Toward Highly-efficient GPU-centric Networking

MASSIMO GIRONDI

Academic dissertation which, with due permission of the KTH Royal Institute of Technology, is submitted for public defence for the Degree of Licentiate of Technology on Wednesday the 10th April 2024 at 09:00 CET via Zoom and Sal C (Sven-Olof Öhrvik) at Electrum, Kungliga Tekniska Högskolan, Kistagången 16, Kista, Stockholm, Sweden.

Licentiate Thesis in Information and Communication Technology
KTH Royal Institute of Technology
Stockholm, Sweden 2024

Mot Högeffektiva GPU-centrerade Nätverk

ISBN 978-91-8040-877-6
TRITA-EECS-AVL-2024:30

Printed by: Universitetsservice US AB, Sweden 2024

Abstract

Graphics Processing Units (GPUs) are emerging as the most popular accelerator for many applications, powering the core of Machine Learning applications and many computing-intensive workloads. GPUs have typically been considered as accelerators, with Central Processing Units (CPUs) in charge of the main application logic, data movement, and network connectivity. In these architectures, input and output data of network-based GPU-accelerated application typically traverse the CPU, and the Operating System network stack multiple times, getting copied across the system main memory. These increase application latency and require expensive CPU cycles, reducing the power efficiency of systems, and increasing the overall response times. These inefficiencies become of higher importance in latency-bounded deployments, or with high throughput, where copy times could easily inflate the response time of modern GPUs.

The main contribution of this dissertation is towards a GPU-centric network architecture, allowing GPUs to initiate network transfers without the intervention of CPUs. We focus on commodity hardware, using NVIDIA GPUs and Remote Direct Memory Access over Converged Ethernet (RoCE) to realize this architecture, removing the need of highly homogeneous clusters and ad-hoc designed network architecture, as it is required by many other similar approaches. By porting some `rdma-core` posting routines to GPU runtime, we can saturate a 100-Gbps link without any CPU cycle, reducing the overall system response time, while increasing the power efficiency and improving the application throughput.

The second contribution concerns the analysis of Clockwork, a State-of-The-Art inference serving system, showing the limitations imposed by controller-centric, CPU-mediated architectures. We then propose an alternative architecture to this system based on an RDMA transport, and we study some performance gains that such a system would introduce.

An integral component of an inference system is to account and track user flows, and distribute them across multiple worker nodes. Our third contribution aims to understand the challenges of *Connection Tracking* applications running at 100 Gbps, in the context of a Stateful Load Balancer running on commodity hardware.

Keywords

Low-Latency Internet Services, Packet Processing, Network Functions Virtualization, Middle Boxes, Commodity Hardware, Multi-Hundred-Gigabit-Per-Second, Low-Level Optimization, Graphics Processing Units, Inference Serving, Remote Direct Memory Access

Sammanfattning

Grafikprocessorer (GPU) håller på att bli den mest populära acceleratorn för många tillämpningar och utgör kärnan i maskininlärningsapplikationer och många dataintensiva beräkningar. GPU:er har vanligtvis betraktats som acceleratorer, med CPU:er (Central Processing Units) som ansvarar för den huvudsakliga applikationslogiken, förflyttning av data och nätverksanslutningen. I dessa arkitekturer passerar in- och utdata för nätverksbaserade GPU-accelererade applikationer vanligtvis processorn och operativsystemets nätverksstack flera gånger och kopieras över systemets huvudminne. Detta ökar applikationslatensen och kräver dyra CPU-cykler, vilket minskar systemens energieffektivitet och ökar de totala svarstiderna. Denna ineffektivitet får större betydelse i latensbegränsade distributioner eller med hög genomströmning, där kopieringstider lätt kan öka svarstiden för moderna GPU:er.

Det huvudsakliga bidraget i denna avhandling är en GPU-centrerad nätverksarkitektur som gör det möjligt för GPU:er att initiera nätverksöverföringar utan inblandning av CPU:er. Vi fokuserar på kretsar befintliga på marknaden och använder NVIDIA GPU:er och Remote Direct Memory Access over Converged Ethernet (RoCE) för att realisera denna arkitektur, vilket tar bort behovet av mycket homogena kluster och ad hoc-designad nätverksarkitektur, vilket krävs av många andra liknande tillvägagångssätt. Genom att portera vissa `rdma-core` posteringsrutiner till GPU runtime kan vi mätta en 100-Gbps länk utan någon CPU-cykel, vilket minskar systemets totala svarstid, samtidigt som energieffektiviteten ökar och applikationens genomströmning förbättras.

Det andra bidraget handlar om analysen av Clockwork, ett toppmodernt inferenssystem, som visar begränsningarna med controller-centrerade, CPU-medierade arkitekturer. Vi föreslår sedan en alternativ arkitektur till detta system som baseras på en RDMA-transport, och vi studerar några prestandavinster som ett sådant system skulle medföra.

En integrerad del av ett inferenssystem är att redovisa och spåra användarflöden, och distribuera dem över flera arbetsnoder. Vårt tredje bidrag syftar till att förstå utmaningarna med *Connection Tracking*-applikationer som körs med 100 Gbps, i samband med en Stateful Load Balancer som körs på hårdvara befintlig på marknaden.

Nyckelord

Internettjänster med Låg Fördröjning, Paketbearbetning, Virtualisering av Nätverksfunktioner, Mellanutrustning, Tillgänglig Datorhårdvara, Flera-Hundra-Gigabit-Per-Sekund, Lågnivå-Optimering, Grafikprocessor, Inferensserving, Remote Direct Memory Access

Acknowledgments

First and foremost, I would like to thank my supervisors Professor Dejan Kostić and Associate Professor Marco Chiesa. They trusted me and gave me opportunities to work and grow within NSLAB, touching many interesting topics and exploring various technical and scientific aspects of modern network systems.

I would also like to thank Prof. Em. Gerald Q. Maguire Jr., whose efforts in the early days of this thesis helped shape it from a disordered set of ideas into a scientific report. Without his initial push to explore the launch of RDMA verbs from GPUs most of this work would not exist. Mariano Scazzariello helped in the initial development of the RDMA stack implementation and dug the undocumented code with me.

Assistant Professor Tom Barbette played a significant role during the early stage of my journey at KTH, leading to the HPSR 2021 and NSDI 2022 publications. Thomas Sjöland promptly offered his time to help with the Swedish abstract, and Associate Professor Masoumeh Ebrahimi provided insightful comments serving as advance reviewer of this thesis.

Finally, I must thank all the friends and colleagues who supported and tolerated me over these years: there is no need to list them, they know who they are!

Last but not least, I would like to thank my family for fostering my journey far from home and supporting me over the years.

Massimo Girondi,
Stockholm, April 2024

The research leading to this thesis has received funding from the European Research Council (ERC) under the European Union Horizon 2020 Research and Innovation Program (grant agreement No. 770889). It was also partially funded by the Swedish Foundation for Strategic Research (SSF).

Contents

1 Introduction **1**

1.1 Thesis limitations 2

1.2 Thesis contributions 3

1.3 Publications 4

1.4 Research sustainability and ethical aspects 5

1.4.1 Sustainability 5

1.4.2 Ethical aspects 6

1.5 Thesis structure 6

2 Background **7**

2.1 AI and ML 7

2.1.1 What is behind an inference 8

2.1.1.1 At the core of inference 9

2.2 AI accelerators 10

2.2.1 Typical accelerator offload processing 10

2.2.2 The CUDA architecture 11

2.2.2.1 CUDA GPU connectivity 12

2.2.2.2 CUDA streams 14

2.2.2.3 CUDA Graphs 14

2.2.2.4 Advanced memory management in CUDA 16

2.2.3 NVIDIA is not the only player 18

2.2.4 GPUs as machine learning accelerators 19

2.3 PCIe 20
2.3.1 PCIe topology 22
2.3.2 PCIe DMA 22
2.3.3 Extending DMA over the network: RDMA 23
2.4 Connectivity in the datacenter 23
2.4.1 RDMA in the datacenter 24
2.4.2 Future steps for Ethernet 25
2.5 TVM 26
2.6 How much processing does an inference take? 27
2.7 The cost of the network in inference tasks 30
2.8 How long does it take to do an inference? 32
2.9 A use case for bandwidth-intense inference 34
3 Related Works **39**
3.1 Interconnecting GPUs and other devices 39
3.1.1 HPC-oriented technologies 40
3.1.2 Data-center GPU networking 41
3.2 AI-oriented efforts 43
3.3 Improving GPU runtime efficiency 44
4 Research Problem and Challenges **46**
4.1 The problem 46
4.1.1 The performance of GPUs is growing faster than network speeds 47
4.1.2 Traditional GPU systems scaling is limited by CPU performance 48
4.1.3 Problem definition 48
4.2 The challenges 49
4.2.1 Challenge 1: High-speed connection tracking 49
4.2.2 Challenge 2: Network operations from GPU without CPU interventions 49
4.2.3 Challenge 3: High-performance inference serving architectures .. 50
4.3 Research question 50
4.4 Research methodology 51

5 Experimental Setup **52**
5.1 Compute nodes ... 52
5.2 Data-plane network ... 53
5.3 Statistical validity of the results ... 54
5.4 Results reproducibility ... 54
5.5 AI models used in this thesis ... 55
5.6 NVIDIA DGX-H100 ... 55
6 Connection Tracking **57**
6.1 Introduction ... 57
6.2 Data-structures for connection tracking ... 59
6.2.1 Hash tables ... 60
6.2.2 Multi-core approaches ... 61
6.2.3 Flow aging and deletions ... 62
6.3 Evaluation ... 64
6.3.1 Test methodology ... 64
6.3.1.1 Traffic generation ... 64
6.3.1.2 CPU cycle measurement ... 65
6.3.2 Single-core performances ... 65
6.3.3 Multi-core scaling ... 67
6.3.4 Deletion ... 69
6.3.4.1 Scaling with deletion ... 69
6.3.4.2 Additional controls imply additional overhead ... 69
6.4 Related works ... 71
6.4.1 Hash tables ... 71
6.4.2 Flow ageing ... 71
6.4.3 Future work ... 72
6.5 Conclusions ... 72
7 RDMA from the GPU Side **73**
7.1 RDMA, InfiniBand, and RoCE ... 73
7.2 Structure of a typical RDMA application ... 74

7.3 The life of an RDMA verb 75
7.3.1 Under the hood of RDMA verbs posting 77
7.3.2 Ringing the doorbell 78
7.3.3 Memory registration and protection domains 79
7.4 Receiving 1-sided RDMA verbs 80
7.4.1 NVIDIA GPUDirect RDMA and peer-to-peer memory accesses 80
7.4.1.1 Enabling technologies 80
7.4.1.2 NVIDIA GPUDirect 81
7.4.1.3 Active and passive roles 83
7.4.1.4 The AMD way 83
7.5 Extending rdma-core for GPU-driven operations 83
7.5.1 Challenges 84
7.5.2 Memory allocation and registration 85
7.5.3 Verb posting from GPU side 86
7.5.4 Receiving data on the GPU side 87
7.5.5 Consuming CQE 87
7.5.6 GPU-side continuous loop 88
7.5.7 Final application workflow 88
7.5.8 Towards CPU-free execution 94
7.5.9 Event-based synchronization 95
7.5.10 Multi-stream architecture 95
7.5.11 Time accounting 97
7.6 Performance evaluation 97
7.6.1 The performance of GPU-controlled RDMA 97
7.6.2 Number of CPU interrupts 101
7.6.3 An inference serving prototype 103
7.6.3.1 How costly is the networking? 105
7.6.3.2 The burden of CPU-GPU synchronization 107
7.6.3.3 Does the transport type matter? 108
7.6.3.4 Zero-CPU inferencing 109
7.6.3.5 Buffer allocation method 110

8 A System for Inference Serving **113**
8.1 The Clockwork model ... 113
8.1.1 Assessing the Clockwork model ... 114
8.1.1.1 Analysis limitations ... 117
8.1.2 Limitations of the Clockwork model ... 118
8.2 Single or concurrent execution ... 119
8.3 Towards a CPU-free architecture ... 123
9 GPUs in the UNIX Way **128**
9.1 ML applications are pipelines ... 128
9.2 The UNIX way ... 130
9.3 Use-cases ... 131
10 Conclusions and Future Works **133**
10.1 Future works ... 135

List of Figures

2.1 Typical life cycle of a ML model. 8
2.2 Benefits of GPU-offloading compared to CPU longer processing time 11
2.3 Typical I/O workflows for training and inference applications..... 20
2.4 MLPerf® inference results for some selected systems..... 29
2.5 Pure inference throughput and time across a selection of models on an NVIDIA A100 GPU. 33
2.6 Estimated network bandwidth required by different models running on an NVIDIA A100 GPU. 34
2.7 Estimated network bandwidth required by different models, NVIDIA A100 GPU, with $8\times$ linear scale. 36
2.8 Estimated network bandwidth required by *superresolution* at different upscaling factors. 37

5.1 DGX H100 Topology, adapted from [1]. 56
5.2 Detail of the Cedar-Fever architecture..... 56

6.1 Performance of the 6 methods using a single core, under 2 different traffic scenarios..... 65
6.2 Number of cycles to insert and lookup entries, campus trace 16x, 2M entries..... 66
6.3 Scaling of the connection tracker using core-sharding..... 68
6.4 Scaling of connection tracker using locking techniques for a single hash-table 68
6.5 Comparison of deletion methods under an increasing garbage collection frequency running the Cuckoo implementation, single core 70

6.6 Scaling of two deletion techniques with sharded tables or a single lock-free Cuckoo table 70

7.1 Structure of a generic RDMA communication for a 1-sided verb..... 76

7.2 Data-flow for a RDMA Write verb. 78

7.3 RDMA vs. GPUDirect RDMA data flow 82

7.4 Multi-stream structure of our CPU-mediated inference serving prototype. 96

7.5 Event-based time measurement. 97

7.6 Structure of our benchmark application, GPU driven. 98

7.7 Maximum bandwidth reached by generating RDMA Write verbs on CPU. 99

7.8 Average time needed to post a Write verb. 100

7.9 Average number of CPU interrupts per second received when posting RDMA verbs. 102

7.10 Average number of CPU interrupts per second received when posting RDMA verbs from the CPU and GPU, when changing the frequency for CQE generation and retrieval. 103

7.11 Structure of our GPU-driven Inference Serving prototype. 105

7.12 Runtime contributions to the total inference latency when all processing is performed on GPU via an Unreliable RDMA connection. 106

7.13 Runtime contributions to the total inference latency when doing all processing on GPU via an Unreliable RDMA connection, with and without I/O memory copy kernels..... 106

7.14 Structure of our CPU-mediated Inference Serving prototype. 107

7.15 Time contributions to the total inference latency, when executing two different models with either GPU or CPU initiated networking and A100 GPU. Unreliable RDMA connection. 108

7.16 Time contributions to the total inference latency, under different transports and synchronization mechanisms. 109

7.17 Fractional CPU usage, over time, for two synchronization techniques and multiple workloads. 110

7.18 Time contributions to the total inference latency, when executing *denoise* with different memory allocation strategies. 112

8.1 Main components of the Clockwork architecture..... 114

8.2 Breakdown of client latency for Clockwork 116

8.3 Latency experienced by the clients, showing a 2× performance gain when using RDMA..... 117

8.4 Distribution of average inference rate and average inference time for some models on an NVIDIA A100 as a function of different levels of concurrency..... 121

8.5 Inference time distribution for *squeezenet_tuned* on an NVIDIA A100 across multiple concurrency levels..... 122

8.6 Data flow in Clockwork and our proposed architecture..... 124

9.1 Components of a typical GPU-accelerated application, with respective placement. 129

10.1 Overview architecture of an inference system, highlighting this thesis contributions. 134

List of Tables

5.1 Summary of the hardware used in this work. 52

5.2 Summary of the models used in this thesis. All inputs and outputs have a batch size of 1. 55

8.1 Testbed for assessing Clockwork performance. 115

List of Codes

1 C++ code implementing the *tail launch* mechanism..... 16

2 Pseudocode for a simple GPU-accelerated application, transmitting data with RDMA and traversing a GPU kernel..... 90

3 Pseudocode for a simple GPU-accelerated application, transmitting data with RDMA controlled by GPU. 91

4 Pseudocode for a simple GPU-accelerated application, transmitting data with RDMA controlled by GPU, using CUDA Graphs. 92

5 Pseudocode for a simple GPU-accelerated application, transmitting data with RDMA controlled by GPU, using CUDA Graphs and tail launching..... 93

6 Standard CUDA application structure: the CPU immediately *waits* for the GPU to complete workload. 95

7 *Interruptible-sleep* approach: synchronization is invoked only when the stop event has been generated. 95

List of acronyms and abbreviations

AI	Artificial Intelligence
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
BAR	Base Address Register
BF Register	Blue Flame Register
CAPEX	Capital Expenditure
CPU	Central Processing Unit
CQ	Completion Queue
DL	Deep Learning
DMA	Direct Memory Access
DNN	Deep Neuural Network
DPDK	Data Plane Development Kit
ECN	Explicit Congestion Notification
GP-GPU	General Purpose Graphics Processing Unit
GPU	Graphics Processing Unit
HCA	Host Channel Adapter
HPC	High Performance Computing
LB	Load Balancer
ML	Machine Learning
MMIO	Memory Mapped Input/Output
MMU	Memory Management Unit
MPI	Message Passing Interface
MR	Memory Region
NCCL	NVIDIA Collective Communication Library
NIC	Network Interface Card
NTP	Network Time Protocol
NUMA	Non-Uniform Memory Access
NVMe	Non-Volatile Memory express

NVSHMEM	NVIDIA OpenSHMEM Library
OPEX	Operating Expenses
OS	Operating System
OSFP	Octal Small Format Pluggable
P2P	Peer-To-Peer
PCIe	Peripheral Component Interconnect Express
PD	Protection Domain
PFC	Priority-based Flow Control
PTP	Precision Time Protocol
PTX	Parallel Thread Execution
PU	Processing Unit
QP	Queue Pair
RDMA	Remote Direct Memory Access
RoCE	RDMA over Converged Ethernet
ROCm	Radeon Open Compute
SAN	Storage Area Network
SLA	Service Level Agreement
SLO	Service Level Objective
SLOC	Source Lines of Code
SM	Streaming Multiprocessor
SoC	System on-a-Chip
SSD	Solid State Drive
TCO	Total Cost of Ownership
TDP	Thermal Design Power
TPU	Tensor Processing Unit
TVM	Apache TVM
UCX	Unified Communication - X Framework
UN	United Nations
VPI	Virtual Protocol Interfaces

Chapter 1

Introduction

The increasing amount of digitization in the world, together with the desire to explore and exploit this data, puts an increasing burden on networking technologies [2]. The rapid growth of processing capabilities in CPUs, GPUs, and accelerators in general, pushed the current computer science world in what has been called the *AI Spring* [3, 4], with Artificial Intelligence (AI) permeating more or more aspects of our daily life and provoking a huge impact in the computer science environment, both on the theoretical, ethical, and practical aspects.

Many companies already run huge volumes of inference every day [5, 6], which represents the major cost in many AI-based applications deployments [7, 8]. Most of these applications run on GPUs and commodity hardware, as modern trends suggest [9]. In these contexts, the ability of modern GPUs to complete inferences at *millisecond-scale* times poses interesting challenges compared with the *millisecond-scale* times to traverse a typical Linux Kernel Network stack, as already demonstrated in other work [10, 11]. This, along with the physical connectivity limits and processing power of modern CPUs, requires a shift in the application paradigms, going beyond the classic CPU-centric network stacks.

This thesis focuses on the optimization of the networking aspect of GPU-enabled applications, running on commodity hardware and in datacenter environments, with a particular attention to AI-driven workloads, which represent one of the most relevant workloads nowadays. Due to the increasing data volumes, the evolution of interconnecting speeds, and the increasing of processing speeds of GPUs, the cost of data-transfers in these scenarios is becoming more and more important, although it has traditionally received less attention than other aspects related to these technologies.

In particular, we focus on Ethernet-based infrastructures, running at speeds of

100 Gbps and faster, and NVIDIA CUDA GPUs. The latter represents the *de facto* standard runtime for at-scale AI workloads.

Our research began by looking at *stateful load balancing*, a necessary network function to be able to scale large services up, and down. The outcomes of this first part are reported in Chapter 6, and have been published as a conference paper [12].

Next, following the recent trends that have shown an explosive increase for Machine Learning (ML)-oriented tasks, both for data processing and generation, the thesis focuses on efficient methods to provide inferences, which we identified as a critical aspect in any ML application.

On this aspect, the contribution of this thesis is towards a system that can entirely bypass the CPU, exploiting commodity technologies that enable Network Interface Card (NIC) and GPU direct communication. We focus on an efficient method to send data directly from the GPU, allowing a *worker*'s GPU to directly send inference results without involving the CPU. This would result in lower system requirements, higher power efficiency, and lower round-trip latency for requests. We begin by examining the impact of networking overhead in State-of-The-Art AI systems, analyzing the common problems of these applications, and we propose an implementation to send responses directly from a GPU, bypassing the CPU stack.

We believe that our *RDMA from GPU* represents a key enabler for large-scale GPU driven applications, especially in datacenter environments, but also in edge computing facilities, where power budgets and costs are even tighter. In larger deployments, the ability to perform efficient *Connection Tracking* is a requirements for any *Load Balancing* function, and in general for any service that needs to maintain some state, such as a *front-end* for an inference serving infrastructure.

1.1 Thesis limitations

In this thesis, we will limit our research to *commodity* hardware and open-source platforms. We analyze the ML workload field from an NVIDIA CUDA point of view, which represents the current *de facto* choice for AI accelerators. Similarly, we limited our analysis to Ethernet networks, focusing on the broadly-available NVIDIA Mellanox Connect-X family of NICs. We studied the software-hardware interactions, and the dynamics behind the network stack, in a standard Linux environment, leveraging the standard libraries that are offered as part of the vendors' software support (*e.g.,* NVIDIA Mellanox OFED [13, 14] and CUDA [15]).

1.2 Thesis contributions

This work's main contribution is a novel approach to perform RDMA operations directly from the NVIDIA CUDA runtime.

This potentially allows any GPU workload, with particular focus on AI applications, to process network-bound data without any major CPU involvement. While the underlying GPUDirect-RDMA technology is broadly supported by many NVIDIA GPUs, it does not directly allow GPU programs to launch RDMA verbs,* but rather only to act as the memory involved by such verbs actions (*e.g.,* as a source or destination buffer).

Our approach allows such verbs to be *launched directly* from the GPU side, without any interaction with the CPU, both for data residing on the system's main memory and on the GPU memory, decoupling the verb-posting device from the device where the memory is physically allocated.

One major difference from other works in the area, such as the UCX [17] and NCCL [18] frameworks, is the decoupling from both the hardware vendor and the target applications. Both of these frameworks are mostly designed to run on homogeneous, purpose-built infrastructures, typical of High Performance Computing (HPC) environments, and fostering multi-GPU collaboration, where multiple GPU-equipped workers collaborate to achieve better performance, or solve bigger computation problems. This is commonly done using RDMA technologies on lossless network fabrics, and using out-of-band mechanisms to establish the communication channels.

Our contribution, instead, is proposed as a *drop-in replacement* for standard RDMA libraries (*e.g.,* `rdma-core` [14]), allowing any application to move the processing of RDMA transfers from CPU to GPU, potentially abstracting from a single-vendor, single-technology infrastructures, but rather fostering the use of heterogeneous computing resources, and allowing end-to-end communications between these devices without any active work from CPUs.

By moving the processing of verbs to the GPU runtime, we reduce the overhead involved in the synchronization between GPU and CPU, and we can potentially reduce the need of CPU cores to the initial, and periodic, house-keeping of resources, without any active involvement on the processing, nor on the data movement. This contributes both to save energy (by not having a power-hungry component), costs (by not purchasing expensive, large CPUs), and processing time (avoiding expensive transfers and synchronizations with the CPU).

Beside the inherent cluster-wide ability to expose GPU processing to other applications without active CPU involvement, other scenarios may include many

*A *RDMA verb* is an abstract description of the functionality of an RNIC Interface [16], and usually represents a data *transfer* that could be done over an RDMA connection.

real-time oriented tasks that are bandwidth intensive and which processing could be performed mostly on GPUs. Such applications are typically, but not exclusively, video-based such as (i) on demand transcoding services, (ii) remote rendering facilities, and (iii) cloud gaming platforms, where high-bitrate video should be processed, while guaranteeing the lowest possible latency.

In order to design, and develop our RDMA implementation, we have performed an in-depth analysis of some State-of-The-Art works, with a particular focus on Clockwork [19]. This allowed us to understand the needs, and the challenges of such a system, and explore the strategies that can be used to solve these, which resulted in the GPU-driven RDMA stack introduced above. We also implemented a standard, CPU-driven, RDMA transport in Clockwork, showing how the straightforward switch to this communication mechanism could reduce many of the bottlenecks in the architecture.

In addition to our *RDMA from GPU* approach, we study the performance of *Connection Tracking*, a fundamental ability to enable the scaling of any service outside *a single machine*, *i.e.,* as used in large systems front-ends.

1.3 Publications

This research project involved some collaborations that lead to three conference papers:

C1 "**High-speed Connection Tracking in Modern Servers**"
M. Girondi, M. Chiesa, and T. Barbette
IEEE 22nd International Conference on High Performance Switching and Routing (HPSR)

C2 "**Packet Order Matters! Improving Application Performance by Deliberately Delaying Packets**"
H. Ghasemirahni, T. Barbette, G. Katsikas, A. Farshin, **M. Girondi**, A. Roozbeh, M. Chiesa, G. Q. Maguire Jr., and D. Kostić
19th USENIX Symposium on Networked Systems Design and Implementation (NSDI)

C3 "**Toward GPU-centric networking on commodity hardware**"
M. Girondi, G. Q. Maguire Jr., and D. Kostić
Submitted for review at the 7th International Workshop on Edge Systems, Analytics and Networking (EdgeSys).

I am the first author of the first paper (**C1**) and all contributions related to the implementation, testing, and analysis of the results have been done by me. Tom

Barbette contributed significantly with the initial definition of the project and the continuous support of FastClick source code [20]. Marco Chiesa contributed with the definition of the problem, the background analysis, and the text of the article. The paper is included integrally in this thesis as Chapter 6.

The second paper (**C2**) is a collaboration with Hamid Ghasemirahni and the colleagues of NSLAB. My contribution for this paper concerned the deployment of a Virtual Network Function chain on a Mellanox Bluefield SmartNIC [21], and it has been discussed in detail in Hamid's licentiate thesis [22]. This contribution has not been included in this document.

The third contribution (**C3**) summarizes our *RDMA from GPU* implementation, and covers the content of Chapter 7. The co-authors Gerald Q. Maguire Jr. and Dejan Kostić helped with the text and the article structure. Additionally, Mariano Scazzariello helped in the initial implementation of the RDMA stack on the GPU.

1.4 Research sustainability and ethical aspects

In this section, we present the contributions of this research to the development goals established by the United Nations (UN), and we reflect on the ethical aspects of this thesis.

1.4.1 Sustainability

We believe that the outcomes of this work mainly contribute towards 3 domains of the UN sustainable development goals.

Environmental Sustainability The lower need for CPU cycles of the RDMA approach proposed in this thesis reduces the need for CPU-based computation in networking tasks. This would increase the share of CPU cycles that can be used by applications, reducing also the need for newer, faster CPUs, and consequently prolonging the life of existing hardware.

Economic Sustainability Reducing, or excluding, CPUs would in general reduce the power requirements to provide a given service, and thus reduce the total costs to run datacenters. Similarly to the previous point, service providers could potentially postpone hardware replacement, reducing both Capital Expenditure (CAPEX) and Operating Expenses (OPEX).

Social Sustainability We believe that reducing the costs of large AI applications deployments, and GPU-powered services in general, would allow more people to gain access to these technologies, potentially improving their quality of life and productivity. The availability of cheaper services would make them more affordable, and thus improving the equity in the society.

1.4.2 Ethical aspects

This work does not raise significant ethical problems.

To enhance the transparency and integrity of our results, we plan to release all source code developed for this work, fostering collaboration and encouraging other researchers to reproduce and verify our results. We encourage the community to take advantage of our findings and contributions to realize better and more efficient systems.

There are large contrasting opinions on some ethical aspects of ML and AI, mainly related to datasets used for training and the use of these applications' outputs [23–25]. We believe that the work presented in this thesis is not affected by these problems but provides an orthogonal technological improvement to better support any of these applications.

We invite the users of the contributions of this thesis to critically reason about the ethical aspects of their systems, and the impact these would have in the society, conducting an in-depth analysis of these problems.

The results reported in Chapter 6 have been collected by replaying some real traffic traces, striving to provide a realistic analysis of a practical application:

CAIDA: This trace has been collected, and made available, by the Center for Applied Internet Data Analysis based at the University of California's San Diego Supercomputer Center [26]. The traffic represented by these traces has been collected on a commercial backbone link and has been anonymized by the authors to reduce any possibility of user, or payload, identification.

Campus: This represents a packet trace that has been captured at one of our campus' main routers. To enforce confidentiality of the data, we have rewritten all IP addresses with anonymous values and removed the body of the packets. In order not to introduce any bias, we strove not to analyze the dynamics or patterns of this packet trace.

1.5 Thesis structure

Chapter 2 presents the necessary background to put this thesis in context. Chapter 3 summarize some related works and State-of-The-Art systems, which allows us to identify the problems and research questions presented in Chapter 4. Chapter 5 describes the main characteristics of the hardware setup used to analyze the results of this work. Chapter 6 presents a study on Connection Tracking mechanism for high-speed networked systems, while in Chapter 7 our implementation of a GPU-driven RDMA stack is proposed. Chapter 8 and Chapter 9 present two use cases for our building block. Finally, Chapter 10 draws the conclusions and outlines some future research direction to continue this thesis path.

Chapter 2

Background

This chapter provides the reader with an essential background for this thesis. Section 2.1 introduces the essential concepts about AI and ML, Section 2.2 presents an overview of the relevant execution platforms used for AI workloads, with a particular focus on the CUDA architecture. Section 2.3 gives an overview on the Peripheral Component Interconnect Express (PCIe) subdomain, Section 2.4 discusses current trends in datacenter connectivity. In Section 2.5 we give a summary of Apache TVM (TVM), the ML framework used in this thesis to evaluate the performance of ML applications. Section 2.6 reports an analysis of modern GPUs performance, while Section 2.7 provides some discussion on networking costs. Finally, Section 2.8 and Section 2.9 give an evaluation of the raw GPU performance for the hardware used in this thesis.

2.1 AI and ML

The umbrella term AI describes the area of science and engineering that concerns the construction of *intelligent entities*, in particular machines and software, as opposed to the intelligence of humans and animals.

Although AI has been founded as an academic discipline in 1956 [27], it has reached major importance only in the last decade, when *deep learning* techniques' performance have surpassed all previous methods, thanks to algorithmic improvements, faster processing speeds, and the availability of large dataset to *train* models.

In recent years, this has become one of the most strategic technologies, with many applications based on, or assisted by, the output of AI agents, with a particular focus on *ML* algorithms. ML is the branch of AI that is concerned with the development of programs that can automatically improve their performance (*e.g.,*

precision or accuracy) through a phase of *training*, binding closely *Computer Science* and *Statistics*, to deliver programs whose output is *as precise as necessary* to a desirable target level of performance.

Although the first fundamentals for Artificial Neural Networks have been proposed more than one century ago [28], only the recent processing capabilities offered by dedicated accelerators, mainly General Purpose Graphics Processing Units (GP-GPUs), allowed these methods to be used with satisfactory performance and reasonable hardware requirements.

We refer the interested reader to [29] for a general overview of the ML field, while [30] presents a survey of many commonly used Deep Learning (DL) methods.

2.1.1 What is behind an inference

While this thesis is concerned with the interaction of GPUs and NICs, this is evaluated in the perspective of a ML model serving scenario. Therefore, it is important to understand the foundations of the computation behind a ML inference.

Typically, all ML models (and DL in particular) follow a similar life cycle, shown in Figure 2.1: (i) *Model design*, (ii) *training*, and (iii) *inference*.

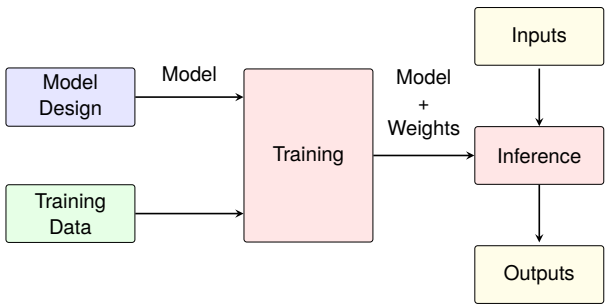


Figure 2.1: Typical life cycle of a ML model.

The first phase involves the appropriate choice of the model structure, the mathematical functions used, and the sizes of these (*e.g.,* the *width* of the model). There is a vast literature on the different techniques that can be used [30, 31]. Often, the choice of the model's structure and complexity depends on the type of application and the desired speed and accuracy.

After an initial model has been designed, a *training* phase follows, during which the model's *weights* are repeatedly adjusted to reach a good minimum point for errors produced by the models. This phase is usually time-consuming; with very large models, this training phase can last for months and usually involves large-scale systems whose hardware architecture has been designed to maximize

the performance for this specific task. For reference, it has been estimated that OpenAI's GPT-3 training took 34 days [6, 32]. Similar State-of-The-Art models have also been trained on comparable time scales [33, 34].

Finally, after a model has been *designed* and *trained*, applications use the model by submitting inputs, which go through the sequence of *layers* of the model. Ideally, the results of the model should be very close to the desired output. This application of the model on the inputs to produce outputs is referred to as *inferencing*.

The motivation and possible use cases for the contributions presented in this thesis will focus on inference serving and the technology behind the execution of inference, as these aspects of applying ML have received less interest from the research world, particularly when large-scale systems are involved. For most models, the deployment of the model in production produces economic & social value and will require even more resources than the earlier phases. For example, early *back-of-the-envelope* calculations on OpenAI's ChatGPT, estimated the daily costs to be on the scale of US$100 000, with a volume of 10 million queries/day [2, 35, 36]. We note that the volume of requests has increased by two orders of magnitude by January 2023, and ChatGPT introduced a paid service to be more economically sustainable [37, 38].

2.1.1.1 At the core of inference

Abstracting away many details, a DL inference, at its core, is a sequence of mathematical operations, often involving *vectors* and *matrices*.* Inputs to the models are encoded in some numeric form (typically, in a *floating point* representation) through a preprocessing phase. Similarly, outputs are mapped to the desired output type (*e.g.,* strings, labels, images, …) through a post-processing phase.

In many cases, the mathematical operations involved in the inference are highly deterministic, with little or no state maintained by the model architecture. This allows applications to execute DL models by interacting with DL frameworks through simple Application Programming Interfaces (APIs) without the need to know the details of the implementation, and abstracting the implementation details from the consumer application perspective.

Due to the parallel nature of the mathematical operations used during inferences, which are mainly matrix multiply operations, GPUs and other parallel processors have emerged as the main execution platforms for these models. To improve even more the performance of these devices, most of them include specific hardware components designed to achieve better throughput without sacrificing precision and often achieving higher power efficiency. We discuss AI accelerators, and GPUs in more detail in Section 2.2.

*This concept can also be extended to other ML methods.

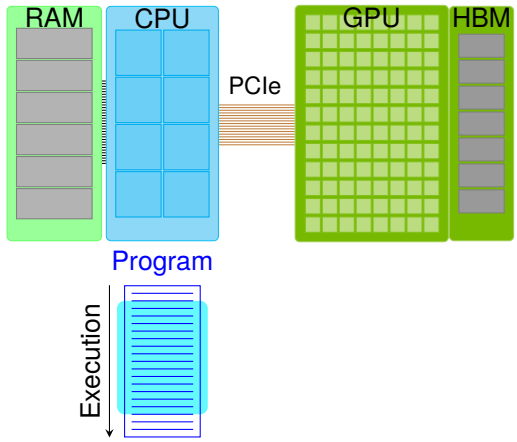
2.2 AI accelerators

In recent years, many companies have developed specialized hardware to support the emergent AI workloads, providing specialized hardware accelerators targeted to the specific mathematical operations that form most of these workloads, which can in most cases be executed in parallel. CPUs are rarely the most optimized target for this type of operations: virtually any at-scale AI workload leverages one, or more, AI hardware accelerators.

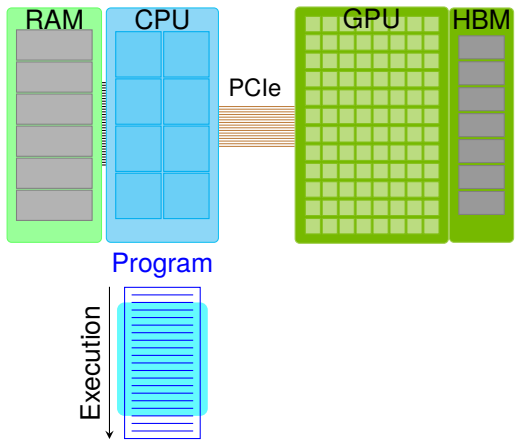
Among the other accelerators that have been developed over the years, GP-GPUs stand out as the dominant type of accelerators, with NVIDIA CUDA architecture representing the heavy share of the sector [39–41]. We treat the CUDA architecture, which represents the target platform of this work, in Section 2.2.2, while we provide a general overview on other AI accelerators in Section 2.2.3.

2.2.1 Typical accelerator offload processing

GPUs, and other specific DL hardware components, are often used as *offload accelerators* in an asynchronous fashion: applications can continue to process other data or perform other tasks while inferences are run on dedicated hardware, as shown in Figure 2.2. This allows scaling accelerators and CPUs independently for each application, according to the specific requirements and the type of workload needed (*e.g.,* more intensive preprocessing, hybrid architectures, or memory-bounded models). We note that most of these accelerators are specifically optimized for running ML operations and would be severely limited for general-purpose programming [42], *de facto* forcing these applications to run on heterogeneous platforms, where some operations are performed on general-purpose CPUs, putting even more importance in the communication mechanisms that are used to exchange data between the different devices involved.



(a) CPU Only



(b) CPU + GPU

Figure 2.2: Benefits of GPU-offloading: highly-parallelizable applications can run faster and complete in a shorter time. During offloading, the CPU can execute other tasks.

2.2.2 The CUDA architecture

Among the different architectures available for GP-GPU, the NVIDIA CUDA architecture maintains a stable and growing share of the sector [39–41] with many applications explicitly developed and optimized for this architecture.

The release of the GeForce 3 series by NVIDIA in 2006 represents the start of the *general-purpose GPUs* sector, with the first components *ad-hoc* designed to perform generic computation, as opposed to the *graphic-oriented*

APIs and architectures available in previous generations. These newer GPUs featured not only the usual graphic components (*e.g.,* to render video-games, the practically dominant market at the time), but also to perform floating-point calculations and arbitrary memory accesses. To exploit these hardware features, NVIDIA released the CUDA programming language, which extends industry standard C with specific instructions to use all newer hardware features, which represented the first programming language designed to facilitate general-purpose GPU programming [43].

The execution model of GPUs differs from the traditional execution model of CPUs, which usually features a larger number of threads to execute each instruction, rather than one or two as found in traditional CPUs [44].

The CUDA execution unit is represented by *CUDA cores*, which are grouped in *Streaming Multiprocessors (SMs)*. Each SM would execute a specific workload, as configured by the programmer through the software-level abstractions of *Threads* and *Grids*. CUDA cores inside each SM are further grouped into fixed subsets called *Warps*, with each *thread* scheduled to run on one or more of these warps.

Since the CUDA architecture does not implement any branch prediction or out-of-order execution, and all cores in a warp execute the same instruction at a given time, the maximum efficiency is reached when all 32 threads of a warp follow the same execution path [45]. In other words, every GPU program should be as similar as possible for all threads composing a warp, consequently reducing all possible branching by avoiding any if/else statements that would increase this branching.

Current CUDA GPUs offer specific hardware accelerators to improve the performance of specific tasks. The most notable of these, *Tensor Cores* [39] are highly relevant for AI applications, as they execute matrix operations (in the form of $D = a * B + C$) faster than the SM cores. Additionally, the mixed precision computations supported on these accelerators and the narrower data types available on the latest generations allow a multiplicative increase in some applications' performance while maintaining sufficient precision in the final results and have been proven to be effective for many AI related tasks.

Other accelerators worth noting are (i) *Memory Copy Engines*, which are used by default to move data between the GPU memory and other devices (either via PCIe Peer-To-Peer (P2P) transactions or to the main memory), and (ii) NVENC/NVDEC engines, used in video-related tasks to transcode video formats.

2.2.2.1 CUDA GPU connectivity

GPUs are usually not deployed on their own, but rather are installed in a host machine that has the role of initializing, orchestrating, and issuing workloads to them. At the datacenter scale, GPUs are often inserted in x86 server machines

and interconnected through the PCIe bus, which represents the *de facto* industry-standard for both server, desktop, and edge devices.

Current GPUs (*e.g.,* NVIDIA H100 [46]) supports the last PCIe release, *i.e.,* PCIe 5.0. With an ideal bus width of 16 lanes, this would allow the GPU to interact with the CPU, and other PCIe peripherals, with up to 512 Gbps of bidirectional bandwidth. Following the last decade of PCIe evolution, these speeds are expected to double for each new generation, with PCIe 6.0 specifications already published [47], and devices supporting it are expected to be released next year.

We discuss some relevant characteristics of PCIe in Section 2.3.

NVLink Beside the standard PCIe bus, NVIDIA has also developed a family of proprietary interconnect solutions, evolving from Scalable Link Interface (SLI) to NVLink and NVSwitch [48], with the goal to maximize the performance of multi-GPU workloads. The first iteration, SLI, was originally designed for graphic tasks, and has been replaced by NVLink, both in consumer and in server GPUs. NVLink, now upgraded to its fourth release, represents the primary means of communication between multiple GPUs cooperating to deliver a common workload, both within a single chassis and between multiple machines, as supported in its latest release.

NVLink has been designed as a point-to-point protocol, similar to PCIe, where each device has a direct physical connection with another device, but has been later expanded to support switches to increase the possible communication schemes that can be realized between GPUs.

The current NVLink 4 connectivity present on last-generation NVIDIA H100 GPUs, can transfer up to 7 200 Gbps, as opposed to the maximum theoretical bandwidth achievable on its PCIe 5.0 x16 bus of 512 Gbps,* which can be directly served from the HBM2 on-board memory. This memory is rated for over 25 Tbps of bandwidth [39, 46]. NVLink has also been used in HPC environments to interconnect PowerPC CPUs [49], and is the basis of the Grace-Hopper architecture, a hybrid CPU-GPU chip [50, 51].

NVSwitch Beside the intrinsic higher inter-connectivity speeds for GPU-to-GPU communication, and the capacity of operating as an all-to-all mesh fabric instead of a point-to-point architecture, the *NVSwitch* provides acceleration of some collective operations,† speeding-up *All-Reduce* executions, among other MPI operations, as part of the bigger NVIDIA SHARP technology [52]. These accelerations further reduces the load on each GPU participating in the collective operation, while at the

*A PCIe 5.0 lane can perform 32 GT/s, which translates to a raw, un-encoded, bit-rate of 512 Gbps.

†We refer to collective operations as the communication mechanisms commonly used in HPC and MPI environments, where (typically many) machines communicate to solve a bigger problem by periodically exchanging intermediate results.

same time reducing the total amount of data that needs to be transferred across the NVSwitch network. While the initial generations of NVSwitch have been designed as internal server components, interconnecting devices inside a single chassis, with the third generation this capability has been extended to rack-scale systems [53].

Fitting in a single rack unit, *NVLink Switches* interconnect up to 32 endpoints NVLink-enabled endpoints (*i.e.,* 32 GPUs). Each of the 32 ports on the switch carries 4 NVLink lanes, each able to transport up to 125 Gbps of bidirectional traffic, designed to be connected to NVIDIA DGX-H100's Cedar Fever ports.

