DISS. ETH NO. 29551

FPGA-Based Systems for Stream Data Analytics and I/O Data Transformations

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES

(Dr. sc. ETH Zurich)

presented by

MONICA CHIOSA

MSc in Electrical and Electronic Engineering, EPFL

born on 05.09.1991

accepted on the recommendation of

Prof. Dr. Gustavo Alonso

Prof. Dr. Ce Zhang

Prof. Dr. Lana Josipović

Dr. Thomas Preußer

2023

IKS-Lab
Institute for Computing Platforms
ETH Department of Computer Science

Monica Chiosa

A dissertation submitted to
ETH Zurich
for the degree of Doctor of Sciences

DISS. ETH NO. 29551

examiner:
Prof. Dr. Gustavo Alonso
co-examiners:
Prof. Dr. Ce Zhang
Prof. Dr. Lana Josipović
Dr. Thomas Preußer

Examination date: September 12, 2023

FPGA-Based Systems for Stream Data Analytics and I/O Data Transformations

Abstract

The distributed nature of the cloud, regarding resource placement and application execution, incurs large data transfer. As data movement is unavoidable, it lies within the systems' capacities to ensure its effectiveness. This thesis takes a step towards enhancing data movement with compute capabilities, more precisely with data analytics and privacy computational features, with the ultimate goal of making data movement more efficient and secure using Field Programmable Gate Arrays (FPGAs) based compute engines.

From the data analytics perspective, a correlation coefficient compute engine is designed and analyzed. This selection is based on the widespread adoption of correlation across database and ML systems that require knowledge of the correlation relationship between the input data before processing it. The experimental evaluation shows that correlation computation offloaded to an FPGA-based smartNIC can sustain streams arriving at 100 Gbps over an RDMA network, while performing a single pass over the data and requiring an order of magnitude less time for computing compared to CPU or GPU designs.

Maintaining the continuity in the data characterization approach, a streaming analytics compute engine is analyzed. The system objective is to sustain a 100 Gbps TCP/IP data rate while characterizing input data (cardinality, second frequency moment, frequency distribution) in a manageable space and with a single pass over the data. In addition to accomplishing its intended objective, the system deployment on a single FPGA can match the performance of 70 CPU cores.

From the data privacy perspective, the performance benefits of offloading the I/O path data transformation operations (encryption and decryption) are analyzed, taking into account the requirements of a relational analytics engine (SAP HANA), while providing three levels of security.

The findings indicate that substantial benefits can be achieved, both in terms of performance and freeing CPU resources, in the cloud environment, and protecting the data while it is at rest or transmitted over the network.

Keywords: FPGA, correlation computation, sketch algorithms, encryption and decryption, data movement, data characterization, data transformation, Pearson correlation coefficient, HyperLogLog, Count-Min, Fast-AGMS, AES, Remote Direct Memory Access (RDMA), TCP/IP

RÉSUMÉ

La nature distribuée de l'informatique en nuage (cloud computing), en ce qui concerne l'allocation des ressources et l'exécution des applications, implique un important transfert de données. Le mouvement des données étant inévitable, il appartient aux systèmes de s'assurer que cela se fasse de manière efficace. Cette thèse contribue à améliorer le mouvement des données grâce aux capacités de calcul, plus précisément avec des fonctions d'analyse et protection des données. L'objectif est de rendre le mouvement des données plus efficace et plus sûr à l'aide de moteurs de calcul basés sur des circuits intégrés programmables (Field Programmable Gate Arrays, FPGA).

Du point de vue de l'analyse des données, nous concevons et analysons un moteur de calcul du coefficient de corrélation. Ce choix est basé sur une grande adoption de la corrélation dans les bases de données et les systèmes d'apprentissage automatique, qui nécessitent la connaissance des relations de corrélation qui existent entre les données entrantes avant de les traiter. L'évaluation expérimentale montre que, si l'on confie le calcul de corrélations à un smartNIC basé sur un FPGA, il est possible d'atteindre des flux arrivant à 100 Gbps sur un réseau RDMA. Ces performances sont obtenues en un seul scan des données et nécessitent un temps de calcul inférieur d'un ordre de grandeur à celui des systèmes CPU ou GPU.

Afin de maintenir la continuité de l'approche de caractérisation des données, un moteur de calcul analytique en continu est analysé. L'objectif du système est de maintenir un débit de données TCP/IP de 100 Gbps tout en caractérisant les données d'entrée (cardinalité, second moment de fréquence, distribution de fréquence) dans un espace gérable et en un seul passage sur les données. En plus d'atteindre l'objectif visé, le déploiement du système sur un seul FPGA peut égaler les performances d'un processeur de 70 cœurs.

Du point de vue de la confidentialité des données, nous analysons les avantages en termes de performance, de la délocalisation des opérations, de la transformation des données du chemin de lecture/écriture (cryptage et décryptage), en tenant compte des exigences d'un moteur d'analyse relationnel (SAP HANA), et en offrant trois niveaux de sécurité.

Les résultats indiquent que des avantages substantiels peuvent être obtenus dans l'environnement en nuage, à la fois en termes de performances, de la libération de ressources du processeur, et de la protection des données lorsqu'elles sont au repos ou transmises sur le réseau.

Mots clés: FPGA, calcul de corrélation, algorithmes sketch, cryptage et décryptage, mouvement de données, caractérisation des données, transformation des données, coefficient de corrélation de Pearson, HyperLogLog, Count-Min, Fast-AGMS, AES, Remote Direct Memory Access (RDMA), TCP/IP

Acknowledgments

There are many people to whom I would like to express my gratitude. If I have inadvertently omitted your name, please know that you are in my thoughts.

First and foremost, I would like to thank Gustavo Alonso for providing me with the opportunity and guidance to enhance my research skills and complete my PhD. I would like to express my gratitude to Lana Josipović and Ce Zhang for being part of my PhD committee and providing feedback on this thesis. Additionally, I want to thank Thomas Preußer for mentoring me not only during my internship at AMD but also during his time at ETH. I truly appreciated our discussions, your feedback, and the opportunity to collaborate with you.

Special appreciation is due to David Sidler for his contributions to offloading network protocols on FPGAs, enabling my research and paving the way for FPGAs in cloud computing. During my PhD I was also able to collaborate with many brilliant people Ingo Müller, Norman May, Michaela Blott, Kaan Kara and Amit Kulkarni.

The long hours at the office would not have been the same without the joyful atmosphere created by Daniel Schwyn, Fabio Maschi and Anastasiia Ruzhanskaia. Anastasiia, you had a meaningful impact on my time during my PhD, and I am grateful for our friendship. Thank you for the coffee meetings, market visits, hikes, and events - these experiences helped me maintain a sense of normalcy.

Similarly, I would like to express my gratitude to Dimitris Koutsoukos, Dario Korolija, and Zhenhao He for our walks, beers and dinner meetings. I enjoyed our discussions on different subjects. Furthermore, I want to thank to all my fellow colleagues in the Systems Group for creating an enjoyable experience as a member of the group: Dan Graur, Michal Friedman, Michal Wawrzoniak, Tom Kuchler, Michael Giardino, Abishek Ramdas, Susie Rao.

I am also grateful to my friends outside of the Systems Group: Delia, Paula, Attie, Viki, Étienne, Cristina, Bogdan, Bojan, Muriel, Eve, Corina, Camilo, Fernando, Andrei, Mădălina, Christian,

Laura, Ana, Cătălin and Patricia. I enjoyed meeting you all over the world and I appreciate your availability to meet whenever I reached out, especially after the PhD deadlines. Thank you for making the time.

My PhD life would have not been the same without the support of my partner, Igor. Thank you for your patience and encouragements, for believing in me and in my aptitudes when I was doubting them. Even if sometimes we were miles apart, we were always close at heart.

I am grateful for my sister, Diana. Thank you for being by my side, for making me laugh and for introducing me to psychology. The pandemic period was difficult for many of us, but it would have had a much higher toll on me if not for you.

Finally, I would like to express my gratitude towards my parents, Sabina and Mihai, to my grandparents, Elena and Faust, my aunt, Gabriela, and my cousin, Victor. Thank you for supporting and believing in me and my weird ideas, processes and plans. Having you as my base gave me courage to try new things and dream big.

Monica Chiosa
Zürich, 2023

In the darkest times, hope is something you give yourself.
That is the meaning of inner strength.

-Uncle Iroh, 'Avatar: The Last Airbender'

If I try, I fail. If I don't try, I'm never going to get it.

-Aang, 'Avatar: The Last Airbender'

CONTENTS

CHAPTER 1

INTRODUCTION

1.1 General Context

One would argue that nowadays we produce a greater amount of data compared to the past, but we have consistently generated it in large quantities, and data has always existed abundantly around us, ready to be collected, analyzed, and comprehended. The factors that have evolved in the past five decades are our methods and mechanisms of examining and collecting it. Starting with the introduction of SQL in 1979 and that of the personal computers in 1980, our means have increased exponentially. Today, almost anybody can collect and analyze data by using digital phones and watches, laptops, tablets, smart home devices, smart cars, and culminating with the use of the powerful tool of internet.

All of these have been possible due to a human invention, the transistor, that represents the core of digital data processing and compute devices (CPU, GPU, FPGA, ASIC). Traditionally, the CPU has been responsible of processing the data, giving rise to comprehensive areas of study on the efficiency of doing it. The end of Dennard Scaling [65], the slowdown of Moore's Law [64], and Amdahl's Law [188] have shifted the processing paradigm from general compute devices (CPU) to specialized devices (GPU, FPGA and ASIC), allowing data processing to remain a constantly advancing field. From an economic perspective, the combined utilization of general and specialized devices can lead to cost savings, better energy efficiency, less security flaws as well as higher performance [90]. Figure 1.1 illustrates a scale on how these digital

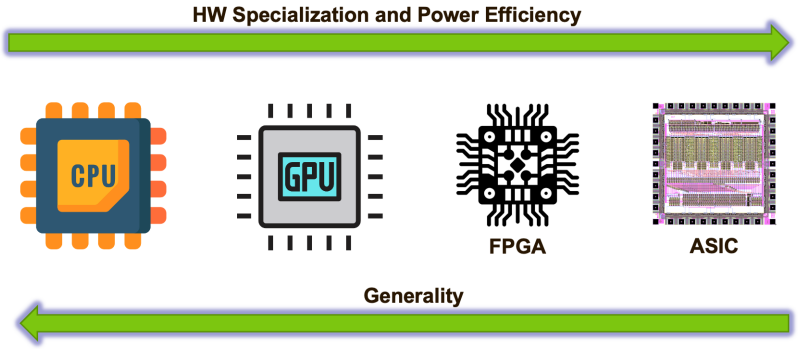


Figure 1.1: Compute generality vs hardware specialization and power efficiency.

devices align in terms of computational general purpose, hardware specialization and power efficiency, with the CPU being at the far end of the generality scale, and the ASIC positioning itself at the far end of the hardware specialization and power efficiency scale.

