 44).

[82] Jon Postel. "Internet protocol—DARPA internet program protocol specification, rfc 791". In: *(No Title)* (1981) (cit. on p. 50).

[83] WS Ring and OPTOELECTRONICS INDUSTRY DEVELOPMENT ASSOCIATION WASHINGTON DC. "100 Gbit Interconnects and Above: The Need for Speed". In: (2007) (cit. on p. 2).

[84] RIPE NCC. *Routing Information Service (RIS)*. 2024. URL: https://www.ripe.net/analyse/internet-measurements/routing-information-service-ris/ (cit. on p. 52).

[85] Luigi Rizzo. "netmap: a novel framework for fast packet I/O". In: *21st USENIX Security Symposium (USENIX Security 12)*. 2012, pp. 101–112 (cit. on p. 6).

[86] Sartaj Sahni and Kun Suk Kim. "Efficient construction of multibit tries for IP lookup". In: *IEEE/ACM Transactions on Networking* 11.4 (2003), pp. 650–662 (cit. on p. 88).

[87] Mark Silberstein et al. "GPUfs: Integrating a file system with GPUs". In: *Proceedings of the eighteenth international conference on Architectural Support for Programming Languages and Operating Systems*. 2013, pp. 485–498 (cit. on p. 1).

[88] Mark Silberstein et al. "GPUnet: Networking abstractions for GPU programs". In: *ACM Transactions on Computer Systems (TOCS)* 34.3 (2016), pp. 1–31 (cit. on pp. 22, 26).

[89] Weibin Sun and Robert Ricci. "Fast and Flexible: Parallel Packet Processing with GPUs and Click". In: *Architectures for Networking and Communications Systems*. IEEE. 2013, pp. 25–35 (cit. on pp. 3, 22, 24).

[90] Herb Sutter et al. "The free lunch is over: A fundamental turn toward concurrency in software". In: *Dr. Dobb's journal* 30.3 (2005), pp. 202–210 (cit. on p. 2).

[91] The Khronos Group Inc. *OpenCL*. 2024. URL: https://www.khronos.org/opencl/ (cit. on pp. 1, 14).

[92] The Khronos Group Inc. *OpenCL – Conformant Products*. 2024. URL: https://www.khronos.org/conformance/adopters/conformant-products/opencl/ (cit. on p. 14).

[93] The Linux Foundation Wiki Authors. *napi*. The Linux Foundation Wiki. URL: https://wiki.linuxfoundation.org/networking/napi/ (cit. on p. 6).

[94] The Linux Foundation Wiki Authors. *sk_buff*. The Linux Foundation Wiki. URL: https://wiki.linuxfoundation.org/networking/sk_buff/ (cit. on p. 5).

[95] Arnout Vandecappelle and Mind. *Networking – Kernel Flow*. The Linux Foundation Wiki. URL: https://wiki.linuxfoundation.org/networking/kernel_flow/ (cit. on p. 6).

[96] Marcos AM Vieira et al. "Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications". In: *ACM Computing Surveys (CSUR)* 53.1 (2020), pp. 1–36 (cit. on p. 6).

[97] Keith Wiles. *Pktgen-DPDK*. Intel Corporation. URL: https://github.com/pktgen/Pktgen-DPDK (cit. on p. 57).

[98] Xilinx. *Xilinx Alveo U25 SmartNIC Platform*. 2020. URL: https://www.xilinx.com/publications/product-briefs/alveo-u25-product-brief.pdf (cit. on p. 10).

[99] Tianzhu Zhang et al. "Comparing the performance of state-of-the-art software switches for NFV". In: *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*. 2019, pp. 68–81 (cit. on p. 8).

[100] Xiantao Zhang et al. "High-density multi-tenant bare-metal cloud". In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2020, pp. 483–495 (cit. on p. 2).

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**