2.2.2.2 CUDA streams

Given the highly-parallel architecture of GPUs, having control on how different tasks are executed simultaneously represents a key characteristics of an efficient system. The CUDA runtime provides these mechanisms through *CUDA streams*, which abstract a queue of actions to be executed. *CUDA streams* ensure that tasks are run in the strict order they have been *posted*. By default, every *kernel* is associated with a default stream (*stream 0*), but developers can create multiple streams, which can run in parallel (*e.g.,* in a *time-sharing* fashion), maintaining the execution orders inside each stream.

Other *asynchronous* operations can be associated with streams, such as memory copies, and specific synchronization mechanisms are provided to both *exhaust* the work in a stream and *pause* a stream until some event happens (*e.g.,* when the previous stage is completed in a multi-stage application.

We refer the reader to [54] for a detailed description of this mechanism. We analyze the advantages of multiple, concurrent, streams in Section 8.2.

2.2.2.3 CUDA Graphs

GPU workloads are usually composed of many *kernels*, *i.e.,* small workload units executed in sequence. For instance, ML models are usually translated with a 1-to-1 association between *layers* (or *operators*) to *kernels*, all executed in sequence [55]. Execution times of these *kernels* can often be in the order of microseconds, the same time-scale required by the runtime APIs to transfer the workload to the GPU to be executed [56, 57] (*e.g.,* the binary code to be executed needs to be transferred to the GPU physical memory, potentially with the associated data). Through *CUDA Graphs*, a sequence of GPU operations can be *captured*, instantiated once, and played in sequence. This *sequence* is launched as a single operation, thus greatly reducing the API overheads introduced by these transfers, and could generally complete quicker than single-launched kernels. In addition to kernels, other operations can be part of CUDA Graphs, such as host functions, synchronization

blocks, or memory copy operations, which are guaranteed to be always executed in the same order (*e.g.,* as they have been captured).

CUDA Graphs are generally launched similarly to stand-alone kernels, and would execute asynchronously until a synchronization point is encountered by the runtime (*e.g.,* a `CudaDeviceSynchronize`). However, under certain conditions,* it is possible to instantiate self-calling Graphs, which would execute continuously by self-enqueueing themselves in the CUDA runtime, without any direct CPU involvement at runtime (*e.g.,* to issue each individual kernel). We call this mechanism *tail launch* and we use it implement our contribution described in Section 7.5.8. Listing 1 reports an example on how this mechanism translates in C++ code.

*More precisely, when the Graph is instantiated with the `cudaGraphInstantiateFlagDeviceLaunch` flag, it involves a single CUDA device, and only some types of memory operations are used [58].

```
__global__ void foo(executor_data * data) {}
__global__ void bar(executor_data * data) {}

__global__ void looper(executor_data *data) {
auto g = cudaGetCurrentGraphExec();
if (g && !data->stop) {
   int ret = cudaGraphLaunch(g, cudaStreamGraphTailLaunch);
   }
}

void executor(executor_data * data)
{
  cudaGraph_t graph;
  cudaGraphExec_t instance;

  // Build the graph with all the operations
  cudaStreamBeginCapture(data->stream, cudaStreamCaptureModeGlobal);
  // Add some operations
  foo<data->blocks, data->threads, 0, data->stream>>>(data);
  bar<data->blocks, data->threads, 0, data->stream>>>(data);
  // ... and the looping function (for self-launch)
  looper<1,1, 0, data->stream>>>(data);
  cudaStreamEndCapture(data->stream, &graph);

  // Instantiate the graph
  cudaGraphInstantiate(&instance, graph,
        cudaGraphInstantiateFlagDeviceLaunch);

  // Run the graph in a specific CUDA stream
  cudaGraphLaunch(instance, data->stream);
}
```

Listing 1: C++ code implementing the *tail launch* mechanism. `foo` and `bar` represent two generic functions. The graph would run asynchronously until the `data->stop` variable would be set to false, *e.g.,* after a kill signal, invoking the two functions `foo` and `bar` continuously.

2.2.2.4 Advanced memory management in CUDA

Traditionally, GPU-aware applications allocate memory in distinct areas for GPU and CPU processing, relying on primitives exposed by the GPU runtime driver to move data between them. In CUDA, this is typically done via calls to `cuMemcpy`, which offloads the copying task to dedicated accelerators (so-called *Memory Copy*

engines). This need to explicit request areas of memory to be copied across devices introduces a further complexity for developers, that need to understand, and embrace, the different type of memories and their location. To remove this limitation, starting with CUDA 6 and Kepler class GPUs, *Unified Memory* has been introduced, allowing (i) GPUs and CPUs to share the same address space and (ii) a single memory allocation to be visible from both the CPU and the GPU, without any need to explicitly copying memory.

CUDA Managed Memory The first type of memory allocation that benefits from *Unified Memory* is *CUDA Managed Memory*, as allocated through `cudaMallocManaged`. The underlying mechanism is implemented with a paging system, with these pages moved closer to the processor using the memory. While this simplifies the general programming (*e.g.,* the applications do not need to explicitly call `cuMemcpy`), it introduces more unpredictable performance, especially when data are continuously accessed by both processors, which would provoke an enormous number of page faults and constant page movement between the two sides of the PCIe bus. Thus, managed memory must be used wisely to avoid destroying performance due to page movement.

Pinned Memory Another approach to provide a unified memory allocation to application is *Page-Locked Host Memory*, also known as *Pinned Memory*. When *page-locking* a range of host memory (*e.g.,* fixing the virtual to physical mapping), the GPU device can directly access this area. While this mechanism provides the best performance for inter-device memory transfers, when large areas of memory are allocated the global system performance can quickly degrade and should be carefully used [59].

The main drawback of this approach is the continuous PCIe bus traversal for every operation, which would increase each access operation latency. For example, an application accessing a counter of Page-Locked Host Memory 1 000 times would cause 1 000 read/write transactions over the PCIe bus, while when using unified memory (and assuming that the CPU is not concurrently accessing the same counter), this would optimistically cause only two transactions: one at the beginning of the execution and the other when the CPU needs to read this counter.

GDRCopy Leveraging the P2P capability of modern GPUs, GDRCopy [60, 61] improves the performance of CPU-GPU copies, using more CPU cycles to save *GPU's Memory Engines* actions, while both reducing the latency and improving the throughput of these copies. In this implementation, the CPU acts as a DMA controller, moving the data to, or from, the GPU without involving the specialized hardware components on the GPU.

2.2.3 NVIDIA is not the only player

While NVIDIA is considered the top player in the GPU market [41], the CUDA architecture is not the only accelerator technology that can be exploited by ML and other scientific workloads. For example, AMD, with its CDNA hardware architecture [62], offers similar characteristics, with a comparable software runtime, Radeon Open Compute (ROCm™) [63], and similar interconnect technologies [64], both for GPU interconnection and RDMA support. Major AI frameworks [65,66] also support other runtime targets, but these are usually either low-power edge devices (*e.g.,* mobile GPUs, microcontrollers or other low-power embedded devices), or massive-computing accelerators, mainly focused on large-scale training and processing (*i.e.,* in an HPC setting).

Various vendors have made major efforts to deliver specialized chips specifically optimized for AI tasks, partially by moving to domain-specific architectures [67, 68].* For example, Google's proprietary Tensor Processing Unit (TPU) accelerators [69] represent the first type of dedicated hardware made available to the general public in a cloud platform. Currently, TPUs power most of Google's DL training workloads. TPUs have now been updated to their 4th generation, with *ad-hoc* built-datacenters to better support these platforms, also available for external use through public cloud services [70]. Other major efforts have resulted in the Cerebras CS-2 platform (which features the currently largest single-wafer chip ever produced) [71], Graphcore IPU [72], SambaNova DataScale [73], and the Groq GroqChip [74]. Although all of these systems support the most popular DL frameworks (*e.g.,* TensorFlow [75], PyTorch [76], and MXNet [77]), and offer potentially orders of magnitude better performance, due to their limited availability for the public and their generally higher price,† these cannot be considered commodity hardware. Moreover, the few deployments mainly focus on model training rather than inference. We refer the interested readers to [68] for a complete review of these platforms.

On the other side of the spectrum, many vendors are improving edge CPUs and GPUs with AI capabilities, improving the AI performance of mobile and end-user devices. Intel is starting to ship AI accelerators in common CPUs [78], with server-grade Xeon CPUs already implementing built-in accelerators and specific AI-oriented instructions [79]. The latest Intel Xeon MAX family [80] embeds HBM2 memories directly in the CPU's package, with specific optimizations for AI applications. AMD is already deploying dedicated accelerators in laptop CPUs [81]. Apple is doing something similar [82]. This same trend is followed by most mobile device vendors, with Apple [82], Samsung [83], Google [84], and

*However, this requires large investments and is hard to sustain (as this requires continuing large investments).

†For most of these components, no price is published.

Qualcomm [85], among others, embedding such accelerators in the System on-a-Chip (SoC) used in their smartphones.

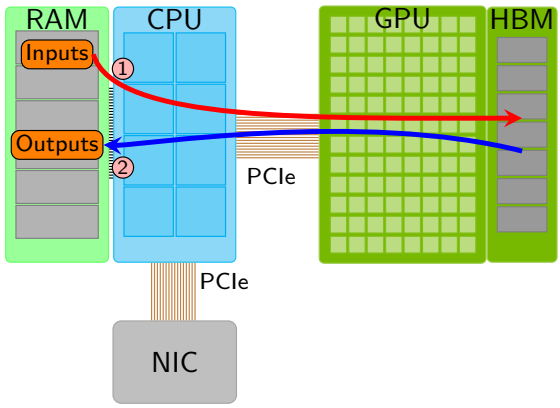
Although general software support is usually provided (*e.g.,* via ONNX [65], TVM [66]) or directly by some popular ML frameworks [86, 87], their limited performance, together with the single-user and device-local inferences, excludes these devices from datacenter-style applications. As a result, practical large-scale inference deployments are mainly based on NVIDIA CUDA GPUs [41].

2.2.4 GPUs as machine learning accelerators

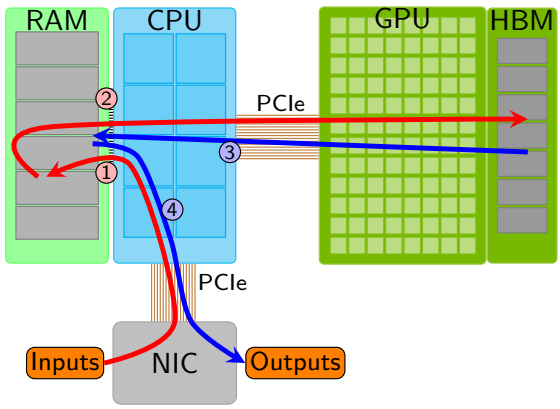
Typically, GPUs have been used to accelerate part of the computation by moving the burden of it from the CPU cores to an external device, usually interconnected via a PCIe bus of the host system. The application needs to transfer inputs (doing a so-called *Host-To-Device copy*) through this interconnect and then instructs the GPU hardware what operations to execute (*e.g.,* loading the kernel code), and finally, the CPU copies back the result. These operations are typically sequential and asynchronous, meaning that CPU can soon execute other instructions (*e.g.,* pre-processing the next batch of data) while waiting for the results to be ready.

While this execution model is suitable for many applications where data is resident (or preloaded) in the system's memory (*e.g.,* for ML training tasks), this execution model introduces many potential limitations when data is arriving (or leaving) the system over the network or via other peripherals. As can be seen in Figure 2.3a, this approach requires *at least* four memory copies between the different devices, with more introduced as soon as any pre- or post-processing needs to be queued up in the pipeline.

Although most GPUs (and AI accelerators in general) support means of transferring data between parties at low latency and with low CPU overhead, they are seldom used for inference applications, mainly because they require specific hardware [88] (*e.g.,* PCIe switches) and due to the lack of plug-and-play support in common ML frameworks.



(a) Training/local Inference



(b) Serving Inference

Figure 2.3: Typical I/O workflows for training and inference applications

2.3 PCIe

PCIe [89] is an expansion bus found in the vast majority of modern servers,* with very few exceptions, and it is used to connect the CPUs with most of the system's peripherals. Although modeled after the older bus-based PCI architecture, PCIe has evolved as a high-speed, point-to-point connection, with direct serial links between every device and the host system, defined as the *Room Complex*. Communications on the PCIe interface occur through packets in full duplex mode, controlled by the

*More specifically, it can be found in virtually any x86 platform, and also in many ARM, PowerPC, and RISC-V systems.

transaction layer of the PCIe protocol.

PCIe devices can establish connections with the Root Complex at different speeds by using multiple *lanes*. The maximum bandwidth of the link scales linearly with the number of lanes used and lanes can be allocated as powers of two (*i.e.,* 1x, 2x, 4x, 8x, and 16x*), with packet data spread across multiple lanes to increase peak throughput. The number of lanes used, and their speed, are negotiated during the port initialization, with the possibility of devices to downgrade the link speed, or downsize the bus width, improving the resilience when unreliable lanes are present. This flexibility allows the PCIe bus to be used by a large set of peripherals, both supporting low-requirement devices and high-speed, low-latency ones, choosing the best link size, and speed, for each use case.

This auto-negotiation feature also allows backward compatibility with older devices, both when using a newer peripheral in an older system (*e.g.,* a PCIe 4.0 NIC is supposed to work on a PCIe 3.0 slot), and when older devices are used in a newer system (*e.g.,* a PCIe 4.0 server with a PCIe 3.0 NIC). In these cases, as the PCIe subsystem is organized as a tree, all endpoints will work at the maximum agreed speed and width, with only the links with a mismatching characteristics being downgraded.

Electrically, each lane is composed of a differential signaling path, which is implemented as a bonded serial bus, as opposed to a traditional parallel bus, thus avoiding the time skew typical of parallel connections, which would limit the bandwidth. Following the last decade of PCIe evolution, the speeds of PCIe lanes are doubling with each new generation, with the PCIe 6.0 specifications already published [47], and devices supporting these specifications are expected to be released this year (*i.e.,* 2024).

At technical level, the bandwidth of PCIe devices is measured in *transfers per second*, which refers to the raw, un-encoded, data rate of the bus. The data to be transferred are then encoded with a line code to provide enough clock recovery to the receivers, resulting in a lower effective bit rate. PCIe 1.0 and 2.0 used 8b/10b encoding, PCIe 3.0, 4.0 and 5.0 use 128b/130b, improving the efficiency of the channel, while future generations will use a 242B/256B FLIT encoding schema.

The current commercially available generation of PCIe 5.0 supports a rate of 32 GT/s per lane, which translates to a maximum raw bandwidth of 504.12 Gbps on a 16x PCIe slot in each direction. However, the effective data rate that can be achieved could be severely limited by the access pattern, especially when small data packets are transferred across the bus [90].

*32x and larger links are also possible but not available in most systems.

2.3.1 PCIe topology

The standard PCIe-equipped platform usually implements the PCIe subdomain as a tree.* The CPU's *Root Complex* represents the tree's root. Typically, each device in the PCIe domain is directly connected to one endpoint in the Root Complex, without any bandwidth contention with other devices. To augment the flexibility of the system, increase the number of available lanes, and potentially achieve better performance, PCIe-switches can be used.

Similarly to an *Ethernet Switch*, a PCIe switch routes packets between the different endpoints connected to it, allowing *direct P2P transactions* between these endpoints. These P2P transactions allow data to be transferred between endpoints *without* traversing the Root Complex, which should reduce the latency **and** increase the data throughput. It also has the advantage of not consuming bandwidth on the PCIe endpoints that do not participate in the P2P transaction, allowing greater scaling of systems (*i.e.,* by oversubscribing the CPU's PCIe lanes) and potentially offering better power efficiency.

PCIe P2P transactions require both support from the hardware devices (*e.g.,* by GPUs, Solid State Drives (SSDs), and NICs) and from the software stacks, typically by specific kernel modules (in the Linux environment) that set up the devices' driver allowing access to each other's memory areas. These memory areas are later used to exchange data between the devices.

In this work, we heavily rely on PCIe switches and P2P transactions to support our GPU-NIC interactions, as described in details in Section 7.4.1.

2.3.2 PCIe DMA

The performance of modern systems is highly dependent on Direct Memory Access (DMA). DMA frees up the CPU's resource by offloading data transfers to a *DMA engine*, which is instructed by the CPU (or in our case by the GPU) to take care of the data transfers. By shifting this workload from the CPU, the CPU can execute some other operations during the time that the transfer is performed. After the transfer, the DMA engine interrupts the CPU when the transfer has finished. When PCIe P2P transactions are used, the devices' DMA engine performs the memory transfer operations, reading and writing to the other devices' memory regions. This occurs completely without needing any CPU cycles.

We refer the interested reader to [91] for an in-depth analysis of how drivers handle DMA in the Linux system, while [92, 92] provide a good discussion of the DMA mechanisms.

*Or as multiple trees, when multiple CPUs are present.

2.3.3 Extending DMA over the network: RDMA

Extending the DMA mechanisms over the network, it becomes possible to allow remote networked devices to write to each other's memory without (or with only minimal) CPU intervention. We call this family of technologies RDMA. RDMA is made possible by specific functionalities embedded in modern NICs, or by specific software stacks.* In RDMA, data transfers are programmed by the CPU, while the network adapter performs the actual data transfers using dedicated embedded processors, either to or from a remote network endpoint. We refer to each of these operations as a *verb*, and most of the verbs can be directly mapped to a single DMA transaction over the PCIe bus. The most common network protocols supporting RDMA are InfiniBand and Ethernet, the latter being the focus of this thesis. We note that most RDMA concepts are in common between Ethernet, InfiniBand, and other protocols implementations, as the CPU and memory operations to which the network *verbs* are translated are very similar between all stacks.

We give a general overview of RDMA in Section 2.4.1, while the relevant technical details used in this work are presented in Chapter 7. We refer the interested reader to [93] for a more in-depth introduction to the technology.

2.4 Connectivity in the datacenter

All computers are connected together to build larger systems, with *The Internet* being the largest of these systems. The boundary between the *network* and the *internal system* is usually identified by the NIC, a hardware device (or rather a PCIe-attached card, especially in server environments) with the purpose of mediating the communications between the physical network layer, typically *Ethernet*.

NICs are usually equipped with one or more processing units that handle these translations and perform DMA access to the system's main memory, reducing the CPU overhead for network communications. Beside the basic DMA functionalities, almost any NICs used in datacenter deployments employ some type of optimization to transfer data more efficiently, either by leveraging modern CPU functionalities (*e.g.,* CPU caches [94]), PCIe bus functionalities, or by working in cooperation with more advanced software drivers.

Traditionally, Linux-based operating systems handle networking at a kernel level: every time an application needs to send data to some peer over the network (i), the kernel gets informed about the incoming data transfer, (ii) the software driver programs the DMA engine in the NIC and (iii) the data transfer follows. A

*In the case of software RDMA, the benefits of the CPU usage reduction are limited to only one side of the data channel.

similar approach is followed when packets arrive from the network, with the NIC notifying the operating system about the incoming packet, typically via an interrupt.

Although this interrupt-based approach has been shown to provide good performance for general applications, the scale of line rates of modern network deployments introduced the need for more advanced approaches, such as NetMap [95] and DPDK [96], aiming to reduce latencies and the number of CPU cycles needed to handle CPU interrupts.

Both of these frameworks leverage a polling mechanism, with the NIC continuously polled from the user-space program, avoiding the typical context-switching of kernel-based traditional applications, reducing the overall latencies and the CPU cycles needed to process each network packet.

This technique relies on the assumption that, at high loads, the CPU would always have packets to process, and thus it becomes more efficient to busy-wait (*i.e.,* wasting CPU cycles) packets rather than receiving interrupts, since the time lost waiting for a packet to be ready in the NIC would be in general low.

Another approach, which is the main technology that powers this work, is represented by RDMA, already introduced in Section 2.3.3.

2.4.1 RDMA in the datacenter

RDMA is the commercial implementation of the ideas proposed in 1995 by Von Eicken *et al.,* [97], who proposed the first concepts behind the technology. Initially targeted to HPC clusters and used to increase the performance of large distributed computations, RDMA has gradually evolved and has been integrated in many datacenter applications.

The core idea of RDMA is the ability to access (*e.g.,* read and write) remote hosts' main memory, through an extension of some DMA operations. These are mapped to RDMA's *verbs*, which are transmitted over the network. A more detailed description of *verbs* is given in Section 7.1.

The main appeal for this transition from classic network stacks to RDMA-based applications is the lack of context switching (*i.e.,* the kernel is not involved in data transfers), the wide availability of RDMA-capable hardware (*i.e.,* NICs), and specific additions to the network physical protocols to support this type of communication. Thanks to these additions, RDMA (and in particular RDMA over Converged Ethernet (RoCE)) has also been shown to provide good performance over regular network infrastructures lacking typical guarantees of HPC fabrics (*i.e.,* lossy network) [98, 99].

Software support for RDMA has been introduced in many common datacenter applications (*e.g.,* NFS, RPC, Spark, Ceph, SMB, NVMeOF, …), and RDMA is supported by multiple physical implementations, such as RoCE [100], Infini-

Band [101], iWarp [102], and OmniPath [103]. Most commodity datacenter NICs [104] provide support for some RDMA stack, such as NVIDIA's Mellanox Connect-X family [105], Intel cards [106], Chelsio Terminator [107], Marvell FastLinQ [108], Broadcom [109], among others.

In this work, the focus is on NVIDIA Mellanox Connect-X cards, which are widely adopted in the datacenter market, feature in many similar research works, and used in all NVIDIA-branded servers (*e.g.,* the NVIDIA DGX family). The latter are specifically designed to run AI workloads and are starting to be widely adopted in inferencing serving workloads.

As mentioned above, many physical and transport layers can be used to provide RDMA functionalities, with InfiniBand representing for many years the main technology used in modern large clusters [110]. However, with the release of additions to Ethernet physical layer to provide *lossless* functionalities (which are intrinsic in the InfiniBand fabrics) [93], the lower cost of Ethernet infrastructure [5, 110] and the desire to converge to a single infrastructure to reduce the overall Total Cost of Ownership (TCO), is pushing more and more providers to switch to *Converged Ethernet* architectures and RoCE for RDMA deployments. This has been possible thanks to the recent additions to the Ethernet standard [93, 98, 100], the release of many NICs support RoCE, and switches that accelerate it.

RoCE, typically used in its *v2* version, transports RDMA verbs (*i.e.,* InfiniBand semantics) over Ethernet, encapsulating them in UDP packets. RoCE v2 packets are routable (*i.e.,* they can traverse multiple networks), and make use of IP Explicit Congestion Notification (ECN) bits, Pause Frames, and Priority-based Flow Control (PFC) to provide the same guarantees of an InfiniBand physical layer.

Although InfiniBand and Ethernet are incompatible, *tunneling* protocols have been defined to transport either type of traffic on the other physical layer. In addition, the two share most electrical standards (*e.g.,* optical transceivers, cables, and physical connectors). Mellanox, the main vendor for InfiniBand hardware [111, 112], released Virtual Protocol Interfaces (VPI) devices that can be configured in either InfiniBand and Ethernet physical layers, helping the shift toward the RoCE protocol. We refer the interested reader to [93] for a general overview of RDMA, InfiniBand, and RoCE technologies, while we treat some other relevant technical details in Section 7.1.

2.4.2 Future steps for Ethernet

The rapid growth of GPU performance, which supports the demand for a large amount of AI workloads, poses a large burden on the interconnectivity means in datacenters, which needs to be redesigned to support this *revisited* type of workloads, which is for some aspects similar to a HPC workload, often with many

nodes dividing processing of requests. This translates in an increase of RDMA-based applications that can alleviate the pressure on CPUs, while offloading some network functionalities directly to network cards.

Although RDMA and RoCE in particular work well on small networks, when traversing multiple switches, or in the presence of network congestion, maintaining a loss-free fabric becomes a harder task, and many limitations have been encountered by large providers deploying at scale [113].* This becomes even more relevant in the presence of multiple paths and out-of-order packets, as it could happen in a very large network.

Solving this problem is an open question, with many contributions from the research community. The industry is responding to these limitations with both commercial solutions (*e.g.,* implementing features in switches) and through the Ultra-Ethernet consortium [114].

2.5 TVM

TVM [66] is a project hosted by the Apache Foundation that aims to provide an open-source framework for ML, enabling efficient optimization of algorithms for CPUs, GPUs, and other accelerators. The core of TVM is the ability to compile ML models into deployable modules that can be *optimized* and run in the appropriate back-end for each execution platform. These compilation phases can benefit from a further *tuning* process, which would improve the runtime performance by carefully packing the execution of the different layers to reach maximum efficiency [55].

TVM takes as input a high-level implementation of a deep learning program, as exported by common frameworks, and transforms it into a low-level optimized module, typically in the form of a library, which can be executed on different hardware back-ends, exploiting the different characteristics of each of these. To produce efficient code without manual intervention, an intermediate vector-based representation is used, which allows separation between hardware intrinsics. This simplifies the support and optimization for new instruction sets and accelerators, decoupling it from the pure optimization phase. The latter is based on a large space optimization problem, where an ML-based cost model guides the research, which automatically adapts the model by collecting metrics from the hardware back-end.

TVM optimization has been further improved by Ansor's contributions [115], which fine-tunes the models with an evolutionary search approach, potentially going beyond the search space of other State-of-The-Art mechanisms, improving the performance on all TVM supported platforms.

In this thesis, two characteristics of TVM has been exploited:

*Similar issues happen with NFS, iSCSI, and other protocols expecting a loss-less fabric, but these have rarely been deployed at-scale.

Optimization allows each model to run as fast as possible on the hardware, demonstrating the actual achievable performance of State-of-The-Art GPUs. Thanks to the automated search for the optimal model, no manual fine-tuning has been necessary to reach a good execution rate on the hardware examined.

Deployable modules by importing the TVM output at runtime as stand-alone modules, our application can abstract from the low-level implementation of the ML algorithm, allowing large flexibility in the workloads supported by the final system and increasing the reusability of the general application structure for other use cases.

We note that, due to some current idiosyncrasies in the support of TensorRT by TVM, we have not been able to use these accelerators in our evaluation. This is not a fundamental limitation, and the performance of a ML application that would use these accelerators would just increase the impact of the networking cost on the total inference time (since they would reduce the execution time of the inference).

2.6 How much processing does an inference take?

In order to provide a more transparent and fair approach to measure performance of ML systems, both on the hardware and software sides, the MLPerf® framework [116] has been proposed, gaining wide adoption by the principal hardware manufacturers. MLPerf® is a testing and benchmarking suite aimed to deliver consistent and comparable results, with open-source results and testing procedures.

In this section, we take MLPerf®'s results as a baseline to understand the current trends for the performance of AI systems and, in particular, AI accelerators.

A performance insight from MLPerf According to the latest publicly available results [117], the current record for image classification on a single system is 600 179 images per second. This result was obtained on an NVIDIA DGX-H100 machine equipped with 8 NVIDIA H100 GPUs, which represents the current top-of-the-line of NVIDIA products, with images not preloaded, using *ResNet50* [118] and the *ImageNet* [119] dataset. We discuss the architecture of this system in Section 5.6.

We report a relevant selection of the results for this test in Figure 2.4a. Assuming that each inference takes the same time and that only one inference is

performed at a time,* we can derive an approximate average inference time of 1.66 µs and an average rate of 75 022 inferences per second on each GPU. We report these normalized values in Figure 2.4b.

Although the specific architecture of this system may introduce bias in the result (*e.g.,* each inference may be spread among multiple GPUs), we note that the inference rate for a single GPU, in the same operations is reported to be 76 001 images per second.†

GPU vs. CPU efficiency for inferencing As a point of comparison, the same test reports a rate of 16 498 images per second for the top-of-the-line dual Intel® Xeon® 8480+ CPUs. We note that the pair of CPUs in the latter system has the same Thermal Design Power (TDP) of a single H100 GPU, which results in the GPU-based system executing $\approx$ 4.5 times more inferences per each nominal consumed watt w.r.t. the latter.‡

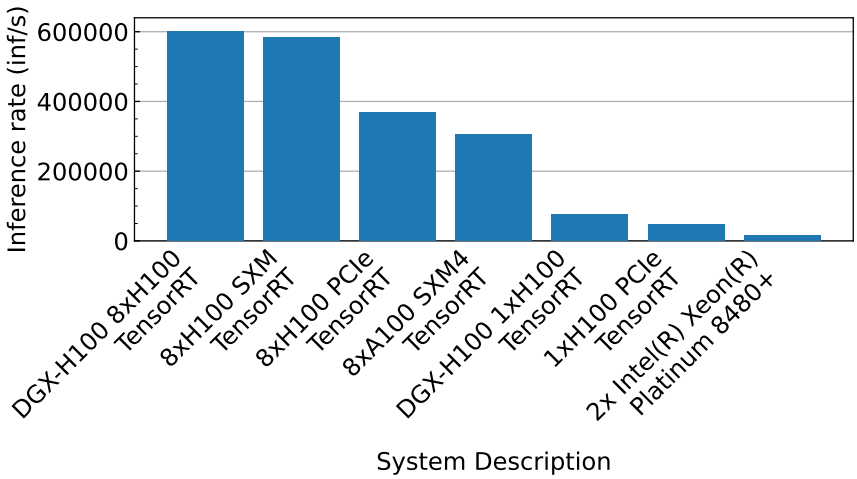
However, for some specific models, performing inferences on a CPU may represent a viable alternative, and the claim *GPUs are always better than CPUs for AI tasks* should not be taken as *ground truth* without proper testing.§

*When running multiple inferences, the single inference time, on average, will be higher, but the total throughput would increase as well. See Section 8.2 for a practical analysis of it.

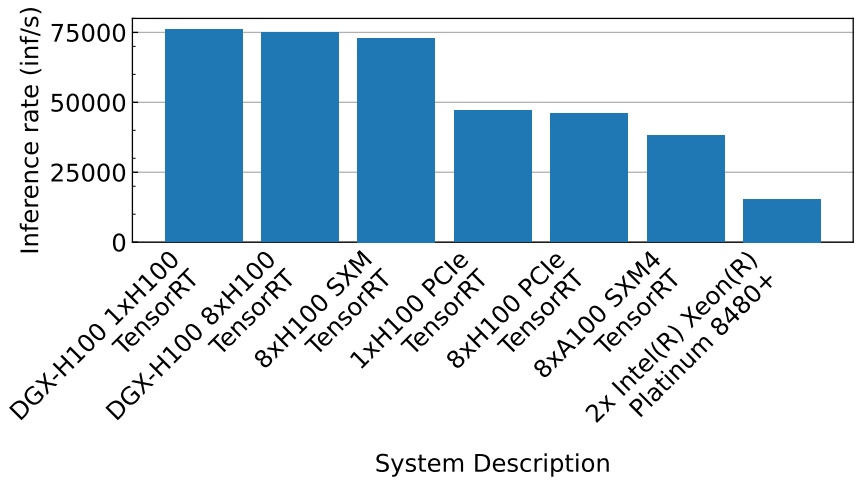
†We note that MLPerf® guidelines are discouraging the extrapolation of partial results, although we use them only as a point of comparison and we are not making any claim based on these.

‡If we also account for the power used by the CPUs in the system, assuming a 100% load, the GPU system would process $\approx 4\times$ more inferences per consumed watt.

§Similar concerns could be raised for many other technologies used in this work [120].



(a) Cumulative per-machine



(b) Normalized per-accelerator

Figure 2.4: MLPerf® inference results for some selected systems, *closed* scenario with ResNet serving benchmark. Replot from MLPerf® Inference DataCenter v3.0 [117].

2.7 The cost of the network in inference tasks

Network speeds have grown substantially in recent years, with up to 400 Gbps NICs available in the market. On such links, moving the typical I/O data for an Imagenet dataset requires around 10 µs, which becomes relevant when compared to a realistic inference time of ≈ 3 µs with ResNet on an NVIDIA H100 GPU.

With today's high inference rates, when performing inference serving it becomes extremely important to carefully design the network I/O stack of inference-serving applications to minimize these additional costs, as close as possible to the minimum achievable. This is necessary both to reduce the serving latency and to maximize the system's throughput.

Network operations are typically handled by the CPU, which would run the network stack, and moves data between the different application and driver buffers. Assuming the aforementioned results for an NVIDIA DGX-H100 system, a system running the ResNet model needs to ingest 2.9 Tbps of input data from the network through multiple NICs over the PCIe bus to saturate its GPU capacity.

Kumar, Sivasubramaniam, and Zhu [10] studied the costs of data-transfers for inference serving. For a ResNet model running on a single NVIDIA A100 GPU, the data-copy was reported to cost between $\approx$ 40–50 % of the total inference serving time when using gRPC to bring data in and out of the serving machine when serving the *ResNet* model, while it grows up to 73% for models that run for a shorter time.

Assuming a direct scaling of performance on an H100 GPUs based upon these MLPerf® results, this would result in $\approx$ 60-70% total serving time spent on network I/O and memory copying tasks.[*] Similar insights have been shown by Meta [121], suggesting that in major application deployments a large share of the response time is spent in networking tasks.

An architectural issue: PCIe lanes distribution A top-of-the-line Intel® Xeon® Platinum 8480, as used in the NVIDIA DGX H100, offers 80 PCIe 5.0 lanes. Assuming a *typical* achievable bandwidth of 400 Gbps for a 16x device,[†] a system with two of these CPUs with *all* lanes allocated for network I/O could handle a total of 4 Tbps of throughput (per direction). However, in such a GPU-based machine, PCIe lanes are also needed for communication with GPUs and other peripherals. Assuming a fair split of the available lanes in the system between GPUs and NICs (*i.e.,* ignoring all other peripherals), the maximum bandwidth that could traverse the system via the *network-GPU* path is limited to less than 2 Tbps, unless

[*] Assuming that the copy time is the same in both cases and with the NVIDIA H100 inference rate for ResNet50 twice as fast inference as the NVIDIA A100, *i.e.,* 70 k inferences/s versus 35 k inferences/s

[†] A PCIe 5.0 16x link would handle 512 Gbps per direction, but standard Ethernet NICs are available at 100, 200 and 400 Gbps rates.

an extremely asymmetric traffic profile is expected from the application (*e.g.,* the I/O size ratio is very high or low).

The hardware design used by the DGX-H100 machine and similar GPU-dense servers expands the number of PCIe lanes by exploiting a network of PCIe switches. These switches allow oversubscribing the PCIe lanes available on CPUs, which could be allocated in different priorities to GPUs and other peripherals, depending on the specific design. This will inevitably introduce bottlenecks in the system (*e.g.,* it is impossible for all devices to saturate their PCIe lanes when communicating with the CPU), but it greatly improves the P2P communication capabilities of all devices, which then requires a more careful design, adjusted to the physical architecture of the machine.

With traditional approaches for inference serving, where the CPU is in charge of all the network communication along with pre- and post-processing of data, it becomes even harder to reach full performance, with throughput likely limited by the contention that occurs on the shared PCIe lanes, as is the case on NVIDIA DGX-H100 (see Section 5.6 for a technical analysis of its internal structure).

The problem is even more exacerbated in general applications by auxiliary data movement between CPUs, GPUs, and potentially Non-Volatile Memory express (NVMe) disks (*e.g.,* to load or replace a running model's weights). These transfers would participate in the PCIe lane contention and increase the system load, requiring even more careful scheduling to limit performance drops that may be introduced by these overheads. We also note that, in most cases, additional control traffic occurs between CPU and the various devices, increasing the contention (although minimally) on the PCIe bus.

Moving 2 Tbps of traffic We also note that although many works aimed at improving the performance of the network stack [10, 122–128], processing 2 Tbps on a single machine is still a demanding task, with most work in the field trading CPU cycles for lower latency (*e.g.,* using CPU polling as in DPDK [96]). These works, mainly focusing on *Network Virtual Functions*, are often targeting simple programs that can process each packet (or a unit of work in general) very quickly, and thus could trade the free CPU cycles to poll network interfaces.

A *back-of-the-envelope calculation* based on PacketMill [128] suggests that a DPDK-based router on modern CPU models (such as the one used in a DGX H100) could process around 100 Gbps of traffic on each core. Assuming a perfect linear scale (*e.g.,* without considering any synchronization, locking, cache, or bus contention), a pair of the above CPUs would be able to process ≈ 10.12 Tbps in a routing application, when *all cores* are used for packet processing.

We note that in an inference serving setting the application running on the CPU may be more complex (*e.g.,* it could include some data conversion, pre-, and post-

processing), and this would reduce the total number of CPU cycles that can be dedicated to data movement, making it even harder to handle this amount of traffic.

Furthermore, considering commonly available NICs speeds, such a *Terabit-scale* machine would need to use multiple network ports to receive and transmit traffic, which would require careful design to ensure that traffic is distributed evenly among them (*e.g.,* through ECMP), landing on different, sharded, physical cores.

2.8 How long does it take to do an inference?

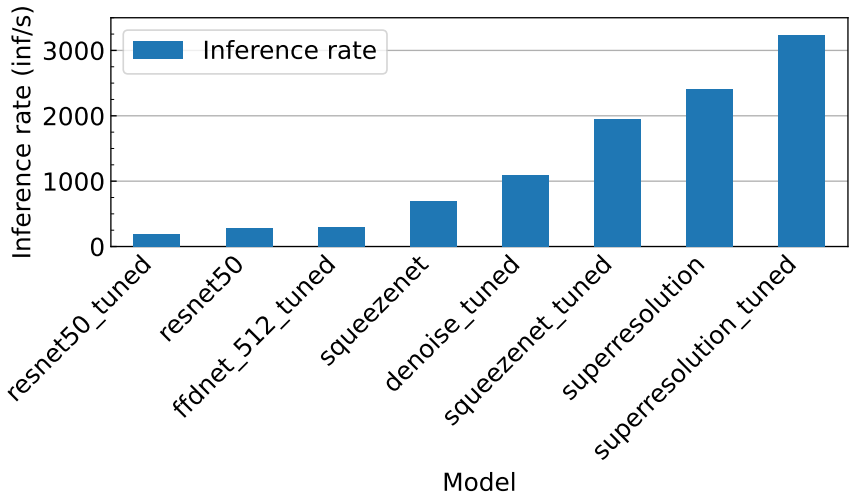
To quantify the real inference rate that an application may obtain, we deployed a simple benchmark on an NVIDIA A100 GPU, where we ran a series of inferences in the most optimized settings.

We sourced our models from State of The Art and publicly available repositories [129, 130], covering different I/O ratios and different computing complexities, ranging from image classification (*e.g.,* ResNet [118], ShuffleNet [131], and SqueezeNet [132]), to image denoise models (*e.g.,* fddNet [133, 134], and resolution upscaling models [135]). While restricted to specific fields (*e.g.,* without considering recommendation models, Large Language Models, or generative models), we believe that these are the most critical applications, where it is more likely to have low-latency requirements and where the I/O size has a significant weight in resources usage: most observations could be generalized to different models.

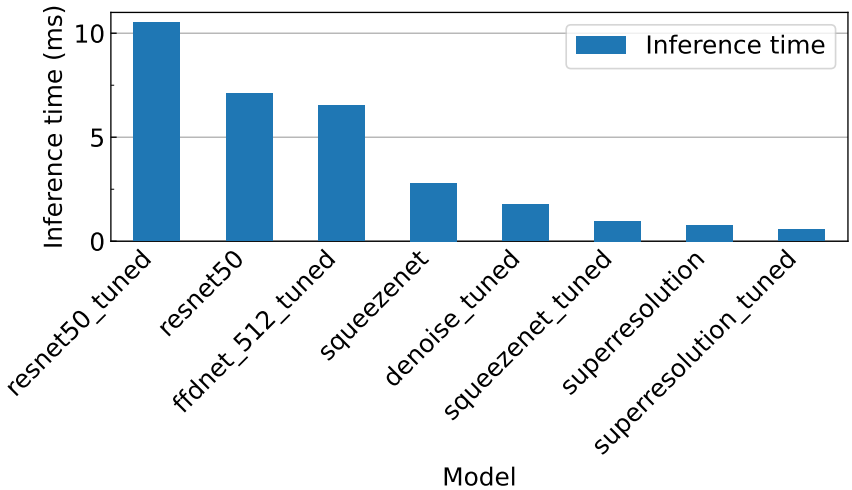
We tuned and compiled our models with TVM, improving the performance of the näive deployment, although without using specific acceleration techniques (*e.g.,* mixed precision and TensorRT), and without any further pre-/post-processing, to evaluate the throughput and inference time of every model.

TVM [66] is an open-source project hosted by the Apache Foundation, aimed to deliver a *Machine Learning Compiler Framework*, which could be compiled and run on a multitude of hardware targets, decoupling the definition of the model and training from compilation and optimization of it. The main TVM features are described in more detail in Section 2.5.

We report the results of this benchmark in Figure 2.5, both for the inference rate and the pure inference time. While some models have inference times in the order of milliseconds, we can see how most commonly-used models can complete inferences in submillisecond time frames on modern hardware, even with a single GPU running the workload. As discussed in Section 2.7, at this time-scale every memory copy, RAM access, and synchronization point in the application becomes relevant since all of them could easily hurt performance by introducing queuing and head-of-the-line blocking.



(a) Average Inference Throughput



(b) Average Inference Time

Figure 2.5: Pure inference throughput and time across a selection of models on an NVIDIA A100 GPU.

All the models have been compiled using the default TVM auto-tuning parameters without studying the performance of each single model in detail. We note that the performance difference between an untuned model and its tuned version can be as high as 2×, potentially drastically changing the performance of the system.

Any inference service provider should *not underestimate* the benefit of model tuning, which could save resources at the cost of a brief offline tuning, which would easily amortized, especially for long-running deployments. Tuning techniques, and the mechanisms behind them, have not been studied as part of this work. We refer the interested reader to [55, 115, 136] for some relevant works.

2.9 A use case for bandwidth-intense inference

To quantify the amount of data that need to flow in and out of a generic inference system, we can multiply the pure inference rate with the input and output model sizes, which leads to the results reported in Figure 2.6.

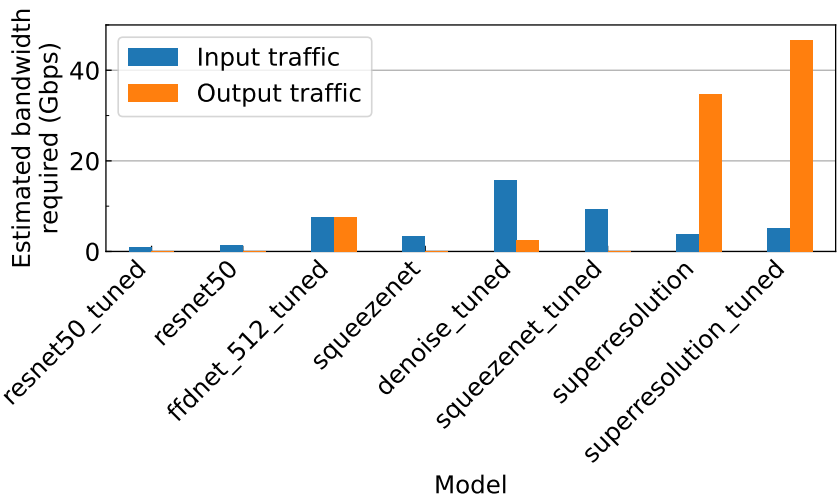


Figure 2.6: Estimated network bandwidth required by different models running on an NVIDIA A100 GPU.

These results represent the theoretical bandwidth that the inference service would consume if the network were not a limitation and without accounting for any time or overhead related to the network I/O (*i.e.,* all data are already located in the GPU memory). We do not account for headers and potential serialization/deserialization of the data, assuming that the client sends inputs as raw bytes and

receives the outputs in the same form, which closely represents a scenario where clients submit requests through RDMA.*

Although most models have low bandwidth requirements, we can see how some models exceed 10 Gbps of I/O bandwidth, with peaks of 40 Gbps when running on a single NVIDIA A100 GPU. While these could be considered modest requirements by modern datacenter standards [137], we note that most providers aim to deploy GPUs with a higher density than 1 GPU-per-chassis. Assuming a linear scaling of the inference rate, the I/O bandwidth can easily saturate a single 100-Gbps link, which justifies the need to scale the network speeds accordingly.