Field Programmable Gate Array chips, FPGA for short, fall in the category of integrated circuits that provide an intermediate point between flexibility, high-throughput, and low-latency performance. Unlike the CPU, which operates in the GHz clock frequency range, the FPGA operates at lower frequencies, in the MHz range, but with a much higher level of resource parallelism. The FPGA represents a pool of integrated hardware resources (e.g., low-latency on-chip memories, combinatorial logic blocks, digital signal processing units-DSPs) with a programmable interconnect between them. If the GPU represents a network of identical processors with access to fast memories and specialized for numerical applications, the resource heterogeneity of the FPGA enables it to be configured to arbitrary different digital circuits, from communication interfaces (e.g., PCIe, UART, I^2C, CAN, network stacks [197]) to complex analytical engines (e.g., regulator expression matching [200], query operators such as projections, selections, group-by aggregation [224, 138], compression and encryption [42, 126]) or, crucially, both at the same time. Given sufficient digital hardware resources for both communication and computation tasks, the FPGA can act as a bump-in-the-wire computational node placed between two processing units or between storage and processing units. This system integration of the FPGA shifts away from the co-processor concept that has been associated to specialized digital devices, towards a data-flow processing model, taking advantage of the FPGA's pipeline parallelism feature [212].

Simultaneously with the advent of specialized hardware, there has been a paradigm shift regarding the location for data storage and processing, moving from personal computers and business oriented data centers to cloud computing (i.e., accessing computational or storage resources

without knowing their precise location). Many companies have directed their focus towards this trend and have started offering associated services. For example, Amazon Web Services (AWS), Microsoft Azure, Google Cloud Platform (GCP), Alibaba Cloud and Oracle Cloud, are the companies whose cloud services cover more than 25 regions (i.e., physical locations) across the globe [237]. Together with the migration of traditional systems to the cloud, the providers have recognized the cost-effectiveness and convenience associated with deploying also FPGAs [112].

Cloud computing offers a collection of high-performance heterogeneous compute and storage devices that are inter-connected by high-bandwidth communication links and can be used on-demand either at the infrastructure or application level, under the label of "as-a-Service" [98]. Even if cloud computing comes with the perception of deployment being effortless by outsourcing machine maintenance and infrastructure development to other third-party providers [176, 99], it raises questions on resource orchestration [154, 135], on what is the proper level of abstraction for "as-a-Service", on how to make the cloud secure [20, 21] and how to increase its energy efficiency [55, 236, 158]. Large-scale cloud deployments are governed by complex performance/cost/energy consumption equations [90] and point out to data movement as one of the biggest source of energy consumption in computing infrastructures [58, 57, 222, 208].

While research attention is dedicated to reducing data movement [196, 115] and creating the green cloud [236], there is a lack of focus on exploring strategies to enhance data movement with compute capabilities.

This thesis addresses the open question of enhancing data movement with compute capabilities, more precisely with data analytics and privacy computational features, with the ultimate goal of making data movement more efficient and secure using FPGA based compute engines. Through the integration of compute engines alongside data movement, it becomes possible to reduce the number of passes performed on the data and eliminate the necessity for intermediate storage. To achieve this, the performance of several database-oriented algorithms is analyzed when deployed as compute engines on an FPGA. Additionally, the behavior of the engines when connected to network communication protocols like TCP/IP and RDMA is examined on an FPGA-based smartNIC. The structure of the thesis can be divided into three main components.

Firstly, exact computation algorithms that are preserving the full representation of the data are considered. One such example is the algorithm for calculating Pearson Correlation Coefficient, used to assess and quantify the correlation between two or more data attributes. Correlation is frequently used in machine learning [83], data mining [56], databases [155, 121], business analysis [157], and statistics [206] to interchangeably represent different types of relations (linear or non-linear), mutual dependencies, or causality, with the ultimate goal of summarizing large

amounts of data by observing patterns between data attributes [31]. Correlation is important for data processing and data management systems [36, 121, 195] and usually finds its place in exploration, data cleaning, or data pre-processing stages, all of which constitute a significant amount of effort for a data scientist [144].

Secondly, retrieving statistical information using sub-linear memory complexity with respect to the input data size is considering by looking into sketch algorithms. Unlike exact computation algorithms, in sketch algorithms, input data loses its representation through hashing and the final result is an approximation of the result that would have been obtained by an exact computation. This set of algorithms uses fixed size data structures to deliver approximate statistical information over data values, such as: membership to a set [28], the frequency of an element [51], cardinality [71], and different skew degrees of the data [9]. Beside pure statistical information, sketch algorithms can be used to monitor network traffic [223], to perform query optimization [39, 173] or to optimize the way database engines perform multiple joins [148].

Finally, irrespective of the use of exact or approximate computation, for services offered in the cloud, where hardware resources can be shared among multiple users, data needs to be secured when it is in transit or at rest. To address the domain of data privacy, it is shown how the AES encryprion-decryption algorithm behave on the FPGA in the context of databases, addressing multiple security levels, and expanding the utilization of FPGA further than ”only parallelizable algorithms behave well on the FPGA”.

Processing data as stream. The transfer of data over the internet can be seen as a quasi-continuous flow of bits carrying information from a source that generates the data to a destination (sink) that consumes it, making the transfer very similar to a stream. Different from batch processing that processes data at predefined time intervals, stream processing starts processing the data as soon as it becomes available to the processing engine. In the context of the thesis, a stream does not mean endless data, but rather that the end of the data may come at an undefined time. Moreover, the usage of the term 'stream interfaces' depicts how data is processed inside the FPGA through a direct engine-to-engine communication, avoiding communication via memory interfaces and resulting in significant performance improvements. Also, the limit of the number of elements in a continuous stream can also come from saturating the computing capacity of the processing module consuming the stream. For example, for a system accumulating values over streams, the limit is imposed by saturating the accumulators' capacity; for a sketching algorithms module, the limits is imposed by exceeding the allowable error. Nevertheless, the term 'stream' is not used from the point of view of a stream processor, where processing data is grouped in windows, being them sliding or tumbling windows.

1.2 Motivation & Thesis Statement

Cloud computing targets the decoupling of compute units from the storage ones [12, 216], while allowing the elastic allocation of resources (i.e., scaling up and out) to meet demands. This distributed resource placement [75], together with the more distributed nature of the applications running on top of the cloud, indicate only an increase in the data movement. As data movement cannot be avoided, it remains within the systems' power and ability to make it effective.

FPGA as smartNICs. As mentioned before, operating data applications in the cloud translates to data movement, several data copies, and large I/O costs. With 100G network interfaces becoming the network communication speed norm, the CPU can easily become the bottleneck when processing incoming data packets [97, 233, 167]. Therefore, offloading data processing operators into the Network Interface Card (NIC) can address these shortcomings and relieve the load on the CPU. A NIC is a printed circuit board that connects to a server via PCIe and enables server's connectivity to a network, whereas a smartNIC adds also processing in addition to network traffic. SmartNICs are available in multiple forms, ASIC-based, FPGA-based, or SoC (system-on-chip)-based, and the choice of one over the other depends on the data workload and acceleration tasks that are sought.

For example, Microsoft's Catapult project deploys FPGAs as line-rate or remote accelerators for distributed systems, with the FPGA being positioned between the data center network switch and server's NIC, with all the network traffic passing through the FPGA [159] with the aim to accelerate a wide range of use cases from key-value stores [150], network function virtualization [70], search engines [159], to AI/ML applications [152, 46, 73]. Amazon's AQUA employs FPGAs together with SSDs to offload parts of SQL operators (selection, projection, LIKE predicates) to a network-attached caching layer for large-scale data analytics [23], a design also explored in research [120, 224]. Sidler et al. [198, 191, 197, 201] have advanced the research path for in-network data processing by developing the first open source FPGA-based scalable network stack module supporting TCP/IP, RoCEv2, UDP/IP at up to 100 Gbps. The availability of this network stack has enabled the evolution of various FPGA frameworks (shells) that target operating system abstractions [139], network access for FPGA boards operating with Vitis shell [86], remote direct memory access [201], FPGA board resource management [60], and which facilitate the deployment of compute kernels along the network.

This dissertation adds a building block to the state-of-the-art of in-network data processing by showing the benefits FPGAs could bring to on-the-fly data analysis, and their flexibility to attach

to different communication protocols. FPGA's flexibility makes them easy to be positioned either between two compute nodes or between storage and compute, acting as a bump-in-the-wire. When put on the wire, an FPGA-based compute engine can be designed so as not to decrease the communication link performance. The endpoints should perceive the communication as if no other compute unit is placed in between. Since the FPGA can host both compute engines and I/O interfaces, it becomes a good candidate for performing compute-intensive operations while data is on the move. This contribution will show that FPGAs can minimize I/O cost, data movement and data compute at the CPU level by acting as a bump-in-the-wire processor.

In order to understand the impact FPGAs could make on cloud computing it is important to analyze the compute and I/O bounds of high-end CPUs. For this, to each analyzed algorithm deployed on the FPGA is associated a multi-threaded CPU implementation. This allows the comparison with a solid baseline and the study of the computational resources that could be gained by the CPU by offloading intense computational tasks to the FPGA.

Thesis Statement. The FPGA can extract information from and characterize the data while it is flowing through the data center system with low overhead. By taking the network stack and the PCIe connections on an FPGA-based NIC as I/O modules (i.e., the source and sink of the network data), and the analytical functions implemented on the FPGA as units of continuous non-blocking processing, the FPGA is a suitable candidate for in-network stream data analytics.

1.3 Contributions & Structure

The contributions brought by this thesis to the state-of-the art of in-network data processing are summarized as follows.

In **Chapter 3**, we analyze the behavior of a stream analytics FPGA accelerator integrating correlation computation when attached to the CPU as a co-processor or as a network interface card, with RDMA as the underlying network communication protocol, showing the low overhead induced by the FPGA when compared with a multi-threaded CPU implementation and making the following contributions:

- We describe the architecture of an FPGA-based accelerator that computes the Pearson correlation coefficients between parallel data streams;
- We demonstrate the functionality of the design for the concurrent computation of the correlation coefficient among up to 64 data streams and the analytical study of the capacity

of the design, which is as high as several thousand concurrent data streams (well beyond the I/O capabilities of modern devices or networks);

- We study the generalization capacity of the design, which is as high as several thousand data streams (well beyond the I/O capabilities of modern devices or networks);
- We demonstrate the ability to embed the accelerator on an RDMA-capable FPGA-based smartNIC;
- We show through experimental evaluation that the design offers several orders of magnitude performance gains (4.4×) in both throughput and compute time over CPU and GPU-based approaches.

In **Chapter 4**, we support the thesis statement by analyzing the behavior of a stream analytics FPGA accelerator integrating three different sketch algorithms (Hyperloglog, Fast-AGMS and Count-Min) when attached to the CPU as a co-processor or as a network interface card, with TCP/IP as the underlying network communication protocol:

- We explore how to merge the three sketch algorithms into a single design both on CPUs and FPGAs;
- We analyze in depth the trade-off between resource utilization and accuracy of the algorithms;
- We demonstrate the ability to embed the accelerator on an TCP/IP-capable FPGA-based smartNIC and show the compromise between the number of pipelines that run in parallel within an accelerator and the saturation of a 100 Gbps communication link;
- We extensively explore the resulting performance with comparisons against CPUs with different processing capacities, showing gains of up to 1.75× over a high-end server with 2× Intel®Xeon®Gold 6248 Processors, and 3.5× over a smaller, more conventional Intel Xeon Gold 6234 Processor.

In **Chapter 5**, we address the data security aspect of the cloud, by implementing and evaluating an FPGA-based accelerator with two complementary processes (encryption and decryption) that are used to secure data files produced by an in-memory databases (SAP HANA). This work is part of a larger effort of introducing in-memory databases ”as a Service” in the cloud, with I/O operations such as heavy-weighted compression and encryption offloaded to an FPGA-based

accelerator that is placed on the I/O path of SAP HANA. In this thesis the focus if only on the data security problem, making the following contributions:

- We propose a design that is configurable at build-time and implements three different modes of the Advanced Encryption Standard (AES), targeting three different degrees or tiers of security;
- We show how RTL design can be integrated as function calls, abstracting away the hardware description language;
- We differ from "only parallelizable algorithms" can be offloaded to the FPGA, showing that a non-parallelizable algorithm can have better performances over the CPU when combined with a compute-intense data reduction module;
- We show that encryption and decryption can achieve each up to 15 GB/s of throughput;
- We open source a design that shares similarities with the proprietary designs used by multiple cloud providers.

Chapter 2 provides an overview of the platforms and FPGA devices used in this thesis, whereas **Chapter 6** summarizes the thesis and offers a view on how FPGA should play their role in the CPU-GPU-FPGA-in network data processing ecosystem.

Related Publications This dissertation is based on work (* - equal contribution) that has also been presented in the following peer reviewed publications:

- **AMNES: Accelerating the computation of data correlation using FPGAs** *by M. Chiosa, T.B. Preußer, M. Blott, G. Alonso, in Proceedings of the Very Large Databases (VLDB) Endowment, Volume 16, Number 13, August 2024.*
- **SKT: A one-pass multi-sketch data analytics accelerator** *by M. Chiosa, T.B. Preußer, G. Alonso, in Proceedings of the Very Large Databases (VLDB) Endowment, Volume 14, Number 11, July 2021.*
- **Hardware acceleration of compression and encryption in SAP HANA** *by M. Chiosa*[*]*, F. Maschi*[*]*, I. Müller, G. Alonso, N. May, in the 48*[th] *International Conference on Very Large Databases (VLDB), September 2022.*

Apart the above-mentioned publications, participation in other research projects has resulted in the following publications:

- **StRoM: Smart Remote Memory** *by D. Sidler, Z. Wang, M. Chiosa, A. Kulkarni, G. Alonso, in Proceedings of the 5th European Conference on Computer Systems (EuroSys), April 2020.*
- **Using DSP Slices as content-addressable update queues** *by T.B. Preußer, M. Chiosa, A. Weiss, G. Alonso, in the 30th International Conference on Field-Programmable Logic and Applications (FPL), August 2020.*
- **Hyperloglog sketch acceleration on FPGA** *by A. Kulkarni, M. Chiosa, T.B. Preußer, K. Kara, D. Sidler, G. Alonso, in the 30th International Conference on Field-Programmable Logic and Applications (FPL), August 2020.*

CHAPTER 2

BACKGROUND

This chapter provides an overview of the FPGA, the FPGA frameworks and platforms, and the network communication protocols used for this thesis.

2.1 Field Programmable Gate Arrays

FPGAs are considered ”field programmable” integrated circuits because once taken out from the electronic wafer, their hardware can be reconfigured via programmable interconnects and SRAM writes to carry out different digital functionalities. Their versatility spans many industry areas, from telecommunications [217] to medical field [110], and aerospace to defense [16, 129, 162]. For the last decade, FPGAs have been proven effective to accelerate database operators [212, 112], stream processing applications [238] and security sensitive applications [21]. Only in the last few years, they have started being deployed into data centers [112, 23, 70, 168] and used for data analytics and machine learning tasks.

For example, Microsoft Azure uses FPGAs for deep neural network inference and Bing search ranking [164] and Azure Synapse deploys them for big data analytics [24]. Additional examples that place FPGAs near-data include a write-optimized storage engine developed at Alibaba, X-Engine, to efficiently implement the Log-Structured Merge (LSM) tree compaction [96], in-storage computing capabilities to optimize data-intensive queries [120], or an intelligent distributed storage layer [113].

Prior to introducing the FPGA-based platforms used for this work, two FPGA environments are identified. On the one hand, there is the internal FPGA environment which is composed of lookup tables (LUTs), flip-flops (FFs), carry-logic and multiplexers that are grouped into logic blocks. Depending on the FPGA vendor, the logic blocks are named configurable logic blocks (CLBs) for AMD-Xilinx FPGAs or logic array blocks (LABs) for Intel FPGAs [215]. In addition to the logic blocks, other types of resources can be found in the FPGA fabric such as low-latency memories, Block RAMs (BRAMs) or Ultra RAMs (URAMs), or computational units, such as the digital signal processing (DSP) units. All these internally available resources are coupled by a programmable interconnect and can be configured to implement a wide range of digital functionalities from communication protocols to regular expression matching engines [200] by leveraging hardware description languages (HDL), like VHDL or (System)Verilog, or high-level languages, like C/C++ or OpenCL.

On the other hand, there is the external FPGA environment which is usually referred to as the FPGA board or platform and encompasses: (1) external DDR memory banks, with each bank bandwidth capacities being around 16 GB/s, (2) PCI Express (PCIe) (Gen3x16 or Gen4x8) interfaces, (3) network physical interfaces (QSFP28) with bandwidth capabilities of up to 100 Gbps, (4) additional serial communication peripherals (UART, I^2C, SPI, USB), (5) video controllers or (6) analog to digital converters (ADCs). Nowadays, external to the FPGA integrated circuit, but within the same chip package, it is coupled a High Bandwidth Memory (HBM) [27] that overcomes the memory throughput bottleneck of DDR memories by offering up to 420 GB/s bandwidth capabilities for 32 physical memory channels [128].

FPGAs used in this thesis. For the work presented in this thesis, various FPGA boards have been utilized depending on their availability or the specific requirements of the tasks involved. All the boards used are data center class boards, that target deployments in heterogeneous clusters. Table 2.1 summarizes them, their board level specifications and vendors. In addition to this, the table also includes a reference to the chapter where the FPGA boards have been employed.

Table 2.1: FPGA boards that are used for this thesis.

Board	Specification					Producer	System
	DDR and HBM [GB]	SRAM Capacity [MB]	PCIe	Network	Power[W]		
U250	64	54	Gen3×16	2×QSFP28 (100 GbE)	225	AMD-Xilinx	SKT [Chapter 4]
U280	40	41	Gen4×8	2×QSFP28 (100 GbE)	225	AMD-Xilinx	SKT [Chapter 4]
U55c	16	43	Gen3×16, 2× Gen4×8	2×QSFP28 (100 GbE)	150	AMD-Xilinx	AMNES [Chapter 3]
D5005	32	30.5	Gen3x16	2×QSFP28 (100 GbE)	215	Intel	(En—De)cryption [Chapter 5]

Data types used in this thesis. While it is possible to implement floating-point arithmetic on the FPGA using specialized IPs, the true advantage of the FPGA lies in its capability to operate at the bit-level with custom data types, whether they are integer or arbitrary precision fixed-point data types. Significant research has thrived in the field of ML concerning reduced precision representations with Microsoft's FPGA-based Brainwave [46], Google's TPU [123] and NVIDIA's Turing GPUs [34] looking into reducing the precision down to 16-bit floating-point, along with 8-bit (integer) and 4-bit (integer) fixed-point. Rajagopal et al. [183] propose MuPPET that is using quantized fixed-point training for convolutional neural networks (CNNs). The benefit of reduce precision is also shown for deep neural networks (DNN) by Sun et al. [210]. Furthermore, AMD-Xilinx Research Lab has proposed quantized neural networks inference on FPGA through the FINN open source framework [143], offering customizable dataflow architectures with low latency and high throughput [218], and Alistarh et al. [7] have shown that quantization can reduce communication and memory overhead in distributed training. Another aspect that motivates the usage of integer and fixed-point representation on FPGA is the fact that fixed architectures are highly optimized for floating-point data types. For example, the Fused Multiply-Add (FMA) unit of the CPU is optimized for single and double precision floating-point data types. However, this feature does not yield any performance advantages when it comes to integer values. In this thesis, the data analytics engines' focus is on integer data representation. The arithmetic behavior of this representation is the same with fixed-point data types, therefore the conclusions can be extended to both data types.

FPGA Shells. In this thesis, the FPGA focus is twofold: (1) how compute-intensive data analytics engines behave when attached to data streams coming from the network, and (2) how a security engine behaves while encrypting the data flow created between two FPGA DDR memory channels. Since data center setups are targeted, the compute kernels are deployed on FPGA-based boards provided by the two main vendors. AMD-Xilinx offers Heterogeneous Accelerated Compute Cluster (HACC) across five world wide academic centers (ETH Zurich, University of Illinois in Urbana-Champaign – UIUC, National University of Singapore – NUC, Paderborn University, University of California in Los Angeles – UCLA) [13], and Intel offers the IL Academic Compute Environment (ACE) [105]. Each of the academic clusters is equipped with high-end servers and FPGA-based boards, and utilizes similar programming environments, Vitis unified software platform by AMD-Xilinx [230] and Open Programmable Acceleration Engine by Intel (OPAE) [103], to handle the interaction between the CPU and the FPGA as regards control and data movement.

Regardless of these different denominations, the user application on the host CPU is developed in

C/C++ and interacts with the FPGA and its compute engine (kernel) through standard OpenCL API calls. The physical interaction between the CPU and the FPGA is abstracted away from the user application with the help of runtime libraries that run on the host CPU. They include user space libraries and drivers (i.e., Linux and kernel drivers) and have different functionalities such as: (1) downloading the FPGA binary file on the hardware platform, (2) managing the execution of the kernel (i.e., triggering, sequencing and synchronizing computation), (3) moving data between the host CPU and the FPGA's internal memory or DDRs, and (4) providing board management (i.e., debug, power management and board recovery). The runtime library of each vendor interacts with the proprietary static shell and the configurable dynamic hardware region on the FPGA. The configurable dynamic region implements the compute kernel, whereas the static shell ensures the communication with all the peripherals (i.e., PCIe, HBM, DDR).