A GPU-dense environment estimation Similarly to how we extrapolated the inference rate for a single H100 GPU from MLPerf® results in Section 2.6, we can linearly scale the bandwidth of a single GPU to simulate a system equipped with eight, without any PCIe bus contention. We report the results of this simulation in Figure 2.7, from which we can see how the throughput for some models could exceed 100 Gbps, currently one of the most common interconnection speeds in datacenters [137].

We note that most commercial server platforms designed for GPU-intensive environments can either support 10 standard PCIe double-width GPUs or 8 GPUs in proprietary form-factors (*e.g.,* NVIDIA SXM and AMD UBB), the latter usually providing higher performance thanks to the higher TDP.

*We note that similar approach is followed in State-of-The-Art works [19].

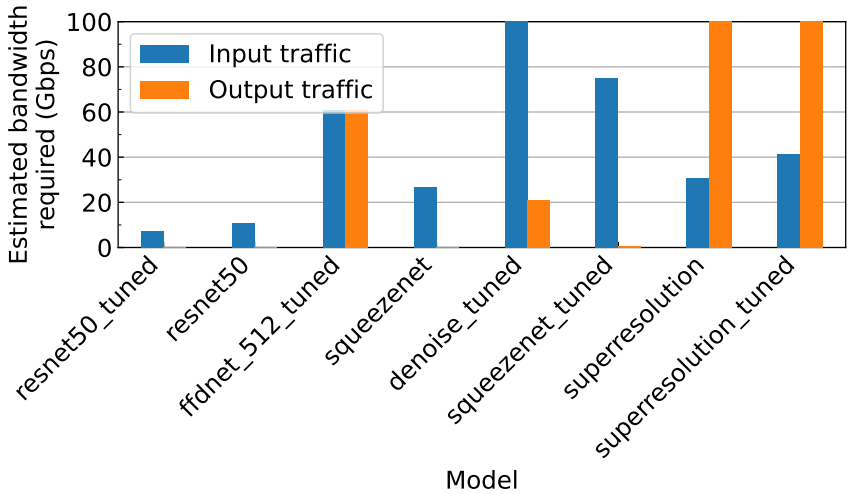
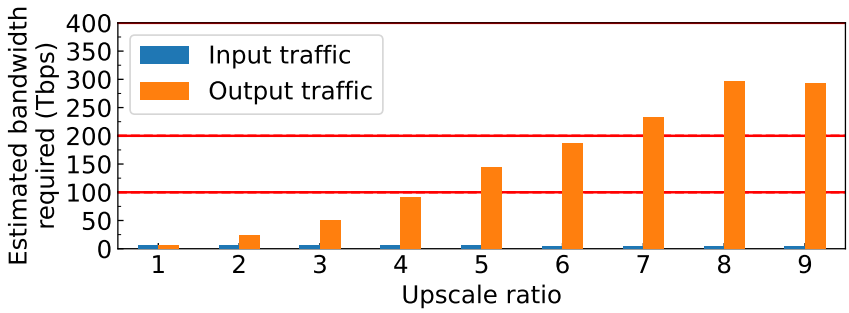


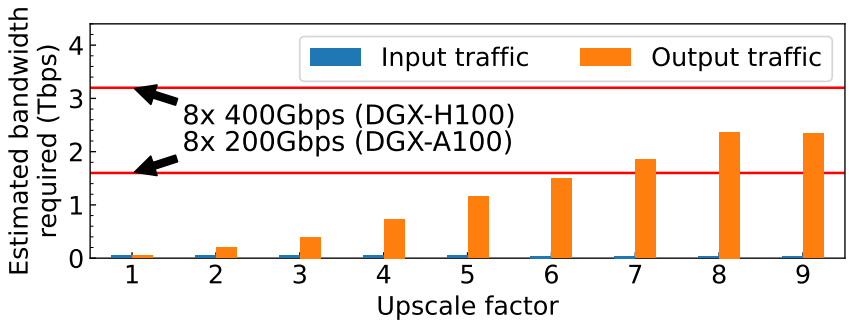
Figure 2.7: Estimated network bandwidth required by different models, NVIDIA A100 GPU, with 8× linear scale.

Resolution augmentation To understand how the I/O bandwidth would scale for a realistic GPU-heavy application, we have generated multiple versions of the *superresolution* [135] model, changing the upscaling factor and setting the input size to the frame size of a 480p video (*e.g.,* 480x640 pixels). This simulates the workload sustained by a service wanting to deliver better video quality, on demand, at different resolutions.

The results in Figure 2.8a clearly show how the bandwidth could quickly exceed the common link capacities of 100 Gbps and 200 Gbps.



(a) 1xA100 GPU



(b) 8xA100 scaling

Figure 2.8: Estimated network bandwidth required by *superresolution* at different upscaling factors. The inference rate has been measured on a physical A100 GPU.

As a reference, we note that a 4K frame (*e.g.,* 3840x2160 pixels) would be filled with an upscaling factor of 6, while 8K would require $10\times$ upscaling.* In Figure 2.8b, we report the linearly scaled results for a system equipped with 8 GPUs, which shows how a 4K workload (*i.e.,* with an upscaling factor of 6) would already generate more than 1 Tbps of output traffic. For an 8x NVIDIA A100 chassis, this would saturate the PCIe lanes and the physical network ports before reaching the maximum GPU throughput, without accounting performance issues that may arise from PCIe bus contention and suboptimal access patterns.

Most GPUs used in datacenter are not targeted to video-processing, and thus their encoding/decoding capabilities may be worse than those on the desktop GPUs. We also note that, when used in pipelines (*e.g.,* super-resolution as part of a greater video-enhance process), developers may want to avoid any intermediate transcoding to reduce artifacts, thus obtaining a setup similar to the one simulated in this measurement.

*Multiple standards exists for video resolution. In this example, we consider a classic 480p (VGA) frame and a 4K or 8K YouTube video frame size.

Chapter 3

Related Works

This chapter provides an overview of relevant related works. Section 3.1 focuses on enabling GPUs as networked devices, both in HPC environments (Section 3.1.1) and in more general datacenter deployments (Section 3.1.2). Section 3.2 presents relevant works that aim to facilitate AI workloads. Finally, Section 3.3 summarizes relevant contributions from the research community aimed at improving GPU runtime, with a focus on AI- and ML-oriented workloads.

3.1 Interconnecting GPUs and other devices

Given the increasing demand of GPU-accelerated large-scale workloads, and the large problem space that could benefit from running on GPUs, several solutions have been proposed to improve performance on multi-GPU deployments, both from hardware manufacturers, software supports, and from the research community.

Besides the ubiquitous PCIe bus (introduced in Section 2.3), modern GPUs leverage vendor-specific connection technologies to solve inherent limitations of PCIe. Neugebauer *et al.,* [90] studied the performance of the PCIe bus, in its 3rd generation, showing how, in many cases, access patterns can reduce the usable bandwidth on the bus. NVIDIA datacenter GPUs are usually equipped with *NVLINK* connectors, which allows multiple GPUs to talk directly skipping the PCIe bus. We analyze some characteristics of this technology in Section 2.2.2.1. Appelhans *et al.,* have compared earlier versions of NVLINK with PCIe, developing a model to understand what performance benefits can be achieved by using the former in place of PCIe [49]. Newer hybrid GPU-CPU architectures from NVIDIA (*e.g.,* Grace-Hopper) are expected to use a unified memory topology, dropping PCIe as the main interconnection between CPU and GPU [50].

On top of the latest PCIe 5.0 physical layer revision, CXL has been proposed as an extension of PCIe semantics, fostering heterogeneous computing infrastructure and more advanced resource pooling [138]. Arif *et al.,* have shown this technology could improve resource usage and memory performance for large-scale multi-GPU systems [139]. We note that NVIDIA is part of the CXL consortium, and there are plans to support CXL on top of the proprietary *NVLink-C2C* Chip-to-Chip technology, although no current product supports it [140].

FLEET [141] proposes a disaggregated architecture where GPUs are interconnected through optical switches and custom *fabric adapters*, extending the PCIe protocol across the chassis boundaries. Google has followed a similar approach with its 4th generation TPU, which employs direct chip-to-chip interconnections through optical switches [70].

Li *et al.,* [48] have analyzed technologies and interactions between GPUs, CPUs, and in hybrid architectures, in datacenter environments. They provide a good overview of the different available technologies and how these affect the performance of applications in various contexts, focusing in particular on multi-GPU interactions.

3.1.1 HPC-oriented technologies

NVSHMEM [142] provides a facility to exchange data between multiple GPUs involved in collective operations, such as large-scale simulations or training. This provides a large virtual memory space between NVIDIA GPUs, with data being transported between them either on PCIe, NVLink, or over RDMA, always preferring the less expensive path. GPU applications can directly exchange data with other GPUs without CPU intervention, exploiting a drop-in replacement to other standard Message Passing Interface (MPI) libraries.

UCX [17] provides similar functionalities, offering a complete library stack to support a variety of applications, interconnection technologies, and hardware vendors. The main difference with NVSHMEM (with whom it shares most functionalities) is the more open nature, and the possibility to interact with other vendors' hardware.

Microsoft Collective Communication Library (MSCCL) [143] is an extension proposed by Microsoft to improve communication performance inside Azure datacenters, building on top of NCCL, focusing mainly on collective algorithms, such as distributed AI trainings [144, 145].

We can separate all these works from those discussed in Section 3.1.2 based on the deployment scenario and the communication patterns. NVSHMEM, UCX, and MSCCL are all aimed at improving *collective operations*, where multiple GPUs (potentially in the order of hundreds or thousands) collaborate to solve a larger

common task, typically using a many-to-many style of communication in a highly homogeneous cluster (*e.g.,* using the same hardware architecture), as typically found in HPC clusters.

3.1.2 Data-center GPU networking

This section will present some relevant works in the field of networking, where GPUs represent a central component in the architecture, focusing towards a more *datacenter* environment. In this scenario, the communication pattern is typically *1-to-1*, with a more heterogeneous hardware selection involved.

GPUrdma [146] represents one of the pioneering works in the extension of RDMA semantics to GPUs, delivering a first version of network-aware capabilities to NVIDIA GPUs, while working around to the limitations imposed by the drivers & the PCIe bus implementations available at the time. NICA [147] provided similar capabilities using FPGA-equipped NICs.

GPUnet [148] proposes a network abstraction framework for GPUs, allowing them to receive & transmit data directly without CPU intervention. This is done by re-implementing the network routines directly on the GPU, and relying on InfiniBand for the physical layer. The main difference with our contribution described in Section 7.5 is the use of a custom network stack (which potentially needs to be adapted to the different applications) and the need for a I/O proxy layer on the CPU. Our approach is based on a minimally modified *rdma-core* which could be used as drop-in replacement, and characterized by the total absence of CPU operations during runtime.

Some efforts in the direction of improving the integration of GPUs and high-speed NICs, pushed by the industry, have resulted in the *l2fwd-nv* demo project [149, 150], and later in the integration of specific facilities in Data Plane Development Kit (DPDK) [151,152], from the 21.11 release. This is mainly driven by the ability of some NICs (*e.g.,* Mellanox) to split the memory buffer locations between the GPU and RAM, and thus skip unnecessary copies. Similar integrations are also being addressed for 5G use-cases [153]. GPU-Ether [154] propose a GPU I/O framework to exchange Ethernet frames directly from a GPU, following an approach similar to the one followed by DPDK for CPU applications. However, this system does not provide application-level semantics, which is oriented to Virtual Network Functions rather than *L7* applications. Tseng *et al.,* proposed a network packet processing framework, exploiting the (small) GPU integrated in Xeon processors [155]. A follow-up work [156], improves the performance of OpenVswitch, improving the throughput up to 3x on a CPU-only, DPDK-based, architecture. Although the capabilities of these are very limited, we note that this work could have an increased attention in the near future, following the increase of

GPU-like functionalities in modern CPUs, and the imminent release of integrated hybrid platforms, such as NVIDIA Grace-Hopper [51].

BPCM [157] proposes a system to improve the efficiency of GPU clusters by exploiting an approach similar to DPDK and Load-Balancing traffic across multiple network ports, interconnected through simpler L2 switches, using the CPU resources to move the data to and from the network. We note that this approach is not scalable on higher bandwidth, both due to the intensive requirements for CPUs and for the lack of physical space in modern server chassis: operators often prefer higher computing density to *many-ports* systems, especially when dealing with a low number of available PCIe slots.

A different approach has been taken by Furukawa *et al.*, proposing to accelerate the training phase of ML algorithms by running on multiple GPUs, interconnected by a DPDK-based software switch and ExpEther boxes, interconnected with 40-Gbps Ethernet links [158]. PacketShader [159] proposes a high-performance software router accelerated by GPUs, improving performances outperforming CPU-based commodity solutions, especially for small packets. SSLShader [160] implements a similar architecture to accelerate the performance of SSL cryptographic function leveraging GPUs on commodity machines.

Kargus [161] implements a high-speed Intrusion Detection System on GPU, offloading pattern matching and regular expression processing on GPUs, while providing compatibility with the popular Snort IDS. GASPP [162] is a GPU-based packet processor, using a hybrid schema to solve the memory copy bottlenecks between CPU and GPU, depending on the packet size. Plante *et al.,* use DPDK split buffers to deliver data directly to the GPU through UDP [163], but keeping the CPU in the *critical path*, which maintains the role of the system orchestrator, handling the packet reception, with all the processing performed on the GPU. Thoustrup *et al.,* improve the performance of distributed DBMSs *join* operations by combining GPUs in pipelines, interconnected by 100-Gbps InfiniBand network, and leveraging GPUDirect RDMA to move data between GPUs [164].

Li *et al.,* have analyzed the technologies and the interactions between GPUs and CPUs in hybrid datacenter architectures, providing an extensive overview of the different available technologies and how these affect the performance of applications in various applications, focusing in particular on multi-GPU interactions [48]. Katsikas *et al.,* studied the performance of modern NICs, highlighting the problems they can face under particular traffic patterns [165]. A major series of efforts have explored the performance of network applications, and NIC in general, studying the interactions with the other components and the performance these have [94, 128, 166, 167]. While most of these techniques are not directly applicable to the GPU subsystem, we note that the search for faster packet processing and more optimized interactions between devices has also brought benefits for RDMA-

centric applications.*

NVIDIA Bluefield-2X [168] combines ARM and Ampere-generation GPU on a single board with 100 Gbps connectivity. GPUNetIO [169] explicitly targets this platform, allowing GPUs to directly receive or transmit Ethernet packets [170,171]. Today, the main targets of these frameworks are Network Functions rather than AI Input/Output transfers. GPUNetIO represents one of the closest frameworks to our contribution described in Chapter 7. However, for AI applications, its packet-level semantics represent a tight limitation, given the input & output sizes usually larger than a single packet. Therefore, manual segmentation and congestion control may need to be implemented, which are provided as an integral part of the RDMA network stack.

GPU Triggered Networking [172] implements a direct GPU-centric networking model, allowing GPU kernels to exchange data directly over RDMA. The verb processing gets triggered by a CUDA-side operation, with the CPU is in charge of preparing the data-structures (*i.e.,* preparing the verbs) and performing all operations that are not complex to offload to a GPU. Agostini *et al.,* presented early efforts in this direction by NVIDIA in [173], showing how GPU-driven networking can improve performance in a MPI-like environment.

3.2 AI-oriented efforts

The rise of the AI- and ML-based workloads pushed the research community to explore the different issues in common applications' architecture, trying to solve different limiting factors.

The closest work to our contribution presented in Chapter 8 is described in [174], where an RDMA-based inference serving system is evaluated. The proposed architecture uses RDMA to transfer data across the network, but initiate data communication from the CPU, using expensive CPU cycles and without removing CPU-GPU synchronization points.

Clockwork [19] proposed an inference service system with a centralized controller that handles all requests, with very strict Service Level Objectives (SLOs). By carefully controlling what model is run and when the data is loaded or evicted, they can predictably reach very high performance with very stable tail latency. We analyze Clockwork in more detail in Section 8.1.

SHEPHERD [175] addresses the scalability of large inference serving systems. INFaaS [176] tackles the scalability of the problem by automatically generating model variants to best suit the hardware architecture and the query type while autoscaling on AWS EC2 resources. Nexus [177] addresses the issue of scheduling

*We note that most NICs use the same Application Specific Integrated Circuit (ASIC) to deliver standard kernel-level IP processing, user-level DPDK-style applications, and RoCE functionalities

requests across a large cluster of GPUs performing Deep Neuural Network (DNN)-based video analysis, using a periodic per-stream scheduling approach.

Triton [178] is an inference serving system developed by NVIDIA as a production-ready platform, handling the scheduling and runtime of the models and directly focusing on the integration with other *MLOps* tools. Being based on gRPC and HTTP protocols, its CPU-based architecture and its internal scheduling mechanisms are not optimal for low-latency applications where strict control on the runtime is desired. ORCA [179] is an inference serving system designed to efficiently serve Large Language Model (LLM) workloads, targeting clusters of NVIDIA GPUs. The inference workload is distributed through a NVIDIA Collective Communication Library (NCCL)-based mechanism.

Some works explored the efficiency of network transfers to and from GPU devices. SplitRPC [10] highlights the need to optimize network transfers, showing how on a 100-Gbps network fabric, data movement may be as expensive as 50% of the total inference serving time, when operating on realistic workloads. SplitRPC improves the efficiency of inference serving, using DPDK [96] to deliver requests payload directly to GPU's memory. These requests are transported over RPC, with the requests' metadata delivered to RAM (*e.g.,* by splitting the network packets' bytes). Similar support has been introduced in DPDK [96] and can be used to improve the performance of high-speed network applications that involve GPU workloads [151].

A further insight is shown in [174], where the overhead of a traditional CPU-driven network stack is quantified as consuming between 15% and 50% of the total inference time compared to transports based on RDMA.

Jasny *et al.,* proposed a zero-sided RDMA implementation to allow FPGAs and GPUs to interact over RDMA without any of the two being actively involved in data exchange [180]. Their implementation involves a programmable switch with the role of translating *read responses* in *write requests*, coordinating the data pipelining, and it is focused on a use-case where the two devices involved exchange data continuously, such as in an AI inference-serving pipeline.

3.3 Improving GPU runtime efficiency

A branch of works has focused on improving the pure performance of inferences, either by exploiting more efficient features in modern GPUs or by following accurate strategies in planning the execution of the different components in the GPU subsystem. Hardware vendors have made great efforts to develop more advanced accelerators (both GPU hardware and other types of accelerators). We discussed major relevant advances in this field in Section 2.2. As already introduced in Section 2.5, Chen, *et al.,* introduced the TVM framework [55], allowing

applications developers to easily compile DL models into efficient runtime code, supporting multiple hardware targets. This framework has been later improved by Ansor [115].

With similar goals, Meta's AITemplate aims to transform DNN code into highly efficient low-level NVIDIA CUDA or AMD HIP C++ code [136]. Collage [181] aims to join multiple DL frameworks and integrate multiple back-ends, placing each operation on the most suitable (and optimized) hardware resource. Mingzhen, *et al.,* [182] published a survey of DL compilers.

Chapter 4

Research Problem and Challenges

The rapid growth of high-performing AI deployments, and the increase in the performance of AI accelerators put stress on network systems supporting these applications, which are rarely contained completely in a single node.

In Section 4.1 this problem is presented in more detail, discussing how this affects current and future application deployments. Section 4.2 presents some of the challenges associated with the problem, together with the approach followed to solve them in this thesis. This leads to the research question formulated in Section 4.3, while the research methodology followed in this thesis is presented in Section 4.4.

4.1 The problem

The recent growth of AI accelerators' performance, the high demand for AI features, and the subsequent scaling of hardware systems supporting these workloads, pose huge demands on networks. As discussed in Section 2.2.3 most of these applications run on commodity hardware, mainly to reduce initial costs and for the easier access to off-the-shelf equipment. In most cases, this translates in the use of GPUs, taking advantage of the large parallelism offered by these, as discussed in Section 2.2.

While large AI application may spread across many nodes, as discussed in Section 3.1, the vast majority of GPU-accelerated applications are executed on a single machine, with some other *pre-* and *post-* processing executed by a CPU, especially in the context of ML applications. Most of these applications need to

interact with some data, either in the form of online requests (*e.g.,* in an *inference serving* scenario) or from a mass storage resource (*e.g.,* in an offline analysis or during a training).

In all of these cases, one or more CPU *traversals* are experienced by the data, introducing higher latency and CPU usage, which directly translates into the need of building bigger and more powerful worker nodes, where the CPU represents a central party involved in the data-path.

In this thesis, we are mostly interested in the performance of an application processing online requests, and in particular to scenarios where the processing time is small enough to be comparable to the network processing time, as discussed in Section 2.7. Beside the low-level implementation that would improve the performance of these systems, as presented in Chapter 7, we studied the performance of Connection Tracking applications running at 100 Gbps, which play a fundamental role in scale-out deployments for both accounting and Load Balancing of the resources, as presented in Chapter 6.

4.1.1 The performance of GPUs is growing faster than network speeds

During the last 60 years, the computational power of CPUs has been mainly driven by semiconductors miniaturization, and the consequent ability to pack higher computational resources in the same physical space. In the early age of computing, Intel founder Gordon Moore predicted that the number of transistors in a CPU would have doubled every 18 months [183], and this was proved to hold until 2003, after which the trend started to slow down [184], both due to physical production limits and the higher costs of manufacturing at smaller transistor sizes [185].

In recent years, GPU performance has been shown to grow faster than CPU performance [186, 187], despite the diminishing returns of transistor miniaturization. This has been mainly possible by algorithm improvements, specific built-in accelerators, and better integration of GPUs in the software ecosystem.

The growing rate of AI-driven workloads, and the increasing complexity of the computing behind these, put extreme pressure on memory and network devices, which have historically shown a slower growth compared to GPU performance [2, 188]. The need to optimize network-related aspects has also been demonstrated by the growing interests of GPU manufacturers in inter-networking technologies, and their integration with their software stacks [112, 169, 189].

4.1.2 Traditional GPU systems scaling is limited by CPU performance

While large-scale GPU-based datacenter deployments are often based on custom-designed chassis, commodity server platforms often suffer from physical constraints regarding the number of GPUs that can be installed in a single unit (*e.g.,* the *GPU/rack-unit* density). These are mainly a consequence of the limited number of PCIe lanes available on common CPUs, and the traditional consideration of GPU as system *peripherals*, rather than first-class citizens.

As we have analyzed in Section 2.7, GPU-based server machines can move *Tbps-scale* volumes of traffic, posing great challenges in the CPU stack.

In practical terms, this results in hardware architectures where (i) all data in & out of a GPU need to traverse the entire host stack (*e.g.,* CPU and RAM), (ii) strict *trade-offs* between the number of PCIe lanes allocated for GPU-interconnectivity and network I/O, and (iii) general inefficiency with respect to latency and power usage of the systems, caused by the multiple memory copies that are usually part of the process, even when no computation is required by the CPU.

The latency aspect becomes even more relevant when analyzed in the context of real-time applications: as presented in Section 2.8, modern GPUs may complete processing within 1 ms, which is on the same time scale of traversing a standard Linux system's network stack [11, 127].

Removing CPUs from the critical paths in GPU-based systems, and increasing the integration of GPUs and NICs, represent thus a key step towards higher efficiency, lower latency, and better power efficiency for GPU-driven applications.

4.1.3 Problem definition

The observations discussed above lead us to a major problem in the *Networked Systems* research area:

> **Problem definition**: current State-of-The-Art networked GPU-accelerated architectures are CPU-centric, introducing *high CPU utilization* and *unneeded latency* costs at server side.

This problem is clearly related to current State-of-The-Art software implementations and commodity hardware technologies, as discussed in Chapter 2, but represents a *major*, often underestimated, problem for most AI-enabled applications. As introduced in Section 2.6, we believe this would be exacerbated by scaling GPU performance and the huge demand for AI-enabled workloads, making the

problem *highly relevant* for any high-performance workload at scale, potentially also applicable to other accelerator technologies and applications beside AI.

4.2 The challenges

In this section, the key challenges addressed by this work are discussed, as responses to the problems discussed above. We then describe the approaches used and the contributions related to these.

4.2.1 Challenge 1: High-speed connection tracking

Any high-speed networked system needs to be able to coherently and consistently associate incoming packets with some *flow*, and thus with the data associated to it. This is a fundamental step to be able to correctly execute any application, but also to guarantee isolation, accounting, authentication, and authorization. This problem is exacerbated when the inter-packet arrival rate is below 10 ns, at which timescale a single memory access can exhaust the whole CPU time that could be used to maintain the lane-rate, as it is the case for minimum-sized packets on a 100-Gbps link [94, 190].

Approach We analyzed the performance, and issues, of a State-of-The-Art Stateful Load Balancer based on FastClick [167], studying the performance of it at 100 Gbps. This *Network Function* represents a key functionality in any at-scale system, without which it would be impossible to realize server applications spreading across multiple machines, as required by modern processing-heavy network workloads, such as *inference serving*.

Scientific contributions We identified the main limitation of these applications in the slow and complex *table format* commonly used in this type of applications. We then analyzed different implementation of these data-structures, studying how they behave when hit by real-world data-traffic. We report our insights and results in Chapter 6.

4.2.2 Challenge 2: Network operations from GPU without CPU interventions

With the growth of AI-drive applications and the general lower pace of CPU evolution, enabling GPUs to perform network transfers actively becomes a crucial step to sustain the scaling of large systems, while maximizing the performance and reducing the overheads.

Approach We leverage RDMA and its support in current NVIDIA GPU drivers to perform data transfers from and to GPU memory, in the context of an *inference serving* application.

Scientific contributions We propose a novel implementation of RDMA routines running directly on NVIDIA GPUs and completely on commodity hardware, reducing the current need to synchronize GPU activities with the CPU in order to initiate data transfers over the network. We describe our solution in Chapter 7, studying some relevant performance metrics in the context of an *inference serving* application. We then propose another use case for this *building block* in Chapter 9.

4.2.3 Challenge 3: High-performance inference serving architectures

Inference serving is a type of workload that is becoming very popular as a consequence of the wide-spread introduction of AI-driven features in many applications. In näive implementations, but also in most common architectures, all data to & from clients traverse one or more CPU stacks (*e.g.,* multiple networked nodes' CPUs), increasing the total processing latency and overhead that these traversals can introduce. We believe that these workloads are ideal candidates to study and improve the interaction of high-speed network and GPUs, especially considering the current evolution of computing performance of these architectures.

Approach We deployed and analyzed a State-of-The-Art inference serving architecture, Clockwork [19]. We identified the main issues of this system as the centrality of the controller, and the high CPU requirements needed to relay the requests & responses through the controller's CPU stack. These introduce unnecessary latency in the system and high load on all nodes' CPU, which could be avoided by carefully redesigning the transport layer.

Scientific contributions We present our insight on Clockwork in Section 8.1.1 and propose an alternative implementation to solve some of these limitations in Section 8.3, based on the building block presented in Chapter 7.

4.3 Research question

Having acquired the relevant background (Chapter 2), discussed the State-of-The-Art contributions (Chapter 3), and the major problem we are addressing (Sections 4.1 and 4.2), we can summarize the main research question behind this thesis as:

Is it possible to realize a High-Bandwidth, Low-Latency, GPU-enabled building block without CPU involvement?

4.4 Research methodology

This thesis has been driven by an empirical research approach. Several measurements and observations have been made during this work to understand and analyze the dynamics behind the technologies involved, which are mainly based on proprietary hardware and closed-source drivers.

We continuously collected and compared data between each revision of our prototype, striving to maintain statistical validity for all data (*e.g.,* repeating all experiments multiple times and analyzing the distribution of values).

During our experimental exploration, we followed a closed-loop approach, where we (i) evaluated the performance of a system (ii) identified opportunities for improvements, and (iii) hypothesized & implemented a solution to address each of these opportunities.

Chapter 5

Experimental Setup

All the results presented in this thesis have been measured on a physical cluster, with dedicated hardware allocation to limit the possible interferences with other experiments or applications. This chapter describes the experimental setup, the hardware, and the techniques followed to collect these results.

5.1 Compute nodes

We performed all the experiments presented on this thesis on a physical testbed, composed of many physical server machines. For simplicity, we refer to the machines as *nslrackXX*, and we report the main characteristics of the systems in Table 5.1. All machines are based on dual-socket Intel architecture, with both sockets populated with Intel Xeon Gold CPUs. All GPUs and NICs are NVIDIA products. We list only the model numbers for all of these components.

A key characteristic of the last system, *nslrack34*, is the presence of PCIe

Table 5.1: Summary of the hardware used in this work.

Name	nslrack28	nslrack{29,30}	nslrack34
OEM	Dell	Dell	SuperMicro
Model	PowerEdge R740xd	PowerEdge R750	SYS-420GP-TNR2
CPU	2× 6246R	2× 6346	2× 6336Y
RAM	384 GB	256 GB	256 GB
GPU	Tesla T4	Tesla T4	A100 80GB + L40
NIC	ConnectX-5 Ex 100 Gbps	ConnectX-6 200 Gbps	ConnectX-6 200 Gbps
PCIe	3.0	4.0	4.0 + PCIe switches

switches, which are used as *fast-path* between the GPUs and the NICs, allowing the maximum performance achievable by *peer to peer transactions*, which represents the driving technology for the contribution described in Section 7.4.1. We note that this system is configured in a so-called *single root* topology, with all interested peripherals connected to the first CPU of the system.

All machines run Ubuntu 20.04 and the upstream kernel. We used the latest stable version of the rdma-core libraries, together with the LTS version of NVIDIA Mellanox OFED,* and the upstream version of NVIDIA GPU Drivers. All interactions with the GPU subsystem are powered by the upstream CUDA driver and libraries (respectively, 535.86.10 and 12.2 at the time of testing).

To maximize the reliability of our results and maximize the consistency of metrics, we disable all power saving features in the CPU, GPU, memory, and PCIe system and fix the clock speeds at their nominal clock speeds (*i.e.,* disabling Turbo Boost features). While these settings would be undesirable in a production environment, they reduce the variables that may affect the measurements. We also isolate the CPU cores (through the *isolcpus* kernel option and *taskset*) and force our application to **not** run on hyper-threads. This allows us to collect CPU usage metrics from these cores in isolation and reduces the contention that might be introduced by other tasks. Given the dual node Non-Uniform Memory Access (NUMA) architecture of all machines used in the testbed, we rely on the Linux Kernel to allocate resources on the correct NUMA node, *i.e.,* allocating local memory on the same node where GPU and NIC are connected. This would avoid any latency-expensive cross-NUMA data movements.

By fixing the GPU clock speed we can also reliably (and in a lightweight approach) measure execution times of each component running on the GPU, computing the number of clock cycles elapsed between two time points.

5.2 Data-plane network

Most of the experiments have been conducted by connecting the machines through a 100-Gbps Noviflow S9180-32X switch, using Active Optical Cables. Only the measured application traffic was traversing this network, allowing us to measure performance in isolation from other network activities. The switch is controlled by an OpenFlow [191] controller, which pre-installs static forwarding rules, preventing delays related to MAC address learning. We isolated the ports used in these experiments in a separate VLAN to block other hosts to affect the measures, and

*OFED (OpenFabrics Enterprise Distribution) is an independent distribution of drivers and libraries related to the Linux network stack (with RDMA in particular) which includes vendor-specific contributions, which usually have not been included yet in the upstream kernel code-base. NVIDIA Mellanox OFED is the version of this distribution specifically targeted to Connect-X products. [13]

we ensure that no other bandwidth-intensive traffic was traversing the switch (*e.g.,* in other VLANs or ports).

If not stated differently, all *network throughput* measures have been obtained by collecting the port counters on the switch via a Ryu-based custom controller [192]. We collect these counters at regular intervals during the experiments and compute all relevant statistical values.

This data represents the actual amount of bits that transit through the physical switch port and takes into account all overheads that the *Ethernet*, *IP*, *TCP*, or *RDMA* introduce. We note that this represents the most accurate way to measure physical throughput when dealing with *RDMA* or other kernel-bypassing techniques, since it does not rely on the number of packets handled by the NIC nor on the performance of the end host. Moreover, it avoids any overhead that a host-based packet counter collection might introduce.

5.3 Statistical validity of the results

To ensure statistical validity of the results, we run each experiment multiple times and aggregate the results through standard statistical techniques. Unless otherwise stated, we collect metrics at regular intervals and aggregate them, ensuring results represent all data points and not only a general average.

Each experiment is run long enough to reach a stable behavior (in most cases, a stable throughput), and we exclude the beginning and tail of each run, where performance may be unstable either because queues are filling-up, or because we have on-the-fly data at the end of the experiments.

5.4 Results reproducibility

Most of the measurements presented in this work have been obtained through the automatic testing framework *NPF* [193]. This is an improved version of the tool already used in many works (*e.g.,* [12, 127, 128, 165, 166]).

NPF reads a test description file, opens ssh connections to the servers involved in the experiments, runs the specified applications on the different machines, and collects the data printed by these for later processing. The resulting data-sets, exported in CSV format, are then read by a Python script that aggregates the results (*e.g.,* on the basis of the variables of the X axis), calculates the statistical figures, and generates the plots.

To maximize the reproducibility of the results, and foster follow-up works to this thesis, we intend to make all relevant source code available online. The source code related to the work presented in Chapter 6 has already been published in [194].

Table 5.2: Summary of the models used in this thesis. All inputs and outputs have a batch size of 1.

Name	Input shape	Output shape	Input size	Output size
ResNet50 [118]	1x224x224x3 FP32	1x1000 FP32	602 kB	4 kB
denoise [134]	1x224x224x9 FP32	1x224x224x3 FP16	1.8 MB	300 kB
superresolution [135]	1x224x224 FP32	1x672x672 FP32	200 kB	1.8 MB
superresolution_N* [135]	1x640x480 FP32	1x640Nx480N FP32	307 kB	$307N^2$ MB
fddNet_128 [133]	3x128x128 FP32	3x128x128 FP32	196 kB	196 kB
SqueezeNet [132]	1x224x224x3 FP32	1x1000 FP32	602 kB	4 kB

5.5 AI models used in this thesis

We report in Table 5.2 a summary of all the models used in this work, listing the input/output sizes used for the experiments.

5.6 NVIDIA DGX-H100

We use the NVIDIA DGX-H100 as a comparison system in several parts of this thesis, although not being directly the focus of our experiments. This system represents one of the main architectures the industry is moving towards for GPU-accelerated tasks, and it is currently the best show-case for NVIDIA technologies, including the latest versions of both GPUs, NICs, NVLink switches, together with latest generation CPUs.

The DGX-H100 represents the latest server machine (at the time of writing) commercialized by NVIDIA, featuring 8 NVIDIA H100 80 GB GPUs. These are interconnected through an NVLink subsystem, which features 4 NVLink switches, with up to 900 Gbps of throughput between each pair of GPUs (provided as 18 NVLink-4 lanes, each supporting 50 Gbps).

The system is controlled by a pair of Intel Xeon Platinum 8480C processors, coupled with 2 TB of main memory. Each GPU is connected through a 16x PCIe 5.0 link to a PCIe switch, which interconnects a GPU, a NIC, and a CPU endpoint. This setup, made possible through the proprietary *Cedar Fever* architecture [1, 195], oversubscribes the CPU's lanes, with the goal of maximizing the throughput between each GPU-NIC pair. The latter are 400 Gbps NVIDIA Connect-X 7 NICs, which are designed to link multiple chassis together, forming a *Super Pod* [196], by extending the NVLink protocol outside the single chassis domains, as already mentioned in Section 2.2.2.1.

We summarize the main system architecture in Figure 5.1, while the Cedar Fever internal architecture is represented in Figure 5.2.

*Where N is an integer number, *e.g.,* 1,2,3, ...

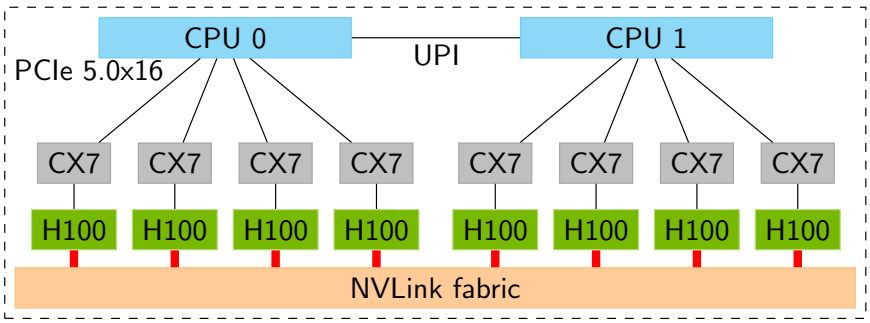


Figure 5.1: DGX H100 Topology, adapted from [1].

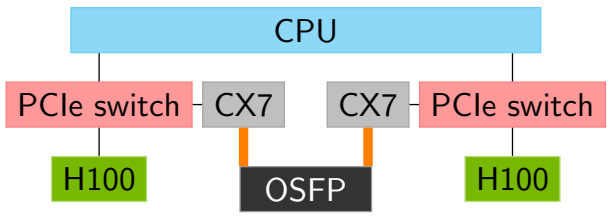


Figure 5.2: Detail of the Cedar-Fever architecture: a PCIe 5.0 switch connects a CPU endpoint, a CX-7 NIC and a H100 GPU. All links are PCIe 5.0x16. Each physical Octal Small Format Pluggable (OSFP) port is shared between two CX-7, with a 2x400 Gbps bandwidth.

Chapter 6

Connection Tracking

In this chapter, we seek to analyze the performance of different *Connection Tracking* techniques running in a Virtual Network Function chain running at 100 Gbps. The aim of this investigation is to understand what would be the best approach to provide Connection Tracking features to high-speed applications, as it could be adopted in any front-end device in a datacenter environment. We studied these implementations within the context of a *Stateful Load Balancer*, but most of our findings can be directly applied to other types of application running on commodity platforms.

Section 6.1 introduces in more details the problem, Section 6.2 describes the data-structures analyzed, and we report our insights in Section 6.3. Finally, Section 6.5 summarizes this chapter.

The work presented in this chapter has been published at *2021 IEEE 22nd International Conference on High Performance Switching and Routing* [12]. The text material and graphics are subjected to the following copyright:

6.1 Introduction

Connection tracking is a critical task for many modern high-speed network applications including load balancers, stateful firewalls, network address translators,

and general-purpose networking stacks. The goal of connection tracking is to associate the connection identifier of a packet to a *connection state* (*i.e.*, the data that an application stores and uses to process all the packets belonging to a specific connection). It is therefore essential to design the underlying data structures and algorithms used for connection tracking efficiently, so that network applications can operate at multi-gigabits speeds and with minimal latency.

The most common data structures used for connection tracking are hash tables. These data structures efficiently map each input key to its associated value. Different hash table implementations exist and vary in their performance guarantees. The Linux TCP networking stack tracks TCP connections using "chained" hash tables [197]. An eBPF program uses the same type of hash table to share data with user-level programs [198]. The Facebook Katran Load Balancer [199] relies on the eBPF's implementation to map incoming packets to the correct back-end server. CUCKOOSWITCH [200], Krononat [201] and MemC3 [202], use "cuckoo" hash tables [203].

The rise of network card speeds up and above 100 Gbps [204–206] poses some unique requirements on the design of the connection tracking component (and its hash tables). We focus on network functions and stacks running on commodity servers, which is the prevalent deployment approach in the emerging SDN/NFV-based modern networks [199, 207–212]. The following properties should be supported: (i) *efficient operations* into the table as packets arrive at high-speed networking interface cards in the order of tens (or hundreds) of nanoseconds, (ii) *low memory* utilization as efficient packet processing requires keeping the data structures in the memory caches to the largest extent possible, (iii) *scale to multiple cores* as a single core cannot cope with all the incoming traffic, and (iv) *handling up to millions* of flows, a common requirement for datacenter load balancers [213].

Our analysis, presented in Section 6.3, could be divided as follows:

- We first measure the throughput and performance of insertion and lookup operations with five different types of hash tables that run on a single core. We carefully selected some of the most efficient implementations already available in the literature or used in open source projects.
- We then quantify the scalability bottlenecks of six different techniques that can be used to scale a data structure in a multi-core environment.
- We compare different thread-safe hash table implementations with the widely adopted *core sharding* approach [201, 214–217] where each core is assigned its own hash table and the network card sends each incoming packet to the core that has the connection state to process that packet.
- We finally focus on deletion operations on the hash table. Deletion operations

are typically performed after the application does not receive a packet of a certain connection for a certain (configurable) amount of time.

We note that deletion operations on general hash tables have received fewer attentions [218–220] than insertion and lookup operations [200,202,214]. Deletion operations are however performed *as often as* insertion operations and must therefore be implemented efficiently. Deletions are also key to achieve low memory utilization (by promptly evicting non-active connections flows) and avoid the well-known performance degradation that arises when the number of elements in a hash table approaches it maximum limit. We focus our measurement of deletion techniques on three different mechanisms. In a multi-thread application, the deletion is even more cumbersome: other threads may access the connection table while an entry is deleted, requiring further controls.

We derive the following main findings for any developer implementing connection tracking in any application:

- Recently proposed hash tables improve throughput performance by around 15% in multiple scenarios over a traditional "chained" hash table. The performance however sharply degrades as the utilization of the tables get close to its maximum.
- Relatively simple multi-core scaling solutions such as spinlock, mutexes, or hierarchical locking scale extremely poorly, not being able to take advantage of more than three cores for a load balancer use case.
- Core-sharding is the easiest scaling approach for multi-core network applications. Lock-free techniques achieve similar performances of core-sharding at the cost of an increased complexity for deletion operations.
- Traditional deletion techniques based on a "timer wheel" performs equivalently to recently proposed "lazy" deletions without making insertion operations more complex, and scales better. Scanning the table to evict connections should be avoided.

6.2 Data-structures for connection tracking

Connection tracking applications rely on a *connection table* to perform the association between the connection identifier of a packet (*e.g.*, the TCP/IP 5-tuple, a QUIC connection identifier, the GTP TEID of mobile traffic) and the corresponding stored connection state.

As modern network applications must process packets within a very strict time budget to avoid packet queuing (and consequently packet drops), the implementation of the connection table has received much attention in the past. On a 100 Gbps link, the time budget for processing a packet can be in the same order of magnitude of a memory access.* It is therefore of paramount importance to keep the connection table small so that it better fits in the CPU memory caches.

Binary trees and hash tables [222, 223] can both be used to implement a connection table. Binary trees offer good performance when used to classify packets based on general filtering policies (possibly including wildcard matches), but suffer from costly updates and inserts. When the classification task is performed on a specific set of fields with an exact match, as it is the case for connection tracking, hash tables provide better performance (*i.e.*, constant-time lookup and average constant-time insertions). We therefore focus only on hash table data structures.

6.2.1 Hash tables

In its easiest form, a hash table stores elements based on an index obtained by *hashing* the key of the element. When two keys are hashed to the same index, a collision happens. The major difference between different hash table implementations is the technique used to handle such collisions. In this work, we focus on the following hash tables:

- *Chained hash tables* handle collisions using an *open hashing* strategy. Each bucket in the hash table stores all colliding elements in a linked list. A lookup operation simply entails accessing the linked list indexed by the hash of the element key and scan all its elements. Chained hash tables may lead to slow lookup operations as linked lists may grow arbitrarily large (containing all the inserted elements in the worst case).