A limiting factor of the FPGA vendor proprietary shells is the absence of data streaming support between the FPGA and the host CPU. Current shells use the OpenCL memory model and need all data to be transferred to the FPGA's internal or DDR memories before triggering kernel action, thus incurring additional data movement steps. AMD-Xilinx previously offered a shell with support for streaming transfers (QDMA) and a limited transfer capacity of 32 GB. However, the shell got discontinued and only the memory-mapped shell (XDMA) is supported now. For clarification purposes, it is to be noted that the same names that are used to identify the DMA for PCIe Subsystem IPs (QDMA and XDMA) are also used to identify the shells. Regardless of the IPs' interfaces that get exposed by the shells, the two IPs offer two configurable user interfaces: a memory-mapped interface (AXI4-MM) and a separate streaming interface (AXI4-Stream). The proprietary shell's streaming interface limitation has been addressed by open source academic shells, such as Coyote [139], TaPaSCo (Task-Parallel System Composer) [88], and DavOS (Distributed Accelerator OS) [60], that enable streaming interfaces between the host and the FPGA, reducing data movement while providing FPGA peripheral orchestration and OS abstractions.

Another limiting factor of the proprietary static shells is the access to more than the data link layer of the network. Moreover, when taken off-the-shelf, the shells enable only the PCIe communication as access point to the FPGA-based board. For the AMD-Xilinx shells for the HACC cluster at ETH Zurich, this limitation has been addressed by EasyNet [86], an open source academic project, that encapsulates the TCP/IP functionality and facilitates the access to the integrated 100G Ethernet Subsystem and the physical Gigabit Transceiver pins exposed by the shell, from within the dynamic region of the shell. In addition to network functionality in the form of a network kernel, EasyNet also allows the deployment of compute kernels along the network one, hence enabling streaming processing with data coming from the network. Irrespective of the

network access, EasyNet is limited on the host CPU side, where the interaction lacks streaming support, being only memory-mapped.

Coyote [139] and DavOS [60] extend the proprietary shell for the HACC cluster and EasyNet, enabling not only TCP/IP, but also RDMA over Convereged Ethernet (RoCE), with a stream processing line from the network physical interface to the host CPU, therefore enabling the bump-in-the-wire behavior. More than DavOS, the Coyote shell enables dynamic reconfiguration of multiple isolated FPGA regions, kernel support for both HLS and RTL rather than only RTL support by DavOS, a unified host and FPGA memory and HBM support, making it the practical and desirable shell for the HACC cluster's boards at ETH Zurich. Throughout this thesis, the streaming compute kernels are attached to EasyNet in Chapter 4, to Coyote in Chapter 3 and to the Intel shell in Chapter 5.

The kernels (engines) running on the dynamic region of the FPGA can be developed in HDL, OpenCL (C-like) or C/C++. If the former aims to achieve resource efficiency and high clock frequencies, the latter two aim to increase productivity by providing C/C++ abstraction layers over register transfer level (RTL) structures, and make the FPGA accessible to software developers. Nevertheless, building abstraction layers over RTL is a constantly evolving process, both in industry and academia [35, 94, 122]. Currently, specific words named *pragmas* need to be added to the source code in order to obtain an optimized RTL design (i.e., reduce latency, larger level of pipelining, reduce FPGA resource usage). Therefore, the kernel developer still needs to have hardware knowledge and understanding to create efficient compute kernels.

AMD-Xilinx has initially developed and maintained Vivado High Level Synthesis (Vivado HLS) to later move to Vitis HLS. While both target the same vendor FPGAs, Vivado offers a hardware-centric approach to designing hardware, while Vitis offers a software-centric approach to developing both hardware and software [95] with the aim to unify both under the same tool and development flow. A similar approach is adopted by Intel with the oneAPI toolkit, subsequent to Intel SDK for OpenCL. Nevertheless, Intel FPGA SDK for OpenCL tool is used in Chapter 5.

Throughout this thesis, Vivado and Vitis HLS are used in Chapter 3, Intel FPGA SDK for OpenCL and hardware description language (VHDL) are used in Chapter 5 to develop the kernel modules. The development of the analytical kernel presented in Chapter 4 has been carried out in collaboration with Thomas Preußer, and integrated into EasyNet using Vitis HLS.

2.2 Networks in the Data Center

The transport layers used in data center network communication and their associated physical support environments come in a wide variety of choices with the ultimate goal of achieving high bandwidth and low latency [92]. TCP/IP legacy has facilitated its adoption into the data center network [124], but its large communication latencies observed at software stack implementation levels have challenged its usage for intra data center communication. The protocol is still preferred for wide area networks (WAN) due to its reliable transmission and control flow, but in order to be adopted internally in the data center, to support the multitude of applications that still depend on it, it has to overcome the software stack shortcomings. For this, the TCP/IP stack has been partially [38, 72] or fully [198] offloaded to the Network Interface Card (NIC), with the latter solution being optimal in terms of performance scalability of up to 100 Gbps and the supported number of connections [199]. When the TCP/IP stack is totally offloaded to an FPGA, the traditional TCP/IP communication pass through the Kernel space is simplified (i.e., unnecessary data copies from the user space to kernel space are avoided), since the application running on the host CPU interacts directly with the driver that orchestrates and synchronizes the communication with the DMA (Direct Memory Access) engine running on the FPGA. In Chapter 4 a multiple sketch design is deployed on an FPGA-based smartNIC and processes data coming from a TCP/IP interface.

Remote Direct Memory Access (RDMA) enables the direct data transfer from one CPU's memory to a remote CPU's memory by partially or totally bypassing the CPU, leading to lower communication latencies than TCP/IP. This performance advantage has made RDMA attractive to both research and industry, being adopted in cloud deployments (Microsoft Azure [82], Alibaba Cloud [76], Oracle Exadata [178], distributed storage systems [3, 62] and databases [233, 142, 193, 220]. Currently, there are three network supports for RDMA: Infiniband [6], RDMA over Converged Ethernet (RoCE) [5] and iWARP [174]. Infiniband requires its own physical infrastructure to deliver reliable RDMA, whereas the other two run over existing Ethernet physical infrastructures. On the one hand, iWARP has a top-down architecture that aims to implement RDMA functionality within the framework of the existing TCP transport layer. The intention to run over pre-existing layers has lead to a convoluted architecture with large latencies, defeating the low-latency RDMA initial purpose [211]. On the other hand, RoCE has a bottom-up approach, being defined by the InfiniBand Trade Association (IBTA) standard as a network protocol that enables RDMA over Ethernet networks, which leads to a simpler and straightforward architecture than iWARP. For the first version of RoCE (RoCEv1), the RDMA packets are en-

capsulated into Ethernet frames. This approach limits the utilization of RoCE to only the same IP subdomain. RoCEv2, also known as routable RoCE, overcomes this limitation, enabling RoCE in different IP subnets and network scaling. This is possible through the addition of IP and UDP headers information into the RDMA packets. By utilizing UDP port 4791, which is reserved for RoCEv2, RDMA payloads are encapsulated as UDP payloads.

Even if Infiniband is set to offer the best performance for RDMA in terms of latency and throughput, RoCE has a very large data center adoption due to its cost, very good performance and resource scaling capabilities [76, 82, 178]. In Chapter 3 the Pearson correlation coefficient kernel processes data coming from an RDMA interface.

CHAPTER 3

CORRELATION - EXACT COMPUTATION

In this chapter we analyze how correlation computation on data streams can be offloaded to the FPGA. The design goes beyond matrix-matrix multiplication approach and offers a customized solution based on accumulators. The end result, `AMNES` – a stream analytics engine, can sustain data streams arriving at 100 Gbps over an RDMA network, while requiring only 10 ms to compute the correlation coefficient among 64 data streams.

3.1 Motivation

Computing the correlation coefficient among two or multiple attributes finds extensive applications across a wide range of disciplines, including machine learning (ML), databases (DB), data privacy, medicine, physics and economics. For example, knowing the correlation among data can affect the scheduling decision of whether to offload the computation to a General Purpose GPU (GPGPU) or another accelerator that involves data movement [32]. Similarly, correlated data affects the error of the selectivity estimators used by query optimizers [63], i.e., highly correlated data leading to higher errors [87]. In machine learning, knowing which features or dimensions are correlated serves in both dimensionality reduction, by pointing at data that can be removed as it does not provide additional information [232], and correlation clustering since the correlation relationship between data is a useful input to clustering algorithms [136]. Finally, correlation computation can be utilized by vector databases to find data vector similarity [175], or in data privacy applications to detect when too much public data leads to information leakage [239]. In all these applications, knowing in advance whether data sets are correlated is

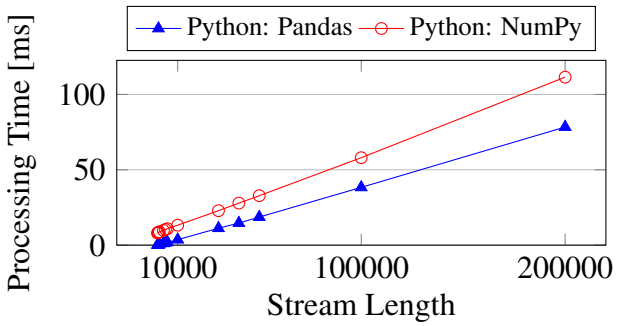


Figure 3.1: Performance metrics of Python NumPy and Pandas libraries for computing the Pearson correlation coefficient for 16 attributes.

important to be able to process the data sets efficiently. Additionally, correlation becomes a basic quantity for many other modeling techniques.

Finding the correlation between data sets (columns in a relational table, two data streams, or dimensions in a set of vector data) is typically an expensive operation. On the one hand, its computation typically involves calculating several statistics (i.e., standard deviation, covariance) over each data set that feed into computing the correlation coefficient. This requires a full pass over the data. On the other hand, in most cases, the correlation has to be calculated across many data sets (e.g., wide tables in databases, or high-dimensional vectors in ML applications). As a reference, the widely used NumPy and Pandas Python libraries can take up to 100 ms to compute the Pearson correlation coefficients between just 16 attributes (streams) with 200'000 elements each as illustrated in Figure 3.1. In this chapter we show how to compute the correlation for 64 streams and 2 million elements per variable in about 10 ms, i.e., an order of magnitude less than these established libraries and processing 10 times more items. In databases, due to the cost of computing correlation, it is often approximated, especially when used to optimize the creation of indexes, as data correlations can significantly affect their performance, particularly for clustered indexes [134, 61]. One common approximation is to compare the number of distinct values across attributes [134]. However, this measure is far less precise than statistical correlation measures.

Pearson Correlation in RDBMS (Relational Database Management System). Relational engines such as PostgreSQL, Oracle DB, MySQL, or Snowflake and vector databases (Milvus) compute correlation between pairs of attributes via either an intrinsic operator (PostgreSQL, Oracle DB, Snowflake, BigTable) or by combining statistics over the data (MySQL). For more than two attributes, a manual query with the explicit pairs of attributes has to be written. As the amount of data to be processed grows, there is a need to understand how to efficiently correlate

many attributes in parallel and whether the computation can be offloaded to an accelerator, e.g., without CPU involvement. Therefore, in this chapter, we explore this question by looking at how computing the statistical correlation can be accelerated using an FPGA acting on streams of data arriving from or being sent to the network.