- *Cuckoo hash tables* [203] implement a *closed hashing* approach, where collisions are resolved by assigning two (or more), independent, buckets to a given key. During a colliding insertion, all the buckets for that key are checked and, if all are occupied, a swap process is started. The incoming element is inserted in any of the buckets and the insertion procedure is repeated for the replaced element. The main advantage of this algorithm is the constant-time lookup operation, which is independent of the table utilization. Several improvements have been proposed upon the original schema, such as partial hashing [202], concurrent optimizations [224], and

*On a 100 Gbps link, minimum size packets arrive each 6.72 ns, while a memory access requires around 60 ns [184, 221].

pre-fetching predictions [214, 224]. Cuckoo hash table implementations are broadly available in many data-plane libraries, such as DPDK [225]. We evaluate the implementation offered by DPDK [226], using a traditional design enhanced with buckets that can store up to K elements instead of only one.

- *Cuckoo++* [214] improves the cuckoo baseline by adding a Bloom filter [227] to each bucket. This filter indicates whether a key is certainly not stored in the other possible bucket, saving one unnecessary memory access to that bucket.

- *Hopscotch* [228] is a closed hashing scheme where entries are relocated closely to the original location of a key. It resolves collisions by storing the colliding entries in a *neighborhood* of the initial location. If the neighborhood fits in a cache line, the cost to access any colliding entry will be close to finding an entry in the primary location.

- *Robin-hood* [229] is a closed hashing scheme that relocates entries using a linear-probing approach: when a collision is found, the following adjacent entries are iteratively inspected, swapping the entries if the inspected entry is closer to its original location than the one that needs to be inserted.

All of these implementations behave similarly on a global scale. However, the slightly different implementation details induce distinct performance when employed in a high speed environment.

6.2.2 Multi-core approaches

As networking speeds have grown at a much faster pace than CPU core clock frequencies, handling full line-rate traffic on a single CPU core had become an increasingly elusive feat. Consequently, it is paramount to distribute the network application processing among several CPU cores. In connection tracking applications, this typically requires sharing the data structure used as connection table. To operate consistently and reliably, the different processes (or threads) of the application must coordinate their read and write operations to keep the table in a consistent state. The main existing approaches to share a data structure are:

- *Lock-based methods* provide mutually exclusive access to a data structure through an explicit synchronization primitive, called a *lock*. A lock on a data structure can be acquired or released by one process and relies on hardware atomic primitives to avoid race conditions. Different types of locks exist that differentiate on how the lock is acquired.

A *spinlock* is a locking mechanism that performs a busy-wait loop on the lock acquisition operation until it succeeds (*i.e.*, the lock on the resource has been released). This approach minimizes the accessing time for a usually-free resource, but it wastes CPU cycles when waiting for the lock.

A *mutex*, or mutually exclusive lock, is an improvement upon spinlocks. When waiting for a lock to be released, processes are moved to a waiting state, allowing the CPU to schedule other tasks. While this mechanism reduces the CPU usage, the transition from one state to another one introduces additional latency. Both spinlocks and mutex can be acquired either on an entire data structure (*i.e.*, the entire connection table) or on a smaller portion (*e.g.*, per-bucket granularity). In this work, we refer to these fine-grained locking mechanisms as *hierarchical* locks.

- *Lock-free methods* [230, 231] solves a pressing problem of lock-based mechanisms in which a failed process that holds a lock may indefinitely stuck the progress of the other processes. A way to implement lock-free data structures is to keep one or more version counters that are updated whenever the data structure (or some parts of it) are updated. Other processes check the initial and final version number to verify whether their read/write operations were performed correctly or should be corrected (or reverted).

- *Core sharding* is a well-known technique to completely overcome issues related to sharing a resource by assigning a distinct instance of a resource to each process. In modern network cards, RSS [232] distributes incoming packets to each core deterministically based on the connection identifier, *i.e.*, it sends packets with the same connection identifier to the same core using a hash-based load balancer. In the context of connection tracking, each process only stores the connection state for the connections that it receives from RSS, thus eliminating any need to share resources. For some connection tracking applications such as NATs, one may want to guarantee that packets belonging to a connection are delivered to the same core in both directions.

6.2.3 Flow aging and deletions

Handling connection termination is a delicate task in a connection tracking application. Most network protocols are time-based, relying on timing to update their status. For instance, even after sending the last FIN, a TCP connection must wait for a certain amount of time before being closed. Therefore, connection tracking applications most often recognize expired connections using a time interval: if packets have not been seen for a certain connection for a given amount of time, the connection is considered expired and its state can be deleted.

When deleting an entry, especially in a multi-thread scenario, attention should be paid in controlling that other threads are not accessing it, either simultaneously or after the deletion. In particular, the deletion process may interleave with a read-access by another thread: while the deleting thread is proceeding to delete an expired entry, the reading thread would prevent the deletion.* Thus, the deletion must be protected with some additional mechanisms, always leaving the table in a consistent state.

In this work, we analyze three approaches to delete entries from the connection table: a scanning-based, a timer wheel, and lazy-deletion.

- *Scanning-based deletion* is the simplest approach that merely consists in periodic scanning of the connection table: when we find an expired entry, we remove them from the connection table. The ratio between the scan interval I (*i.e.*, how frequently the connection table is parsed) and the timeout t (*i.e.*, what is the maximum age for a flow before being deleted) determines the aggressiveness of the algorithm: lower t will delete flows sooner (possibly incorrectly), while higher I will increase the number of flows that are deleted at a single round and requiring a larger table size.
  Aggregating more deletions at a single round may be more efficient for some connection table implementations: if the deletion requires a lock of the connection table, this may be kept during the entire maintenance process. While all the other packets will be delayed, the deletion can proceed faster thanks to the absence of interleaved accesses, speeding up the entire operation.
- *Timer Wheels* are abstractions to efficiently maintain a set of timers in software, used also in major software projects like the Linux kernel [233–235]. At high level, we associate a timer to every entry in the connection table, triggering the deletion of the entry on expiration. We implement it by keeping a set of *time-buckets* for different time ranges, and we store a connection identifier in a bucket with time range $[T; T+\delta)$ if the connection should be deleted in that time range. At regular time intervals, the timers registered in the current bucket are fired, deleting the corresponding connections.
- *Lazy deletion* follows a similarly proposed approach in Cuckoo++ [214], where an extra `last_seen` field is added to the hash table. The field is updated every time an entry is read from the hash table. During the insertion of a new entry, we detect a collision if the existing resident entry has not yet expired. We evict the expired entry otherwise.

*In that case, the entry would not be expired anymore.

6.3 Evaluation

We now evaluate the performance of different hash table designs using a simple L4 Load Balancer (LB) implemented in FastClick [167], a faster version of the Click Modular Router [236]. We use the different hash table implementations to store the connection identifiers (*i.e.*, the TCP/IP 5-tuple). The data-structures will then return an integer identifier which is used as an index to access the *connection states*. This array, pre-allocated during the start-up, contains the states of every connection. For the Load Balancer (LB), the state is the selected destination server. While we limit our analysis to a LB, the results are not closely related to the chosen application and could be generally applied to a multitude of other connection tracking application such as middleboxes, networking stacks, and key-value stores.

6.3.1 Test methodology

We perform our evaluation using two server-grade machines, interconnected via a 100 Gbps NoviFlow switch. One machine logically acts as a Traffic Generator (both client and server sides) while the other acts as a LB. The first machine is equipped with two Intel® Xeon® Gold 6246R CPUs, 192 GB of RAM and a Mellanox® ConnectX®-5Ex NIC, while the LB machine is equipped with a Intel® Xeon® Gold 6140, 192 GB of RAM and a Mellanox® ConnectX®-5 NIC. The LB receives and transmits the traffic on the same NIC port. FastClick uses DPDK 20.11 to access the network cards, with each thread statically assigned to a physical core, polling one receiving queue. The application runs on the NUMA node where the network card is connected, accessing only local resources on the same node. Tests are repeated 5 times to show the average value and the standard deviation in the figures.

6.3.1.1 Traffic generation

We use FastClick to replay a traffic trace recorded at our campus router (with $\sim$ 200k connections) and, in some experiments, we also rely on traffic traces from CAIDA [26], which has roughly twice the number of connections and represents traffic transiting at a tier-1 router. To achieve 100 Gbps, we replay up to 32 different windows of the trace simultaneously, shifting the IP and port pairs. This method has the advantage of keeping temporal spacing between the arrival of new flows and subsequent packets. Thus, the resulting traffic looks like the connections generated by a campus up to 32 times bigger or an IXP router with higher speed ports.

6.3.1.2 CPU cycle measurement

We measure CPU cycles for different types of operations by performing differences of the TSC counter, whose values are read before and after each operation on the hash table. While this may introduce an additional cost and for each packet processing operation, this is constant across all the analyzed implementations, giving a fair comparison between all methods.

6.3.2 Single-core performances

We conduct our research on six different hash table implementations, adopting the schemes already introduced in Section 6.2.

- **Chained**: two chained hash table using per-bucket lists to handle colliding entries, which are stored at the head of the list. We report the performance of the implementation distributed with FastClick, and C++'s *unordered_map* implementation from the standard library.
- **Cuckoo**: a cuckoo based hash table offered by the DPDK framework [226]. It improves the original cuckoo idea with partial hashing [202] and pre-fetching of the buckets [224].
- **Cuckoo++**: a cuckoo based hash table optimized with bloom filters [214].
- **HopScotch**: an Open Source hopscotch hash table implementation [237].
- **RobinMap**: an Open Source hash table implementation based of robin-hood hashing [238].

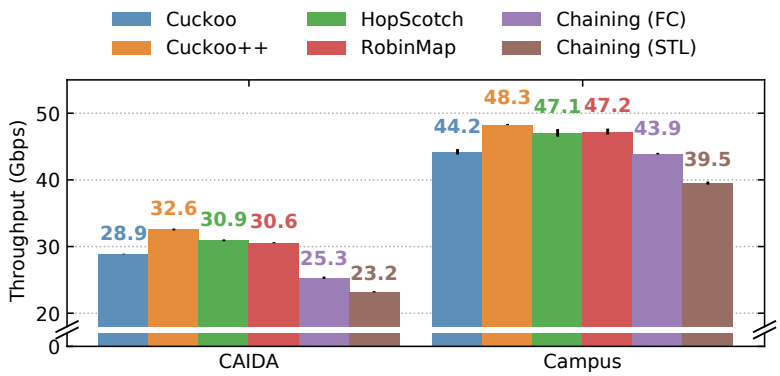


Figure 6.1: Performance of the 6 methods using a single core, under 2 different traffic scenarios. **Both cases shows identical trends, with up to** 28% **difference in performance.** ©2021 IEEE.

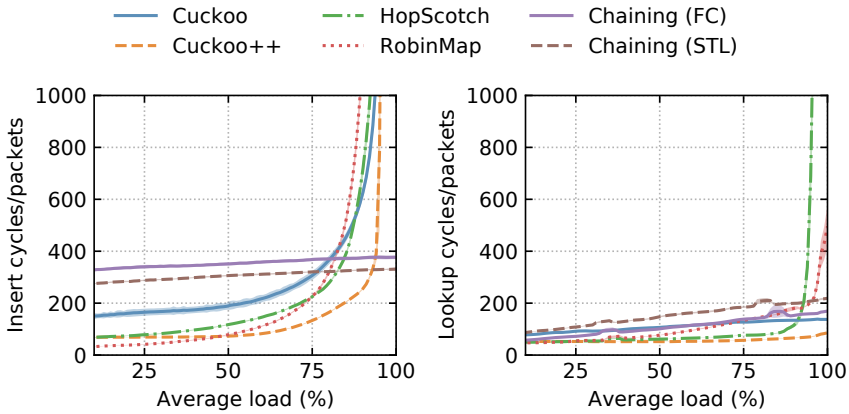


Figure 6.2: Number of cycles to insert and lookup entries, campus trace 16x, 2M entries. **Chained hash table cost increases linearly but performs worst than other methods.** ©2021 IEEE

Figure 6.1 shows the throughput of the LB when processing traffic with only one core using the six implementations with 16 windows of both traces (∼55 Gbps), ensuring the throughput is only limited by the performance of the LB. All hash-tables are provisioned with 4M entries that fit the 10 seconds of trace replayed (as we do not bring up recycling at this stage). The CAIDA traces contain twice more flows than the campus trace, exhibiting around 33% worse performance. However, despite those very different conditions, the trend between the methods is identical: Cuckoo++ performs best, up to $\sim 28\%$ better than FastClick's (FC) chaining hash table. The hash table from the C++ standard library (STL) performs up to $\sim 15\%$ worse than FastClick's due to heavier memory management while the latter relies on memory pools and a simpler, more efficient API instead. HopScotch and RobinMap perform similarly, slightly below Cuckoo++. DPDK's Cuckoo implementation performs, averagely, $7 - 10\%$ worse than Cuckoo++.

Figure 6.2 reports the number of cycles required to process a packet under an increasing table load factors. We do not report load below 10%: at this network speed rate as the load grows to 5% in a fraction of seconds and shows unstable results. While all the implementations slowly increase the number of cycles required to lookup for entries when the tables are becoming full, HopScotch and RobinMap lookup times become unsustainable earlier whereas Cuckoo and Cuckoo++ maintain an almost constant lookup time complexity. Recall that the lookup operation in Cuckoo searches an element in at most two buckets while RobinMap and HopScotch may have to search through more than two adjacent buckets, possibly through the entire table. The number of cycles for Cuckoo and Cuckoo++ lookups slowly increases despite an expected constant complexity:

when storing more connections in the table, only a small fraction of them can fit in the cache. We note the chaining method is always more costly up to 80% of capacity, but does not degenerate. The reader should not extrapolate that chaining is a safer solution, the trivia here is that tables should be provisioned big enough to ensure load does not go beyond 75% of their maximum capacity.

6.3.3 Multi-core scaling

We study the scaling of the connection table across multiple cores using six methods:

- **Spinlock**: all the access operations to the connection table are protected with a spinlock.
- **Mutex**: readers and writers are protected in a Single-Writer, Multiple-Reader schema. We use C++17 [239] `std::mutex`, acquiring an *exclusive* lock around any write operation on the flow-table. A *shared* lock is instead acquired by the readers.
- **Hierarchical Locking**: we adopt this schema to protect the *chained hash table*, locking either the whole table, the bucket, or the single entry depending on the operation. The STL implementation does not support such mechanism, hence we only study hierarchical locking with the FastClick implementation.
- **Cuckoo LB**: the table is shared among different threads using the *locking* mechanism offered by the *DPDK Cuckoo* hash table implementation. This mechanism offers multiple-reader, single-writer protection using a lock [240] around critical operations on the table, with a strict integration with the hash table code.
- **Cuckoo LF**: the table is shared among different threads using the *lock-free* mechanism offered by the *DPDK Cuckoo* hash table implementation, based on a *version counter* and atomic operations [241, 242], similar to the idea discussed in Section 6.2.2.
- **Core Sharding**: we duplicate the connection table data structures per each thread, exploiting RSS and assigning one receiving queue to each processor, as discussed in Section 6.2.2.

Figure 6.3 shows the throughput for the core-sharding scaling approach and an increasing number of CPU cores used by the load balancer. We can observe how the throughput scales almost linearly for the six implementations using sharding up to 3 cores. From this point, the bottleneck of the system is the 100 Gbps network link, which saturates. To put the overheads of connection tracking into perspective, we note a forwarding configuration achieves 80 Gbps with a single core, and 100 Gbps with 2 cores.

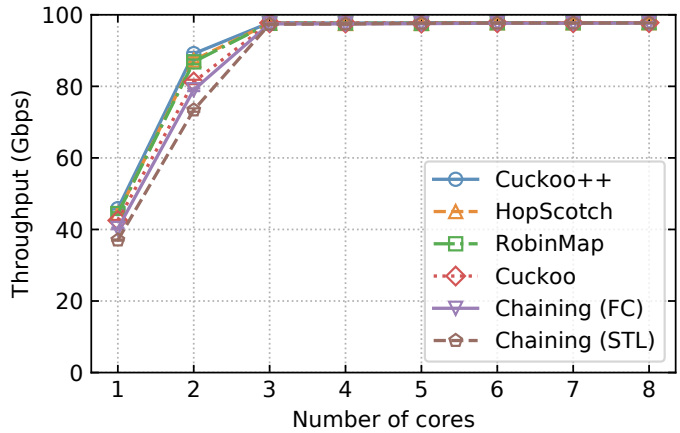


Figure 6.3: Scaling of the connection tracker using core-sharding: one hash-table per core, campus trace 32x.**All the implementations scale linearly until network bottleneck.** ©2021 IEEE.

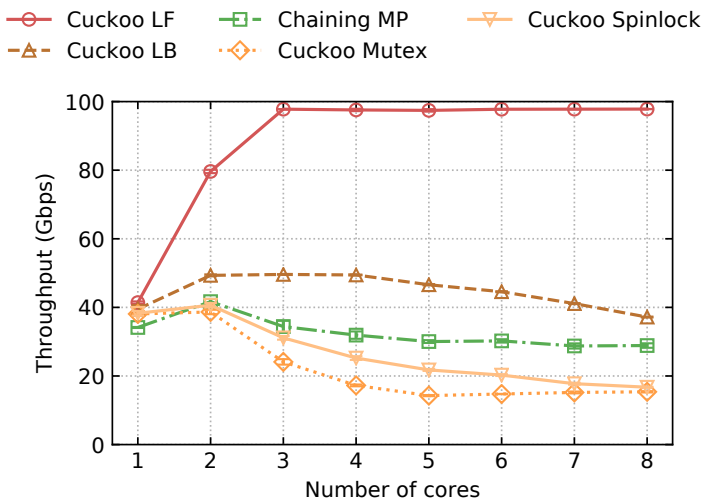


Figure 6.4: Scaling of connection tracker using locking techniques for a single hash-table, campus trace 32x. **Cuckoo Lock-Free scales linearly on the number of cores, while all lock-based mechanisms can achieve an almost constant throughput among cores.** ©2021 IEEE.

Figure 6.4 shows the throughput of different locking techniques for an increasing number of CPU cores used by the load balancer. We observe that the performance of lock-based methods sharply degrades when increasing the number of cores. The Cuckoo lock-free is the only one to compete with core-sharding, reaching 100 Gbps and offering an almost-linear scaling. Still, with 2 cores, Cuckoo lock-free delivers 67 Gbps while Cuckoo++ with core-sharding reaches 76 Gbps. The other implementations cannot process more than 61 Gbps even with eight cores, showing the bottleneck is the locking mechanism.

6.3.4 Deletion

We then compare the garbage collection techniques presented in Section 6.2.2 in Figure 6.5, using a single core and the Cuckoo hash table. The load of the CPU is around 70%. The scanning technique is heavy on the CPU cache, and induces an order of magnitude higher number of LLC cache misses. Moreover, even when scanning $1/1000$ of the connection table at a frequency of 1 kHz, the scanning is taking too long and packets are dropped as they accumulate in the receiving queue. We note that at a high enough frequency, the timer wheel mechanism performs similarly to the lazy-deletion. Surprisingly, if the length of the timer wheel buckets does not increase too much (ensured by a frequency high enough), the cycles spent in recycling are equal to the overhead of looking for expired entries and updating the time in the lazy deletion. One could argue that scanning could be performed by a remote, eventually dedicated, core. This would waste resources, breaking sharding and forcing costly synchronization mechanisms.

6.3.4.1 Scaling with deletion

When sharding is not possible, deletion becomes more complicated as one thread could recycle entries of other threads. Figure 6.6 shows the single-table using Cuckoo LF exhibits slightly lower performance than sharding because of the increased contention due to recycling. Similarly, the lazy method do not scale as well as the timer wheel because all threads have to scan buckets for recycled entries which force cache-line transfers.

6.3.4.2 Additional controls imply additional overhead

We also observed that the presence of multi-thread support influences the performance of the system even when using a single core (not shown in a graph). In particular, the Lock-Based approach of DPDK Cuckoo tables reduces the throughput by 10%, similarly to Mutex and Spinlock implementations, even when if there is no thread-race.

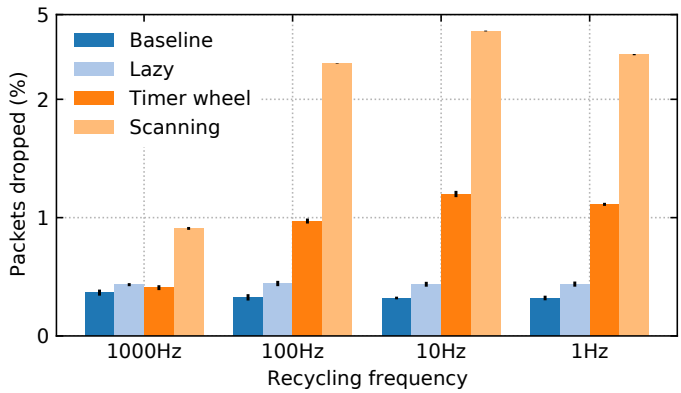


Figure 6.5: Comparison of deletion methods under an increasing garbage collection frequency running the Cuckoo implementation on a single core. **Lazy deletion is comparable to timer wheel at higher recycling frequencies.** ©2021 IEEE.

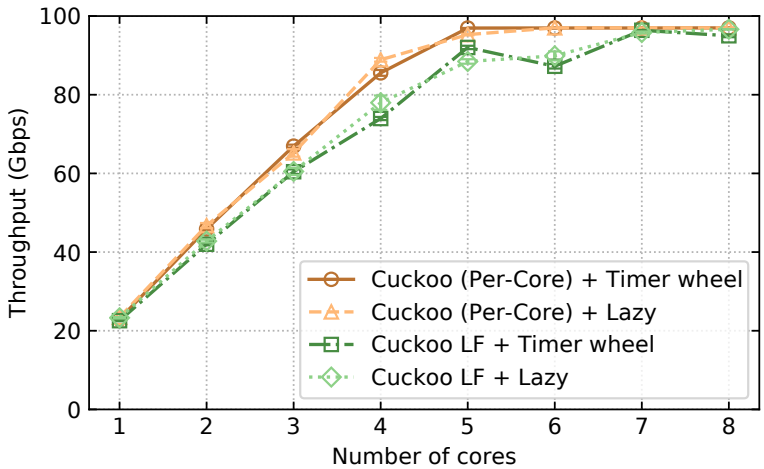


Figure 6.6: Scaling of two deletion techniques with sharded tables or a single lock-free Cuckoo table, Caida trace 32x. **Timer-Wheel per-core scales best, performing up to** 20% **better than Cuckoo LF using Lazy deletion.** ©2021 IEEE.

6.4 Related works

6.4.1 Hash tables

On top of the methods presented earlier, several works have focused on hash tables optimizations and implementations. CUCKOOSWITCH [200] optimizes the underlying data structures to exploit instruction reordering in x86 CPUs and dynamic batching. MemC3 [202] improves Cuckoo Hash tables with algorithmic and work-load based optimizations, some of which have been implemented in the cuckoo implementation used in this work. A stash-memory based solution has been proposed by A. Kirschet et al. [243], demonstrating how a small external area of memory can be used to amortize the cost of collisions and improving the performances at high load factors. Cache Line Hash Tables [244] tightly designs the implementation around the CPU cache structure to reduce the latency of the operations.

Othello [218] and Ludo [219] hashing take a more algorithmic approach, separating the implementation of the hash tables in a control plane and a data plane parts. Concury [245] uses Othello hashing to implement a fast load balancer while Concise [246] implements a packet forwarding mechanism based on Othello, where the control plane part is run on the SDN controller. RUBIK [247] and DUET [248] enhance the packet classification by moving some logic to the ECMP tables of modern switches. Krononat [201] presents a CG-NAT application built on top of [214], using core-sharding to scale for high bandwidths. SilkRoad [213] provides low latency Load Balancing by using ASIC switches, where a digest of the hashes is stored. Cheetah inserts the index value of the entry where the flow state is stored into each packet of a connection, thus relying on simple arrays instead of hash tables [249]. A hardware implementation of hash tables can be realized with Content Addressable Memories (CAMs), which provide a constant and low access time. However, the power consumption and the small capacity limit the scaling of this solution [250]. Recent releases in programmable hardware [251–253], together with recent advancements in this field [213,254,255], could open the path to further developments, where connection tracking is offloaded to the network card.

6.4.2 Flow ageing

Iptables implements connection tracking by relying on a Finite State Machine, with ageing regulated by timeouts [256–258]. Cuckoo++ [214] implements ageing with a lazy strategy, similar to our implementation, updating a timestamp when an entry is accessed. However, the authors do not evaluate its efficiency, claiming lazy deletion is supposedly better than timer wheels, which we could not confirm in our experiments. MemC3 [202] leverages a linked list that stores the last recently

used keys. Whenever space is needed, the key at the tail is removed from the table. Bonomi et al. [259] propose to use a flag to distinguish between recently used and expired entries, parsing all the table at regular intervals. Binned Duration Flow Tracking [260] divides the hash tables in a fixed number of ordered bins, across which the flows are swapped. When space is needed, the oldest bin is deleted. Bloom filters are used to speed up the lookup of the flows in the bins. Katran [199] tracks expired UDP connections with a timer, while TCP connections are removed when space is needed in the tables.

6.4.3 Future work

Recent releases of NICs with connection tracking mechanisms [205] may open the path to implementations that exploit hardware offloading on commodity hardware, freeing the servers' CPU from the connection tracking duty [165].

To improve the memory efficiency and processing time, the structure and the size of flow table data-structures may be changed at run-time. While static allocation of the structures is usually preferred, limiting the variables that may influence the packet processing time, some works have already explored this possibility [211, 261]. The transformation between different data structure types may result in a more optimized response under some specific loads (*e.g.*, one implementation may optimize insertions in an insert-intensive workload, while another may optimize lookups or deletions). This conversion requires a careful design in order to reduce the packet-processing time during the transition.

6.5 Conclusions

This chapter presents an analysis of six different hash tables implementations based on a load balancer application, showing up to 30% differences in performance. Cuckoo++ resulted the most efficient hash table implementation, gaining a 10% higher throughput than a basic Cuckoo implementation and up to 28% better than basic chained hashing.

The need to scale to multiple cores showed that only core-sharding and lock-free connection tables can achieve high-throughput. However, lock-free implementations heavily depend on the workload (*e.g.*, write or read intensive), requiring a careful design to keep the number of cycles low. Core-sharding is the only approach that can truly scale linearly on the number of cores, independently to the offered traffic load. We then explored recycling, an angle generally forgotten by other works. We found that timer wheels, when run frequently enough, compensate the cost of real-time garbage collection, delivering similar performances to lazy deletion. The latter is penalized by the additional cycles required for timestamp comparisons on all operations, achieving slightly lower performance.

Chapter 7

RDMA from the GPU Side

This chapter introduces the reader to RDMA operations in the context of a Linux application, providing a low-level analysis of the operations, and focusing on the relevant mechanism for our implementation. After an initial introduction to the RDMA concepts in Section 7.1, Section 7.2 presents the typical structure of a RDMA-enabled application.

Section 7.3 analyzes the *life* of an RDMA verb in a typical application and Section 7.4 describes the mechanics related to the *reception* of a verb. Finally, in Section 7.5 we describe the operating principles behind our GPU-driven RDMA stack implementation, which we evaluate in Section 7.6.

7.1 RDMA, InfiniBand, and RoCE

As already introduced in Section 2.3.3, RDMA can be considered a direct extension of DMA functionalities over the network, allowing remote access to a computer's memory from another computer without Operating System (OS) intervention and potentially without using any CPU cycle on the remote system. This functionality is made possible by using specialized NICs, which usually implement these functionalities in specialized embedded processors that can directly read or write data to & from the host's main memory through DMA.

Data transfers occur without direct CPU intervention or context switch, typically asynchronously with other system operations, thus reducing the latency of data transfers. While initially developed and designed for large HPC clusters and *ad-hoc* designed infrastructures, RDMA functionalities are nowadays available on commodity hardware (*e.g.,* NVIDIA Mellanox Connect-X adapters), and can achieve good performance even on ordinary network infrastructures [98].

On Linux, the RDMA stack is usually implemented through the `rdma-core` component [14]. This set of user-space libraries works with the Linux Kernel drivers to realize the complete RDMA functionalities, exploiting the hardware features exposed by the supported hardware, software, and NICs.

Among the vendors and models of NICs supported by this library, in this work we focus on NVIDIA Mellanox Connect-X® 5 and 6 NICs, but many high-level principles are common to all adapters, regardless of the physical hardware vendor. Being part of the umbrella of Linux projects, `rdma-core` is mainly developed by individual vendors, who periodically introduce new functionalities and support for newer devices. One of the main contributors, according to the repository statistics [262] is NVIDIA, which contributed significantly to the development of these libraries and to the RDMA standards.

Although multiple transport layers may be used to implement end-to-end physical communications, for a long time, InfiniBand represented the main technology used in large clusters to leverage the high performance achievable through RDMA [110]. Traditionally, RDMA has been used in large HPC systems and to deploy high-performance Storage Area Networks (SANs), while Ethernet was the main means of transport for general-purpose applications and for connections traversing the boundaries of datacenter.

We refer the interested reader to [93] for a general overview of RDMA, InfiniBand, and RoCE technologies.

7.2 Structure of a typical RDMA application

To gain a better understanding of the dynamics and mechanisms behind an RDMA-based application, it is beneficial to analyze the general structure of such an application. We will analyze a simple *ping-pong* application, where a client would write some data to a server, and receive the results back through one-side RDMA operations.*

1. At the application start, the *RDMA stack* is initialized, similarly to a TCP three-ways-handshake: Queue Pair (QP) are established, sequence numbers exchanged, and the remote memory addresses are transmitted between the client and the server. This typically happens on an *out-of-band* connection, *e.g.,* a standard TCP connection that is used only for these control messages.

2. Memory areas need to be *registered* with the RDMA stack to be accessible

*In general, RDMA is agnostic to *client/server* terminology. But for ease of explanation we will refer to *client* or *requester* as the active communication party, and *server* as the other side of the communication.

from the NIC. This is usually an *expensive* operation and should not be performed frequently at runtime.

3. When the client application wants to send some data (*e.g.,* the content of a certain location in memory) to the remote party for processing, a *verb* would be *posted*. The NIC would perform the transfer (typically, asynchronously and without CPU intervention), optionally informing the client application about the completion of the processing.

4. On the server side, a busy-waiting mechanism would wait for data to arrive (typically, looking at completion events or polling some area of memory for changes). This approach is similar to *DPDK queues polling*, but *without* the need to process any TCP/UDP stack.

5. With the data ready and already in the main memory, the server application would call the *application logic* to process it. In this simple *ping-pong* case, no significant processing would be performed.

6. The server application would prepare a *RDMA verb* to send the data back to the client, performing the same operations that happened when sending the initial data, in the opposite direction.

7. Eventually, the application terminates and the *RDMA stack* would be cleaned-up, with related data-structures destroyed.

Performing these operations at line-rate (*e.g.,* 100 Gbps and more) represents some great challenges, mainly due to the number of different devices that need to interact, and the tight timings that are allowed in these situations. These become even more relevant when further peripherals are involved in the processing of such applications, such as when data should be retrieved from disks or, as our focus, when external accelerators are involved. Multi-threading and information sharing across multiple cores could also increase the complexity, introducing the need for further synchronizations, or lock-free mechanisms.

7.3 The life of an RDMA verb

Having described the general high-level architecture of an RDMA application, we will now analyze more in details the mechanisms behind *posting a verb*. These are very similar in all RDMA stacks, but we will analyze them from the *RoCE* perspective. While understanding the low-level transmission techniques and the accurate implementations of RDMA semantics over the network is beyond the scope of this thesis, having a view of the user-level processing of verbs in a

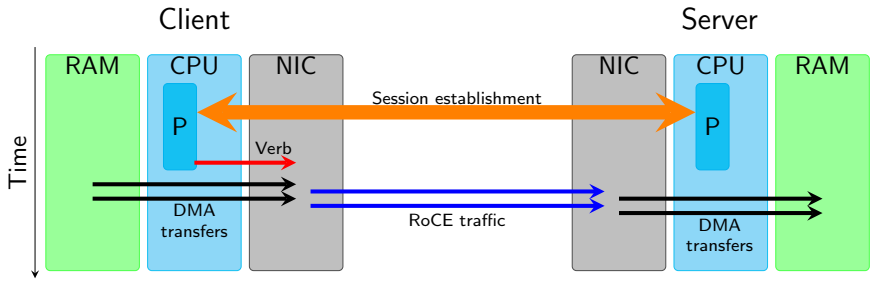


Figure 7.1: Structure of a generic RDMA communication between a client and a server for an unreliable 1-sided verb. (i) the client program establish the connection with a out-of-band channel (orange arrow), (ii) a verb is *posted* (red arrow), (iii) data is retrieved via DMA (black arrows) and (iv) translated in network packets (blue arrows). Finally, (v) the payloads are written to the remote peer memory. **CPUs are not involved after the initial communication establishment and verb posting.**

RDMA application is fundamental to analyze the GPU-side implementation of our RDMA posting routines. We refer the interested reader to [263, 264] for in-depth implementation analysis and to [93] for a high-level overview of the underlying technologies, while [265] presents the structure of a traditional RDMA application in more detail.

To simplify the analysis of the system, we will continue our journey in the RDMA system focusing only on *simple* RDMA verbs (*i.e.,* operations that write or read memory areas), without considering atomic verbs or other types of more complex interactions. We note that the software-NIC interactions for all of these verb types would be similar, and our analysis and contribution could be extended to other verb categories. Figure 7.1 shows the basic structure of a RDMA application, with a client writing data to a remote server's memory through a *1-side verb* over an *unreliable* connection (*i.e.,* a *WRITE* operation over a *UC* channel).

1-sided and 2-sided verbs We can distinguish RDMA verbs based on their interaction with the software stack on the remote side, which can be actively involved in the verb processing, or completely unaware of operations. We call this last class of operations *1-sided verbs*, which represents the most interesting aspect of RDMA since it allows the potentially complete offload of CPU on the remote side, as has been demonstrated by many works [266–268]. The most common verbs in the *1-sided* family are *Write* and *Read*, with their *2-sided* counterparts *Send* and *Receive*. We say that we *post a verb* when we instruct the adapter to perform an RDMA operation, and we distinguish between the two family of verbs also on the API calls used to *post* them. In the case of the *ibverbs* library:*

*Similar API calls are available on other libraries (*e.g.,* `rdma-cm`).

1-sided verbs are typically posted through a call to `ibv_post_send` by the *requester*, and no operation is required on the *server* side.

2-sided verbs are posted with a call by the *requester* to `ibv_post_send`, but a prior call to `ibv_post_recv` must have been done on the *server* side to assign the server's local memory destination (*e.g.,* the server-side verb needs to have a large enough buffer registered for the incoming verb). Sometime after the requester calls `ibv_post_send`, the server will receive the data that was sent, and a consequent operation on the server CPU will be executed as response.

We will continue our discussion focusing on *1-sided* verbs, which naturally fit the paradigm of *zero-CPU operations*.

7.3.1 Under the hood of RDMA verbs posting

Having presented the general architecture of an RDMA-based application, we can now delve inside the mechanisms that allow user-space applications to execute RDMA operations, focusing on *1-sided* verbs in the context of a Linux application.

Considering an application that uses the `ibverbs` library* to operate with RDMA, posting a verb translates into a single user-space call to `ibv_post_send` or `ibv_post_recv`, depending on the class of the operations, as discussed in Section 7.1, with the first used for *1-sided* verbs.

The core of these routines, beside constraints and error checks, is composed by the preparation of some data-structures containing all the metadata that *describe the operations*: these get copied from the user-supplied parameters, and re-arranged in NIC-readable structures, which are allocated in a specific area of memory, created during the initialization of QPs.† Once these have been prepared, the NIC must be informed, and processing *from the NIC's Processing Units (PUs)* would follow. We describe this mechanism in Section 7.3.2.

This represents a fundamental difference from a typical TCP (or UDP) stack, where the OS is usually in charge of the actual transmission protocol, typically involving kernel-space functions and, thus, expensive context-switches. When using RDMA the processing of the transmission is delegated to the hardware adapter,‡ with a minimal CPU cycle usage.

*The `ibverbs` library, part of `rdma-core`, is one of the principal approaches to RDMA at the application level, usually linked dynamically at runtime.

†At low level, the user-prepared data-structures get translated in the driver-specific counterparts, *e.g.,* `ibv_send_wr` would get translated into `mlx5_wqe_*` data structures for NVIDIA Mellanox CX-5 NICs using the `mlx5` driver.

‡Unless the system adapter is *virtualized* (*i.e.,* implemented in software).

Although more complex operations (*e.g.,* signaled operations, in-line data, or atomic verbs) generally use a higher number of CPU cycles, most RDMA operations require a constant number of CPU cycles, independent of the size of the payload, and much smaller than the number of cycles required for an equivalent TCP- or UDP-based transport.

7.3.2 Ringing the doorbell

Once the data-structures have been prepared within the *RDMA-stack* routines (as presented in Section 7.3.1), the NIC must be informed about the presence of operations to be performed, which would be offloaded to the PUs in the NIC. The mechanism that implements this signaling in RDMA applications is commonly referred as *ringing the doorbell*.

This process consists in writing the starting address and the size of these specific data-structures to a specific PCIe register (*i.e.,* `doorbell record`), and advancing a counter to notify the NIC about this data (*i.e.,* `wqe_counter`). These operations usually happen through *DMA transactions* over the PCIe bus.

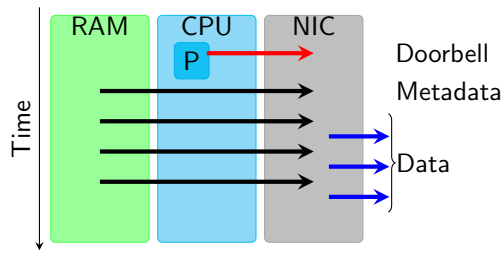


Figure 7.2: Data-flow for a RDMA Write verb: the user program is involved only initially (represented by the *P*, and the *red arrow*). Further data movement (*black arrows*) are performed by DMA engines. The NIC's PUs are responsible to translate the fetched data in network packets (*blue arrows*).

WRITE verbs Like most other RDMA operations, WRITE operations would result in two initial DMA transactions over the PCIe bus (*i.e.,* one for the *metadata* and one for the *doorbell record*). Once the NIC got the *doorbell notification*, one or more DMA operations will follow to fetch the relevant metadata, and subsequently the content of the memory area to be transferred. This data would be eventually translated in network packets' payload to be transmitted to the remote side. A sample RDMA WRITE operation is summarized in Figure 7.2.

READ verbs In the case of RDMA `READ` operations, instead, a symmetric data flow occurs: once the packets *requesting* the data are acknowledged by the remote

side, the NIC would start to receive a flow of data packets, whose payloads need to be written in memory through DMA operations. Optionally, one or more additional DMA transactions may signal the user-space application about the completion of the operation.

BF register Typically, any RDMA verb posting would result in two initial DMA transactions over the PCIe bus (*i.e.,* one for the *metadata* and one for the *doorbell record*), after which one or more DMA operations may follow accordingly to the verb being processed. To achieve better performance, the Mellanox implementation of the RDMA stack uses the Blue Flame Register (BF Register), an optimized mechanism that allows *ringing the doorbell* **and** *writing the metadata* as a single DMA write, reducing the PCIe traversals (and thus the overall operation latency). This is particularly relevant for small operations, where the PCIe traversal time would have a higher cost with respect to the actual retrieval and transmission of the content of the referenced memory areas.

We refer the interested reader to the Mellanox Adapters Programmer's Reference Manual [269] for an in-depth description of the posting mechanisms and to the patch that implements the functionality for a low-level technical description of the BF Register [270].

We use only the *BF Register* verb posting strategy in our prototype to reduce complexity and maximize performance.

7.3.3 Memory registration and protection domains

In order to provide security guarantees and prevent unauthorized hosts from performing any operation to RDMA endpoints, different isolation techniques have been implemented in the RDMA stack. At the basis of these mechanisms, there is the concept of *memory registration*, which configures a *Memory Region (MR)* in the RDMA stack. A *MR* represents an area of memory that can be accessed through RDMA verbs.

Memory registration happens through a `rdma-core` API call (*e.g.,* `ibv_reg_mr`), which generates the MR based on the parameters passed. A MR is characterized by a starting address, a size, and a *rkey-lkey* pair (respectively, *remote* and *local* keys). These *keys* are unique in the context of a single RDMA device, and need to be specified every time a verb is posted (*i.e., lkey* needs to be specified when referencing local MRs, *rkey* when remote MRs are targeted). Each Memory Region is also associated with Protection Domain (PD), which would prevent accesses from verbs posted or received in QPs belonging to different PDs. Each MR can be created with different *access policies*, for instance limiting remote RDMA endpoints to read-only operations.

Protection Domains allow isolation of objects (*e.g.,* QPs and MRs) when initializing the RDMA stack, similarly to *namespaces* in the Linux kernel. Objects belonging to different PDs will be prevented from interacting with each other, providing additional isolation guarantees.

Many concerns have been raised about the security model of RDMA, which is believed to not be suitable for untrusted, potentially hostile, networks such as multi tenant datacenters. We refer the interested reader to [271–274] for an analysis of some of these issues, which we do not study in this work.

7.4 Receiving 1-sided RDMA verbs

Having introduced the mechanisms behind the posting of a RDMA verb from the sender perspective, it is helpful to understand how these are processed at the receiving side, focusing on a RDMA `WRITE` scenario. Once a packet is received from the NIC, and all checks are successful (*e.g.,* the corresponding memory area has been correctly registered and referenced), the NIC would start to translate the received packets in DMA transactions to the main memory, without the need to notify the CPU.* When the last packet belonging to the operation is received, the NIC may inform the user-space application about the completed operation (if configured with specific flags), and/or send an acknowledgment response to the sending side, notifying the success of the operation.

7.4.1 NVIDIA GPUDirect RDMA and peer-to-peer memory accesses

Given the delicate, and tight, integration that must happen between all components involved in a RDMA-based applications, many efforts have been put from hardware producers and application designers to simplify, and improve, these interactions, often resulting in addition to the PCIe and RDMA standards.

7.4.1.1 Enabling technologies

Building upon PCIe Peer-to-Peer transactions (see Section 2.3.1), efforts from both the industry (*e.g.,* from GPU vendors) and the scientific world have proposed solutions to improve GPU interactions with other devices, NICs in particular.

In 2016, the first milestone in enabling GPU to control NICs was set by Daoud, Watad, and Silberstein in their GPUrdma [146], where an initial implementation was proposed to allow GPUs to launch InfiniBand RDMA verbs without any CPU intervention. To achieve this, a custom proprietary driver was developed, and the

*Unless explicitly requested in verb's metadata (and thus in the RDMA packets' flags).

GPU runtime required modifications to make the memory areas accessible both from the GPU and the InfiniBand Host Channel Adapter (HCA).

Although most concepts and operations can be transparently mapped to a RoCE stack, the need for a custom-developed driver, and the InfiniBand semantics of this work limit its applicability in modern datacenters, which are shifting to higher speeds and Converged Ethernet architectures.

7.4.1.2 NVIDIA GPUDirect

At the same time of GPUrdma publication, the first release of `nv_peer_memory` was released [275]. This is the result of a joint effort between NVIDIA and Mellanox,* enabling a direct data path between NVIDIA GPUs and Mellanox NICs (or HCAs). Being developed as a kernel module, this works in tandem with the driver package (*i.e.,* Mellanox OFED) to enable memory registration and mutual access between devices.

The result is the ability of RDMA verbs to read, write, and in general access, *registered memory areas* backed by memory in the GPU address space. To allow this interaction between third-party devices and NVIDIA GPUs, bilateral integration needs to be done in the respective drivers, allowing them to exchange information on the memory addresses involved, *e.g.,* exchanging information on the used memory pages.

Considering the NVIDIA driver, this page mapping between GPU space and the other device's driver was initially provided by the *nv_peer_mem* kernel module [275], and later by the NVIDIA driver directly through the *nvidia-peermem* kernel module [88].

Mellanox network drivers, and any third party device driver in general, need to obtain the physical addresses of the GPU device, which are exposed through the Base Address Register (BAR). Each PCIe drive can expose up to six BAR registers, which then represent up to six linear memory regions that can be accessed in GPU memory. Once one of such a region has been set up, the other PCIe devices can access without any difference from a system main memory access, issuing DMA read and writes to the BAR address.

Device resources are commonly mapped in the user or kernel address spaces, so that applications can access them transparently through the CPU's Memory Management Unit (MMU), in the form of Memory Mapped Input/Output (MMIO) addresses. Common Operating Systems (*i.e.,* the ones based on the Linux Kernel), lack mechanisms to support the exchange of information between drivers, such as MMIO regions, hence the need for the *nv_peer_mem* driver, which exposes specific APIs to directly obtain the address of these exposed memory areas, translating them

*At the time of the first release (May 2016) NVIDIA and Mellanox were two separate companies.

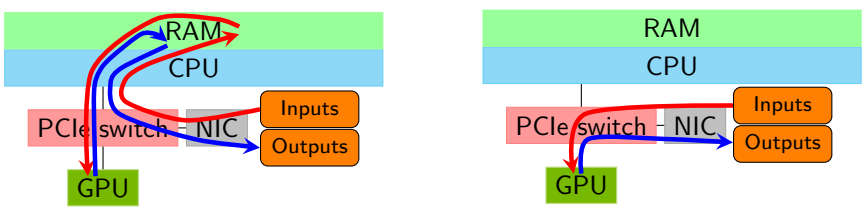
to usable memory pages. Third-party device drivers need then to interact with this kernel module, similarly to how the standard *user pages* are obtained from the Linux kernel.

When a user program issues a transfer, the device's driver needs to verify whether the backing physical address is ready for transfer, or if it has to be handled by the kernel driver. In the latter case, the kernel driver will interact with the *nvidia-peermem* driver to translate the virtual address in the physical pages of the GPU, which would be fixed for the application duration. We refer to this operation as *pinning the memory*. The addresses of these memory pages are then used to program the third-party DMA engines, and the transfers should follow accordingly.

A similar approach is needed for RDMA transfers, with the NIC's driver (within the RDMA stack) programming the DMA engines with the physical page addresses obtained from the *nvidia-peermem* driver. To allow RDMA operations to work correctly on these memory areas (*e.g.,* to ensure consistency), some optimizations must be disabled on the GPU memory management system. This is done by specifying the flag `CU_POINTER_ATTRIBUTE_SYNC_MEMOPS` when pinning the memory pages. We note that these attributes disable some optimizations, which may affect the performance.

The benefits of this GPU-NIC interaction are summarized in Figure 7.3: when GPUDirect RDMA can not be used (Figure 7.3a) four memory copies are needed to move data to GPU from network endpoints and back, while GPUDirect allows the NIC to write the data directly without any CPU intervention or RAM intermediate copy. This both saves CPU cycles, reduces PCIe bus contention, and improves the total round-trip time.

As discussed in Section 2.7 and in Section 2.2.4, this is a fundamental step to achieve the maximum performance of the hardware while at the same time reducing both latency and CPU usage.



(a) RDMA - data is intermediately written to main memory.

(b) RDMA - directly to/from GPU RAM with GPUDirect.

Figure 7.3: RDMA vs. GPUDirect RDMA data flow. **At least two memory copies can be avoided when using GPUDirect RDMA.** The *input* path is represented by the *arrows*, while the *blue* arrows represent the output path.

7.4.1.3 Active and passive roles

While these technologies, together with more HPC-oriented frameworks, such as NVSHMEM [142] and UCX [17] provides the basis for GPU-NIC interactions, there is no straightforward implementation, in upstream stacks, to *control* a RDMA NIC from a GPU. In other words, the GPU's memory can serve as *target* for RDMA verbs, but these verbs need to be posted from the CPU side, or generated from a remote host (*e.g.,* when reading data from a client).

We note that both NVSHMEM and UCX provides mechanisms to perform these operations *from the GPU*, but the semantics, and the granularity, offered is usually oriented to *many-to-many* interactions, which are the main style of operations in the target environments for these frameworks, which are HPC-like clusters and highly-homogeneous networks, usually with single-vendor deployments, as already introduced in Section 3.1.1.

7.4.1.4 The AMD way

Beside NVIDIA implementation, other vendors proposed solutions to improve GPUs and NICs interactions. AMD ROCm documentation [189] mentions how their MPI/UCX implementation is based on the *ib_core* module, the same on which the NVIDIA peer-to-peer implementation relies, enabling AMD GPUs' memory to be read, and written, through RDMA verbs. We believe that a similar implementation as given in this work could be easily ported to the AMD HIP platform, given the little platform-specific code present in our library implementation and the support to the same underlying standards in both architectures.

7.5 Extending rdma-core for GPU-driven operations

Having analyzed the general structure of RDMA applications and their semantics, the support by NVIDIA GPUs for PCIe P2P operations, and the mechanisms behind these technologies, we can now analyze our contribution, which aims to answer the following question:

> Can we improve the performance of GPU-accelerated RDMA-based applications by seamlessly integrating these technologies together while reducing CPU usage?

Given the capability of modern GPUs, we believe that a solution to this problem consists in ***performing RDMA operations directly from a GPU***, without CPU

intervention, leveraging standard drivers, and without *replacing* the whole network stack for existing applications. We will analyze these aspects from a *server point-of-view*, although a client-side applications could benefit from the same advantages, given the use of the same underlying mechanisms.

7.5.1 Challenges

We can summarize the main challenges to solve this problem as follows:

1. **GPU memory registration**: to access GPU memory from the RDMA domain, an interaction between the GPU and the NIC drivers need to be established. We describe our approach in Section 7.5.2, based on the functionalities provided by *nvidia-peermem* module as described in Section 7.4.1.

2. **Sending data from the GPU**: the ability of *posting RDMA verbs* from the GPU runtime is the enabler to transmit data to remote endpoints (*e.g.,* clients) without CPU intervention. We describe our approach in Section 7.5.3.

3. **Receiving data on the GPU and trigger processing**: the driver integration described above naturally allows RDMA verbs to read, and write, data to GPU memory. However, we need to design a mechanism to avoid involving the CPU to receive the *"Data is ready!"* notification and trigger processing. We describe it in Section 7.5.8.

4. **Consuming Completion Events from the GPU**: the RDMA stack needs to be periodically *checked* for errors and successful completion of the operation. This is fundamental in order to allow applications to run for a long period of time (*e.g.,* for more than the size of a *working queue*). We describe this mechanism in Section 7.5.5.

5. **Continuous execution without CPU intervention**: while submitting workload to GPU is an asynchronous process, and no explicit `cuda*Synchronize` call is required when everything is performed on the GPU, the length of this *outstanding work queue* is limited, and the operation becomes synchronous when the queue is full (*e.g.,* invoking kernels would take considerable CPU time). We need to design a mechanism to continuously process data on the GPU without the CPU submitting work. We use CUDA Graphs to implement this functionality and describe it in Section 7.5.6.

We describe the solution of these challenges and the structure of our implementation in the following sections.

A simplified, modular implementation Our approach relies on a minimally modified `rdma-core` library,* mainly with functions exposing underlying driver data-structures.

Similarly to how a normal RDMA-based application would be dynamically liked to the `rdma-core` library, an application exploiting our GPU-capable routines would need to link to our version of the `rdma-core` library, together with our `rdma_shim` library, which provides the functions reimplementing the critical components of verb-posting. All other operations involved in the setup and management of the *RDMA stack* would use the original implementations provided by `rdma-core`. Following this two-stages implementation, it is possible to easily re-base our changes on top of different `rdma-core` versions, while maintaining an independent implementation for our routines. Moreover, this approach allows applications to be able to use both the standard and our custom routines interchangeably, since our implementation is designed to be a drop-in replacement requiring minimal code changes.

The NVSHMEM and UCX approach to RDMA Although this approach is similar to the implementation used in NVSHMEM [277] and Unified Communication - X Framework (UCX®) [278], we believe that our approach is more flexible, as it can be integrated, updated, and maintained easier. Both of these libraries import directly part of the `rdma-core` code in the code-base, tightly coupling with the specific releases of `rdma-core`. When new features are introduced in the `rdma-core` libraries, or bug are fixed, manual porting is required to map the changes to the imported code bases in these libraries, while our approach would require manual changes only when changes are made on the specific *posting routines*.

7.5.2 Memory allocation and registration

In a standard *RDMA stack* implementation (*e.g.,* when using `rdma-core`), during the initialization of QPs, Completion Queues (CQs), and other data-structures, memory areas are allocated with the standard C++ API (*e.g.,* `malloc`). Pointers to these areas are then used throughout the application to interact with the NIC and perform RDMA operations. When these operations are performed from the GPU side, these memory areas must be exposed to the respective address-space, which is by-default separated. Through *Unified Memory*, CUDA applications can access host memory that has been previously registered *transparently*, as described in

*For only the NVIDIA Mellanox `mlx5` driver implementation, we modified 330 lines of code, while the implementation of this driver is composed by 41084 Source Lines of Code (SLOC) [276]. As a reference, the whole `rdma-core` stack is composed by 180944 SLOC.

Section 2.2.2.4. To allow the `rdma-core` library to allocate memory portions in custom memory regions (*i.e.,* accessible by CUDA runtime) we replaced all invocations to `malloc` and `calloc` functions in the `rdma-core` library with some wrappers, which by default invokes the respective original functions (*e.g.,* `malloc` and `calloc`), without changing anything for a *standard* application.

An application using our libraries can then arbitrarily provide a custom definition for the `malloc` and `calloc` functions, allowing to implement custom allocation functions. These could, for instance, allocate memory in a previously initialized and registered CUDA-accessible memory region (*i.e.,* through `cudaHostRegister`).* We note that this approach may introduce potential security risks (*e.g.,* reducing the code and memory isolation), but we believe future works can address these issues, perhaps exploiting future features in the CUDA driver (*e.g.,* using CUDA physical memory mapping for driver data-structures) and native security-oriented features of modern GPUs (*e.g.,* NVIDIA Confidential Computing [279]). In our prototype implementation, we allocate these memory areas through `cudaMallocManaged`, which directly expose them to the GPU runtime address space.

In addition to the memory areas manipulated by RDMA operations, the driver regions, and the areas holding the general RDMA stack status, the GPU runtime must be able to successfully *ring the doorbell*. This PCIe register, thus, needs to be exposed to the GPU runtime, similarly to the other memory areas. In *CUDA semantics*, this translates to a *registration* of a further memory area through the memory registration API (*i.e.,* `cudaHostRegister`), specifying the additional `cudaHostRegisterIoMemory` flag.

7.5.3 Verb posting from GPU side

As discussed in Section 7.3.1, most of the operations involved in the *verb posting* consists on *memory copies* and manipulation of data-structures. While these routines are originally coded in plain C language, it is not possible to execute them directly on a CUDA application, mainly due to CPU-specific instructions and very-specific optimizations that are needed to achieve good performance while supporting many different NICs and functionalities.

To solve this issue, we have reimplemented a verb-posting routine in less than 200 lines of C++ code, with some minor components written in CUDA assembly language (*i.e.,* Parallel Thread Execution (PTX)), *without external dependencies on other functions or libraries*. We can thus compile, and run, these routines on both

*Another approach would require to manually map every data-structure used by the `rdma-core` stack, which would reduce the portability and maintainability of it, *e.g.,* when adding newer fields or using different NIC's features.

CPU and GPU, without the need to maintain different code bases for each device (*i.e.,* the majority of the source code is written only *once*). We rely on C conditional preprocessor blocks to select the correct implementation at compilation time for those instructions that are device-specific.

7.5.4 Receiving data on the GPU side

While the RDMA semantics provides specific flags to *signal* to applications running on CPU about completion (or reception) of verbs, receiving these notifications typically involves a busy-waiting loop (*e.g.,* the application poll the `rdma-core` API). Instead of reimplementing this mechanism in the GPU runtime, we propose a *lazy approach* where the application would just read (periodically) the input buffers. When a client's write to a specific *slot* is completed, the verb would set a specific flag (*e.g.,* a specific value is included as part of the verb payload at a fixed location), and thus the *wait* routine on the GPU knows that the data belonging to that input *slot* is ready to be processed. This approach is ideal for fixed input applications, like ML inference, where the inputs are typically fixed, but could be easily extended to other scenarios (*e.g.,* performing two different writes, one for the request's metadata and one for the payload, with this routing checking the metadata area). Complex worker-side schedulers may also be implemented, *e.g.,* by accessing the buffers in specific orders, or by implementing software-level queues.

7.5.5 Consuming CQE

Completion events are needed to allow a RDMA-based application to work seamlessly without exhausting all resources, proceeding through a continuous flow of packets to be verbs and received over the network. Completion events are generated by the NIC when processing some specifically *flagged* verbs, which would result in the NIC populating some metadata data-structures when processing them, similarly to the mechanism used when *posting verbs*.

These are eventually *consumed* by the application with a polling routine, which would advance some specific PCIe registers, similarly to the mechanism used when posting verbs. Instead of relying on the CPU to consume these events (which would potentially require further synchronization), we implemented a simple consumption routine directly in GPU-side code, with a structure similar to the one used for *verb posting*, without explicitly checking for execution errors.

Error handling & future works We believe that in a *run-to-completion model*, as the model used in our prototype application, any error-specific action (*e.g.,* results cannot be sent) would (i) be unrecoverable (*e.g.,* the Service Level Agreement (SLA) would be already expired) or (ii) introduce too high an overhead, so we

simply ignore these cases and proceed to process the following requests. We leave the implementation of these features to future works. We note that in RDMA applications, when a QP would hit an error, this error is often unrecoverable, requiring most data structures to be re-created and reinitialized (*e.g.,* QPs need to be re-created in most cases). In our GPU-driven implementation, this would require CPU intervention, destroying the performance benefits.

With a careful accounting of posted, outstanding, and completed requests, an application can prevent many of these unrecoverable errors from happening. We believe that this approach could be enhanced by using network-level monitoring appliances (*e.g.,* using P4 programmable switches [280]), polling NIC counters periodically, and continuously monitoring the performance of the application: these would be used as feedback for a controller (or more generically to a Load Balancer) to adapt scheduling and avoiding errors that may arise from congestion.

7.5.6 GPU-side continuous loop

When the same sequence of operations is executed on a GPU, CUDA Graphs can be used to improve the efficiency of submitting workloads, as introduced in Section 2.2.2.3. In our implementation, a CUDA Graph containing all the steps needed to process a single input (*i.e.,* waiting inputs, processing, sending outputs, and optionally consuming CQE) are *captured* in a *Graph* during the application start-up, and later executed sequentially (*e.g.,* inside a CPU-driven loop). We further improve this mechanism by implementing this *infinite loop* functionality with *self-launching* Graphs (introduced in Section 2.2.2.3), avoiding any CPU-GPU interaction during the normal run-time of the application.

7.5.7 Final application workflow

We envision a relatively straightforward approach for an application developer who wants to adopt our GPU-side RDMA implementation. At a high level, the steps to be followed to adopt our implementation are:

- Change the `rdma-core` library to our implementation
- **Override the memory allocation** routines, as described in Section 7.3.3.
- **Memory registration:** all data-structures used by the driver and the NIC's registers, should be registered in the GPU memory space.
- **Remove memory copies**: inputs, and outputs, data would be directly located on the GPU memory, hence no explicit memory copy with the CPU needs to happen. The GPU routines would access directly the same memory areas where the NIC write, and read, data through the RDMA stack.

- **Pipeline a waiting routine** before the GPU application logic to trigger the processing.
- **Enqueue a posting routine** to the GPU application logic to send the data to the remote client.
- At regular intervals (*i.e.,* before filling up the receiving completion events queue), a call to the event consumption routine would advance the doorbell register for the completion events queue.
- Optionally, transform the CPU infinite loop to a GPU-side *self-launching Graph*.

In practical terms, this boils down to adding some additional function calls during the initialization of the RDMA stack (using functions defined in our library), and replacing the *verb posting routine* function call with the custom ones, queuing them in the GPU runtime (*e.g.,* in the same stream of the application). We report the pseudocode for a simple CPU-driven GPU-based application in Listing 2, with data received & send on the main system memory through RDMA. Listing 3 presents its GPU-driven counterpart, where all RDMA operations and checks are performed by some CUDA kernels, avoiding the need of copies across the device boundaries. This is further improved by the use of CUDA Graphs, shown in Listing 4, with the program logic shrinking to a single call to *launch* the graph. Finally, Listing 5 shows how the application loop can be entirely offloaded to the GPU, without any active CPU processing during the graph execution.

```
// Initialize CUDA stack
cudaSetDevice(0);
// Initialize RDMA stack and store data-structures
rdma_metadata * rdma_data = init_rdma_stack(rdma_dev, remote_address);
// Allocate a Memory Area and register it (as Memory Region)
void * data = malloc(MAX_SIZE);
rdma_data->mr = ibv_reg_mr(rdma_data->pd, data, MAX_SIZE,
  IBV_ACCESS_REMOTE_WRITE|IBV_ACCESS_LOCAL_WRITE);

// Create a buffer on the GPU
void * gpudata; cudaMalloc(&data, MAX_SIZE);
size_t length;

while(!stop) {
    // Obtain a pointer to RDMA-filled memory
    // This is busy waiting (polling) for data to arrive
    receive_data(rdma_data, &data, &lenght);
    // Copy data to GPU
    cudaMemcpy(gpu_data, data, length, cudaMemcpyHostToDevice);
    // Invoke a kernel
    GPUFunction<<<32,32,0>>>(data, length);
    // Copy data from GPU
    cudaMemcpy(data, gpudata, length, cudaMemcpyDeviceToHost);
    // Wait for all operations to finish (blocking)
    cudaDeviceSynchronize();
    // Transmit the data via a CPU function
    rdma_write_with_imm(rdma_data, &data, &lenght);
}
```

Listing 2: Pseudocode for a simple GPU-accelerated application, transmitting data with RDMA and traversing a GPU kernel. RDMA operations target main system memory, and the CPU is in charge of posting, and receiving, routines.

```
// Initialize CUDA stack
cudaSetDevice(0);
// Pre-allocate a buffer for driver data
void * driver_data; cudaMalloc(&driver_data, DRIVER_DATA_SIZE);
// Replace the default allocators to use the above area
rdma_replace_allocators(driver_data, DRIVER_DATA_SIZE);
// Initialize RDMA stack and store data-structures
rdma_metadata * rdma_data;
rdma_data = init_rdma_stack(rdma_dev, remote_address);
// Allocate a buffer (note: only on the GPU)
void * gpu_data; cudaMalloc(&gpu_data, MAX_SIZE);
// Register the GPU buffer to be used by the NIC
register_rdma_gpu(rdma_data, gpu_data, MAX_SIZE);

size_t length;
while(!stop) {
    // Obtain a pointer to RDMA-filled memory
    wait_data<<<1,1,0>>>(rdma_data, &data, &lenght);
    // Invoke a kernel
    GPUFunction<<<32,32,0>>>(data, length);
    // Send the data through a kernel to be queued
    rdma_write_with_imm_cu<<<1,1,0>>>(rdma_data, data, length);
}
```

Listing 3: Pseudocode for a simple GPU-accelerated application, transmitting data with RDMA controlled by GPU. RDMA operations are controlled directly from the GPU (`rdma_write_with_imm_cu` kernel). Note the lack of any synchronization point (*e.g.*, `cudaDeviceSynchronize`).

```
// Initialize CUDA stack
cudaSetDevice(0);
// Pre-allocate a buffer for driver data
void * driver_data; cudaMalloc(&driver_data, DRIVER_DATA_SIZE);
// Replace the default allocators to use the above area
rdma_replace_allocators(driver_data, DRIVER_DATA_SIZE);
// Initialize RDMA stack and store data-structures
rdma_metadata * rdma_data;
rdma_data = init_rdma_stack(rdma_dev, remote_address);
// Allocate a buffer (note: only on the GPU)
void * gpu_data; cudaMalloc(&gpu_data, MAX_SIZE);
// Register the GPU buffer to be used by the NIC
register_rdma_gpu(rdma_data, gpu_data, MAX_SIZE);
size_t length;
cudaGraph_t graph;
cudaGraphExec_t instance;
cudaStream_t stream; cudaStreamCreate(&stream);

// Build the graph with all the operations
cudaStreamBeginCapture(stream, cudaStreamCaptureModeGlobal);
wait_data<<<1,1,0>>>(rdma_data, &data, &lenght);
GPUFunction<<<32,32,0>>>(data, length);
rdma_write_with_imm_cu<<<1,1,0>>>(rdma_data, data, length);
looper<<<1,1,0>>>();  // This realize the "LOOP" functionality
cudaStreamEndCapture(stream, &graph);


// Instantiate the graph
cudaGraphInstantiate(&instance, graph, 0);

// Run the graph in a specific CUDA stream
while(!data->stop){
    // The following will execute the graph
    cudaGraphLaunch(instance, stream);
}

cudaDeviceSyncrhonize();
```

Listing 4: Pseudocode for a simple GPU-accelerated application, transmitting data with RDMA controlled by GPU, using CUDA Graphs. The *CPU-side* loop will execute the *graph* in sequence.

```
__global__ void looper(executor_data *data) {
auto g = cudaGetCurrentGraphExec();
if (g && !data->stop) {
   int ret = cudaGraphLaunch(g, cudaStreamGraphTailLaunch);
   }
}
// Initialize CUDA stack
cudaSetDevice(0);
// Pre-allocate a buffer for driver data
void * driver_data; cudaMalloc(&driver_data, DRIVER_DATA_SIZE);
// Replace the default allocators to use the above area
rdma_replace_allocators(driver_data, DRIVER_DATA_SIZE);
// Initialize RDMA stack and store data-structures
rdma_metadata * rdma_data;
rdma_data = init_rdma_stack(rdma_dev, remote_address);
// Allocate a buffer (note: only on the GPU)
void * gpu_data; cudaMalloc(&gpu_data, MAX_SIZE);
// Register the GPU buffer to be used by the NIC
register_rdma_gpu(rdma_data, gpu_data, MAX_SIZE);
size_t length;
cudaGraph_t graph;
cudaGraphExec_t instance;
cudaStream_t stream; cudaStreamCreate(&stream);

// Build the graph with all the operations
cudaStreamBeginCapture(stream, cudaStreamCaptureModeGlobal);
wait_data<<<1,1,0>>>(rdma_data, &data, &lenght);
GPUFunction<<<32,32,0>>>(data, length);
rdma_write_with_imm_cu<<<1,1,0>>>(rdma_data, data, length);
looper<<<1,1,0>>>();  // This realize the "LOOP" functionality
cudaStreamEndCapture(stream, &graph);

// Instantiate the graph
cudaGraphInstantiate(&instance, graph,
      cudaGraphInstantiateFlagDeviceLaunch);

// Run the graph in a specific CUDA stream
cudaGraphLaunch(instance, stream);

// The following will wait until the Graph loop would finish
cudaDeviceSyncrhonize();
```

Listing 5: Pseudocode for a simple GPU-accelerated application, transmitting data with RDMA controlled by GPU, using CUDA Graphs. The `looper` function is in charge of executing the application loop. Note the lack of any CPU active instruction after `cudaGraphLaunch`.

7.5.8 Towards CPU-free execution

Leveraging CUDA Graphs, introduced in Section 2.2.2.3, we can greatly reduce CPU intervention at runtime, relegating it to minimal pointer movements at the beginning and end of each inference batch, as shown in Listing 5. With *Tail Launch*, we eliminate the need for any subsequent workload posting, with GPU processing all data perpetually, potentially without any CPU interaction.

The usual approach to CUDA asynchronous applications relies on `cuda*Synchronize` calls, which stop the processing on CPU until all posted work is completed at stream, device, or system granularity. When CPU does not have any other task to execute (as in the bidirectional GPU communication case), the CPU will reach a point in the application where it will have to wait for the CUDA workload to finish, calling one of the above synchronization API. Under the hood, these calls are implemented via a `poll` system call, in a busy-waiting fashion, causing the usage of CPU to increase to 100% (*i.e.,* with a single core used 100% of the time).

We avoid busy waiting by implementing a relaxed synchronization mechanism, similar to a semaphore, where the CPU periodically checks the value of a specific variable, which would indicate when the application should stop (either from a user signal or because some event happened on the CUDA side would stop the execution, *e.g.,* maximum number of iterations). When this happens, the CPU will call the synchronization primitives and, only at this time, the CPU goes into the busy-waiting phase. On the GPU side, the same check would be performed before *tail-launching* each execution, blocking the loop to continue (and hence emptying the queue of standing work). We report the use of the standard CUDA synchronization primitives in Listing 6, while Listing 7 shows our *interruptible sleep* solution.

While this relaxed synchronization introduces some potential delay between the end signal and the actual end of the program, and prevents a real-time feedback loop from collecting metrics, it greatly reduces the CPU usage while increasing the GPU runtime efficiency and the power efficiency of the system.

Although it was not implemented in this work, it is also possible to use more advanced synchronization mechanisms between the CPU and GPU runtime, allowing periodic reporting of runtime metrics while also minimizing the time during which CPU polls for these results to be ready.

```
cudaGraphLaunch(instance, stream);
// The following would wait until the Graph stop running
cudaDeviceSyncrhonize();
```

Listing 6: Standard CUDA application structure: the CPU immediately *waits* for the GPU to complete workload.

```
cudaGraphLaunch(instance, stream);
// Until a "stop signal" is generated, sleep.
while (!stop){ sleep(1); };

// The following would wait until the Graph stop running
cudaDeviceSyncrhonize();
```

Listing 7: *Interruptible-sleep* approach: synchronization is invoked only when the stop event has been generated.

7.5.9 Event-based synchronization

While self-launching CUDA Graphs, and GPUDirect-RDMA provides the technologies to realize our prototype used in Section 7.6, we implemented also a CPU-driven prototype in order to understand the cost of the CPU-GPU interaction.

To reduce any possible *hard* synchronization point between CPU and GPU, and execute multiple tasks in parallel (*e.g.,* copies and inferences), we use *CUDA events*. A CUDA event is a time marker, which can be *recorded* by a CUDA application, and which would hold a time reference to its invocation. Through another API call in the CUDA runtime is then possible to *query* the status of an event, and verify if (i) it has happened or (ii) wait for it to happen, and (iii) measure the elapsed time from when it happened. CUDA Events are realized through specific object instances in the C++ CUDA API. To avoid overheads while creating and deallocating these objects we pre-allocate and recycle them.

We use CUDA events both to synchronize CPU and GPU, and to measure execution times in the *CPU-mediated* prototype.

7.5.10 Multi-stream architecture

While our GPU-driven prototype is implemented as a *single-stream* application, and we queue memory copies and execution in the same CUDA stream, for our CPU-mediated prototype (used as comparison) we want to avoid any head-of-

the-line blocking that may arise from long copying inter-device times, which we measure in Section 7.6.3.2. The most effective way to perform copies and executions in parallel, while controlling the level of parallelism (*e.g.,* preventing more than one inference to be run at a time), is based on *multiple CUDA stream* synchronized with *CUDA events*.

We use three different streams, each fulfilling a specific step of our application:

- **Input stream**: this stream will be responsible for the input copies from CPU to GPU.
- **Execution stream**: this stream will control the execution of our workload (*i.e.,* our inferences).
- **Output stream**: this stream will handle the output copies from GPU to CPU.

Leveraging the ordering guarantees of streams (see Section 2.2.2.2), we *record* an event after each operation, and wait for this event to occur before executing the next operation (which would happen in a different stream). We summarize our architecture in Figure 7.4.

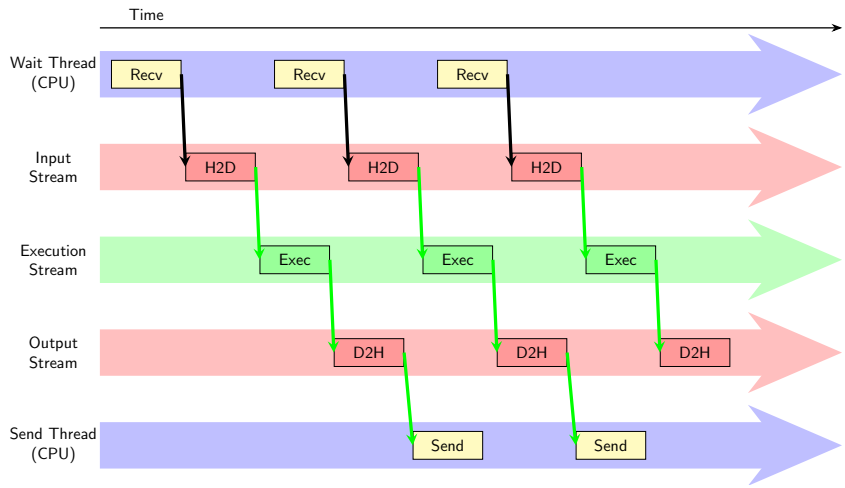


Figure 7.4: Multi-stream structure of our CPU-mediated inference serving prototype, used to measure the benefits of GPU-driven networking. The vertical black arrows represent the *posting* of a kernel, the vertical green arrows represent the *event notification*. The figure shows the situation when the GPU is underutilized, and thus streams would *wait* for the events to be recorded. Under higher load, the wait process would be immediate and the execution would continue immediately after the previous task in the stream has been completed.

7.5.11 Time accounting

While we otherwise collect time measurements by checking the *clock cycle counter* both for CUDA and CPU implementation, these are not synchronized, and it becomes complex to measure the time elapsed between tasks occurring on different devices (*e.g.,* a memory copy CPU to GPU and the start of the execution). To solve this challenge, we insert a first CUDA event in the *copy input stream*, and a second one in the *execution stream*, calculating the elapsed time at the end of the execution. Similarly, we insert a CUDA event before the *memory copy*, which we use to measure the copy duration. Figure 7.5 represents the main structure of this measurement mechanism.

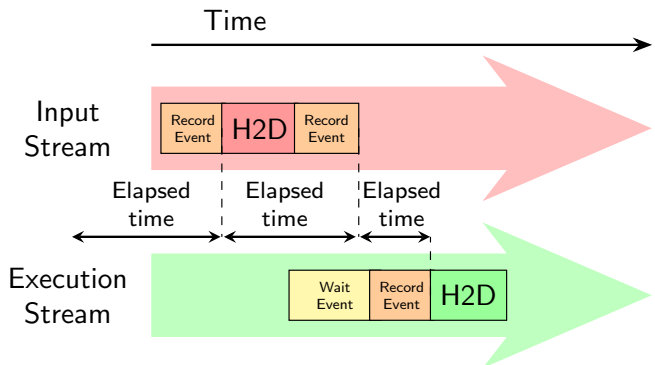


Figure 7.5: Event-based time measurement: the cross-device time is measured between the events. The first *Elapsed time* would be measured from a CPU-side event, not represented.

7.6 Performance evaluation

To understand the benefits, and the limits of our RDMA implementation on the CUDA architecture, we conducted a performance analysis. In this section, we report the relevant results, analyzing the performance of our prototype, and framing the limits in the time-scale of a realistic application.

7.6.1 The performance of GPU-controlled RDMA

As discussed in Section 2.2, GPUs are a good target for parallelizable tasks, such as ML inference and other large matrix-based mathematical operations. On the other hand, GPUs' clock rate is usually lower (*e.g.,* the nominal clock for an NVIDIA

A100 is 1 410 MHz), with CPUs easily having 2× or 3× faster clocks,* although CPUs have much fewer parallel pipelines (*e.g.,* 32 cores for an Intel Xeon 6346 and 6912 *CUDA cores* on a NVIDIA A100).

Due to these architectural characteristics, when sequential workloads with single threads are involved GPUs do not represent ideal targets, potentially leaving most of their resources unused and unable to use all the cores capable of parallel processing. This is particularly relevant for our RDMA posting routines, which are, in most cases, sequential single-threaded code, and thus, by design, cannot fully utilize the GPU's resources. To understand the performance that our näive implementation of the RDMA stack would achieve on a GPU, we implemented a simple benchmark, where a single kernel on the GPU, launched sequentially, posts RDMA WRITE verbs to a remote endpoint. Figure 7.6 shows the structure of our prototype, for the *GPU configurations*.

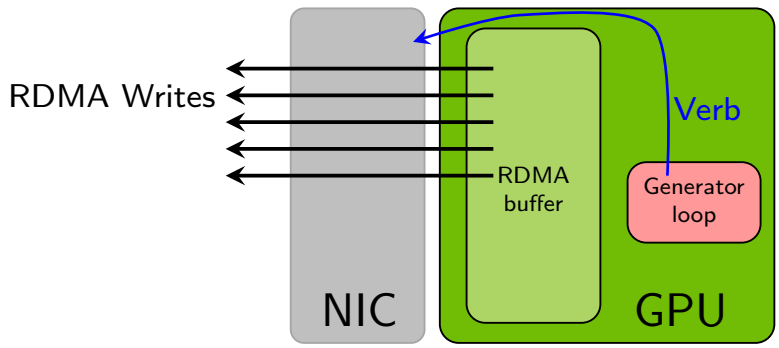


Figure 7.6: Structure of our benchmark application to generate traffic, *for the GPU driven* case. The *blue arrow* represent the verb data-structures being sent directly from the GPU to the NIC, without any CPU traversals. Subsequently, *payload data* is directly retrieved from the GPU memory through NIC's DMA engines.

Figure 7.7 shows the throughput that can be achieved by such an application when run on both NVIDIA A100 and NVIDIA L40 GPUs, installed on the `nslrack34` system described in Section 5.1. The system uses a NVIDIA Mellanox ConnectX-6 200-Gbps NIC on the same PCIe switch as the GPUs, and we measure the pure network throughput by collecting packet counters on a 100-Gbps switch (thus, limiting the maximum transmission rate achievable to 100 Gbps). The application is configured to send RDMA Write operations to a RoCE Unreliable Connection endpoint, excluding any potential overhead that may arise due to network congestion, acknowledgment handling, or other notification-based mechanisms that would involve the CPU.

*An Intel Xeon 6346 has a base frequency of 3.10 GHz, with a turbo boost of 3.6 GHz.

The *Crafted verbs (CPU)* curve represents the performance obtained when running the *same* routines implemented for *the CUDA side* in pure C++ on the CPU. These differ from the GPU implementation only for some device-specific APIs (*e.g.,* memory copies), and allow to understand whether if our routines, when run on the CPU, perform much different from the original `rdma-core` code.

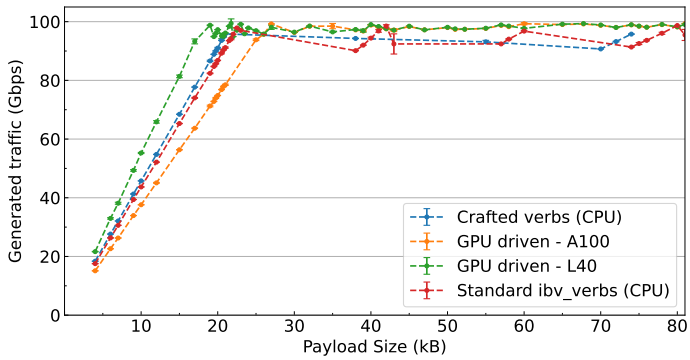


Figure 7.7: Maximum bandwidth reached by generating RDMA Write verbs on CPU (both with *Standard ibv_verbs* and *Crafted verbs* implementations) and two GPUs (*A100* and *L40*), when changing the verb's payload size. **All implementation can saturate the link when packets are larger than 20 kB.**

We can see that, in all cases, the throughput grows linearly with the payload size for both CPUs and GPUs. We note that the instabilities represented by some methods on the rightmost part of the plot are caused by our artificial bandwidth limiter, which prevents the exhaustion of NIC resources by limiting the number of verbs posted at each given time. This limiter is applied at the application level *e.g.*, without considering the effective wire throughput, but only accounting for the payloads of the verbs posted.

The link capacity is saturated for all methods only when packets are larger than ≈ 20 kbps. We hypothesize that this is a limitation of the single-QP design used in our tests, which potentially does not use all available resources on the NIC. We believe that this is not a huge limitation for a realistic application, and we demonstrate this by performing the experiments presented in this section. Some specific dynamics that can be observed in the results requires further study that we leave for future works as it falls outside the scope of this thesis.

The different slopes of GPU and CPU curves are related to the different clock speeds of the devices, which is higher on the CPU, and with the L40 running faster than the A100 (*i.e.,* 2.49 GHz versus 1.41 GHz).

Figure 7.8 shows the actual time required to run the verb-posting code on both the CPU and the two GPUs. The GPUs' processing time is much longer (*i.e.,* worse)

than the CPU cases, both due to the un-optimized code of these routines and due to the lower clock rate of the GPUs' processors. Furthermore, due to the hybrid nature of the driver memory allocation (described in Section 7.5.2) and our need to access memory from both the CPU and the CPU, we must take into account the paging mechanism supporting the *managed memory allocations* as this could potentially increase the latency of the posting operation.

It is interesting to note the *step trend* of the latency for the CPU cases, which we hypothesize being related to PCIe bus dynamics, access patterns, and caching mechanisms. We did not investigate this behavior in detail, and we leave this to future works.

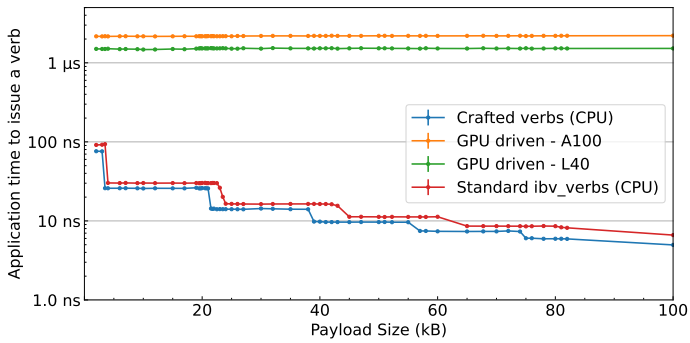


Figure 7.8: Average time needed to post a Write verb (*e.g.,* time required for `ibv_post_send`). **Posting time is not directly dependent by the size of the payload.**

Our *crafted verbs* implementation performs slightly better than the original `rdma-core` APIs, mainly due to the absence of many multi-threading mechanisms and the removal of many optional paths in the code, either because these mechanisms or paths are not needed or supported in our implementation. The loss of the multi-threading and other functionality is collateral due to the porting process since many functions would need to be reimplemented in the CUDA runtime, and their implementation would inevitably introduce (unwanted) conditional branching. Although this crafting greatly reduces the flexibility of the routines (*e.g.,* these may potentially not work with other models of NICs, different architectures, or different drivers), we believe these functions can be more easily ported to other architectures and extended to support more use cases.

In this experiment, the driver data-structures were allocated through the `cudaMallocManaged` API and, given the low control that we can have on the physical location of the underlying pages, we expect a slightly higher latency when operating on these regions compared to pure CUDA allocated memory. However, the CUDA runtime, in the current release, does not support registering CUDA

physically-allocated memory areas to be accessed by the host system, which would require implementing the complete driver initialization routines on the GPU side, as it is done in the NVIDIA OpenSHMEM Library (NVSHMEM™) library.

Limits of current implementation While parallel processing is the natural optimization for these limitations, with most operations performed on the RDMA data structures operating at fixed (*i.e.,* predictable) offsets, we leave this for future work. This is not a fundamental limitation, and parallel operations would only increase the performance of our implementation. Increasing the number of *Queue Pairs* in the system could reduce the need to synchronize operations performed by multiple threads, while also maximizing the use of all *Processing Units* available in the NIC: our evaluation, instead, uses a *single Queue Pair*, a *single GPU thread* and, potentially, a single *Processing Unit*.

Parallelization strategies Beside the classic multi-thread strategies that could be implemented in our routines, *Virtual Functions* (as in PCIe/SR-IOV terminology) can help improve the level of parallelism without introducing any locking. Most NICs allow users to spawn multiple *virtual devices* on top of the physical one, usually appearing as different PCIe devices. When using a 1-to-1 mapping between these instances and applications, it would be possible to run completely lock-free while allowing more to execute in parallel: each would have a separated RDMA stack, and different *doorbell registers*, which could be manipulated simultaneously by multiple applications. We leave this implementation for future work.

The key advantage of doing GPU-driven RDMA operations, overall, is to reduce, if it cannot be completely avoided, all synchronization between CPU and GPU, greatly improving the *overall resources usage and the goodput of the system*, which spends virtually no time waiting for tasks to finish, but instead can spend the entire runtime performing some actual processing.

7.6.2 Number of CPU interrupts

To further understand the benefits of performing RDMA operations from a GPU, we analyzed the number of CPU interrupts per second that are received from the CPU when RDMA posting operations are performed on the CPU and on the GPU. We collect these counters by polling the Linux Kernel, and report them for some selected packet sizes* in Figure 7.9.

*We have verified the metrics are not different for larger packet sizes, but we don't report them for space constraints.

The continuous interaction between CPU and GPU results in a consistent 3× increase in the interrupt rate with respect to a GPU-driven approach. As handling interrupts is very disruptive of the CPU's instruction pipeline, reducing the interrupt rate improves CPU's performance for other tasks. For comparison, we report the number of interrupts collected on the same system when performing no operations, and we report this in the figure as *Idle system*.

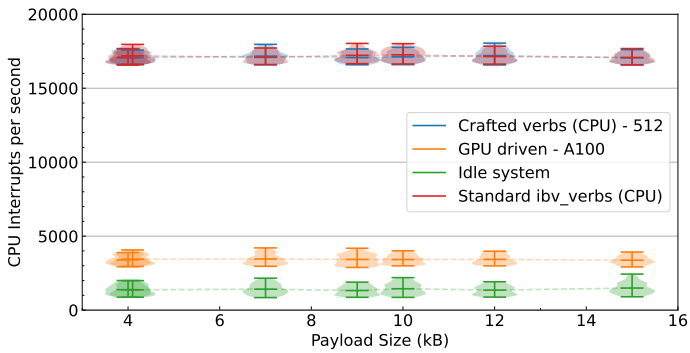


Figure 7.9: Average number of CPU interrupts per second received when posting RDMA verbs from the CPU and GPU. **GPU-driven approaches reduce significantly the number of interrupts caused to CPU when exclusively generating traffic**. The clouds around each point represent the values distributions.

To verify that the number of interrupts received is not directly caused by the *CQE-polling* mechanisms and the intervals we use, we measured the number of interrupts received when changing the interval at which *signaled* operations are performed, and thus *CQE* are created. We report the results of this experiment in Figure 7.10, which shows a stable number of interrupts across all batch sizes, demonstrating how the number of interrupts is not strictly related by the CQE consumption routines. We hypothesize that these are mainly caused by the GPU runtime and the inherent interaction of the GPU hardware with the driver, although we have not performed any in-depth analysis of these dynamics. We leave this analysis, together with a benchmarking of interrupts throughput and latencies, for future work.

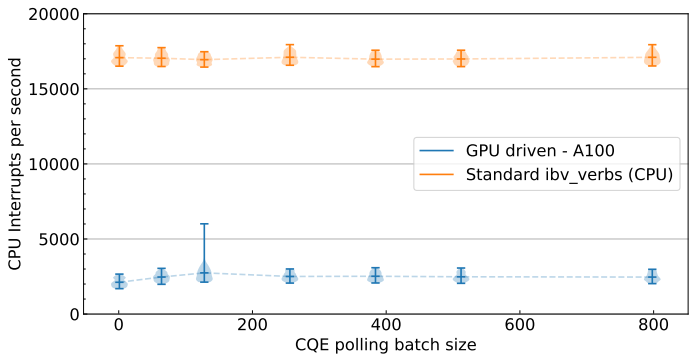


Figure 7.10: Average number of CPU interrupts per second received when posting RDMA verbs from the CPU and GPU, when changing the frequency for CQE generation and retrieval. **The number of interrupts is not significantly affected by this factor.**

7.6.3 An inference serving prototype

To put the performance of our verb posting routine into perspective, and the burden this implementation would have on a realistic application, we built an inference-serving prototype *worker* based on TVM, which receives inputs over RDMA, processes them through a GPU workload, and returns outputs over RDMA.