As mentioned in Chapter 1, the disaggregated nature of today's compute and storage provides motivation to investigate such a design, since processing the data would already involve first reading the data from object storage and bringing it into the computing node. Having the computation offloaded to an accelerator positioned either on the storage nodes or on the network path would be similar to the scenarios provided by Amazon AQUA and Microsoft's Catapult. Another example of on-the-fly data transformation is Oracle Exadata, a database engine with smart disaggregated storage where data is kept in row format for online transaction processing (OLTP), but is transformed on-the-fly into column-based as data is moved from storage to in-memory to accommodate fast online analytical processing (OLAP). Our design enables offloading the correlation operator to the accelerator targeting similar scenarios as the ones mentioned before. Moreover, `AMNES` can also be used in the conventional acceleration model with data residing in the CPU's memory due to the design low latency.

The benefit of having the correlation assessed while data is moving through the network, be it from one compute node to another or from storage to a compute node, is that correlation becomes meta-data that can be used by the subsequent processing tasks (e.g., machine learning pipelines, scheduling algorithms, or analytical jobs), once all the transferred data has reached its destination. Since most of the data to be processed in cloud-based systems has to travel through the network, enhancing network cards with complex computation capabilities reduces data movement, eliminates the need for intermediate storage, and eventually reduces energy consumption. For instance, in the same way Amazon AQUA utilizes FPGAs to process data read from the SSDs in a streaming fashion before forwarding it to the network, our system could be deployed for correlation computation and serve relational DBs to assess correlation across a table, vector DBs or ML systems to identify data similarities directly on the storage (or on a network attached cache like AQUA) or as the data arrives from storage (as in Microsoft's Catapult).

In addition to the architectural flexibility, `AMNES` is also easily adaptable to different data representations than row format cacheline representation. Adapting to different data representations only requires a temporary buffer and, potentially simple transformations (e.g., transposition). For instance, systems such as Snowflake employ a format where groups of rows are mapped into individual micro-partitions, organized in a columnar fashion [204]. Applying our design would require to simply transpose the micro-partition to reconstruct the tuple, something that can be

easily done on-the-fly on an FPGA with no performance loss. This is the reverse (columns to rows) of the transformation performed by Oracle Exadata (rows to columns). Positioned in the storage layer of Oracle Exadata, AMNES would not need to transpose the data. Regarding other types of data transformations that are usually required for data in transit such as compression and encryption, the FPGAs have already proved to efficiently decompress (e.g., delta encoding or run-length) and decrypt both column-based structures [126] and data blocks [42]. Such operations can be easily integrated in our design.

AMNES is an FPGA-based accelerator that computes the Pearson correlation coefficient (PCC) among data streams, and can operate both on data residing in the host memory (i.e., as a conventional accelerator due to its low latency) or on data streams arriving from the network. While the actual network transport protocol used is not relevant for the computation, in our prototype, we have focused on RDMA networks as they are more challenging in terms of throughput but also offer the opportunity to significantly lower the latency of data access. AMNES operates specifically on the Converged Ethernet (RoCEv2) protocol, which is already deployed by many cloud providers such as Alibaba Cloud [76], Microsoft Azure [82], with 70% of Azure traffic being RDMA-based [163], and Oracle Exadata [178]. RDMA has also made its way into database design, with a growing number of systems and prototypes demonstrating its advantages [233, 167].

This chapter focuses on Pearson's correlation as it is the most widely used. If we considered ranked values (ranks being associated to sorted data) instead of actual data, then the same design could be used to compute the Spearman correlation coefficient for non-distinct data. The implementation of equations for Spearman correlation coefficient for only distinct data and for Kendall correlation would require a simplified variation of the design where only the distances between ranks or unitary values get accumulated, respectively. AMNES's modular architecture allows to use only part of the design (i.e., the ACC Engine), to derive a variety of useful statistics such as correlation coefficients, cosine similarity and cosine distance, standard deviation, as well as slope and intercept of linear regression lines over the data. These are all standard operators in relational engines these days and can be supported by our design with minimal changes.

The chapter is structured as follows. We first introduce the Pearson correlation coefficient and the underlying communication protocol to which we associate the compute kernel. Afterwards, we present the overall design of AMNES, offering a thorough analysis of the design choices and finalizing with the FPGA implementation. In Section 3.5 we evaluate our implementation's performance against two multi-threaded CPU baselines with two specific scenarios in consideration: (1) data resides within the memory of the host CPU, from where it is retrieved and processed; (2) data arrives through an RDMA connection from a remote node. We close the chapter with

a comparison regarding the number of streams analyzed in parallel between AMNES and other systems proposed for correlation computation in the database community. In addition to this comparison, we discuss multiple ways to generalize AMNES's backend engine, and how other correlation coefficients, cosine similarity and the least square method may benefit from AMNES.

3.2 Correlation and Remote Direct Memory Access

In this section, we formally define the correlation between two streams, explain the statistical and mathematical background underlying the Pearson correlation coefficient and present our motivation behind using Remote Direct Memory Access (RDMA) at networking level.

3.2.1 The Correlation Coefficients

Correlation can indicate if two data variables co-vary, depend on or predict one another, and can be measured using different types of coefficients, e.g., Pearson, Kendall, Spearman, or Point-Biserial. The latter employs the same formula as the Pearson correlation coefficient, with one variable being binary. The Pearson correlation coefficient is the normalized version of covariance [56]. In contrast to Pearson, which measures linear association, Kendall and Spearman coefficients are non-parametric tests (e.g., do not depend on the underlying data distribution) and measure an ordinal association between ranked data [207]. These two coefficients assume ordered stream values and are mainly used to assess a non-linear association [66]. Database engines often include several measures of correlation as part of their statistical functions (e.g., Oracle includes support for Pearson, Kendall, and Spearman correlation coefficients).

We focus on the Pearson correlation coefficient (PCC), which measures the strength of the linear association between two data streams by taking into account the amount of variation present in each stream and how the streams vary together. The coefficient is a dimensionless quantity within the range $[-1, +1]$. Unlike covariance, which can take infinite values, this well-confined range allows for a straightforward assessment. A Pearson correlation coefficient (ρ) of value 0 indicates that no linear relationship exists between two data streams, i.e., they are independent. A perfect linear relationship is indicated by a coefficient of magnitude 1. A negative sign indicates that the increase in the values of one stream associates with a decrease in the values of the other stream. A positive sign, on the contrary, indicates that the increase in the values of one stream associates with the increase in the values of the other. The stronger the correlation, the closer the

correlation coefficient gets to ±1. If we consider $|\rho|$, the absolute value of the PCC, the strength of the linear relationship can be assessed as follows: (1) weak correlation for $|\rho| \in [0.1, 0.3)$; (2) medium correlation for $|\rho| \in [0.3, 0.5)$; and (3) strong correlation for $|\rho| \in [0.5, 1]$ [18, 209]. Nevertheless, the relationship strength does not imply any causal relationship between the two given streams [1].

The PCC is the most widely used among the correlation coefficients presented above [207]. Intuitively, if a line is drawn as a best fit through the data points of two streams, the PCC indicates the amount of variation that exists around this line of best fit. Nevertheless, PCC does not represent the slope of the line of best fit. Its value in $[-1, +1]$ indicates the variation around this line, with values closer to 0 indicating a large variation as illustrated in Figure 3.2.

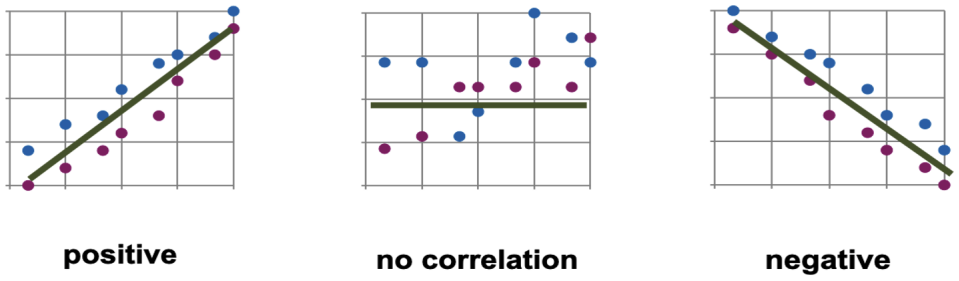


Figure 3.2: Correlation relationship.

Mathematically, the PCC (ρ) for a bound population (e.g., maximum number of items for two analyzed streams is N) is shown in Equation 3.1, where $\mathbb{E}$ is the expectation, μ and σ are the mean and the standard deviation of each data stream, and the sign of $(X - \mu_X)(Y - \mu_Y)$ indicates if an increase in X is associated with an increase in Y. Developed further, Equation 3.1 describes PCC as the centered and standardized sum of the cross-product of two data streams [189].

$$\rho_{XY} = \frac{\mathbb{E}[(X - \mu_X)(Y - \mu_Y)]}{\sigma_X \sigma_Y} = \frac{\sum_{t=0}^{N-1}(x_t - \mu_X)(y_t - \mu_Y)}{\sqrt{\sum_{t=0}^{N-1}(x_t - \mu_X)^2}\sqrt{\sum_{t=0}^{N-1}(y_t - \mu_Y)^2}} \tag{3.1}$$

In Equation 3.2, we replace expectation and standard deviation by their corresponding mathematical formulas and reduce the common members to minimize the number of necessary divisions. Equation 3.2 leads to the core components around which our design centers: the sum of elements ($\sum_{t=0}^{N-1} x_t$, $\sum_{t=0}^{N-1} y_t$), the sum of squares ($\sum_{t=0}^{N-1} x_t^2$, $\sum_{t=0}^{N-1} y_t^2$), and the sum of products ($\sum_{t=0}^{N-1} x_t y_t$). For the remaining of the chapter, we are going to refer to them as *the sufficient statistics*. A similar equation is derived for binary variables and used to find the correlation between graphs [131], aiming at discovering the dependencies within a graph database.

$$\begin{aligned}\rho &= \frac{\sum_{t=0}^{N-1} x_t y_t - \frac{\sum_{t=0}^{N-1} x_t \sum_{t=0}^{N-1} y_t}{N}}{\sqrt{\sum_{t=0}^{N-1} x_t^2 - \frac{(\sum_{t=0}^{N-1} x_t)^2}{N}} \sqrt{\sum_{t=0}^{N-1} y_t^2 - \frac{(\sum_{t=0}^{N-1} y_t)^2}{N}}} \\ &= \frac{N \sum_{t=0}^{N-1} x_t y_t - \sum_{t=0}^{N-1} x_t \sum_{t=0}^{N-1} y_t}{\sqrt{N \sum_{t=0}^{N-1} x_t^2 - (\sum_{t=0}^{N-1} x_t)^2} \sqrt{N \sum_{t=0}^{N-1} y_t^2 - (\sum_{t=0}^{N-1} y_t)^2}}\end{aligned} \tag{3.2}$$

In order for the PCC results to be interpretable and trusted, the analyzed streams should satisfy the following assumptions [209]: (1) no missing values and a continuous scale; (2) the stream values should be normally distributed, have a linear relationship and same variance around the regression line; and (3) the streams should not have outliers, values that do not follow a similar pattern as the rest of the data. Moreover, PCC is applicable only to numeric values [31]. The correlation involving data of other types (i.e., strings) requires their mapping to numeric values.