We use this prototype to measure, and put into perspective, some relevant performance metrics of our GPU-driven RDMA building block, which we report in the following sections of this chapter. The structure of this prototype, and how data traverse it, is represented in Figure 7.11.

Model runtime We patched TVM's CUDA runtime to expose some internal variables related to the streams used to run the model and record the operations in a *CUDA Graph*. We then execute this graph in a loop tailing it with a *wait input*, a *send outputs*, and a *cqe consumption* routines. These realize the functionalities described in Section 7.5.7, and constitute the core of our contribution. This minimizes the overhead that may arise from any synchronization, while guaranteeing in-order processing and allowing to measure accurately the execution time of each sub-component.

Zero-copy I/O We introduced a *wrapper* data-structure in TVM to handle inputs and outputs, allowing an application to use the native interface of TVM with custom-allocated memory areas. We create a *input* and *output* data structures and set them as *zero-copy* inputs and outputs, respectively, of the model, avoiding any inherent copy by the TVM runtime. When the inference graph is run, the inputs

are always read from the input area set as above, while outputs are written to the output region in the same manner.

Memory copy limitations We note that, due to limitations in the API exposed by TVM and CUDA Graphs, it is not possible to change the input and output pointers of the model at runtime (*e.g.,* to process different areas of memory at different times). To tightly control the operations in the graph, and avoid any limitation that may arise from the exhaustion of *memory engines* when running multiple operations in parallel, we implement the copy operations with an *ad-hoc* kernel, which is spawned in parallel across the GPU, with each instance copying a small area of memory. This implementation resembles a model with two additional *identity* layers at the extremities.

While this approach is not ideal (*e.g.,* it does not fully use GPU's capabilities), we prove that it does not introduce high overheads for realistic applications (see Section 7.6.3.1), and we leave as future work the implementation of more efficient copy mechanisms.

Framework agnosticism While our approach has been implemented on top of TVM's runtime, we note this is not a strict requirement, but rather a *blind* replacement of the *model API calls* can extend the support to other frameworks or applications. Our approach is tightly bound to the guarantees exposed by CUDA Graphs and, more precisely, to the ability of executing operations sequentially in a given order: the TVM model APIs are recorded as plain CUDA kernels without any further interaction with the framework after setup. We note that this agnosticism of our implementation is a big difference with State-of-The-Art inferencing frameworks (*e.g.,* Clockwork [19]), which is instead bound in high detail to the framework's semantics.*

Re-usability We believe that the design of our application (*e.g.,* the graph handling and loop mechanisms) could be reused in realistic inference serving applications, with minimal changes in the code, similar to the one presented as pseudocode in Section 7.5.7. We plan to release our code as open-source software.

*In Clockwork, each single layer is cherry-picked from the TVM's shared-object library and executed individually, breaking the isolation & modularity principles, and potentially impeding an easy upgrade path to newer revisions of the framework.

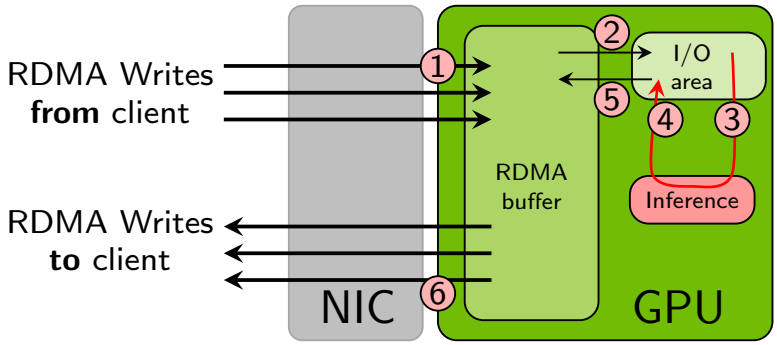


Figure 7.11: Structure of our GPU-inference serving prototype, with the paths followed by the I/O data. *3* and *4* represents the memory accesses by the TVM model, *2* and *5* are explicit memory copies to a local working area, due to NVIDIA Graph limitations. **No memory copy traverses the GPU boundaries.**

7.6.3.1 How costly is the networking?

As reported in Section 7.6.1, the time spent by the GPU performing the RDMA posting tasks may be non-negligible, especially when small and faster models are involved. To quantify how much this impacts our inference serving scenario, we measured the execution time of every single component in the execution pipeline, collecting the clock counter directly on the CUDA side after each execution.

Figure 7.12 shows how in a realistic application the time required to process the outgoing requests (*i.e.,* our *posting routine*) is practically negligible, especially when bigger models are run. The cost of *waiting for the inputs* is small, despite the näive implementation used for this: we look in a round-robin fashion across all the input areas to search for requests ready to be processed. To simulate a *good* system scheduler, we ensure that enough requests are always ready by implementing a similar round-robin logic on the client side. We leave as a future work the implementation of a parallel routine that would use the native multi-threading of the GPU to discover input readiness.

To better visualize the impact of all components, we present a detailed view for the *ResNet50* model in Figure 7.13, comparing it to an ideal case where no memory copy is performed (*e.g.,* a dummy empty kernel is placed instead). We note that the cost of copying inputs and outputs, despite the implementation used, does not affect global processing time by more than 1%.

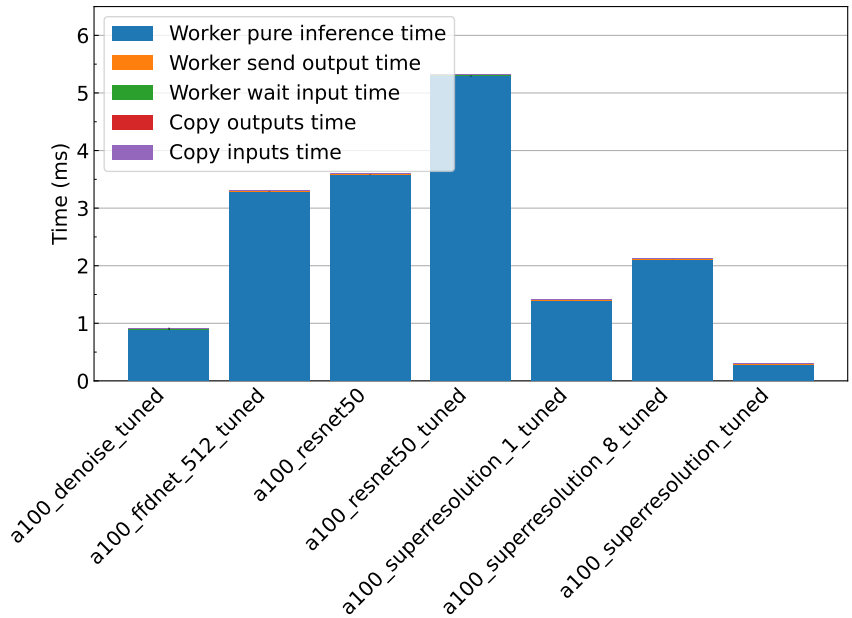


Figure 7.12: Runtime contributions to the total inference latency when all processing is performed on GPU via an Unreliable RDMA connection. **The copying time is negligible compared to the inference time.**

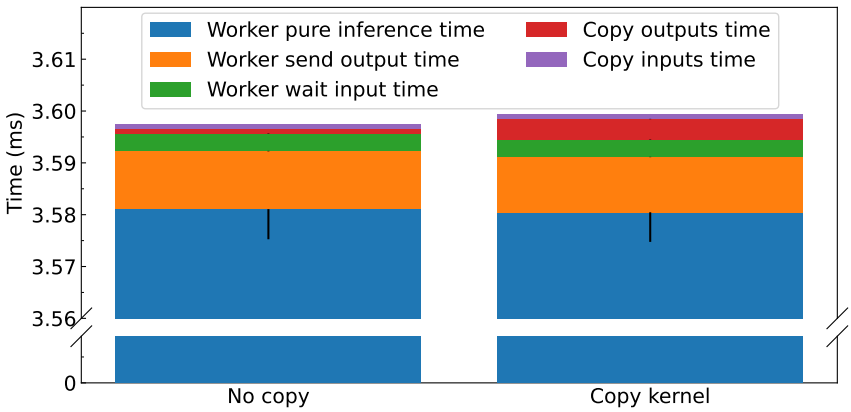


Figure 7.13: Runtime contributions to the total inference latency when doing all processing on GPU via an Unreliable RDMA connection, with and without I/O memory copy kernels. *ResNet50* on A100 GPU. **Memory copy routines affect the total latency by less than 1%.**

7.6.3.2 The burden of CPU-GPU synchronization

To quantify the benefit introduced by avoiding copies between CPU and GPU, we report in Figure 7.15 the latency-breakdown when running two different models with & without I/O data traversing the CPU:

CPU mediated represents the case where RDMA operations have the main system memory as target, and copies are performed to, and from, the GPU memory via `cudaMemcpy` primitives. We ensure these happen in parallel to the execution by using three parallel streams

GPU driven represents our target application, also used in all above experiments, where data is received on the GPU through RDMA, processed, and sent back through RDMA directly from the GPU. No active CPU intervention happens to control execution.

We represent in Figure 7.14 the data paths inside our CPU-mediated prototype, as opposed to the GPU-driven architecture shown in Figure 7.11.

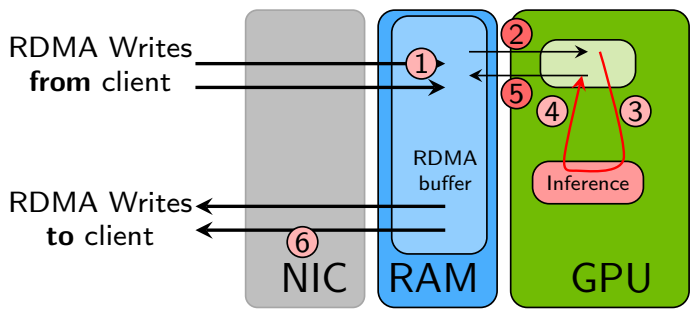
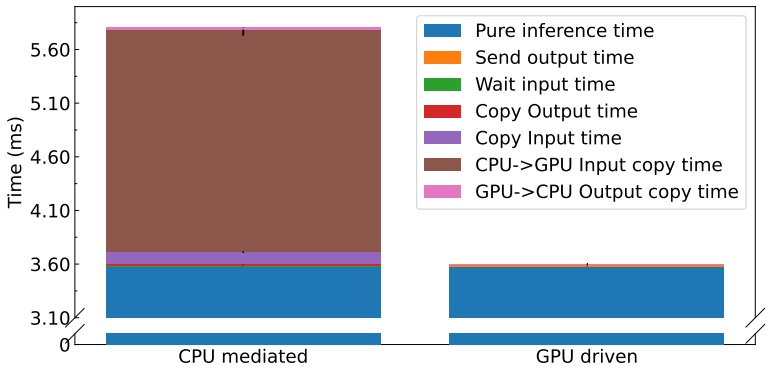
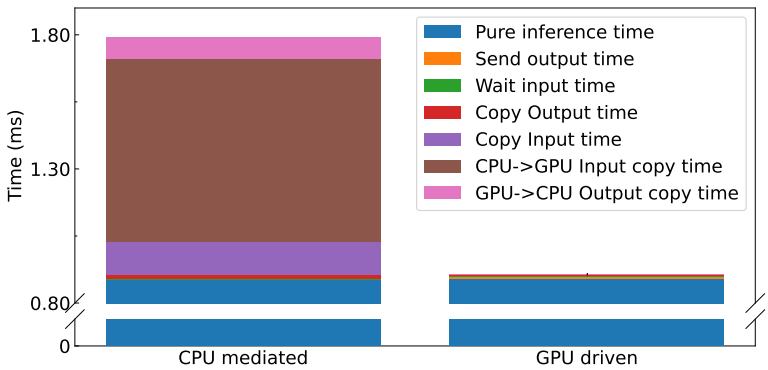


Figure 7.14: Structure of our CPU-mediated inference serving prototype, used for comparison, with the paths followed by the I/O data. *3* and *4* represents the memory accesses by the TVM model, *2* and *5* are explicit memory copies between CPU and GPU.

The cost of transferring data between the two components to the system when copies are made, with a higher cost for the *input data*, due to the model used (*i.e.,* it has large input and smaller output sizes). As already stated, this phenomenon is exacerbated for faster models, as shown in Figure 7.15b for the *denoise* model, with the copying time having similar time costs as the pure inference time.



(a) *ResNet50*



(b) *denoise*

Figure 7.15: Time contributions to the total inference latency, when executing two different models with either GPU or CPU initiated networking and A100 GPU. Unreliable RDMA connection.

7.6.3.3 Does the transport type matter?

In the experiments above, to exclude any network behavior in the measurements, we used RDMA *Unreliable Connection*, which provides similar guarantees to a UDP protocol, but is generally not the best solution as it lacks any of the RDMA loss-less guarantees usually exploited by applications.

To verify the impact of the type of connection used in the *execution times*, we measured the execution times for *Reliable* and *Unreliable* transports, and reported them in Figure 7.16, together with two different CPU/GPU synchronization mechanisms, which we have described in Section 7.5.8.

As we could expect, the offloading performed by the RDMA NIC hardware exhibit a *constant time* for the *posting routines*, without any major notable differ-

ence between the different cases. The *transport time* over the network may instead be different (*e.g.,* due to congestion or error-prone links), but if queues & buffers are always *not-empty* these delays won't be visible at application level (*e.g.,* the application will never wait to receive inputs, with some data always ready to be processed). The self-launching routine, explained in Section 7.5.8, does not affect the processing time on the GPU.

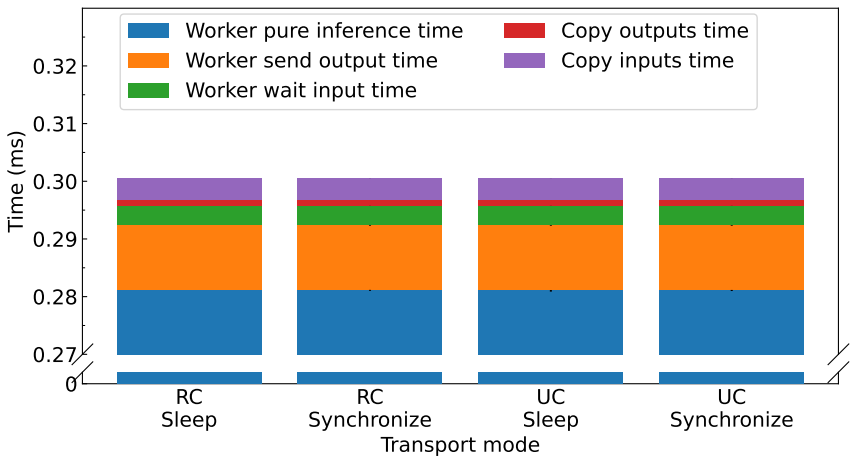


Figure 7.16: Time contributions to the total inference latency, when executing *superresolution_tuned* with GPU initiated networking. *Synchronize* represents the standard `cudaDeviceSynchronize` method, *Sleep* represents our *interruptible sleep* approach. *UC* and *RC* represent the two different RDMA modes. **No combination show significant difference in the time contributions.**

7.6.3.4 Zero-CPU inferencing

To understand the effectiveness of our *CPU-free* mechanism described in Section 7.5.8, we collected CPU usage counters and analyzed them over time. We report the time trace in Figure 7.17, showing that when an *interruptible sleep* approach is followed the CPU usage is *practically* null.

The experiment has been run for 90 s, and we collect CPU counters at 2 s intervals. The application was forced to run on a specific set of cores through `taskset`; these cores have been isolated through the `isolcpus` kernel configuration option.

We can distinguish three sections in the figure, represented by the three different colors:

Initialization colored in yellow, it is the initial phase where the CPU performs all initialization, model loading, and memory registration. An initial

warm-up of the model is also performed to fill caches and reduce start-up overheads.

Inference serving colored in green, represents the actual application serving phase. Here, we can note the *interruptible sleep* requires almost no CPU cycle; hence, the CPU is completely free to execute other tasks.

Cleanup colored in light red, after 60 s of runtime, the application receives a *kill* signal from the operating system. This stops CPU's sleep, signals the *graph loop launcher* to finish, and invokes `cudaDeviceSynchronize` before terminating. In this phase, the CPU returns to being in control of the system and must wait, polling the GPU, for all GPU jobs to end.

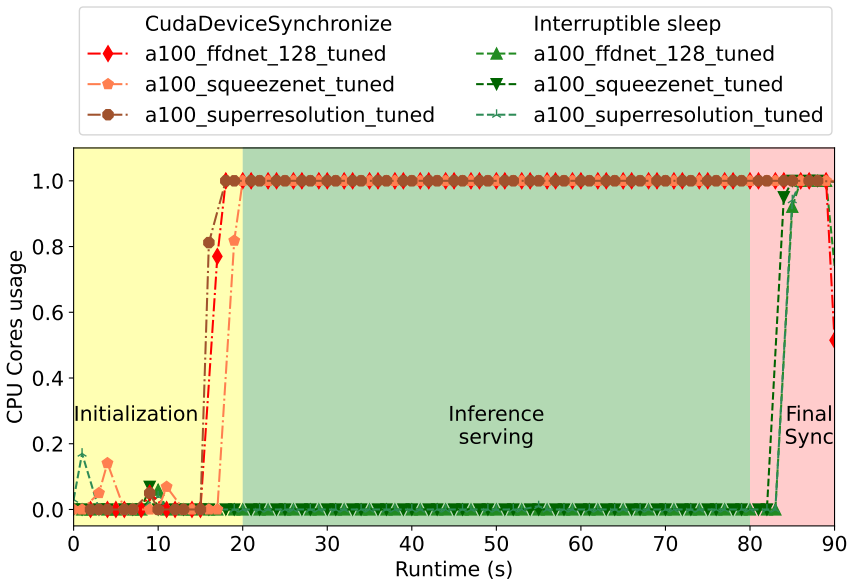


Figure 7.17: Fractional CPU usage, over time, for two synchronization techniques and multiple workloads. **Avoiding `cudaDeviceSynchronize` remove any CPU usage when doing GPU-driven RDMA communication**.

7.6.3.5 Buffer allocation method

As we discussed in Section 2.2.2.4, different methods could be used to allocate memory for CUDA applications, in addition to the CPU-only `memcpy` and the GPU-only `cuMalloc`. While both the above methods provide the best

performances for local accesses (*e.g.,* CPU program segments accessing CPU memory and CUDA sections accessing GPU memory), explicit calls to memory copy APIs are needed when intra-device processing is involved.

The performance of *Unified Memory* approaches, instead, is heavily dependent on access patterns, since page movement between GPU and CPU can easily become an unsustainable cost.

To understand how much this cost would affect our inference serving prototype, we have measured the execution times for three different memory allocation techniques:

cudaMalloc: this represents the canonical way to allocate memory on NVIDIA GPUs, physically allocating the memory areas on the GPU memory.

cudaMallocManaged: this exploits the capability of CUDA runtime to transparently map CPU and GPU memory to the same address space, migrating the memory pages upon access.

malloc: this allocates the memory with standard C++ API, and later registers it to the CUDA runtime to be accessed by the GPU (through *memory pinning*).

In Figure 7.18 we report the latency breakdown of our application, when serving the *denoise* model with the different allocation schemas. As we could expect, *malloc* allocation is the slowest approach, due to the page moving that happens during each memory access, while the other approaches benefit from GPU's L1 caching, and the migration of the backing memory space to the GPU's physical memory in the case of `cudaMallocManaged`.

Although further investigation is needed to fully understand the dynamics introduced by the different approaches, we believe that the best strategy is the simplest, which for this application is `cudaMalloc`, allocating everything directly on the GPU memory. This is particularly effective for *GPU-only* accesses, as in this application scenario.

We performed a similar experiment changing the allocation of the *RDMA stack buffers* (*e.g.,* the area of memory holding all RDMA metadata and status data-structures), and we did not notice any major performance differences between the different cases, mainly due to the highly asymmetric access patterns of the memory areas (*e.g.,* either GPU-only or CPU-only) used in our prototype.

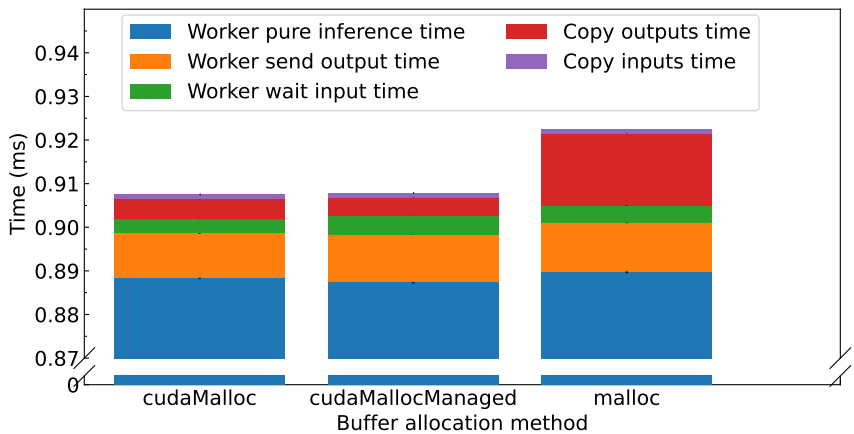


Figure 7.18: Time contributions to the total inference latency, when executing *denoise* with different memory allocation strategies.

Chapter 8

A System for Inference Serving

This chapter presents a performance-analysis of a State-of-The-Art inference serving system, Clockwork, and discusses some issues in this architecture. We then propose a re-design of this system involving RDMA as transport layer, showing the benefits that a direct *client-to-GPU* path could bring, as allowed by our building block described in Chapter 7.

In Section 8.1 a review of the system is presented, with some of system's limits summarized in Section 8.1.2, after the insights gained in Section 8.1.1. We then argue about the benefits of *one-at-a-time* paradigm of Clockwork in Section 8.2, comparing it to a parallel execution of multiple inferences.

Finally, in Section 8.3, we present our improved architecture of a low-latency, scalable, inference serving system.

8.1 The Clockwork model

A seminal work on inference serving systems is Clockwork [19]. The system proposed by Clockwork closely models the runtime of each inference by controlling every single low-level aspect of the inference runtime. This is made possible by the assumption that the execution of inference workloads can be predicted with a low margin of error, mainly due to the static structure of the code executed on the GPU at every step of the inference.

Clockwork proposed a system consisting of a central controller and a series of GPU-equipped workers, which are as a whole serving a large number of clients. Clients connect to the controller and submit inference requests to it. Subsequently,

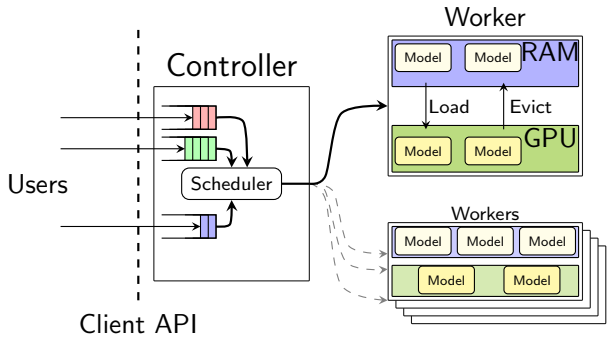


Figure 8.1: Main components of the Clockwork architecture. Adapted from the original [19].

the controller instructs the workers to load the model (*e.g.,* load the kernel code and the model weights into the GPU's memory), schedules the execution on each worker, and eventually evicts models from workers' GPUs. By controlling how and when a specific model code or weights are loaded on a GPU, the controller can predict the inference completion time and thus create an optimal schedule for utilizing the system's resources. The main architecture of Clockwork is shown in Figure 8.1.

At its core, the AI models are compiled with a *custom* TVM [55] version, with the single model's layers invoked *sequentially*. This allows the worker to enforce tight control of the amount of outstanding work, the number of memory copies, and the number of concurrent operations running on the GPU.

This architecture represents a clear step forward from standalone workers that serve inferences in a *best-effort* manner by promising to respect tight SLOs and to maintain high system efficiency. However, this centralized architecture places a high burden on the controller, which is in charge of both handling all scheduling & orchestration tasks and distributing & relaying all input and output inference data. These data, by architectural design, always travel alongside the requests, encapsulated in a custom *ProtoBuf*-based format [281], and traverse the CPU protocol stack of both the controller and the worker.

8.1.1 Assessing the Clockwork model

As already stated by Clockwork's authors in the paper, the centrality of the controller represents a bottleneck in the system, and it is hard to reach good performance when all request, and response, data traverse it. We note that in the paper almost all evaluation is performed by sending 0-length inputs and outputs, which are in turn replaced at the worker side with dummy data. This workaround

allows measuring the performance of the scheduler, and the runtime in general, without being limited by the performance of the data transfers.

To quantify the cost of these protocol stack traversals, we deployed a single-worker instance of Clockwork, with a client submitting, through the controller, a continuous flow of requests. Details of hardware used for this testing are given in Table 8.1. Requests are issued at a rate the worker could easily process without overwhelming it, avoiding queuing. We then deployed a simple RDMA transport* on top of the original Clockwork design, with the controller issuing execution *tickets* to the clients, which in turn write directly to the worker's memory via 1-sided RDMA verbs. The RDMA write operations go directly to the worker's GPU memory, exploiting GPU-direct RDMA support, thus, reducing the number of copies across the PCIe bus and through the CPU. **We don't use our GPU-side RDMA implementation in this evaluation**.

The controller manages the memory addresses where the clients should write the data. The controller relays this information to the worker and the client embeds an address in the ticket, along with the *rkey* value that is needed by the client to access the remote memory area (*e.g.,* the controller receives information about the memory regions from the workers, allocates I/O slots in these, and maintains a list of the assigned slots).

In our initial prototype, we implement this with a single, large, memory area (*e.g.,* with a single *rkey*), with the limitation of not being able to isolate the READs, and WRITEs, of different clients between each other. A more advanced implementation would require an active registration of each client's remote memory region, involving further communication between the controller and workers every time a new client connects, and thus higher start-up latency & higher load on the controller (*i.e.,* there is a need to exchange these data between the three parties at every new connection).

Table 8.1: Testbed for assessing Clockwork performance. Each server detail is reported in Section 5.1.

Component	Description
client	nslrack28
controller	nslrack30
worker	nslrack29
Interconnection	direct 100-Gbps links between each component

We report the latency experienced by requests in Figure 8.2, showing the contribution of the main components involved in the inference serving.

In the base Clockwork implementation (which uses the CPU's protocol stack,

*More formally, a RoCE implementation.

traverses the controller CPU's stack, and requires copies between the CPU and GPU), even with an unloaded system, the time that a request spends at the controller represents a constant 16% component in the final request latency, while the network-related tasks and the physical network transfer times can contribute up to 40% of the total inference time. In contrast with our näive RDMA implementation, we can see the network transfer time can be almost eliminated, thanks to the higher efficiency of the RDMA stack and the use of GPU-Direct RDMA; thus, skipping the memory copy at the worker side. This is also noticeable in the cost of transmitting, the input data: in the RDMA cases, the presence of data does not affect the latency as much as in the baseline case, as it can be noted by the smaller contribution for the *network transfer* in Figure 8.2. This is mainly thanks to the lower CPU time required in the RDMA case with respect to the baseline, with most data-transfers duties offloaded to the NIC, as we also demonstrate in our approach reported in Section 7.6.1.

It should be noted that the original Clockwork scheduling mechanism and the original processing workflow at the worker are used in this experiment, which may result in sub-optimal performances of our implementation. This is particularly restricting when considering the queuing system of Clockwork, which has been realized and tuned with interfaces that are 10x slower than our testbed, and with much loose inference times. We also note that we did not use our modified RDMA stack in these experiments, reducing to minimum the modifications in the original Clockwork code. Thus, RDMA operations are invoked by the CPU, but the memory region read (and write) from the verbs is allocated on the GPU by means of GPUDirect support and peer-to-peer transactions, as explained in Section 7.4.1.

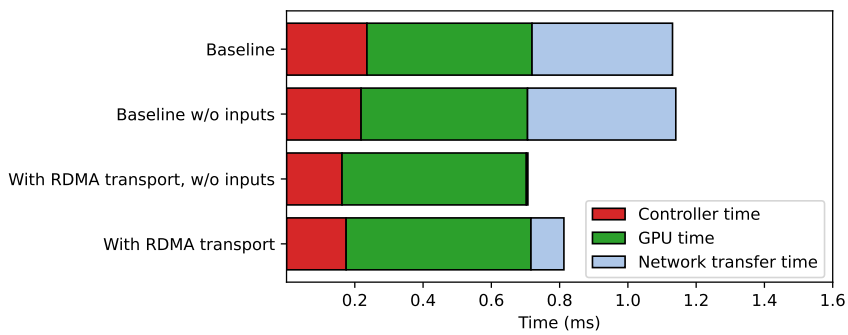


Figure 8.2: Breakdown of client latency for Clockwork, showing how our RDMA-improved prototype could eliminate any network transfer overhead.

In Figure 8.3, we show the measured latency across varied levels of requested load to the system, showing that our modifications are not detrimental to Clock-

work's original design and that the performance is relatively stable across a wide range of system loads. For reference, the average inference execution time is a constant 500 µs for all cases. The cost of transferring the inputs, for all RDMA-based cases, almost disappears thanks to the benefits of offloading the task from the CPU to dedicated hardware components (*i.e.,* NIC's PUs).

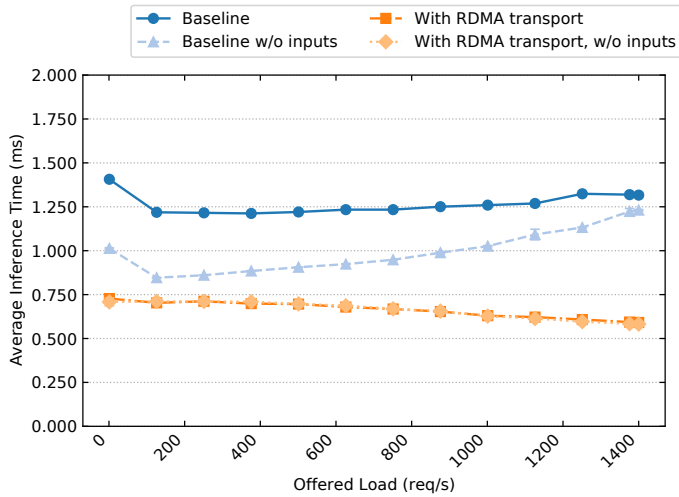


Figure 8.3: Latency experienced by the clients, showing a 2× performance gain when using RDMA.

8.1.1.1 Analysis limitations

Although these results are not directly comparable to the metrics shown in the Clockwork paper due to different underlying system characteristics, we have tuned this deployment of Clockwork to work in the best way possible on our testbed, which is equipped with newer hardware and faster networking. This configuration should reduce any possible bottleneck that may appear from the network layer or from the CPU architecture, given that these were much more powerful than the ones used in Clockwork's testbed.

To mimic the Clockwork testbed and also to reflect the metrics that we would see with a more modern GPU, we have performed this experiment with a lightweight model, *ShuffleNet* [282], which we have compiled with TVM similarly to the original Clockwork models.

It should also be noted that this evaluation has been done working around the limited *BAR0* size available for registering the GPU memory, which represents the maximum amount of memory that could be registered with the RDMA system to

allow remote hosts to perform operations directly to the memory. On the NVIDIA T4 cards used in these experiments, this space is 256 MB, but only 231 MB can be registered. The machines used in this experiment were not equipped with a PCIe switch, but we can assume that the performance gaps would only increase with this architecture, as the RDMA operations would be processed even faster (as introduced in Section 2.3.1). Following the current trends in GPU evolution, we expect these overheads to become even more relevant with newer GPUs.

8.1.2 Limitations of the Clockwork model

Controller centrality We can identify the centrality of Clockwork's controller as the main limitation to its performance and scaling, together with the inability to adapt to requests for large bandwidths. This is confirmed by the scaling experiments in the original paper, where the network usage is limited by sending zero-length input data with the requests. The solution proposed by Clockwork's authors is a front tier of *Load Balancers*, distributing the incoming requests to a pool of scheduler nodes.

We believe that such a solution would be subjected to the same issues of Clockwork's scalability, and it would introduce even more bottlenecks, latency overheads, and further resources to be scaled accordingly. In contrast, we propose a *direct path* between clients and workers, as described in Section 8.3.

Data and control channels coupling Another limitation we identified in Clockwork is the coupling between the request data and the scheduling decisions. While this coupling simplifies the architecture and reduces the number of round-trips between the client and the other components of the system, it becomes a limitation when a very low inference time is required. As already analyzed in Section 2.6, modern GPUs can achieve inference times well below a millisecond, especially in multi-GPU systems. In the Clockwork architecture, low latency inference would be overwhelmed by high controller processing times, queuing, and network overhead.

Avoiding on-critical-path scheduling decisions Similarly to the previous limitation, the on-path scheduling performed by Clockwork limits the complexity that can be implemented in the scheduler: a more complex scheduler would typically require more time to take decisions, and in the Clockwork's architecture this would introduce higher latency for each single request. We believe that a decoupled approach, where the controller is allowed to pre-compute the scheduling decisions, would improve the efficiency of the system, especially in the presence of highly repetitive, periodic tasks. One of such scenario could be a real-time system that process videos: frames would typically arrive at a regular rate, and the amount

of workload could be predicted, at least for the near future. A scheduler aware of these behaviors could compute a more optimized scheduling, issuing multiple time-based *tickets* to clients. These would then write the data at the correct time and, assuming a *well-behaving* time-synced cluster, much lower waiting time would be experienced by requests.

Going beyond one-at-a-time execution model One of the central assumptions of Clockwork is that running inferences sequentially ensures tight control and high efficiency at runtime. We believe that with modern GPUs, and the continuously improving software support, a more loose approach can be used to obtain better performance out of GPUs when doing inferences, allowing multiple inferences to run simultaneously and without controlling each single individual component as in Clockwork. Due to the absence of guarantees for the time needed for each request to be processed on the GPU, further synchronization is needed to obtain good performance (*e.g.,* the CPU should not call `cudaDeviceSynchronize` immediately or after a given predicted time as in Clockwork). Instead, we believe that the GPU should run in a run-to-completion fashion, without any final synchronization to the CPU, requiring support to execute RDMA operations directly from the GPU side. We proposed a building block to realize this feature in Section 7.5.3, we analyze a prototype using it in Section 7.6, and we verify the parallel execution dynamics in Section 8.2.

Kernel-level TCP stack Finally, the general inefficiency of the TCP stack in a standard Linux deployment, coupled with the multiple copies that this stack requires across the CPU, increases the latency and system requirements. While user-level TCP stacks [124], DPDK-based solutions [96] and *ad-hoc* protocol stacks can be used to improve specific use cases [10], we believe that the only practical solution to this problem is to rely on RDMA and 1-side verbs, which are supported in virtually any modern datacenter NIC, as we demonstrated with our prototype in Section 7.6.

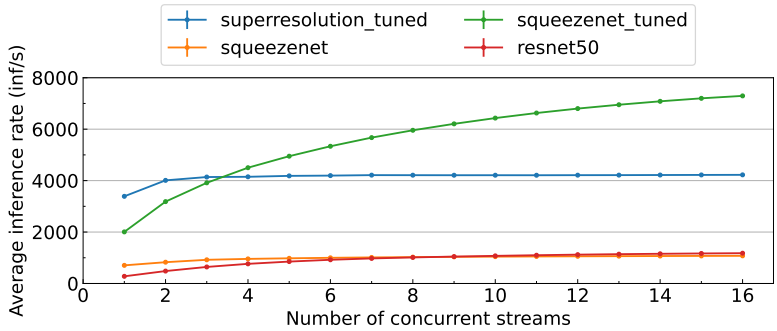
8.2 Single or concurrent execution

One of the central assumptions of Clockwork is that running inferences sequentially ensures tight control and high efficiency at runtime. To verify whether this assumption still holds on modern GPUs, we ran a synthetic benchmark on a NVIDIA A100 GPU, where we sequentially ran a model using a CUDA Graph to reduce the launch overhead, similar to what we did in our RDMA serving prototype (described in Section 7.6.3).

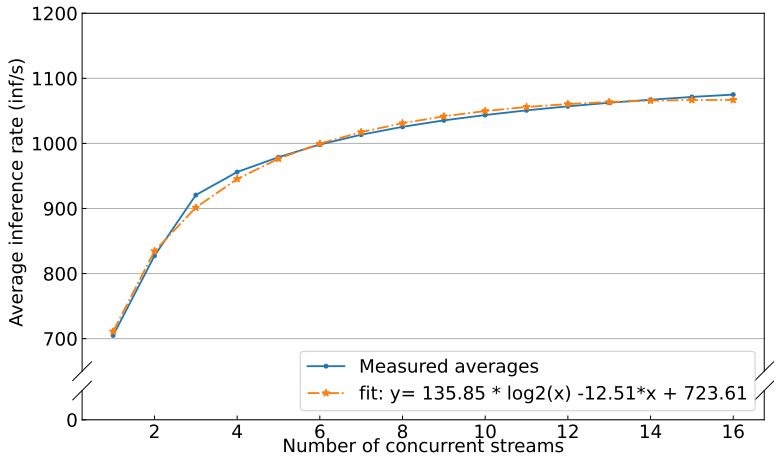
Figure 8.4a shows the average inference rate (calculated as the total number of inferences executed over the total runtime) for several models. As these plots show, increasing the concurrency (*e.g.,* running multiple streams in parallel) improves the inference throughput of the system, which follows a logarithmic trend.

Figure 8.4b shows the result of a *logarithmic curve* fitting for *SqueezeNet* model. The average inference rate is closely approximated by a $y = a \cdot log_2(b) + c \cdot x + d$ curve. This suggests that if resources are not excessively contented and there is not much queuing in the system (*leftmost portion* of the figure), increasing concurrency increases the inference rate of the system with a logarithmic improvement.

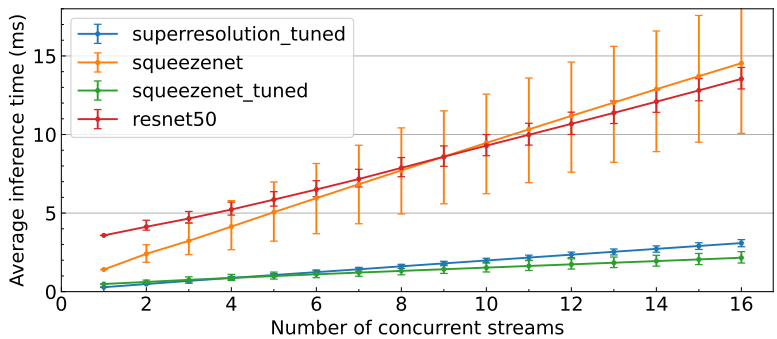
This improvement comes with a linear cost (represented by the negative term), which is quickly dominated by the logarithmic increase, resulting in an overall better inference rate when increasing the number of concurrent streams. The scaling holds as long as there are enough available resources. In the case where there are insufficient resources available, the inference rate quickly reaches a plateau and does not improve further, as it can be seen in the *squeezenet_tuned* case shown in Figure 8.4a.



(a) Average inference rate



(b) Average inference rate for *squeezenet*, with logarithmic curve fitting



(c) Average inference time

Figure 8.4: Distribution of average inference rate and average inference time for some models on an NVIDIA A100 as a function of different levels of concurrency. **Increasing the concurrency increases the average inference rate, at the cost of a higher, and less predictable, latency.**

To visualize the actual inference times, we plot in Figure 8.5 the cumulative distribution of the inference time for the *squeezenet_tuned* model, changing the level of concurrency in the system (*i.e.,* how many inferences are allowed to run simultaneously). As expected from the previous result, increasing the concurrency shifts the distribution curve toward the right and, more importantly, increases the variance of the inference time due to (i) more dynamic scheduling performed at run time and (ii) the higher level of resource contention.

This is shown by the increasingly slanted curves, whereas running a single inference at a time results in a very steep curve due to the near-constant time each inference takes. However, as shown above, running only one inference at a time makes poor use of the system's resources.

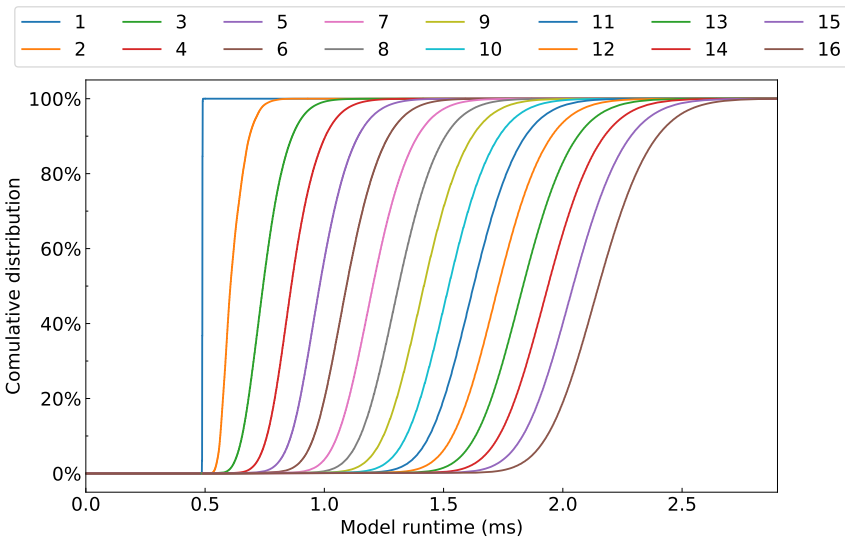


Figure 8.5: Inference time distribution for *squeezenet_tuned* on an NVIDIA A100 across multiple concurrency levels. *The lines, read from left to right, follow the legend order.* **Increasing the concurrency level increases the *slope* of the curves, representing a less predictable inferencing time.**

Lessons learned While these results prove that Clockwork's assumptions still hold (*e.g.,* lower concurrency allows better latency predictability), they also show how an application that could handle multiple parallel streams could significantly improve the serving throughput by freely choosing the trade-off point between lower inference time and higher overall throughput. In a scenario with a higher concurrency, the inference times are less predictable (*i.e.,* have a higher variance);

hence any CPU-driven workflow becomes much more inefficient since it is much more difficult to predict when an inference would end, and thus predict when the CPU can be used to do another task while waiting for the workload to finish (*i.e.,* without immediately calling `cudaDeviceSynchronize`).

The help by RDMA from GPU Building on these observations, we can exploit our *RDMA from GPU* building block (see Section 7.5.3) to build a system where the number of synchronization points and data transfers in the system are minimized, improving the overall efficiency of it. In particular, the ability of invoking a data-transfer from the GPU directly after the inference ends remove the needs to carefully schedule the execution of the return paths' data transfer, and it simplifies the system architecture: it can now adopt a *run-to-completion* approach, with the outputs directly leaving the GPU towards the requesting client. We used a similar *run-to-completion* architecture in our evaluation prototype Section 7.6.3.

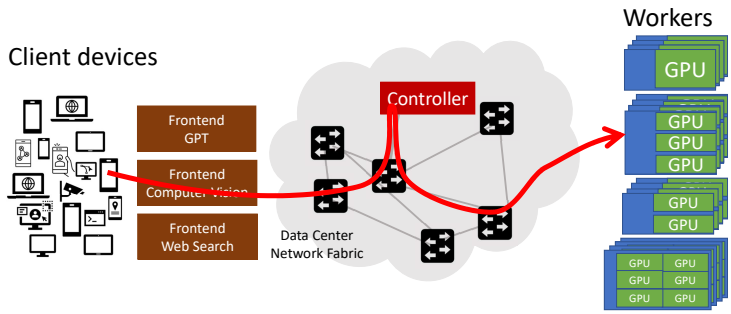
8.3 Towards a CPU-free architecture

Having presented our building block to provide a GPU-driven approach to RDMA (Section 7.5), put it in the context of inference serving (Section 7.6.3), and having analyzed some of the main issues in traditional inference serving architectures (Section 8.1.2), in this section we will now present an improved inference serving system based on RDMA, as a possible use-case for this technology.

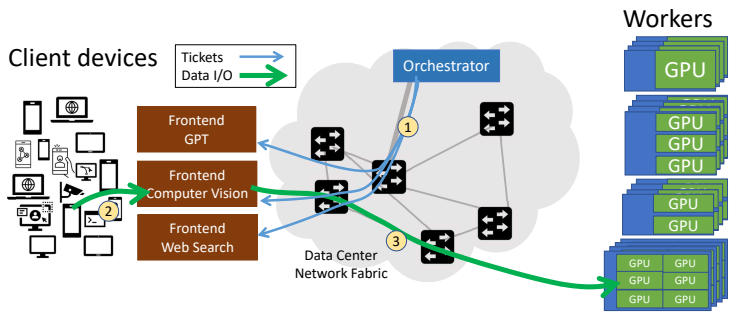
The main goal of this system is to maximize the performance of GPUs, without the need of costly proprietary hardware, or custom-built solutions, but rather using (and potentially re-using) existing commodity platforms.

In the following paragraphs we will present some key features of this system, and how each of them improves the efficiency of the system. Although some aspects of this system has been already implemented in the context of this thesis, the complete architecture has not been fully developed nor experimentally measured. We provide a description of the key characteristics of this system in the following paragraphs, with the aim to provide the reader with a *big picture* view of the system.

Fast and slow path separation In order to reach a closer-to-optimal scheduling and usage of all the resources, we believe that the execution time of the scheduler should not limit the processing of the requests. The system should thus be able to decouple its execution from the request processing. The benefits would be threefold: (i) the decision algorithm could take longer time to complete, (ii) periodic inferences (such as for frames in a video processing system) could be scheduled beforehand for the near future, and (iii) the scaling of the controller



(a) Clockwork model - all connections traverse the controller.



(b) Our proposed architecture - data is directly sent to worker nodes.

Figure 8.6: Data flow in Clockwork and our proposed architecture.

logic would not be coupled to the amount of data exchanged between the clients and the workers.

Clients & Connection Tracking To be able to track all clients, provide authentication, and correctly schedule inferences under high system occupancy, it is of outermost importance to employ a high-speed *Connection Tracking* mechanism. This could be achieved by exploiting the insights reported in Chapter 6, where different methods for tracking connections (*i.e.,* 5-tuples) in a Load Balancer scenario have been analyzed.

The application of these mechanisms are two-folds: (i) in a simple Internet-facing system, clients' TCP/UDP connections need to be translated in RDMA verbs

to GPUs,* with a 1-to-1 mapping, similar to a NAT; (ii) in a scale-out system the controller (and/or translator) functionalities needs to be served by multiple servers, in front of which a high-speed *L3 Load Balancer* is required. This service should distribute the load evenly and consistently across machines and could be implemented by directly deploying the system evaluated in Chapter 6.

RDMA *fast-path* transport Exploiting the support of modern GPUs and NICs to expose memory over RDMA, the clients would transmit inference inputs to the workers' GPUs directly over RDMA, avoiding any further traversal of the CPU network stack. This reduces the latency experience by the data (as reported in Section 8.1.1), avoiding any intermediate hosts' CPU stack traversal.

We note that there is not a strict need to implement all the application logic on RDMA, but rather we envision a system where a *slow-path* channel, implemented with standard tools (*e.g.,* gRPC [283]), is used to transmit all control signals and information, limiting the complexity of the controller and the ability to use widely adopted technologies to implement that.

Configurable concurrent execution Taking advantage of the benefits of GPU concurrent execution, the controller will have the ability to schedule inferences on workers to best suit the SLO of clients. This may involve tight tail latency, and thus less concurrency to be allowed, or best-effort service, and should be configured at runtime depending on the load and on the type of requests, following the dynamics analyzed in Section 8.2.

By allowing this further degree of freedom, we believe that the ability of sending responses directly from the GPUs of workers is of major importance, minimizing time spent waiting for synchronization on CPU, and the load on the system that this would introduce.

By exploiting our CPU-free mechanism described in Section 7.5.8, the system could achieve even better efficiency, avoiding any CPU intervention beside basic bookkeeping of the resources and model management (*e.g.,* when to load or unload a specific model's weight from a specific GPU) and metrics reporting (*e.g.,* actual GPU usage and performance metrics). This would also allow workers to go beyond the inherent scaling limitations derived by the number of PCIe lanes available on CPUs, which can be bypassed by using a topology augmented with PCIe switches, as described inSection 2.3.1, used in most modern multi-GPU machines.

Preventive scheduling Allowing the controller's scheduler to take decision independently of the actual requests' data it is possible to compute, beforehand,

*While RoCEv2 could theoretically work over the Internet, the weak security mechanisms and the limited support to congestion are not practical in a Wide Area Network.

the best schedule for the system. For instance, a client (or a front-end node) could request an inference to happen in the near feature, similarly to how pre-fetching of data happens in storage systems. By the time the actual inference would start on the worker (as instructed by the controller), the client would have delivered the data to its memory (*i.e.,* via a RDMA operation), without any need to wait for them. When the client is time synchronized with the worker, and the input transfer complete *just before* the inference starts, the response time will be the optimal one, and no time will be spent by the request data *waiting for the scheduling*.

Cluster-wide model-loading When multiple models are being served from the workers, and in particular when large models are involved, the load and eviction operations on the workers may introduce a high load in the system. Clockwork addresses this problem by carefully profiling the time that is required to load a model, and accounting for this time when performing the request scheduling, while also loading the entire set of models in all workers' main memory.

We propose to exploit GPUDirect, and PCIe P2P transactions in general, to load the data, reducing the load that these introduce to the workers' CPUs and avoiding bottlenecks that may arise from the limited CPU-GPU bandwidth available in multi-GPU machines. The system can thus store all models' data in a central location (*e.g.,* on the controller), with the workers fetching these data with RDMA READ operations.

It is a burden of the scheduler, which would anyway know all the models currently loaded on each GPU, to decide *when* to source a model, and *from where* (*e.g.,* when multiple sources would be available in scaled-out systems).

A further improvement can be introduced by allowing workers' GPUs to load weights from each other: when multiple GPUs need to load the same model, the weights could be already available on a neighbor worker's GPU, or even on another GPU on the same machine. In these cases loading model's weights from another GPU would be more efficient, and less prone to congestion.

Strict time synchronization In a tightly-scheduled system, it's important to reduce the time between input data writes and execution, *i.e.,* the time that data waits without any processing being done. To achieve this tight scheduling, and exploit the preventive scheduling (Section 8.3) with the littlest idle times, and reduce memory footprints, clients should send the data *as late as possible*. While Clockwork's solution to this synchronization is a home-brew time-stamping system, we believe that the natural solution is a systematic time synchronization of all involved machines' clocks, which would allow accurate measurements of network latencies and correct scheduling of all operations.

Although reaching good time synchronization has always been a complex

task in big networks, and the literature is quite thorough in this area [284, 285], most datacenter NICs provide support for Precision Time Protocol (PTP), allowing to achieve high precision time synchronization between several machines with commodity equipment and standard software support. This network protocol has been proposed as improvement to the widely supported Network Time Protocol (NTP), potentially reaching sub-nanosecond precision in a well-configured infrastructure [286]. Some providers are already deploying tightly time-synchronized clusters [287]. Therefore, we believe that achieving good synchronization between hosts involved in the system, although not a major requirement to achieve good performance, is a *low-hanging fruit* that can be easily achieved with basic configuration of the environment.

Chapter 9

GPUs in the *UNIX Way*

Exploiting our RDMA implementation, we envision an open ecosystem where GPUs, CPU nodes, and other accelerators, could collaborate & communicate to create processing pipelines, using *each resource at its best potential*. This chapter aims to give an overview of this idea and describe the basic features of this approach, which is left as future continuation of this thesis.

9.1 ML applications are pipelines

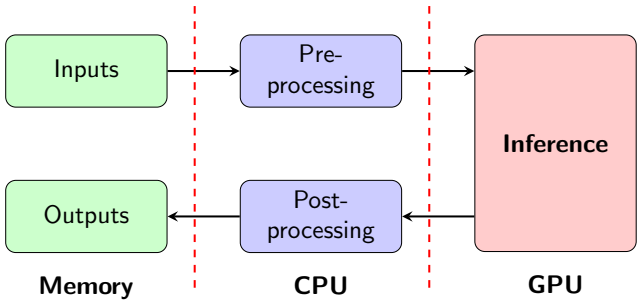
Building upon evidences from literature [141], and analysis of the common structure of ML applications, most ML applications can be seen as *multi-stage* processing pipelines. This is particularly relevant in GPU-accelerated applications, where some processing is offloaded to a specific device, as already discussed in Section 2.2.1, and as represented in Figure 9.1:

Pre-processing: inputs are received through the network, or read from a disk, and prepared to be processed in the *model*. This is usually performed on the CPU and involves type conversion, data re-shaping, data-compression, and other type of operations to transform the raw data in some numeric form that can be processed by the *model*.

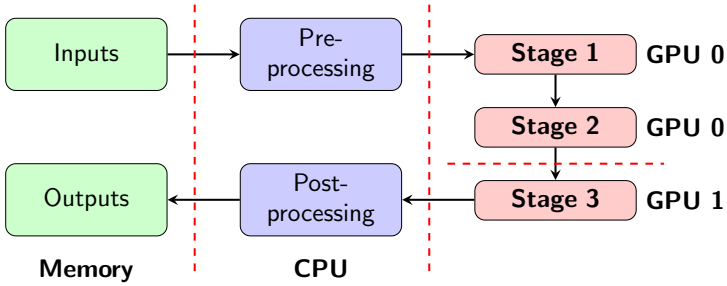
Processing: this is the core of the application and is typically run on an optimized *accelerator*. In most ML applications, this consists of a series of matrix operations, which would transform input data into output data.

Post-processing: the raw data generated by the *processing* phase are often not usable from clients, but rather need some further transformation. This phase,

which is usually run on a CPU, would transform the numeric data in images, text, or any other type that the application is expected to produce. These data would then be saved to a local disk, transmitted to a remote client, or used to trigger some other action.



(a) Single-stage ML inference application.



(b) Multi-stage GPU workload.

Figure 9.1: Components of a typical GPU-accelerated application, with respective placement. Multi-stage GPU workloads can be placed on the same GPU or across multiple devices. Dashed vertical lines represent the boundary across which data movement requires copy.

FLEET An effort in this direction has already started from DARPA's *FastNICs* project [288], with the goal of creating high-speed, dynamically reconfigurable networks, and components, aiming to close the gap between (slower) network interface speeds and current CPU and GPU performance [2]. An initial proposal of a platform to support this system has been described in [141], where a solution based on optical switches and *ad-hoc* designed FPGA-based optical NICs (*O-NICs*) is proposed to extend the PCIe domain of participating nodes across the cluster, improving the performance and resource usage of large-scale applications. This is made possible by a *planner* software, based on FlexFlow [289], which would identify the best topology and the best mapping between resources and tasks.

A commodity hardware alternative We argue that, despite the high performance that the *FLEET* approach could achieve, there are several limitations that will slow the adoption of this technology in the general public domain, and in particular in commodity-based datacenters. Building upon the RDMA framework presented in Chapter 7, we believe a similar system can be built using commodity hardware and using plain *RDMA* semantics to interconnect all components of the cluster.

9.2 The UNIX way

> This is the Unix philosophy: Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface.
>
> *Peter H. Salus [290]*

Taking inspiration from the *Unix Philosophy* [291], we can extend the concepts of modularity and flexibility to GPUs, realizing a pipeline architecture close to the one proposed by FLEET [141]:

Do one thing and do it well: similarly to *FLEET*, we believe it is fundamental to map tasks on the most suitable components. These could be GPUs, CPUs, or any other accelerator that could be interconnected with RDMA (*e.g.,* through an integrated interface or through standalone NICs and PCI-e switches).

Write programs to work together: to use a large resource pool in the best possible way, it is of outermost importance to allow applications to coexist and run in parallel, supported also by the throughput insights presented in Section 8.2. NICs and switches will control the congestion of the network, while the IP addressing space and the RDMA keys on the hosts will allow co-existence of more applications on the same network. Native time-sharing support in GPU runtimes, and *ad-hoc* schedulers may take care of coordinating operations and better allocate resources to different tasks.

Write programs to handle text streams, because that is a universal interface: this could be implemented with the use of a standard RDMA stack and semantics over an Ethernet network, detaching from vendor-specific protocols (*e.g.,* NCCL [18]) and HPC-oriented stacks (*e.g.,* MPI [263] and UCX [17]). RDMA handling would introduce minimal overhead on end devices, which would leverage the specialized Processing Units in NICs

to provide the network stack support (representing the *pipe* element in the UNIX system) and minimal software components to control this offloading (*e.g.,* our RDMA implementation).

Commodity chassis design We believe that this architecture can be built on top of commodity server chassis equipped with PCIe switches.* These can be equipped with a *wimpy* CPU if no useful data processing (*i.e.,* as part of the pipeline) would be performed, and with the bare minimum memory needed to run the *control logic* for the accelerators, potentially re-purposing existing decommissioned servers (*e.g.,* with CPUs that became too old for some useful processing).

SmartNIC-driven setups Similar to the previous design, SmartNICs (*e.g.,* NVIDIA BlueField [253]) can be used to initialize and coordinate the accelerators, taking advantage of the (smaller) processing power available on these, while the network interfaces can be used to establish a control-plane communication channel with a central controller. This is similar to the architecture proposed in Lynx [292], although in the latter the data traverses the CPUs on the SmartNIC, similar to what happens on a traditional server.

Ad-hoc NIC-GPU devices NVIDIA has already announced *converged accelerators* featuring a GPU and a NIC on the same PCB [168,293]. Although these still require a host system to be initialized (*i.e.,* to assign PCIe addresses), we believe that further development may remove this need, making them the perfect target for our pipeline architecture, or as part of a minimally CPU-driven system (*e.g.,* the commodity chassis described above). Similarly, FPGA-based NICs can be used to perform some type of computation, with the RDMA stack implemented in an*ad-hoc* hardware ASIC or as a soft core that shares resources on the FPGA.

9.3 Use-cases

In this section, we propose some use-cases that could leverage the pipelined RDMA-based architecture described above:

Inference serving: extending the idea presented in Section 8.3, *worker* nodes can operate in a pipeline, with data flowing through them as instructed by the controller, with minimal logic executed in the worker node.

*This is not a fundamental limitations, but reduces any possible PCIe bandwidth limitation and latency that may arise from traversing the PCIe Root Complex

Video processing: as most datacenters AI-oriented GPUs lack dedicated components to process videos (*e.g.,* encoders and decoders), and most graphic-oriented GPUs often provide better performance in video processing, video processing chains could be built to exploit characteristics of the two categories. Possible applications could be resolution enhancement, on-demand transcoding, video stabilization, or live event broadcast production [294].

Remote cloud gaming and desktop: similarly, many cloud-based solutions for gaming potentially waste GPU resources [295] to encode the video-stream to deliver to end-users, which needs to be encapsulated to IP streams on the CPU (and thus be transferred across the PCIe bus). Taking advantage of our approach, the output video frames* could be transferred by RDMA over another GPU, or dedicated accelerator (*e.g.,* FPGA), to be encoded and eventually transmitted to the end-user, with minimal overheads on the GPU, and zero overhead on the CPU running the actual software (*e.g.,* the videogame). Higher benefits could be expected in AI-driven content generation, where server-grade GPUs may lack hardware encoders.

Real-time anomaly detection: while many AI-assisted application have loose response time requirements, there is an increasing niche of applications that require very low response time on bandwidth-intensive data streams, such as anomaly detection in network traffic, video-surveilled production plants, and data-intensive sensor-driver applications in general. We believe that this pipeline architecture could augment the performance and capabilities of these applications without disrupting current architectures: scaling of GPU resources dedicated to providing the workload would be independent of the CPU required to run the system, with the latter traditionally scaling linearly to the size of the system. Edge deployments could also benefit from such an architecture, reducing the power, and costs, needed to deliver the services.

*In a classic desktop scenario, these would be output on the physical port (*e.g.,* DisplayPort) of the GPU to be displayed on a monitor.

Chapter 10

Conclusions and Future Works

This thesis focused on analyzing some critical performance aspects of common GPU-driven applications, providing the building blocks to go beyond the common CPU-centric paradigms used by most applications, and identifying strategies both in software and hardware to perform this shift.

First, we have identified the current trends for both GPU and NIC evolution, showing how some commonly adopted hardware paradigms are not scalable for GPU-driven workloads (Section 2.6, Section 2.7). We have analyzed some technical characteristics that would allow better performance for these applications (Section 2.3.1, Section 2.3.3), and how these have been adopted by State-of-The-Art works (Chapter 3).

This allowed us to summarize the main research problems in Chapter 4, together with the challenges associated with them and our main hypothesis.

We explore *Connection Tracking* field, focusing on a Load Balancer application running at 100 Gbps (Chapter 6). A *Stateful Load Balancer* represents a key component to allow a large system to scale up, and down. The capability of tracking *all* connections in a high-speed system represents an important requirement in any at-scale application, such as an inference serving system. This allowed us to gain insights on the performance bottlenecks that such a service would introduce, with a focus on the performance of different *Hash Tables* implementations, the most commonly used data-structure in this field.

We have then explored a way to enable GPU runtime to generate RDMA verbs directly, allowing to reduce the intervention of CPUs in Chapter 7. We identified this implementation as one way to solve the bottlenecks introduced by CPU

traversal in GPU-driven applications, and we analyzed the benefits of this solution in Section 7.6. Our analysis showed that, despite the simple implementation adopted by our design and the lower clock speed of GPUs, it is possible to invoke RDMA posting routines directly from the GPU runtime, without major bottlenecks when used in realistic applications. The major benefit of this implementation is the reduction of inter-device memory copies, which improves the overall application latency. This also reduces CPU usage, while increasing the overall system performance. Consequently, this also reduces the total power usage of the system, which is desirable for the reduction of OPEX, environmental aspects, and in general to widen the adoption of GPU-driven applications, especially in power-constraint spaces, such as mobile and edge computing facilities.

Building on this contribution, we have analyzed how a State-of-The-Art work Inference Serving application is structured, and its main limitations (Chapter 8). In Section 8.3 we propose a re-design of this architecture based on our contributions introduced in Chapter 7, envisioning a system where clients could interact directly over a RDMA channel with GPU-based workers, while a global controller would take the burden to coordinate all interactions, targeting specific SLOs. In this context, our work on *Connection Tracking* could be used as part of the system's front-end application, which would require strong scalability to support to volume of modern workloads, as enabled by our *RDMA*-based GPU communication.

Finally, Chapter 9 proposes a further application for our GPU-driven RDMA stack, transforming GPU-driven applications to *UNIX-style* pipelines, where RDMA would be used as the standard interaction channel. This architecture would allow generic GPU-based applications to take advantage of a disaggregated infrastructure built on commodity hardware, placing every processing stage of the pipeline on the

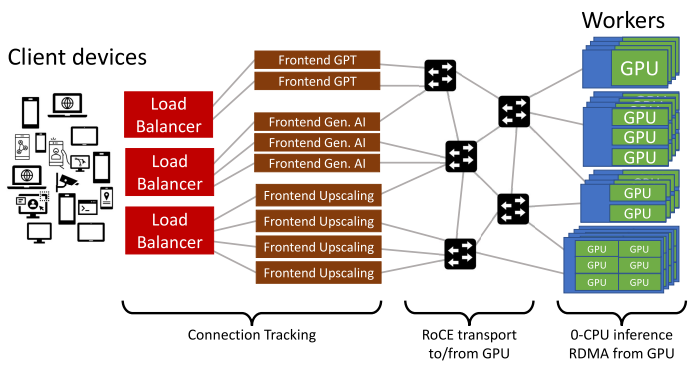


Figure 10.1: Overview of a generic scaled-out inference serving system, with the three main contributions of this thesis: (i) Connection Tracking, (ii) efficient data transport to/from GPU, and (iii) 0-CPU inference processing.

most suitable device (*i.e.,* CPUs, GPUs, FPGAs, or other accelerators).

Figure 10.1 summarizes the contribution of this thesis in a generic inference serving architecture, with a (i) layer of ingress stateful load balancers, (ii) a set of front-end servers (*e.g.,* HTTPs) communicating to a number of GPU-accelerated worker machines over RDMA with (iii) GPU-driven networking and no CPU usage on these.

10.1 Future works

We outline some possible future directions for the topics explored in this thesis as follows, starting from the ideas presented in Section 8.3 and Chapter 9:

Integration of our RDMA implementation in a State-of-The-Art framework Although we demonstrate the raw performance of an Inference Serving *worker* based on our RDMA implementation (in Section 6.3), we did not address the implementation of the bigger Inference Serving *framework*, and neither the integration of our implementation in a State-of-The-Art system. We envision that a *bottom-up* implementation, based on our RDMA stack, would ensure optimal system performance, taking advantage of the relevant novelties presented by other State-of-The-Art works. Further extensions should consider network congestion, and the design of a scheduler able to maintain it to the minimum, overcoming the inherent limitations of RDMA protocols, and Ethernet devices in general, when dealing with overloaded networks.

UNIX-style pipelines for generic GPU-driven applications As introduced in Chapter 9, we believe that most GPU-driven applications could benefit by a pipeline-based architecture, where data are transferred through RDMA and CPUs are minimally involved, reducing the bottlenecks and overheads introduced by CPU traversals and memory copies. We presented our high-level view, and we leave the implementation as a follow-up work to this thesis.

Parallel, multi-thread support for RDMA posting routines Although our näive implementation demonstrated that it is possible to run RDMA posting routines directly from the GPU runtime, without changing the whole RDMA stack, more work is needed to transform our implementation into a production-grade stack. In particular, multi-thread support represents one of the main features that needs to be implemented, allowing multiple parallel tasks (*e.g.,* multiple GPU streams) to access the RDMA stack simultaneously, without compromising data integrity. Similarly, better error handling mechanisms, and support to more RDMA verb types would increase the potential applications of our solution.

GPU, PCIe, and NICs dynamics when performing RDMA operations As introduced in Section 7.6.1, our system presents some performance dynamics that are related to the interaction between all peripherals involved in the system. A throughout analysis is needed to understand how these dynamics affect applications under realistic loads, and what is the best combination of parameters to minimize any overhead in the system.

Graph runtime improvements As discussed in Section 7.5.6, the current proposed implementation for an inference serving system does not remove all memory copies, due to some limitations in both the TVM software support and the native CUDA Graph semantics. A further analysis of these issues could remove the need for these copies and improve the overall performance of the system, providing an *easy to integrate* framework for developers willing to optimize their applications.

Extending support to other vendors Although this thesis focused specifically on NVIDIA GPUs and NICs, all the insights presented could be extended to a more heterogeneous hardware landscape. Building upon standard protocols and interfaces (namely, RDMA and PCIe), it is straightforward to envision support for AMD GPUs, different NIC vendors, and other accelerators in general. This represents a main difference with other industry-adopted frameworks (*e.g.,* NCCL [18]), which target a single vendor and are thus more difficult to extend to other devices.

Controller Connection Tracking The need to track users, connections, and resource usages by different users in any *at-scale system* is directly related to the ability to perform *Connection Tracking* at high speed. This represents the basic characteristics of a *cluster-wide controller*, as the one described for the architecture proposed in Section 8.3, to be able to correctly schedule, scale, and manage resources. The implementation of this controller may take advantage of the features offered by modern NICs [205] or be partially offloaded to dedicated hardware, such as programmable switches [204]. The latter may also have visibility on the data traffic, allowing the controller to better schedule resources avoiding network congestion.

Bibliography

[1] NVIDIA Corporation, "NVIDIA DGX-H100 user guide - introduction", NVIDIA, Aug. 2023. [Online]. Available: https://docs.nvidia.com/dgx/dgxh100-user-guide/introduction-to-dgxh100.html

[2] A. Gholami, Z. Yao, S. Kim, M. W. Mahoney, and K. Keutzer, "Ai and memory wall", Mar 2021.

[3] J. Manyika and J. Bughin, "The coming of ai spring", Oct. 2019, [Accessed: 2024-01-17]. [Online]. Available: https://www.mckinsey.com/mgi/overview/in-the-news/the-coming-of-ai-spring

[4] R. Bommasani, "Ai spring? four takeaways from major releases in foundation models", Mar. 2023, [Accessed: 2024-01-17]. [Online]. Available: https://hai.stanford.edu/news/ai-spring-four-takeaways-major-releases-foundation-models

[5] T. P. Morgan, "Meta platforms is determined to make ethernet work for ai", The Next Platform, Sep. 2023, [Accessed: 2024-02-27]. [Online]. Available: https://www.nextplatform.com/2023/09/26/meta-platforms-is-determined-to-make-ethernet-work-for-ai/

[6] M. Walsh, "ChatGPT statistics (2023) — the key facts and figures", Style Factory, Aug. 2023. [Online]. Available: https://www.stylefactoryproductions.com/blog/chatgpt-statistics

[7] G. Appenzeller, M. Bornstein, and M. Casado, "Navigating the high cost of ai compute", Apr. 2023, [Accessed: 2024-01-17]. [Online]. Available: https://a16z.com/navigating-the-high-cost-of-ai-compute/

[8] A. Goel, "Unraveling gpu inference costs for fine-tuned open-source models v/s closed platforms", Jun. 2023, [Accessed: 2024-01-17]. [Online]. Available: https://mlops.community/unraveling-gpu-inference-costs-for-fine-tuned-open-source-models-v-s-closed-platforms/

[9] K. M. Hazelwood, S. Bird, D. M. Brooks, S. Chintala, U. Diril *et al.*, "Applied machine learning at Facebook: A datacenter infrastructure perspective", *2018 IEEE International Symposium on High Performance*

Computer Architecture (HPCA), pp. 620–629, 2018.

[10] A. Kumar, A. Sivasubramaniam, and T. Zhu, "SplitRPC: A Control + Data path splitting RPC stack for ML inference serving", *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 7, no. 2, May 2023. [Online]. Available: https://doi.org/10.1145/3589974

[11] Q. Cai, M. Vuppalapati, J. Hwang, C. Kozyrakis, and R. Agarwal, "Towards µs tail latency and terabit ethernet: disaggregating the host network stack", in *Proceedings of the ACM SIGCOMM 2022 Conference*. Association for Computing Machinery, 2022, p. 767–779. [Online]. Available: https://doi.org/10.1145/3544216.3544230

[12] M. Girondi, T. Barbette, and M. Chiesa, "High speed connection tracking on modern servers", in *IEEE HPSR 2021*, 2021. [Online]. Available: https://ieeexplore.ieee.org/document/9481841

[13] NVIDIA Corporation, "Nvidia mlnx_ofed documentation", Feb. 2021, [Accessed: 2024-01-17]. [Online]. Available: https://docs.nvidia.com/networking/display/mlnxofedv23100550

[14] "RDMA core userspace libraries and daemons", GitHub, 2023. [Online]. Available: https://github.com/linux-rdma/rdma-core

[15] NVIDIA Corporation, "Cuda zone", [Accessed: 2024-01-17]. [Online]. Available: https://developer.nvidia.com/cuda-zone

[16] R. Recio, B. Metzler, P. Culley, J. Hilland, and D. Garcia, " A Remote Direct Memory Access Protocol Specification ", Internet Requests for Comments, RFC Editor, RFC 5040, Feb 2003. [Online]. Available: https://www.rfc-editor.org/rfc/rfc5040.txt

[17] NVIDIA Corporation, "Unified Communication - X Framework Library", 2023, nVIDIA HPC-X Software Toolkit Rev 2.15. [Online]. Available: https://docs.nvidia.com/networking/display/HPCXv215/Unified+Communication+-+X+Framework+Library

[18] NVIDIA Corporation, "NVIDIA Collective Communications Library (NCCL)", [Accessed: 2023-11-14]. [Online]. Available: https://developer.nvidia.com/nccl

[19] A. Gujarati, R. Karimi, S. Alzayat, W. Hao, A. Kaufmann *et al.*, "Serving DNNs like Clockwork: Performance predictability from the bottom up", in *USENIX OSDI 2020*. USENIX Association, 2020.

[20] T. Barbette, "GitHub - FastClick", GitHub, 2015, https://github.com/tbarbette/fastclick. [Online]. Available: https://github.com/tbarbette/fastclick

[21] NVIDIA Mellanox, "Bluefield SmartNIC", 2020. [Online]. Available: https://network.nvidia.com/files/doc-2020/pb-bluefield-smart-nic.pdf

[22] H. Ghasemirahni, "Packet order matters!: Improving application

performance by deliberately delaying packets", Ph.D. dissertation, KTH Royal Institute of Technology, 2021. [Online]. Available: https://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-293973

[23] A. Chan, "Gpt-3 and instructgpt: technological dystopianism, utopianism, and "contextual" perspectives in ai ethics and industry", *AI and Ethics*, vol. 3, pp. 53–64, 2022.

[24] T. W. Bickmore, H. Trinh, S. Olafsson, T. K. O'Leary, R. Asadi *et al.*, "Patient and consumer safety risks when using conversational assistants for medical information: An observational study of siri, alexa, and google assistant", *J Med Internet Res*, vol. 20, no. 9, p. e11510, Sep 2018. [Online]. Available: https://doi.org/10.2196/11510

[25] K. Bugbee and R. Ramachandran, IMPACT Unofficial - Medium, Feb. 2024, [Accessed: 2023-10-19]. [Online]. Available: https://impactunofficial.medium.com/the-ethics-of-large-language-models-who-controls-the-future-of-open-science-43cca235401d

[26] The CAIDA Anonymized Internet Traces. (2019). [Online]. Available: http://www.caida.org/data/passive/passive_dataset.xml

[27] A. Kaplan, *Artificial intelligence, business and civilization: our fate made in machines*. Routledge, 2022.

[28] S. G. Brush, "History of the Lenz-Ising model", *Rev. Mod. Phys.*, vol. 39, pp. 883–893, Oct. 1967. [Online]. Available: https://link.aps.org/doi/10.1103/RevModPhys.39.883

[29] M. I. Jordan and T. M. Mitchell, "Machine learning: Trends, perspectives, and prospects", *Science*, vol. 349, no. 6245, pp. 255–260, 2015.

[30] J. Schmidhuber, "Deep learning in neural networks: An overview", *Neural Networks*, vol. 61, pp. 85–117, 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0893608014002135

[31] A. Khan, A. Sohail, U. Zahoora, and A. S. Qureshi, "A survey of the recent architectures of deep convolutional neural networks", *Artificial Intelligence Review*, vol. 53, pp. 5455 – 5516, Apr. 2019.

[32] C. Li, "OpenAI's GPT-3 language model: A technical overview", Lambda Labs, Jun. 2020. [Online]. Available: https://lambdalabs.com/blog/demystifying-gpt-3

[33] R. Thoppilan, D. D. Freitas, J. Hall, N. M. Shazeer, A. Kulshreshtha *et al.*, "LaMDA: Language models for dialog applications", *ArXiv*, vol. abs/2201.08239, 2022.

[34] "metaseq:175B Training Run Completed", Jan. 2022. [Online]. Available: https://github.com/facebookresearch/metaseq/blob/main/projects/OPT/chronicles/final_update.md

[35] A. Mok, "ChatGPT could cost over $700,000 per day to operate.

Microsoft is reportedly trying to make it cheaper." Insider, Inc., Apr. 2024. [Online]. Available: https://www.businessinsider.com/how-much-chatgpt-costs-openai-to-run-estimate-report-2023-4

[36] T. Goldstein, "Tweet", Twitter, Dec. 2022, [Accessed: 2023-09-15]. [Online]. Available: https://twitter.com/tomgoldsteincs/status/1600196995389366274?lang=en

[37] "103+ chatgpt statistics & user numbers in sept 2023 (new data)", Nerdy Nav, Sep. 2023, [Accessed: 2023-09-15]. [Online]. Available: https://nerdynav.com/chatgpt-statistics/

[38] C. Hetzner, "Chatgpt moves to cash in on its fame as openai launches plan to charge monthly fee for premium subscribers", Fortune, Jan. 2023, [Accessed: 2023-09-15]. [Online]. Available: https://fortune.com/2023/01/23/chatgpt-openai-plan-monthly-fee-for-premium-subscribers/

[39] J. Choquette, "NVIDIA Hopper H100 GPU: Scaling performance", *IEEE Micro*, vol. 43, no. 3, pp. 9–17, 2023.

[40] S. K. Moore, "The secret to nvidia's ai success", *IEEE Spectrum*, Sep. 2023, [Accessed: 2023-09-11]. [Online]. Available: https://spectrum.ieee.org/nvidia-gpu

[41] A. Patrizio, "Nvidia still crushing the data center market", Network World, Dec. 2022. [Online]. Available: https://www.networkworld.com/article/3684174/nvidia-still-crushing-the-data-center-market.html

[42] S. Mittal and J. S. Vetter, "A survey of CPU-GPU heterogeneous computing techniques", *ACM Computing Surveys (CSUR)*, vol. 47, pp. 1 – 35, 2015. [Online]. Available: https://dl.acm.org/doi/10.1145/2788396

[43] J. Sanders and E. Kandrot, *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[44] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA: Is CUDA the parallel programming model that application developers have been waiting for?" *Queue*, vol. 6, no. 2, p. 40–53, Mar. 2008. [Online]. Available: https://doi.org/10.1145/1365490.1365500

[45] "SIMT Architecture", CUDA C++ Programming Guide, Aug. 2023. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#simt-architecture

[46] NVIDIA Corporation, "NVIDIA H100 Tensor Core GPU - DataSheet", NVIDIA, Jul. 2023. [Online]. Available: https://resources.nvidia.com/en-us-tensor-core/nvidia-tensor-core-gpu-datasheet

[47] "PCI Express 6.0 specification", PCI-SIG. [Online]. Available: https://pcisig.com/pci-express-6.0-specification

[48] A. Li, S. L. Song, J. Chen, J. Li, X. Liu *et al.*, "Evaluating modern GPU

interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect", *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 1, pp. 94–110, Jul. 2019.

[49] D. Appelhans and B. Walkup, "Leveraging NVLINK and asynchronous data transfer to scale beyond the memory capacity of GPUs", in *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*. Association for Computing Machinery, Nov. 2017. [Online]. Available: https://doi.org/10.1145/3148226.3148232

[50] T. P. Morgan, "Nvidia's Grace-Hopper hybrid systems bring huge memory to bear", The Next Platform, May 2023. [Online]. Available: https://www.nextplatform.com/2023/05/29/nvidias-grace-hopper-hybrid-systems-bring-huge-memory-to-bear/

[51] NVIDIA Corporation, "NVIDIA Grace Hopper Superchip", NVIDIA, 2023. [Online]. Available: https://www.nvidia.com/en-us/data-center/grace-hopper-superchip/

[52] NVIDIA Corporation, "NVIDIA Scalable Hierarchical Aggregation and Reduction Protocol (SHARP)™", NVIDIA SHARP Documentation, [Accessed: 2023-09-11]. [Online]. Available: https://docs.nvidia.com/networking/pages/viewpage.action?pageId=19819203

[53] NVIDIA Corporation, "Upgrading Multi-GPU Interconnectivity with the Third-Generation NVIDIA NVSwitch", NVIDIA Technical Blog, Apr. 2022, [Accessed: 2023-09-11]. [Online]. Available: Eassa,AshrafandIshii, AlexandWells,Ryan

[54] NVIDIA Corporation, "Cuda runtime api - stream management", Nov 2023, [Accessed: 2024-01-17]. [Online]. Available: https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__STREAM.html

[55] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan *et al.*, "TVM: An automated end-to-end optimizing compiler for deep learning", in *USENIX OSDI 2018*. USENIX Association, 2018, p. 579–594. [Online]. Available: https://www.usenix.org/system/files/osdi18-chen.pdf

[56] A. Gray, "Getting started with CUDA Graphs", NVIDIA developer - Technical Blog, Sep. 2019. [Online]. Available: https://developer.nvidia.com/blog/cuda-graphs/

[57] C. Yu, S. Royuela, and E. Quiñones, "OpenMP to CUDA Graphs: A compiler-based transformation to enhance the programmability of NVIDIA devices", in *Proceedings of the 23th International Workshop on Software and Compilers for Embedded Systems*. Association for Computing Machinery, 2020, p. 42–47. [Online]. Available: https://doi.org/10.1145/3378678.3391881

[58] "Device graph launch", CUDA C++ Programming Guide, Release v12.2,

Aug. 2023. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#device-graph-launch

[59] "Memory Optimizations", CUDA C++ Programming Guide. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#memory-optimizations

[60] Pak Markthub et al., "NVIDIA/gdrcopy", https://github.com/NVIDIA/gdrcopy, 2023, [Accessed: 2023-09-19].

[61] NVIDIA Corporation, "Magnum IO GDRCopy: Enable faster memory transfers between CPU and GPU with GDRCopy", https://developer.nvidia.com/gdrcopy, 2023, [Accessed: 2023-09-19].

[62] "AMD CDNA Architecture", AMD Inc., 2020. [Online]. Available: https://www.amd.com/system/files/documents/amd-cdna-whitepaper.pdf

[63] "ROCm documentation", AMD Inc., Aug. 2023. [Online]. Available: https://rocm.docs.amd.com/_/downloads/en/latest/pdf/

[64] "AMD Infinity Architecture", AMD Inc., 2023. [Online]. Available: https://www.amd.com/en/technologies/infinity-architecture

[65] Microsoft, "ONNX Runtime". [Online]. Available: https://onnxruntime.ai/

[66] "Apache TVM". [Online]. Available: https://tvm.apache.org/

[67] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture", *Commun. ACM*, vol. 62, no. 2, p. 48–60, Jan. 2019. [Online]. Available: https://doi.org/10.1145/3282307

[68] M. Emani, Z. Xie, S. Raskar, V. Sastry, W. Arnold *et al.*, "A comprehensive evaluation of novel AI accelerators for deep learning workloads", in *2022 IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2022, pp. 13–25.

[69] N. P. Jouppi, C. Young, N. Patil, and D. Patterson, "A domain-specific architecture for deep neural networks", *Commun. ACM*, vol. 61, no. 9, p. 50–59, Aug. 2018. [Online]. Available: https://doi.org/10.1145/3154484

[70] N. Jouppi, G. Kurian, S. Li, P. Ma, R. Nagarajan *et al.*, "TPU v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings", in *Proceedings of the 50th Annual International Symposium on Computer Architecture*. Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3579371.3589350

[71] Cerebras Systems Inc., "Cerebras systems: Achieving industry best AI performance through a systems approach", Apr. 2021. [Online]. Available: https://cerebras.net/wp-content/uploads/2021/04/Cerebras-CS-2-Whitepaper.pdf

[72] S. Knowles, "Graphcore", *2021 IEEE Hot Chips 33 Symposium (HCS)*, pp.

1–25, 2021.

[73] R. Prabhakar, S. Jairath, and J. L. Shin, "SambaNova SN10 RDU: A 7nm dataflow architecture to accelerate software 2.0", *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, vol. 65, pp. 350–352, 2022.

[74] D. Abts, J. Kim, G. Kimmell, M. Boyd, K. Kang *et al.*, "The groq software-defined scale-out tensor streaming multiprocessor: From chips-to-systems architectural overview", *2022 IEEE Hot Chips 34 Symposium (HCS)*, pp. 1–69, 2022.

[75] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis *et al.*, "TensorFlow: A system for large-scale machine learning", in *USENIX OSDI 2016*. USENIX Association, 2016, p. 265–283.

[76] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury *et al.*, "PyTorch: An imperative style, high-performance deep learning library", in *33rd Conference on Neural Information Processing Systems (NeurIPS 2019)*, 2019.

[77] T. Chen, M. Li, Y. Li, M. Lin, N. Wang *et al.*, "MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems", *ArXiv*, vol. abs/1512.01274, 2015.

[78] Intel Corporation, "AI coming to the PC at scale", [Accessed: 2023-09-05]. [Online]. Available: https://www.intel.com/content/www/us/en/newsroom/news/ai-coming-to-pc-at-scale.html

[79] Intel Corporation, "AI accelerated Intel® Xeon® Scalable Processors product brief", [Accessed: 2023-09-05]. [Online]. Available: https://www.intel.com/content/www/us/en/products/docs/processors/xeon-accelerated/ai-accelerators-product-brief.html

[80] Intel Corporation, "Intel® Xeon® CPU Max Series", Jan. 2023, [Accessed: 2023-09-13]. [Online]. Available: https://cdrdv2.intel.com/v1/dl/getContent/765259?fileName=xeon-cpu-max-series-product-brief.pdf

[81] Advanced Micro Devices, Inc., "AMD Ryzen™ AI - windows laptops with AI built in", [Accessed: 2023-09-05]. [Online]. Available: https://www.amd.com/en/products/ryzen-ai

[82] Apple Inc., "Deploying transformers on the Apple Neural Engine", Jun. 2022, [Accessed: 2023-09-05]. [Online]. Available: https://machinelearning.apple.com/research/neural-engine-transformers

[83] Samsung, "Neural Processing Units (NPUs): Semiconductors working like a human brain", Nov. 2019, [Accessed: 2023-09-05]. [Online]. Available: https://semiconductor.samsung.com/support/tools-resources/dictionary/the-neural-processing-unit-npu-a-brainy-next-generation-semiconductor/

[84] M. Gupta, "Google Tensor is a milestone for machine learning", Oct. 2021, [Accessed: 2023-09-05]. [Online]. Available: https://blog.google/products/

pixel/introducing-google-tensor/
[85] Qualcomm Technologies, Inc., "Groundbreaking AI", [Accessed: 2023-09-05]. [Online]. Available: https://www.qualcomm.com/products/mobile/snapdragon/smartphones/mobile-ai
[86] "TensorFlow Lite | ML for mobile and edge devices", https://www.tensorflow.org/lite, [Accessed: 2023-09-05].
[87] R. David, J. Duke, A. Jain, V. Janapa Reddi, N. Jeffries *et al.*, "Tensorflow lite micro: Embedded machine learning for YinyML systems", *Proceedings of Machine Learning and Systems*, vol. 3, pp. 800–811, 2021. [Online]. Available: https://proceedings.mlsys.org/paper_files/paper/2021/file/6c44dc73014d66ba49b28d483a8f8b0d-Paper.pdf
[88] NVIDIA Corporation, "GPUDirect documentation", Aug. 2023, [Accessed: 2023-09-18]. [Online]. Available: https://docs.nvidia.com/cuda/gpudirect-rdma/index.html
[89] D. Mayhew and V. Krishnan, "PCI express and advanced switching: evolutionary path to building next generation interconnects", in *11th Symposium on High Performance Interconnects, 2003. Proceedings.*, 2003, pp. 21–29.
[90] R. Neugebauer, G. Antichi, J. F. Zazo, Y. Audzevich, S. López-Buedo *et al.*, "Understanding PCIe Performance for End Host Networking", in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. Association for Computing Machinery, 2018, p. 327–341. [Online]. Available: https://doi.org/10.1145/3230543.3230560
[91] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers, 3rd Edition*. O'Reilly Media, Mar. 2005.
[92] A. S. Tanenbaum and A. S. Woodhull, *Operating systems: design and implementation*, 2nd ed. Prentice Hall Englewood Cliffs, 1997, vol. 68.
[93] M. Wadekar, "Chapter 11 - InfiniBand, iWARP, and RoCE", in *Handbook of Fiber Optic Data Communication (Fourth Edition)*, 4th ed., C. DeCusatis, Ed. Academic Press, 2013, pp. 267–287. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780124016736000118
[94] A. Farshin, A. Roozbeh, G. Q. Maguire, and D. Kostić, "Make the most out of last level cache in Intel processors", in *Proceedings of the Fourteenth EuroSys Conference 2019*. Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3302424.3303977
[95] L. Rizzo, "netmap: A novel framework for fast packet i/o", in *USENIX Annual Technical Conference*, 2012.
[96] DPDK Project. LF Projects, LLC, "Data Plane Development Kit (DPDK) website". [Online]. Available: https://www.dpdk.org/
[97] T. Von Eicken, A. Basu, V. Buch, and W. Vogels, "U-net: A user-level

network interface for parallel and distributed computing", *ACM SIGOPS Operating Systems Review*, vol. 29, no. 5, pp. 40–53, 1995.