In realistic use cases, it is unavoidable for the data to not have missing values, non-normal distribution or outliers. There are several ways to overcome this and still have an interpretable PCC value: (1) pairwise missing values - compute the correlation using the non-missing streams' values; this results in a partial correlation coefficient [31]; (2) list-wise deletion - compute the correlation only using observations with non-missing values for both streams [31] (Vertica, a unified analytics platform, to discover correlated features, employing list-wise deletion for missing values in the input pairs [69]); (3) replace the missing values by either means among the adjacent values or by constants. Each of these approaches are easy to implement on an FPGA, at the cost of one clock cycle increase in latency. In the design we explore in this chapter, we have not included this feature as it has no impact on the overall result. To assess the normality of stream value distribution, one can employ either histograms [114] or the Jarque-Bera test [117], with the former being successfully implemented on FPGAs [114]. Various techniques exist for outlier detection, ranging from statistical methods such as Z-score and Mahalanobis distance to machine learning approaches such as clustering or support vector machines. However, for FPGA implementations aiming to maintain 100 Gbps rates, moving average or exponential smoothing techniques are more appropriate [116].

Related Work. In databases, correlation has been used interchangeably to capture three different concepts. The first is the semantic relationship (e.g., a functional dependency) between columns. Hermit [225], Correlation Maps [133], CORDS [100], BHUNT [33], CORADD [134] use attributes' semantic correlation to improve indexing, query execution, and query optimization [155] performance. The second concept uses the two attributes covariance to model selectivity for query optimizers [45]. And the third models the relationship between pairs in time

series analysis: BRAID [195] for lag correlation; StatStream [244], Mueen et al. [169], and Li et al. [151] for longest-lasting correlated subsequences; and Wadjet [194] for outlier identification. The last two concepts employ PCC in a CPU based streaming context. Similar to them, we focus on bounded data streams, but adopt newer technologies, e.g. FPGAs, to compute the Pearson correlation coefficient. Moreover, we enhance a network communication link with compute capabilities. As noted, correlation computation on the CPU is expensive and is typically approximated or limited to specific data segments (i.e., Xiong et al. [231] opt to focus on Zipfian distributed datasets for large number of streams and compute PCC for only a subset of pairs; the same applies for Zhang and Feigenbaum [235] to find correlation among large datasets).

Correlation on Heterogeneous Architectures. FPGA-based correlation implementations have been proposed for image processing [137], OFDM (orthogonal frequency division multiplexing) timing synchronization [181], and digital correlation processors [11]. Image correlation is different from PCC by computing the difference in pixel values and energies between two images and similar to it because is based on additions and multiplications. Nevertheless, image processing correlation uses 11×11 window values, leading to computations over 121 pixels, which is much smaller than our target streams' lengths. OFDM timing synchronization [181] aims to reduce DSP utilization by replacing multiplication operations with shift-and-adds, resulting in approximated results. In contrast, our approach focuses on analyzing classic data types, providing exact results, and preserving data representation. There are limited customized GPU-based correlation implementations due to data movement overhead and GPU's constrained memory capacity. Significant speedup of 28× to 38× is achieved in PCC computation for matrix sizes ranging from 4096×16 to 12288×64, utilizing floating-point representations for sequence database search [37]. Their assessment considered both computation time and data transfer time to and from the GPU memory. However, the GPU's memory limitations constrained the maximum analyzed size, and the execution time was in the order of seconds, falling short of `AMNES`' performance and architectural flexibility. Our system is different from these previous ones by offering a single pass over the data, a flexible number of streams, and programmable fixed size data types. Conversely to this previous work, we focus on computing the PCC on data streams and operating at network line rate so that the design can be directly deployed on data arriving from the network without CPU involvement. We show that we can compute the correlation among up to 64 data streams (2016 PCC values) with several orders of magnitude performance gains over existing solutions, and that our design can be generalized to compute correlations among thousands of streams. This ensures that the design remains valid as networks improve and higher bandwidths are provided between CPUs and accelerators (e.g., Intel CXL interconnects).

3.2.2 Remote Direct Memory Access (RDMA)

As the network underlying protocol, we use RDMA over Converged Ethernet (RoCE v2), with the entire network stack [80] being deployed on an FPGA-based smartNIC [139] that we test over a 100 Gbps HACC cluster [13]. On the FPGA-based smartNIC, `AMNES` acts as a bump-in-the-wire accelerator, being placed between the network stack module (ensures the communication of the FPGA with the RDMA network) and the PCIe module (ensures the communication of the FPGA with the host CPU). For testing purposes, we use the low-latency, one-sided RDMA operations, namely the write primitive. The computational kernel is placed at the receiving node, and the correlation coefficients are computed as data is arriving through the network from an RDMA write requests. Although not explored in the chapter, the same could be done on the sending node to compute the correlation near to the data source. The same behavior is expected from `AMNES` if it is utilized together with the read primitive (computing correlation while data is received over the network, after sending an RDMA read request). By utilizing the read operation, the communication latency increases, since one more network trip is required before data gets sent over the network. The concept remains the same for the two one-sided RDMA operations, with the overhead being completely independent of the correlation computation.

Related Work. RDMA has gained prominence in data centers, finding applications in distributed systems [62, 3, 184, 177], databases [233, 142, 193, 220], cloud storage [62, 76, 147] or in-network data analytics [201]. StRoM [201], a system leveraging smartNICs with RDMA support, pioneered the integration of compute capabilities into the RDMA network stack, demonstrating the computation of the Hyperloglog (HLL) cardinality approximation algorithm while data traverses the network. Similarly, we illustrate how a correlation engine can be placed on the smartNIC without impacting network performance. Unlike StRoM, which hashes input values to dissociate input data representation from the algorithm's internal structure, `AMNES` retains the original data representation for processing. Farview, another FPGA-based smartNIC system, utilizes RDMA to offload query operators to a network-attached DRAM module, achieving performance comparable to local memory [138]. Farview serves as a potential deployment example for `AMNES` in future data centers.

Beside flexibility and high availability, cloud computing offers a pool of heterogeneous hardware resources: compute - conventional and specialized architectures, different storage types, and network infrastructures with high throughput and low latency. Its specialized architectures have allowed operations and algorithms normally running on the CPU to be offloaded to hardware accelerators (i.e., GPUs, TPUs, FPGAs, smartNICs, or ASICs) with high-speed networks having

shifted the data processing bottleneck from the network to the CPU [233, 167], making the idea of offloading non-critical operations even more appealing. Research has addressed the many facets of correlation, but there is no work done on offloading the operation to a smartNIC that processes the data as it arrives from the network stack. We address this gap by attaching a novel correlation compute module to an FPGA based smartNIC with an RDMA network connection.

3.3 Correlation Engine Design

In this section, we present AMNES design and focus on its two main components illustrated in Figure 3.3: (1) the accumulator engine (ACC Engine), and (2) the coefficient engine (COEFF Engine). The *pre-* and *post-* processing modules prepare the data either to enter AMNES's compute engines (i.e., augment the data with a control signal marking the last element to be analyzed) or to be sent to the host CPU (i.e., combine the coefficient results to fit into a cacheline). The BRAM temporally stores the results generated by the ACC Engine before being used by the COEFF Engine. The URAM and HBM (High Bandwidth Memory) memories illustrated in the figure are considered for design generalization purposes.

3.3.1 System Overview

We consider as *a stream* a continuous finite flow of fixed-size data items entering AMNES's compute engines. The compute granularity of the design is at cacheline level (64 B - 512 bits) leading to multiple streams to be analyzed in parallel, with each data item in the cacheline being *a value of a certain stream*. The maximum number of streams (M) analyzed in parallel depends on the data item representation ($Width_{data}$), i.e., 64 streams for 8 bits, 32 streams for 16 bits

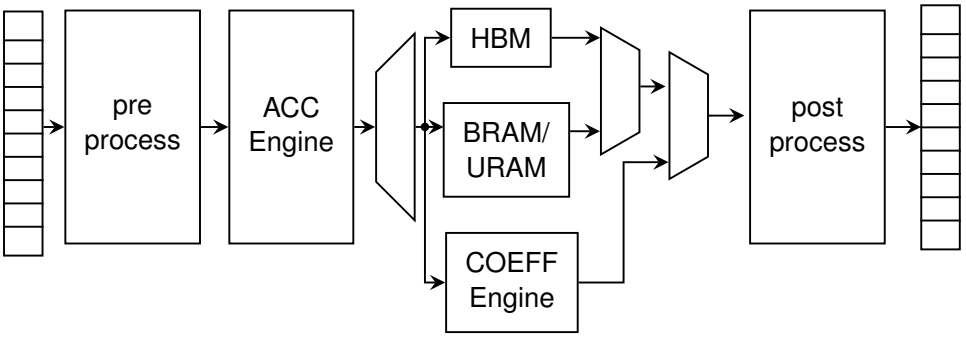


Figure 3.3: AMNES block diagram.

and 16 streams for 32 bits data items, respectively. Table 3.1 summarizes the symbols used in describing the design.

AMNES computes the PCC from the sufficient statistics as required by Equation 3.2. This comprises obtaining, for each stream s_i, the sum of elements–$S_{e_i} = \sum_{t=0}^{N-1} s_{i_t}$ and the sum of squares–$S_{sq_i} = \sum_{t=0}^{N-1} s_{i_t}^2$, as well as, for each unique pair of streams, the sum of products–$S_{p_z} = \sum_{t=0}^{N-1} s_{i_t} s_{j_t}$ with $i < j, i, j \in [0, M)$ and $z \in \left[0, G_p\right)$. After gathering all these statistics, the accumulated values are used to obtain the PCC between the pairs. AMNES's design takes as input M streams with N data items each, and produces $M(M-1)/2$ correlation coefficients, one coefficient for each unique pair of distinct streams. Note that the commuted pairs (s_i, s_j) and (s_j, s_i) are not differentiated as they yield the same coefficient. As depicted in Figure 3.3, the design splits into two parts: the backend part (accumulators – ACC Engine) and the frontend part (coefficients – COEFF Engine). The backend part gathers the sufficient statistics, whereas the frontend computes the coefficient values. Initially, AMNES has been implemented using C++ in Vivado HLS (High Level Synthesis) as a customizable streaming kernel, but the tool limitation got reached for 8-bit data representation. We have upgraded the design to Vitis HLS and deploy it on the FPGA as a compute kernel.

The implementation's challenge is to obtain processing pipelines for each of the engines that guarantee an *initiation interval* of 1 ($II = 1$), i.e., at every clock cycle, the compute engine can consume one cacheline of data items from M parallel streams. An FPGA pipeline is similar in concept to a pipelined processor architecture, with each stage of the pipeline executing a different operation, thus enabling concurrent execution of tasks. On the FPGA, the individual stages are separated by registers. While a deeper pipeline with more stages implies a higher processing latency in terms of clock cycles, its more fine-granular segmentation into stages will reduce the most critical signal path and lead to a higher operational clock frequency and throughput.

Table 3.1: Symbols defining basic design parameters.