[98] A. Shpiner, E. Zahavi, O. Dahley, A. Barnea, R. Damsker *et al.*, "RoCE Rocks without PFC: Detailed evaluation", in *Proceedings of the Workshop on Kernel-Bypass Networks*. Association for Computing Machinery, Aug. 2017, p. 25–30. [Online]. Available: https://doi.org/10.1145/3098583.3098588

[99] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye *et al.*, "Rdma over commodity ethernet at scale", in *Proceedings of the 2016 ACM SIGCOMM Conference*. Association for Computing Machinery, 2016, p. 202–215. [Online]. Available: https://doi.org/10.1145/2934872.2934908

[100] InfiniBand™Trade Association, "RDMA over Converged Ethernet (RoCE) - Supplement to InfiniBand™Architecture Specification volume 1 release 1.2.1", Archived by Internet Archive from the original. [Online]. Available: https://web.archive.org/web/20160309123709/https://cw.infinibandta.org/document/dl/7148

[101] InfiniBand Trade Association, "Infiniband architecture specification", 2023. [Online]. Available: https://www.infinibandta.org/ibta-specification/

[102] Intel Corporation, "Understanding iWARP: Delivering low latency to ethernet", 2010, document number 0710/TS/MESH/PDF 324032-001US. [Online]. Available: https://www.intel.com/content/dam/support/us/en/documents/network/sb/understanding_iwarp_final.pdf

[103] Intel Corporation, "Intel® Omni-Path fabric host software", Jan. 2020, document number H76479 Revision 16.0. [Online]. Available: https://www.intel.com/content/dam/support/us/en/documents/network-and-i-o/fabric-products/Intel_OP_Fabric_Host_Software_UG_H76470_v16_0.pdf

[104] V. Galabov, "Mellanox breathes down Intel's neck for Ethernet adapter market dominance", Jul. 2019, [Accessed: 2023-09-06]. [Online]. Available: https://www.globalsmt.net/articles-and-papers/mellanox-breathes-down-intels-neck-for-ethernet-adapter-market-dominance/

[105] NVIDIA Corporation, "Connectx smartnics", [Accessed: 2023-09-06]. [Online]. Available: https://www.nvidia.com/en-us/networking/ethernet-adapters/

[106] Intel Corporation, "Intel® ethernet network adapter e810-cqda1/cqda2 product brief", Jul. 2023, [Accessed: 2023-09-06]. [Online]. Available: https://cdrdv2-public.intel.com/641671/Intel%20Ethernet%20Network%20Adapter%20E810-CQDA1_CQDA2.pdf

[107] Chelsio Communications, "RDMA – iWARP", [Accessed: 2023-09-06]. [Online]. Available: https://www.chelsio.com/nic/rdma-iwarp/

[108] Marvell, "Universal RDMA", [Accessed: 2023-09-06]. [Online].

Available: https://www.marvell.com/products/ethernet-adapters-and-controllers/universal-rdma.html

[109] Broadcom, "RDMA over Converged Ethernet (RoCE)", Aug. 2023, [Accessed: 2023-09-06]. [Online]. Available: https://www.broadcom.com/products/ethernet-connectivity/network-adapters/n1200g

[110] T. P. Morgan, "The eternal battle between InfiniBand and Ethernet in HPC", The Next Platform, Jul. 2021. [Online]. Available: https://www.nextplatform.com/2021/07/07/the-eternal-battle-between-infiniband-and-ethernet-in-hpc/

[111] D. Raffo, "Largest InfiniBand vendors merge; eye converged networks", Archived by Internet Archive from the original. [Online]. Available: https://web.archive.org/web/20170701002647/http://itknowledgeexchange.techtarget.com/storage-soup/largest-infiniband-vendors-merge-eye-converged-networks/

[112] T. P. Morgan, "Connecting the dots on why NVIDIA is buying Mellanox", The Next Platform, Mar. 2019. [Online]. Available: https://www.nextplatform.com/2019/03/11/connecting-the-dots-on-why-nvidia-is-buying-mellanox/

[113] T. Hoefler, D. Roweth, K. Underwood, B. Alverson, M. Griswold *et al.*, "Datacenter ethernet and rdma: Issues at hyperscale", *arXiv preprint arXiv:2302.03337*, 2023.

[114] UEC, "Overview of and motivation for the forthcoming ultra ethernet consortium specification", Jul 2023, [Accessed: 2023-11-17]. [Online]. Available: https://ultraethernet.org/wp-content/uploads/sites/20/2023/10/23.07.12-UEC-1.0-Overview-FINAL-WITH-LOGO.pdf

[115] L. Zheng, C. Jia, M. Sun, Z. Wu, C. H. Yu *et al.*, "Ansor: Generating high-performance tensor programs for deep learning", in *USENIX Symposium on Operating Systems Design and Implementation*, 2020.

[116] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling *et al.*, "MLPerf inference benchmark", in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 446–459.

[117] "Inference datacenter v3.0 results", MLCommons®, Apr. 2023. [Online]. Available: https://mlcommons.org/en/inference-datacenter-30/

[118] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition", in *Proceedings of the IEEE conference on computer vision and pattern recognition*, Jun. 2016, pp. 770–778.

[119] Stanford Vision Lab, Stanford University, Princeton University, "ImageNet", 2021. [Online]. Available: https://www.image-net.org/

[120] R. Callon, "The twelve networking truths", Apr. 1996, [Accessed: 2023-11-17]. [Online]. Available: https://www.rfc-editor.org/rfc/rfc1925

[121] A. Bjorlin, "Infrastructure for large scale AI: "empowering open"", Nov. 2022. [Online]. Available: https://drive.google.com/file/d/1qqjo-5JtYAcRlK_LWYuQFH-b9MoFOP02/view

[122] M. Scazzariello, T. Caiazzi, H. Ghasemirahni, T. Barbette, D. Kostić *et al.*, "A High-Speed stateful packet processing approach for Tbps programmable switches", in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Apr. 2023, pp. 1237–1255. [Online]. Available: https://www.usenix.org/conference/nsdi23/presentation/scazzariello

[123] FastData.io Project, "A terabit secure network data-plane", FastData.io Project, Apr. 2021. [Online]. Available: https://fd.io/presentations/Fdio-Icelake-Demo-v11i.pdf

[124] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm *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)*. USENIX Association, Apr 2014, pp. 489–502. [Online]. Available: https://www.usenix.org/system/files/conference/nsdi14/nsdi14-paper-jeong.pdf

[125] H. Ghasemirahni, T. Barbette, G. P. Katsikas, A. Farshin, A. Roozbeh *et al.*, "Packet order matters! Improving application performance by deliberately delaying packets", in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI '22)*, 2022, community Award Winner! [Online]. Available: https://www.usenix.org/system/files/nsdi22spring_prepub_ghasemirahni.pdf

[126] H. Li, Y. Dang, G. Sun, G. Liu, D. Shan *et al.*, "LemonNFV: Consolidating heterogeneous network functions at line speed", in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Apr. 2023, pp. 1451–1468. [Online]. Available: https://www.usenix.org/conference/nsdi23/presentation/li-hao

[127] A. Farshin, L. Rizzo, K. Elmeleegy, and D. Kostić, "Overcoming the IOTLB wall for multi-100-Gbps linux-based networking", *PeerJ Computer Science*, vol. 9, p. e1385, May 2023.

[128] A. Farshin, T. Barbette, A. Roozbeh, G. Q. Maguire Jr., and D. Kostić, "PacketMill: Toward per-core 100-gbps networking", in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, Apr. 2021, pp. 1–17. [Online]. Available: https://doi.org/10.1145/3445814.3446724

[129] "ONNX model zoo", GitHub, 2023. [Online]. Available: https://github.com/onnx/models

[130] "ModelZoo". [Online]. Available: https://modelzoo.co/

[131] X. Zhang, X. Zhou, M. Lin, and J. Sun, "Shufflenet: An extremely efficient convolutional neural network for mobile devices", in *Proceedings of the IEEE conference on computer vision and pattern recognition*, Jun. 2018, pp. 6848–6856.

[132] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally *et al.*, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size", *CoRR*, vol. abs/1602.07360, 2016. [Online]. Available: http://arxiv.org/abs/1602.07360

[133] K. Zhang, W. Zuo, and L. Zhang, "FFDNet: Toward a fast and flexible solution for CNN based image denoising", *IEEE Transactions on Image Processing*, vol. 27, no. 9, pp. 4608–4622, May 2018.

[134] "Radeon image filter SDK", GitHub. [Online]. Available: https://github.com/GPUOpen-LibrariesAndSDKs/RadeonImageFilter

[135] W. Shi, J. Caballero, F. Huszár, J. Totz, A. P. Aitken *et al.*, "Real-time single image and video super-resolution using an efficient sub-pixel convolutional neural network", in *Proceedings of the IEEE conference on computer vision and pattern recognition*, Jun. 2016, pp. 1874–1883.

[136] B. Xu, H. Lu, Y. Chen, A. Mathews, and Y. Chen, Terry Zhang, "Faster, more flexible inference on GPUs using AITemplate, a revolutionary new inference engine", Meta AI blog, Oct. 2022, [Accessed: 2023-09-20]. [Online]. Available: https://ai.meta.com/blog/gpu-inference-engine-nvidia-amd-open-source/

[137] "100GbE Trends in Data Center Network Development", FS.com community, Dec. 2022, [Accessed: 2023-10-20]. [Online]. Available: https://community.fs.com/article/100gbe-trends-in-data-center-network-development.html

[138] D. D. Sharma, "Invited: Compute Express Link™ (CXL™): An open interconnect for cloud infrastructure", *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–4, 2023.

[139] M. Arif, A. Maurya, and M. M. Rafique, "Accelerating performance of GPU-Based workloads using CXL", in *Proceedings of the 13th Workshop on AI and Scientific Computing at Scale Using Flexible Computing*. Association for Computing Machinery, 2023, p. 27–31. [Online]. Available: https://doi.org/10.1145/3589013.3596678

[140] C. Edwards, "NVIDIA Opens NVLink for Custom Silicon Integration", NVIDIA Newsroom, Mar. 2022, [Accessed: 2023-10-25]. [Online]. Available: https://nvidianews.nvidia.com/news/nvidia-opens-nvlink-for-custom-silicon-integration

[141] F. Douglis, S. Robertson, E. v. d. Berg, J. Micallef, M. Pucci *et al.*, "FLEET—fast lanes for expedited execution at 10 terabits: Program

overview", *IEEE Internet Computing*, vol. 25, no. 3, pp. 79–87, May 2021.

[142] NVIDIA Corporation, "NVSHMEM", 2023. [Online]. Available: https://developer.nvidia.com/nvshmem

[143] M. Corporation, "Microsoft collective communication library", 2022. [Online]. Available: https://github.com/microsoft/msccl

[144] Z. Cai, Z. Liu, S. Maleki, M. Musuvathi, T. Mytkowicz *et al.*, "Synthesizing optimal collective algorithms", in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. Association for Computing Machinery, Feb 2021. [Online]. Available: https://doi.org/10.1145/3437801.3441620

[145] A. Jangda, J. Huang, G. Liu, A. H. N. Sabet, S. Maleki *et al.*, "Breaking the computation and communication abstraction barrier in distributed machine learning workloads", *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, May 2021.

[146] F. Daoud, A. Watad, and M. Silberstein, "GPUrdma: GPU-side library for high performance networking from GPU kernels", in *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers*. Association for Computing Machinery, Jun. 2016, pp. 1–8. [Online]. Available: https://doi.org/10.1145/2931088.2931091

[147] H. Eran, L. Zeno, M. Tork, G. Malka, and M. Silberstein, "NICA: An infrastructure for inline acceleration of network applications", in *USENIX Annual Technical Conference*, 2019, pp. 345–362. [Online]. Available: https://www.usenix.org/system/files/atc19-eran.pdf

[148] M. Silberstein, S. Kim, S. Huh, X. Zhang, Y. Hu *et al.*, "GPUnet: Networking abstractions for gpu programs", *ACM Transactions on Computer Systems (TOCS)*, vol. 34, no. 3, pp. 1–31, 2016.

[149] E. Agostini, "l2fwd-nv github repository", [Accessed: 2023-12-06]. [Online]. Available: https://github.com/NVIDIA/l2fwd-nv

[150] E. Agostini, "Accelerate dpdk packet processing using gpu", NVIDIA GTC Digital April, Apr 2021, [Accessed: 2023-12-06]. [Online]. Available: https://www.nvidia.com/en-us/on-demand/session/gtcspring21-s31972/

[151] E. Agostini, "Boosting inline packet processing using DPDK and GPUdev with GPUs", NVIDIA Developer Technical Blog posting, Apr. 2022.

[152] E. Agostini, "Boosting inline packet processing using dpdk and gpudev with gpus", NVIDIA Technical Blog, Aug 2022, [Accessed: 2023-12-06]. [Online]. Available: https://developer.nvidia.com/blog/optimizing-inline-packet-processing-using-dpdk-and-gpudev-with-gpus/

[153] A. Kelkar and C. Dick, "Nvidia aerial gpu hosted ai-on-5g", in *2021 IEEE 4th 5G World Forum (5GWF)*, 2021, pp. 64–69.

[154] C. Jung, S. Kim, I. Yeom, H. Woo, and Y. Kim, "Gpu-ether: Gpu-native packet i/o for gpu applications on commodity ethernet", in *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*, 2021, pp. 1–10.

[155] J. Tseng, R. Wang, J. Tsai, S. Edupuganti, A. W. Min *et al.*, "Exploiting integrated gpus for network packet processing workloads", in *2016 IEEE NetSoft Conference and Workshops (NetSoft)*, 2016, pp. 161–165.

[156] J. Tseng, R. Wang, J. Tsai, Y. Wang, and T.-Y. C. Tai, "Accelerating open vswitch with integrated gpu", in *Proceedings of the Workshop on Kernel-Bypass Networks*. Association for Computing Machinery, 2017, p. 7–12. [Online]. Available: https://doi.org/10.1145/3098583.3098585

[157] M. Wu, Q. Chen, and J. Wang, "Bpcm: A flexible high-speed bypass parallel communication mechanism for gpu cluster", *IEEE Access*, vol. 8, pp. 103 256–103 272, 2020.

[158] M. Furukawa, T. Itsubo, and H. Matsutani, "An in-network parameter aggregation using dpdk for multi-gpu deep learning", in *2020 Eighth International Symposium on Computing and Networking (CANDAR)*, 2020, pp. 108–114.

[159] S. Han, K. Jang, K. Park, and S. Moon, "Packetshader: A gpu-accelerated software router", in *Proceedings of the ACM SIGCOMM 2010 Conference*. Association for Computing Machinery, 2010, p. 195–206. [Online]. Available: https://doi.org/10.1145/1851182.1851207

[160] K. Jang, S. Han, S. Han, S. Moon, and K. Park, "Sslshader: Cheap ssl acceleration with commodity processors", in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*. USENIX Association, 2011, p. 1–14.

[161] M. A. Jamshed, J. Lee, S. chul Moon, I. Yun, D. H. Kim *et al.*, "Kargus: a highly-scalable software-based intrusion detection system", *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012.

[162] G. Vasiliadis, L. Koromilas, M. Polychronakis, and S. Ioannidis, "Gaspp: A gpu-accelerated stateful packet processing framework", in *USENIX Annual Technical Conference*, 2014.

[163] J. Plante, D. Gratadour, L. Matias, C. Viou, and E. Agostini, "A high-performance data acquisition on COTS hardware for astronomical instrumentation", in *Proceedings Volume 12189, Software and Cyberinfrastructure for Astronomy VII*, 2022.

[164] L. Thostrup, G. Doci, N. Boeschen, M. Luthra, and C. Binnig, "Distributed GPU joins on fast RDMA-capable networks", *Proceedings of the ACM on Management of Data*, vol. 1, no. 1, pp. 1–26, 2023.

[165] G. P. Katsikas, T. Barbette, M. Chiesa, D. Kostić, and G. Q. Maguire, "What You Need to Know About (Smart) Network Interface Cards", in

Passive and Active Measurement, O. Hohlfeld, A. Lutu, and D. Levin, Eds. Springer International Publishing, 2021, vol. 12671, pp. 319–336, series Title: Lecture Notes in Computer Science. [Online]. Available: https://link.springer.com/10.1007/978-3-030-72582-2_19

[166] H. Ghasemirahni, T. Barbette, G. P. Katsikas, A. Farshin, A. Roozbeh *et al.*, "Packet order matters! improving application performance by deliberately delaying packets", in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Apr. 2022, pp. 807–827. [Online]. Available: https://www.usenix.org/conference/nsdi22/presentation/ghasemirahni

[167] T. Barbette, C. Soldani, and L. Mathy, "Fast userspace packet processing", in *ACM ANCS 2015*, May 2015.

[168] P. Kennedy, "CPU-GPU-NIC PCIe Card Realized with NVIDIA BlueField-2 A100", Jul. 2021. [Online]. Available: https://www.servethehome.com/cpu-gpu-nic-pcie-card-realized-with-nvidia-bluefield-2-a100/

[169] E. Agostini, M. Penn, D. Eustice, and I. Geffen, "Realizing the Power of Real-Time Network Processing with NVIDIA DOCA GPUNetIO | NVIDIA Technical Blog", Jul. 2023, [Accessed: 25-08-2023]. [Online]. Available: https://developer.nvidia.com/blog/realizing-the-power-of-real-time-network-processing-with-nvidia-doca-gpunetio/

[170] NVIDIA Corporation, "GPU Packet Processing - NVIDIA Docs". [Online]. Available: https://docs.nvidia.com/sdk-v2.2.0/gpu-packet-processing/index.html

[171] NVIDIA Corporation, "GPU Packet Processing :: NVIDIA DOCA SDK Documentation", Aug. 2023. [Online]. Available: https://docs.nvidia.com/doca/sdk/gpu-packet-processing/index.html

[172] M. LeBeane, K. Hamidouche, B. Benton, M. Breternitz, S. K. Reinhardt *et al.*, "Gpu triggered networking for intra-kernel communications", in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17. Association for Computing Machinery, Nov 2017.

[173] E. Agostini, D. Rossetti, and S. Potluri, "Offloading communication control logic in gpu accelerated applications", in *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, Jul 2017, pp. 248–257.

[174] W. A. Hanafy, L. Wang, H. Chang, S. Mukherjee, T. Lakshman *et al.*, "Understanding the benefits of hardware-accelerated communication in model-serving applications", *arXiv preprint arXiv:2305.03165*, 2023.

[175] H. Zhang, Y. Tang, A. Khandelwal, and I. Stoica, "SHEPHERD: Serving DNNs in the wild", in *20th USENIX Symposium on Networked Systems*

Design and Implementation (NSDI 23), 2023, pp. 787–808. [Online]. Available: https://www.usenix.org/system/files/nsdi23-zhang-hong.pdf

[176] F. Romero, Q. Li, N. J. Yadwadkar, and C. Kozyrakis, "INFaaS: Automated model-less inference serving", in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, Jul. 2021, pp. 397–411. [Online]. Available: https://www.usenix.org/conference/atc21/presentation/romero

[177] H. Shen, L. Chen, Y. Jin, L. Zhao, B. Kong *et al.*, "Nexus: A GPU cluster engine for accelerating DNN-based video analysis", in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. Association for Computing Machinery, 2019, p. 322–337. [Online]. Available: https://doi.org/10.1145/3341301.3359658

[178] "NVIDIA Triton Inference Server", [Accessed: 2024-04-11]. [Online]. Available: https://www.nvidia.com/en-us/ai-data-science/products/triton-inference-server/

[179] G.-I. Yu, J. S. Jeong, G.-W. Kim, S. Kim, and B.-G. Chun, "Orca: A distributed serving system for {Transformer-Based} generative models", in *USENIX OSDI 2022*, 2022, pp. 521–538.

[180] M. Jasny, L. Thostrup, and C. Binnig, "Zero-Sided RDMA: Network-driven data shuffling", in *Proceedings of the 19th International Workshop on Data Management on New Hardware*. Association for Computing Machinery, 2023, p. 82–85. [Online]. Available: https://doi.org/10.1145/3592980.3595302

[181] B. Jeon, S. Park, P. Liao, S. Xu, T. Chen *et al.*, "Collage: Seamless integration of deep learning backends with automatic placement", in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. Association for Computing Machinery, 2023, p. 517–529. [Online]. Available: https://doi.org/10.1145/3559009.3569651

[182] M. Li, Y. Liu, X. Liu, Q. Sun, X. You *et al.*, "The deep learning compiler: A comprehensive survey", *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, pp. 708–727, 2020.

[183] G. E. Moore, "Cramming more components onto integrated circuits", *Proceedings of the IEEE*, vol. 86, no. 1, pp. 82–85, Jan. 1998.

[184] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan, "Attack of the killer microseconds", *Communications of the ACM*, vol. 60, no. 4, pp. 48–54, 2017.

[185] I. Tuomi, "The lives and death of moore's law", *First Monday*, vol. 7, no. 11, Nov. 2002. [Online]. Available: https://firstmonday.org/ojs/index.php/fm/article/view/1000

[186] C. Mims, "Huang's law is the new moore's law, and explains why

nvidia wants arm", Sep 2020, [Accessed: 2024-01-19]. [Online]. Available: https://www.wsj.com/articles/huangs-law-is-the-new-moores-law-and-explains-why-nvidia-wants-arm-11600488001

[187] P. Tekla, "Move over, moore's law: Make way for huang's law", Sep 2023, [Accessed: 2024-01-19]. [Online]. Available: https://spectrum.ieee.org/move-over-moores-law-make-way-for-huangs-law

[188] D. Patterson, "Latency lags bandwidth", in *2005 International Conference on Computer Design*, Oct 2005.

[189] I. Advanced Micro Devices, "Gpu-enabled mpi", ROCm Documentation, Jul. 2023. [Online]. Available: https://rocm.docs.amd.com/en/latest/how_to/gpu_aware_mpi.html#gpu-enabled-mpi

[190] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan, "Attack of the killer microseconds", *Commun. ACM*, vol. 60, no. 4, p. 48–54, mar 2017. [Online]. Available: https://doi.org/10.1145/3015146

[191] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson *et al.*, "Openflow: Enabling innovation in campus networks", *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, p. 69–74, mar 2008. [Online]. Available: https://doi.org/10.1145/1355734.1355746

[192] "ryu repository on GitHub", [Accessed: 2023-09-07]. [Online]. Available: https://github.com/faucetsdn/ryu.git

[193] M. Girondi, "Network performance framework repository (fork)", GitHub, Oct. 2023, [Accessed: 2023-10-17]. [Online]. Available: https://github.com/MassimoGirondi/npf

[194] "Source code for "High-speed Connection Tracking in Modern Servers"", GitHub, 2021, [Accessed: 2023-12-01]. [Online]. Available: https://github.com/conntrackHPSR21/

[195] P. Kennedy, "NVIDIA Cedar Fever 1.6Tbps Modules Used in the DGX H100", Apr. 2022. [Online]. Available: https://www.servethehome.com/nvidia-cedar-fever-1-6tbps-modules-used-in-the-dgx-h100/

[196] A. Ishii and R. Wells, "The NVLink-network switch: NVIDIA's switch chip for high communication-bandwidth superpods", in *2022 IEEE Hot Chips 34 Symposium (HCS)*, Aug 2022, pp. 1–23, video at https://www.youtube.com/watch?v=S117CO2KL-0.

[197] A. Jaakkola, "Implementation of Transmission Control Protocol in Linux", in *Proceedings of Seminar on Network Protocols in Operating Systems*.

[198] Facebook. (2020) eBPF hashmap implementation in the Linux kernel. [Online]. Available: https://git.io/Jt0mF

[199] Facebook Incubator. Katran GitHub Repository. https://github.com/facebookincubator/katran.

[200] D. Zhou, B. Fan, H. Lim, M. Kaminsky, and D. G. Andersen, "Scalable, high

performance ethernet forwarding with cuckooswitch", in *ACM CoNEXT 2013*, Dec 2013.

[201] F. André, S. Gouache, N. Le Scouarnec, and A. Monsifrot, "Don't share, don't lock: large-scale software connection tracking with krononat", in *USENIX ATC 2018*, Jul 2018.

[202] B. Fan, D. G. Andersen, and M. Kaminsky, "MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing", in *USENIX NSDI 2013*, Sep 2013.

[203] R. Pagh and F. F. Rodler, "Cuckoo hashing", *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.

[204] Intel Barefoot Networks. (2020) Tofino-2 second-generation of world's fastest p4-programmable ethernet switch. [Online]. Available: https://www.barefootnetworks.com/products/brief-tofino-2/

[205] NVIDIA Mellanox. (2020) ConnectX®-6 DX EN IC 200GbE ethernet adapter ic. [Online]. Available: https://www.mellanox.com/files/doc-2020/pb-connectx-6-dx-en-card.pdf

[206] Z. Yao, J. Bagga, and H. Morsy. (2016) Introducing backpack: Our second-generation modular open switch. [Online]. Available: https://engineering.fb.com/data-center-engineering/introducing-backpack-our-second-generation-modular-open-switch/

[207] G. P. Katsikas, T. Barbette, D. Kostic, R. Steinert, and G. Q. Maguire Jr, "Metron:NFV service chains at the true speed of the underlying hardware", in *USENIX NSDI 2018*, Apr 2018.

[208] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda *et al.*, "Clickos and the art of network function virtualization", in *USENIX NSDI 2014*, Apr. 2014.

[209] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda *et al.*, "E2: A framework for nfv applications", in *ACM SOSP 2015*, Oct. 2015.

[210] A. Bremler-Barr, Y. Harchol, and D. Hay, "Openbox: a software-defined framework for developing, deploying, and managing network functions", in *ACM SIGCOMM 2016*, Aug. 2016.

[211] L. Molnár, G. Pongrácz, G. Enyedi, Z. L. Kis, L. Csikor *et al.*, "Dataplane specialization for high-performance openflow software switching", in *ACM SIGCOMM 2016*, Aug. 2016.

[212] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov *et al.*, "Maglev: A Fast and Reliable Software Network Load Balancer", in *USENIX NSDI 2016*, Mar. 2016.

[213] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics", in *ACM SIGCOMM 2017*, Aug. 2017.

[214] N. Le Scouarnec, "Cuckoo++ hash tables: High-performance hash tables for networking applications", in *ACM ANCS 2018*, July 2018.

[215] T. Barbette, G. P. Katsikas, G. Q. Maguire, and D. Kostić, "Rss++: Load and state-aware receive side scaling", in *ACM CoNEXT 2019*, Dec. 2019.

[216] D. Didona and W. Zwaenepoel, "Size-aware sharding for improving tail latencies in in-memory key-value stores", in *USENIX NSDI 2019*, Feb. 2019.

[217] S. Han, S. Marshall, B.-G. Chun, and S. Ratnasamy, "Megapipe: a new programming interface for scalable network i/o", in *USENIX OSDI 2012*, Oct 2012.

[218] Y. Yu, D. Belazzougui, C. Qian, and Q. Zhang, "Memory-Efficient and Ultra-Fast Network Lookup and Forwarding Using Othello Hashing", *IEEE/ACM Transactions on Networking*, vol. 26, no. 3, pp. 1151–1164, 2018.

[219] S. Shi and C. Qian, "Ludo hashing: Compact, fast, and dynamic key-value lookups for practical network systems", *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2020.

[220] Z. Chen, X. He, J. Sun, H. Chen, and L. He, "Concurrent hash tables on multicore machines: Comparison, evaluation and implications", *Future Generation Computer Systems*, vol. 82, pp. 127 – 141, 2018.

[221] A. Farshin, A. Roozbeh, G. Q. Maguire, and D. Kostić, "Make the Most out of Last Level Cache in Intel Processors", in *ACM EuroSys 2019*, Feb. 2019.

[222] P. Gupta and N. McKeown, "Algorithms for packet classification", *IEEE Network*, vol. 15, no. 2, pp. 24–32, Mar. 2001.

[223] S. S. e. Dinesh P. Mehta, *Handbook of data structures and applications*. Chapman & Hall/CRC, 2004.

[224] X. Li, D. G. Andersen, M. Kaminsky, and M. J. Freedman, "Algorithmic improvements for fast concurrent cuckoo hashing", in *ACM EuroSys 2014*, Apr. 2014.

[225] DPDK Project. (2020) Dpdk website. [Online]. Available: https://dpdk.org

[226] DPDK Project. (2020) Hash library. [Online]. Available: https://doc.dpdk.org/guides/prog_guide/hash_lib.html

[227] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors", *Commun. ACM*, vol. 13, no. 7, p. 422–426, Jul. 1970.

[228] M. Herlihy, N. Shavit, and M. Tzafrir, "Hopscotch hashing", in *DISC 2008*, Apr. 2008.

[229] P. Celis, P.-A. Larson, and J. I. Munro, "Robin hood hashing", in *SFCS 1985*, Oct. 1985.

[230] M. Herlihy, N. Shavit, V. Luchangco, and M. Spear, *The Art of Multiprocessor Programming (Second Edition)*, 2nd ed. Morgan Kaufmann, Oct. 2020.

[231] M. P. Herlihy, "Impossibility and universality results for wait-free synchronization", in *ACM PODC 1988*, 1988.

[232] Intel®. (2016) Improving network performance in multi-core systems. [Online]. Available: https://www.intel.com/content/dam/support/us/en/documents/network/sb/318483001us2.pdf

[233] G. Varghese and A. Lauck, "Hashed and hierarchical timing wheels: efficient data structures for implementing a timer facility", *IEEE/ACM transactions on networking*, vol. 5, no. 6, pp. 824–834, 1997.

[234] J. Corbet. (2015, Jun.) Reinventing the timer wheel. [Online]. Available: https://lwn.net/Articles/646950/

[235] J. Corbet. (2016, Jun.) timer: Refactor the timer wheel. [Online]. Available: https://lwn.net/Articles/691064/

[236] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router", *ACM Transactions on Computer Systems (TOCS)*, vol. 18, no. 3, pp. 263–297, Aug. 2000.

[237] Hopscotch-map GitHub repository. [Online]. Available: https://github.com/Tessil/hopscotch-map/

[238] robinmap-map GitHub repository. [Online]. Available: https://github.com/Tessil/robin-map/

[239] *ISO/IEC 14882:2017 Information technology - Programming languages - C++*, ISO Std., 2017. [Online]. Available: https://www.iso.org/standard/68564.html

[240] DPDK Project. (2020) Reader-writer lock library documentation. [Online]. Available: https://doc.dpdk.org/api/rte__rwlock_8h.html

[241] H. Nagarahalli. Patch introducing address reader-writer concurrency in rte_hash. [Online]. Available: https://mails.dpdk.org/archives/dev/2018-September/111016.html

[242] H. Nagarahalli. Lock Free RW Concurrency in hash library. [Online]. Available: https://www.dpdk.org/wp-content/uploads/sites/35/2018/10/am-04-lock_free_rte_hash_Honnappa.pdf

[243] A. Kirsch, M. Mitzenmacher, and U. Wieder, "More robust hashing: Cuckoo hashing with a stash", *SIAM Journal on Computing*, vol. 39, no. 4, p. 1543–1561, 2009.

[244] T. David, R. Guerraoui, and V. Trigonakis, "Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures", in *ACM ASPLOS 2015*, Mar. 2015.

[245] S. Shi, Y. Yu, M. Xie, X. Li, X. Li *et al.*, "Concury: a fast and light-weight software cloud load balancer", in *SoCC 2020*, Oct. 2020.

[246] Y. Yu, D. Belazzougui, C. Qian, and Q. Zhang, "A concise forwarding information base for scalable and fast name lookups", in *IEEE ICNP 2017*,

2017.

[247] R. Gandhi, Y. C. Hu, C.-K. Koh, and M. Zhang, "Rubik: Unlocking the Power of Locality and End-Point Flexibility in Cloud Scale Load Balancing", in *USENIX ATC 2015*, Jul. 2015.

[248] R. Gandhi, H. H. Liu, Y. C. Hu, G. Lu, J. Padhye *et al.*, "Duet: Cloud scale load balancing with hardware and software", in *SIGCOMM 2014*, Aug. 2014.

[249] T. Barbette, C. Tang, H. Yao, D. Kostić, G. Q. Maguire Jr *et al.*, "A high-speed load-balancer design with guaranteed per-connection-consistency", in *USENIX NSDI 2020*, Feb. 2020, pp. 667–683.

[250] F. Baboescu, Sumeet Singh, and G. Varghese, "Packet classification for core routers: Is there an alternative to CAMs?" in *IEEE INFOCOM 2003*, Apr. 2003.

[251] Netronome. (2019) Agilio fx smartnic. [Online]. Available: https://www.netronome.com/products/agilio-fx/

[252] Netronome. (2019) Agilio cx smartnic. [Online]. Available: https://www.netronome.com/products/agilio-cx/

[253] NVIDIA Mellanox. (2020) Bluefield 2 SmartNIC. [Online]. Available: https://www.mellanox.com/files/doc-2020/pb-bluefield-2-smart-nic-eth.pdf

[254] S. Pontarelli, R. Bifulco, M. Bonola, C. Cascone, M. Spaziani *et al.*, "FlowBlaze: Stateful Packet Processing in Hardware", in *USENIX NSDI 2019*, Feb. 2019.

[255] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh *et al.*, "Azure Accelerated Networking: SmartNICs in the Public Cloud", in *USENIX NSDI 2018*, Apr. 2018.

[256] M. Boye. Netfilter connection tracking and nat implementation. [Online]. Available: https://wiki.aalto.fi/download/attachments/69901948/netfilter-paper.pdf

[257] A. Chiao. Connection tracking (conntrack): Design and implementation inside linux kernel. [Online]. Available: https://arthurchiao.art/blog/conntrack-design-and-implementation/#connection-tracking-conntrack

[258] How long does conntrack remember a connection? [Online]. Available: https://unix.stackexchange.com/a/524320

[259] F. Bonomi, M. Mitzenmacher, and R. Panigrahy, "Beyond Bloom Filters: From Approximate Membership Checks to Approximate State Machines", in *ACM SIGCOMM 2016*, Aug. 2016.

[260] B. Whitehead, C.-H. Lung, and P. Rabinovitch, "Tracking per-flow state—binned duration flow tracking", in *IEEE SPECTS 2020*, Jul. 2020.

[261] T. Barbette, C. Soldani, R. Gaillard, and L. Mathy, "Building a chain of

high-speed VNFs in no time", in *IEEE HPSR 2018*, Jun. 2018.

[262] "RDMA core userspace libraries and daemons - contributors", GitHub, [Accessed: 2023-09-20]. [Online]. Available: https://github.com/linux-rdma/rdma-core/graphs/contributors

[263] P. MacArthur, Q. Liu, R. D. Russell, F. Mizero, M. Veeraraghavan *et al.*, "An integrated tutorial on InfiniBand, Verbs, and MPI", *IEEE Communications Surveys & Tutorials*, vol. 19, no. 4, pp. 2894–2926, Fourthquarter 2017.

[264] P. MacArthur, "Userspace RDMA Verbs on commodity hardware using DPDK", in *2017 IEEE 25th Annual Symposium on High-Performance Interconnects (HOTI)*, Aug. 2017, pp. 103–110.

[265] G. Kerr, "Dissecting a small InfiniBand application using the verbs API", *ArXiv*, vol. abs/1105.1827, 2011.

[266] W. Reda, M. Canini, D. Kostić, and S. Peter, "RDMA is Turing complete, we just did not know it yet!" in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Apr. 2022, pp. 71–85. [Online]. Available: https://www.usenix.org/conference/nsdi22/presentation/reda

[267] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson, "FaRM: Fast remote memory", in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Apr. 2014, pp. 401–414. [Online]. Available: https://www.usenix.org/conference/nsdi14/technical-sessions/dragojevi{ć}

[268] C. Mitchell, Y. Geng, and J. Li, "Using One-Sided RDMA reads to build a fast, CPU-Efficient Key-Value store", in *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. USENIX Association, Jun. 2013, pp. 103–114. [Online]. Available: https://www.usenix.org/conference/atc13/technical-sessions/presentation/mitchell

[269] Mellanox Technologies, "Mellanox adapters programmer's reference manual (PRM)", 2016, revision 0.40, Document Number: MLNX-15-4845. [Online]. Available: https://network.nvidia.com/sites/default/files/doc-2020/ethernet-adapters-programming-manual.pdf

[270] E. Cohen, "net/mlx5: Introduce blue flame register allocator", Patchwork - Linux RDMA and InfiniBand, Jan. 2017. [Online]. Available: https://patchwork.kernel.org/project/linux-rdma/patch/1483480528-22622-6-git-send-email-saeedm@mellanox.com/

[271] B. Rothenberger, K. Taranov, A. Perrig, and T. Hoefler, "{ReDMArk}: Bypassing {RDMA} security mechanisms", in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 4277–4292.

[272] J. Xing, K.-F. Hsu, Y. Qiu, Z. Yang, H. Liu *et al.*, "Bedrock: Programmable network support for secure rdma systems", in *USENIX Security Symposium*,

2022.

[273] K. Taranov, B. Rothenberger, A. Perrig, and T. Hoefler, "srdma - efficient nic-based authentication and encryption for remote direct memory access", in *USENIX Annual Technical Conference*, 2020.

[274] S.-Y. Tsai and Y. Zhang, "A double-edged sword: Security threats and opportunities in one-sided network communication", *ArXiv*, vol. abs/1903.09355, 2019.

[275] Mellanox Technologies, "nv_peer_memory repository on github", [Accessed: 2023-09-06]. [Online]. Available: https://github.com/Mellanox/nv_peer_memory

[276] D. A. Wheeler, "Sloccount", [Accessed: 2023-12-07]. [Online]. Available: https://dwheeler.com/sloccount/

[277] NVIDIA Corporation, "NVSHMEM source code", NVIDIA developer account needed to access the archive, NVSHMEM is also included in the NVIDIA HPC SDK. [Online]. Available: https://developer.nvidia.com/nvshmem-downloads

[278] "Verb posting routine for mlx5 in UCX codebase", GitHub. [Online]. Available: https://github.com/openucx/ucx/blob/85e52394050639196c78462138178ab5ff5d231a/src/uct/ib/rc/accel/rc_mlx5.inl#L427

[279] NVIDIA Corporation, "Condential computing deployment guide", Aug 2023, [Accessed: 2024-01-12]. [Online]. Available: https://docs.nvidia.com/confidential-computing-deployment-guide.pdf

[280] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown *et al.*, "P4: Programming protocol-independent packet processors", *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, p. 87–95, Jul. 2014. [Online]. Available: https://doi.org/10.1145/2656877.2656890

[281] K. Varda, "Protocol Buffers: Google's data interchange format", Google, Tech. Rep., Jun. 2008. [Online]. Available: http://google-opensource.blogspot.com/2008/07/protocol-buffers-googles-data.html

[282] X. Zhang, X. Zhou, M. Lin, and J. Sun, "ShuffleNet: An extremely efficient convolutional neural network for mobile devices", arXiv, 2017.

[283] "grpc", [Accessed: 2023-11-07]. [Online]. Available: https://grpc.io/

[284] D. Chefrour, "Evolution of network time synchronization towards nanoseconds accuracy: A survey", *Comput. Commun.*, vol. 191, pp. 26–35, 2022.

[285] T. Ahmed, S. Rahman, M. Tornatore, K. Kim, and B. Mukherjee, "A survey on high-precision time synchronization techniques for optical datacenter networks and a zero-overhead microsecond-accuracy solution", *Photonic Network Communications*, vol. 36, pp. 56 – 67, 2018.

[286] V. Shrivastav, K.-S. Lee, H. Wang, and H. Weatherspoon, "Globally

Synchronized Time via Datacenter Networks", *IEEE/ACM Transactions on Networking*, vol. 27, pp. 1401–1416, 2019.

[287] A. Byagowi and O. Obleukhov, "Ptp: Timing accuracy and precision for the future of computing", Engineering at Meta, Nov. 2021, [Accessed: 2024-02-27]. [Online]. Available: https://engineering.fb.com/2022/11/21/production-engineering/future-computing-ptp/

[288] "Reinventing the network stack for compute-intensive applications", Defense Advanced Research Projects Agency News, Sep. 2019, [Accessed: 2023-11-09]. [Online]. Available: https://www.darpa.mil/news-events/2019-09-26

[289] X. Miao, G. Oliaro, Z. Zhang, X. Cheng, Z. Wang *et al.*, "Specinfer: Accelerating generative large language model serving with speculative inference and token tree verification", 2023.

[290] P. H. Salus, *A quarter century of UNIX*. ACM Press/Addison-Wesley Publishing Co., 1994.

[291] E. S. Raymond, *The Art of Unix Programming*. Addison-Wesley Professional, Sep. 2003.

[292] M. Tork, L. Maudlej, and M. Silberstein, "Lynx: A smartnic-driven accelerator-centric architecture for network servers", in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, 2020, p. 117–131. [Online]. Available: https://doi.org/10.1145/3373376.3378528

[293] NVIDIA Corporation, "Converged accelerators: Real time ai processing", [Accessed: 2023-11-14]. [Online]. Available: https://www.nvidia.com/en-us/data-center/products/converged-accelerator/

[294] E. Services, "Eurovision services successfully tests full cloud production solution", Mar 2021. [Online]. Available: https://www.eurovision.net/insights/technical/eurovision-services-successfully-tests-full-cloud-production-solution

[295] C. Dickson, "The technology behind a low latency cloud gaming service", [Accessed: 2023-11-09]. [Online]. Available: https://parsec.app/blog/description-of-parsec-technology-b2738dcc3842

For DIVA


```
{
"Author1": {
                "Last name": "Girondi",
                "First name": "Massimo",
                "Local User Id": "u1l0lizh",
                "E-mail": "girondi@kth.se",
                "ORCiD": "0000-0002-9400-324X",
                "organisation": {"L1": "School of Electrical Engineering and Computer Science",
                                 "L2": "Computer Science"}
            },
"Degree": {"Educational program": "Information and Communication Technology"},
"Title": {
                "Main title": "Toward Highly-efficient GPU-centric Networking",
                "Language": "eng"},
"Alternative title": {
                "Main title": "Mot Högeffektiva GPU-centrerade Nätverk",
                "Language": "swe"
},
"Other information": {
                "Year": "2024",
                "Number of pages": "xix,161"},
"Supervisor1": {
                        "Last name": "Kostić",
                        "First name": "Dejan",
                        "Local User Id": "u12eursm",
                        "E-mail": "dmk@kth.se",
                        "organisation": {"L1": "EECS",
                                         "L2": "Computer Science"}
                    },
"Supervisor2": {
                        "Last name": "Chiesa",
                        "First name": "Marco",
                        "Local User Id: "u18vjbx4",
                        "E-mail": "mchiesa@kth.se",
                        "organisation": {"L1": "EECS",
                                         "L2": "Computer Science"}
                    },
"Opponent": {
                        "Last name": "Siracusano",
                        "First name": "Giuseppe",
                        "E-mail": "giuseppe.siracusano@neclab.eu",
                        "Other organisation": "NEC Laboratories Europe"
                    },
"Presentation": {
                        "Date": "2024-04-10 09:00"
                        ,"Language": "eng"
                        ,"Room": "via Zoom and Sal C (Sven-Olof Öhrvik) at Electrum, Kungliga Tekniska Högskolan"
                        ,"Address": "Kistagången 16"
                        ,"City": "Kista"
                    },
"National Subject Categories": "20203, 20206",
}
```