N	Total number of elements of each stream.
M	Number of streams in a cacheline ($M \geq 2$).
G_p	Number of unique pairs between M streams.
s_i	Stream belonging to a cacheline, $i \in [0, M)$.
s_{i_t}	Data items of the stream s_i, $t \in [0, N)$.
L_{ACC}	Latency of the ACC engine [clock cycles].
L_{COEFF}	Latency of the COEFF engine [clock cycles].
$Width_{data}$	Input data representation [B].
$Width_{ACC}$	Accumulator representation [B].

3.3.2 The ACC Engine

The backend part gathers the sufficient statistics in parallel for M streams through a network of accumulators and multiply-accumulate (MAC) units. For M streams, M accumulators are required for sum of elements–S_e, and $M(M+1)/2$ MAC units are required for sum of squares–S_{sq} and sum of products–S_p, M units for S_{sq} and $M(M-1)/2$ units for S_p. The number of MAC units has a quadratic dependency on the number of streams that are analyzed in parallel. Table 3.2 shows the number of MAC units in dependence on the data types we consider for our implementation, i.e., 8 bits, 16 bits, 32 bits. In C++, each type of sum (S_e, S_{sq}, S_p) is associated with a class that exposes a set of functions that act upon the data sent to them. Since S_e and S_{sq} can be computed independently for each stream in the cacheline, `AMNES` associates objects from these two classes to each of the streams of the cacheline ($CORR\_SUM < TR > sums\_elements[M]$; and $CORR\_SUM\_SQ < TR > sums\_squares[M]$;). The snippet of class $CORR\_SUM$ is illustrated in Listing 3.1, where TR represents the accumulators data type, and TV represents the input values data type. The accumulator's loop-carried dependency is hidden in the iterative calls, cycle by cycle. S_p depends on the values coming from all the streams, so only one object ($CORR\_SUM\_P < TR > sums\_products$;) is associated from this class to all the streams included in a cacheline.

ACC engine latency lower bound (L_{ACC}) is given by the latency of the multiply operator, e.g., 3 clock cycles. Vitis HLS might introduce a few more cycles as latency on top of this lower bound for larger number of streams (e.g., 64). The bit representation of the stream's items is customized via $Width_{data}$ parameter, whereas the bit representation for the accumulators associated with each sum is customized via $Width_{ACC}$ parameters. In our implementation, the choice of $Width_{ACC} = 2 * Width_{data}$ accommodates stream lengths of up to 2 millions items per stream, with the consideration that some values are repeated. Since the FPGA offers customizable data width representations, Table 3.3 analyzes maximum bit representations of supported unsigned integer representations for accumulators when no pre-processing is applied for corner cases.

FPGAs are known for their flexibility in terms of customizable widths for different data types: from integer and fixed-point to floating-point representations; with the latter being the most resource intense and slowest of the three (i.e., modules working with floating-point values have a lower operating frequency than the modules working with integer or fixed-point values).

Table 3.2: ACC resources characterization for M streams.

Data Width [bits]	Streams	Accumulators	MAC Units	AMNES Op. Freq. [MHz]
32	16	16	136	300
16	32	32	528	250
8	64	64	2080	190

Listing 3.1: Sum of elements class.

```
1  template<typename TR>
2  class CORR_SUM {
3      // accumulator to store the sum of elements
4      TR accu = (TR)0.0;
5
6  public:
7      template<typename TV>
8      void accumulate(TV value){
9          accu += value;
10     } // accumulate()
11
12 public:
13     TR read_accumulator() {
14
15         TR const result_value = accu;
16         accu = (TR)0.0;
17         return  result_value;
18     } // read_accumulator()
19 }; // CORR_SUM
```

Since our focus is on relational data and potentially machine learning systems that use fixed-point or low-precision representations, our design centers on integer and fixed-point representations. Moreover, on FPGA, a fixed-point value is stored as an integer that is scaled by a specific factor, leading to similar resources and operating frequencies for both representations. In fact, the FPGA resource consumption (i.e., LUTs, FFs, DSPs) is slightly smaller when fixed-point representation are used instead of the integer ones.

Table 3.3: AMNES ACC width analysis.

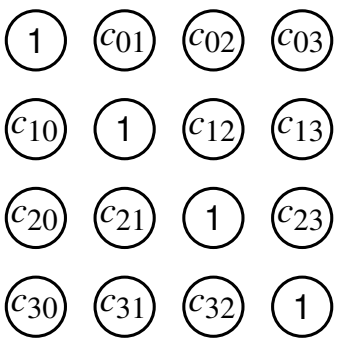
Data Width [bit]	All Values	$\text{Width}_{\text{ACC}}$ for 1 million [bit]	$\text{Width}_{\text{ACC}}$ for 2 millions [bit]
32	1	21	22
	$2^{32}-1$	85	86
16	1	21	22
	$2^{16}-1$	52	53
8	1	21	22
	$2^{8}-1$	28	29

3.3.3 The COEFF Engine

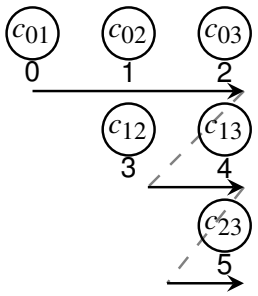
The COEFF Engine takes all the sufficient statistics previously gathered and generates G_p floating-point values representing the Pearson correlation coefficient (c_{ij}) between the unique stream pairs, (s_i, s_j), with $i < j$ and $i, j \in [0, M)$. The frontend computation is triggered once all the input data from the M streams has been consumed by the ACC Engine. When multiple streams (attributes) are correlated, a square matrix of correlation coefficients outputs is usually presented as in Figure 3.4a. This matrix exhibits a diagonal line consisting of '1' values, representing the correlation of each stream with itself. The remaining

portion of the matrix is symmetrical around this diagonal line, accounting for each pair of streams being considered twice (e.g., (s_i, s_j) and (s_j, s_i)), and resulting in identical correlation coefficients ($c_{ij} = c_{ji}$). Consequently, only one half of the matrix (either the upper or lower triangle) contains meaningful results, while the other half comprises duplicates or '1' values.

The COEFF Engine computes only the useful results, no duplicates, and employs techniques from matrix parsing in order to achieve an II of 1. We parse the indeces of G_p coefficients as if parsing the upper triangle of the square matrix as pointed by the arrows in Figure 3.4b following the index values from 0 to 5. Each index value is associated to two sub-indices i and j, where i represents stream s_i and acts as "parsing each row of the matrix", and j represents stream s_j and acts as "parsing each column of the matrix". For each circle (c_{ij}) in Figure 3.4b, we retrieve from the temporally internal storage the sum of elements (S_{e_i}, S_{e_j}) and sum of squares (S_{sq_i}, S_{sq_j}) associated with each stream, and the sum of products associated with their pair $S_{p_{ij}}$ (located at an address given by the index value) in order to compute the correlation coefficient. This operation is sequential and employes floating-point arithmetic to generate the correlation coefficients, which leads to a long pipeline. COEFF engine latency lower bound (L_{COEFF}) is 120 clock cycles. The difference in latency between the two engines arises from their design particularities. The ACC engine design consists of multiple pipelines that work in parallel and lead to a low latency result. The COEFF engine design consists only of one long pipeline that generates results after data have moved through the entire pipeline, leading to a higher latency. Obtaining an $II = 1$, instead of $II > 1$, resumes in the end to trading resources. For systems aiming to leverage the extremely low latency of the ACC Engine and a smaller resource footprint, only the backend can be deployed on the FPGA, with the correlation coefficients being computed on the CPU when the pair of streams are queried for their corresponding PCC.



(a) Correlation coefficient matrix for 4 streams.



(b) COEFF Engine - Index parsing for 4 streams.

Figure 3.4: Correlation matrix vs. COEFF engine.

Table 3.4: Vivado HLS vs Vitis HLS working frequencies for an II=1.

Data Width [bits]	Vivado HLS		Vitis HLS	
	Working Frequency [MHz]	Theoretical Throughput [GB/s]	Working Frequency [MHz]	Theoretical Throughput [GB/s]
32	250	16	300	19.2
16	200	12.8	250	16
8	No RTL generated.		190	12.1

3.4 FPGA Implementation

`AMNES` has been implemented in C++ as both a Vivado HLS (v2020.1) and a Vitis HLS (v2022.1) compute kernel. The performance disparity demonstrated in Table 3.4 between the two implementations has clearly pointed out the superiority of the second implementation as the preferred choice. We have deployed the Vitis HLS compute kernel on the FPGA together with Coyote [139], an open source FPGA shell. Coyote establishes streaming interfaces between both the DMA (Direct Memory Access)/Bridge Subsystem for PCI Express® [228] or an RDMA network stack and the compute kernel and implements the virtual memory management and the synchronization with the host CPU.

Besides the AXI4-Stream interfaces (`hls::stream<ap_axiu<512,0,0,0>>`) to stream data into and out of the kernel, the compute kernel also exposes an AXI4-Lite register interface (`s_axilite`) that allows software to access kernel configuration and parameter data. We use these registers to program the number of items the correlation is computed on, and the number of streams. The memory address results are sent to is handled by the Coyote shell. Both the `AMNES` kernel and the Coyote shell offer versatility across AMD Alveo's portfolio of data center accelerator cards (i.e., U50, U55C, U200, U250, U280). We have deployed `AMNES` together with Coyote on three of these cards, U250, U280 and U55C, and achieved a maximum kernel operating frequency of 300 MHz.

The flow of the implementation is illustrated in Figure 3.5. Even though the cacheline accommodates from 16 to 64 streams for our engine, we illustrate the implementation only for 4 streams (s_0, s_1, s_2, s_3) for simplicity. Nevertheless, the insights can be extrapolated to the number of streams that fit into a cacheline for any given data representation. When a new cacheline arrives, the pre-processing stage will augment it with an asserted last signal in the case that the cacheline represents the last elements of the streams, otherwise the last signal remains deasserted. The implementation of this stage is combinatorial and takes only one clock cycle. The function call to the ACC engine encapsulates through a lamba expression the partitioning of the cacheline into individual stream values. These values are used to simultaneously update the mesh of accumulators and multiply-accumulate units that compose the ACC engine as illustrated in Figure 3.5. The depth of the multiply-accumulate pipeline of 3 stages is also projected onto the plain accumulators, which would otherwise only use a single clock cycle, so as to match all the operational latencies. The accumulator state in the ACC engine is kept in the FPGA's fabric registers which consists of: (1) CLB registers; (2) CLB LUTRAM as a shift register LUT; (3) ILOGIC and OLOGIC, which are

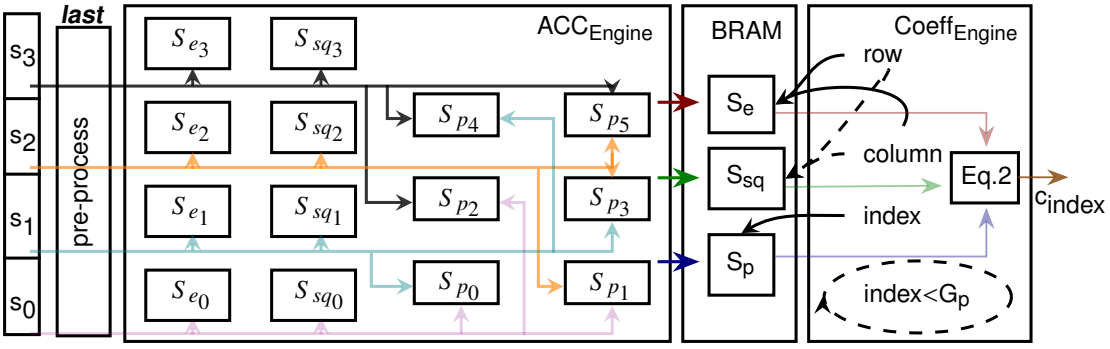
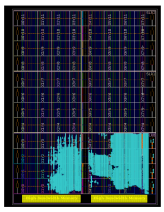


Figure 3.5: AMNES implementation.

synchronous elements for capturing/sending the data that comes into/gets out of the FPGA through the I/O block, respectively; (4) DSPs; and (5) BRAMs. DSPs and BRAMs are chosen if the register is associated to arithmetic or memory functionality. Even if DSPs should have been the choice for our design since the ACC Engine is a heavy arithmetic unit, the HLS tool maps the state storage to CLB registers, with the DSP's registers being used only for pipelining, and no accumulation state storing. For example, for 32-bit data representation, 136 MAC units are required. Since the accumulator representation is 64-bit, 4 DSPs instances are necessary for each MAC unit, leading to a usage of 544 DSPs for the ACC Engine. Beside DSPs, each MAC unit uses 34 CLB registers, 6 Carry8 units and 47 LUTs as logic.

When an asserted last signal is encountered, all the obtained values in the ACC engine are read and temporally moved to the internal memory of the FPGA, namely to BRAM. This temporary storage enables the instant reset of all accumulators to '0' such that new incoming data can be processed immediately. The COEFF Engine reads the BRAM addresses indicated by the row, column for S_e, S_{sq}, respectively, and the memory location indexed by S_p. For each $index < G_p$, five values are retrieved and used to compute the floating-point correlation coefficient associated to the index. The COEFF Engine leverages the power of HLS to implement Equation 3.2 from a regular high-level floating-point expression. The way we parse the index for unique stream pairs ensures an II=1, meaning that the COEFF engine does not put back pressure onto previous modules. Other parsing methods as index loop unrolling result in an II=15 for this engine. The correlation coefficients are collected and sent to the host CPU in the post-processing stage.

Implementation - resource consumption. In the following, we differentiate between the FPGA resource utilization required for the U55C board among the three data representations we analyze for AMNES using Vitis HLS, 32 bits in Figure 3.6, 16 bits in Figure 3.7, 8 bits in Figure 3.8. The quadratic grows of the MAC units number is visible between Figure 3.6 and Figure 3.8. The high SLR occupancy density determined also the drop in the working frequency of the compute kernel.

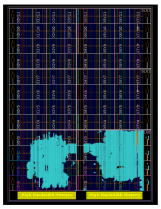


(a) SLR occupancy.

	DSP	FF	LUT	BRAM
AMNES	783 [~ 8.67%]	55'503 [~ 2.12%]	41'283 [~ 3.16%]	21 [~ 1.04%]
U55C Total	9'024	2'607'360	1'303'680	2'016

(b) Resources.

Figure 3.6: AMNES- 32 bits implementation and resource utilization.

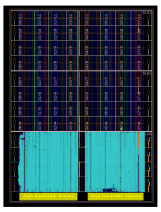


(a) SLR occupancy.

	DSP	FF	LUT	BRAM
AMNES	734 [~ 8.13%]	65'413 [~ 2.5%]	42'242 [~ 3.24%]	21 [~ 1.04%]
U55C Total	9'024	2'607'360	1'303'680	2'016

(b) Resources.

Figure 3.7: AMNES- 16 bits implementation and resource utilization.



(a) SLR occupancy.

	DSP	FF	LUT	BRAM
AMNES	2274 [~ 25.19%]	147'346 [~ 5.65%]	89'644 [~ 6.87%]	21 [~ 1.04%]
U55C Total	9'024	2'607'360	1'303'680	2'016

(b) Resources.

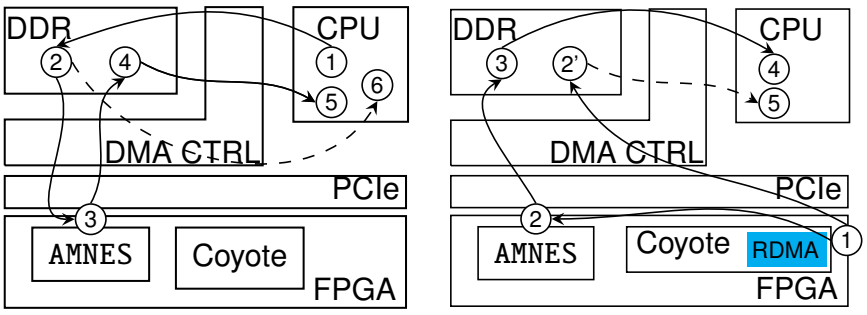
Figure 3.8: AMNES- 8 bits implementation and resource utilization.

3.5 Experimental Evaluation

In this section we present the results obtained when AMNES is deployed as a co-processor or as bump-on-the-wire on the U55C data center card, the two CPU-based baselines that we have developed to compare with the FPGA implementation together with the synthetic dataset we use for testing. Beside the CPU-based baselines, we retrieve the compute time required to compute the Pearson correlation coefficient for relational databases operators, and GPU-PyTorch.

3.5.1 Setup

We have evaluated AMNES on the AMD Heterogeneous Accelerated Compute Cluster (HACC) deployed at ETH Zurich using the U55C data center accelerator card with the FPGA in two configurations: (1) as a co-processor–data to be correlated is produced by the host CPU and resides in its attached memory, and (2) as a smartNIC–data is produced by a remote CPU and moved via RDMA-Write operations. The two configurations are represented in Figure 3.9 and comprise the FPGA and host CPU as compute units. The host CPU is also referred to as local CPU to differentiate between the two CPUs (remote–sending the RDMA-Write requests and local–receiving the RDMA-Write requests) in the RDMA experiment. In both cases, the FPGA is connected to the host CPU via a PCIe Gen3x16 link. The smartNIC configuration differs from the co-processor one by enabling the RDMA stack in the Coyote framework on the FPGA in Figure 3.9b. For each of the configurations, we compare the FPGA performance with the local CPU baseline performance in terms of throughput and latency. Irrespective of the configuration, the local CPU baseline reads the streams' values from its DDR memory; with the way the DDR memory gets populated being different for each configuration: (1) the local CP U populates the DDR memory before executing the correlation computation, and (2) the DDR memory is populated by a remote CPU via RDMA-Write transfers.



(a) Co-processor setup. **(b)** SmartNIC setup.

Figure 3.9: AMNES evaluation setup.

3.5.2 Software Baselines

We have developed two multi-threaded software baselines for the backend computation. One baseline collects the sufficient statistics in the same way as the FPGA implementation, and another extracts them from a matrix-matrix multiplication operation. If the first one is a standalone implementation, the second uses the Eigen Library (v3.4.0) [81], a C++ template library highly optimized for linear algebra, specifically for CPU-based matrix-matrix multiplication operations. We will henceforth refer to them as noEigen and Eigen, respectively. The Eigen Library offers explicit vectorization, compiler support (C++14), a straightforward integration with C++ code, adaptable matrix sizes and numeric types. It expects the stream values to be stored in column-major layout to optimize data partitioning between the processing threads.

Irrespective of the baseline, each thread receives a data chunk size inversely proportional to the number of threads used for computation. For the noEigen baseline, the values are distributed to the corresponding accumulators, whereas for the Eigen baseline, each thread maps the received values to a matrix whose number of rows is the number of streams plus one (M+1) and the number of columns corresponds to the length of the stream segment associated with the thread, i.e. N/thread_count. Each thread's matrix has an additional padded row of '1's to compute the sum of elements (S_e) for each stream. Sum of squares (S_{sq}) are obtained from multiplying each streams' row representation with its own column representation, whereas sum of products (S_p) are obtained when other streams' column representation is used. Figure 3.10 illustrates the matrix-matrix multiplication between the padded streams matrix A and its transpose B when one thread is allocated for 4 streams (s_0, s_1, s_2, s_3) with N items. The transposed matrix (B) is obtained using Eigen's *.transpose()* function. Still, the transpose is not materialized in memory, but returns a proxy object without doing the actual transposition. For a single thread, the sufficient statistics are directly extracted from the matrix product (i.e., the resulting matrix). For multiple threads, the matrix products of each thread undergo a subsequent summation process into a single (M + 1) square matrix to obtain the sufficient statistics. Then an upper module extracts them and computes the coefficients. We

$$\underbrace{\begin{bmatrix} s_{0_0} & s_{0_1} & \cdots & s_{0_{N-1}} \\ s_{1_0} & s_{1_1} & \cdots & s_{1_{N-1}} \\ s_{2_0} & s_{2_1} & \cdots & s_{2_{N-1}} \\ s_{3_0} & s_{3_1} & \cdots & s_{3_{N-1}} \\ \hline 1 & 1 & \cdots & 1 \end{bmatrix}}_{\mathbf{A}} \times \underbrace{\begin{bmatrix} s_{0_0} & s_{1_0} & s_{2_0} & s_{3_0} \\ s_{0_1} & s_{1_1} & s_{2_1} & s_{3_1} \\ \vdots & \vdots & \vdots & \vdots \\ s_{0_{N-1}} & s_{1_{N-1}} & s_{2_{N-1}} & s_{3_{N-1}} \end{bmatrix}}_{\mathbf{B}} \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} = \begin{bmatrix} S_{sq_0} & S_{p_{01}} & S_{p_{02}} & S_{p_{03}} & S_{e_0} \\ S_{p_{10}} & S_{sq_1} & S_{p_{12}} & S_{p_{13}} & S_{e_1} \\ S_{p_{20}} & S_{p_{21}} & S_{sq_2} & S_{p_{23}} & S_{e_2} \\ S_{p_{30}} & S_{p_{31}} & S_{p_{32}} & S_{sq_3} & S_{e_3} \\ S_{e_0} & S_{e_1} & S_{e_2} & S_{e_3} & N \end{bmatrix}$$

Figure 3.10: Eigen matrix-matrix multiplication operation between the padded streams matrix (A) and its transpose (B).

deploy the multi-threaded C++ implementation on the AMD EPYC 7302P 16-Core Processor@ 3.0 GHz base frequency, with each two adjacent cores sharing 16 MB of L3 Cache, and each core having 512 kB L2 Cache, and 32 kB of data and instruction L1 Cache.

The evaluation setup for the two use cases (co-processor, smartNIC) is illustrated in Figure 3.9, with (1) marking the source of the data values when entering our heterogeneous compute node composed of the host CPU and the FPGA. AMNES's working frequency depends on the chosen data width representation as it is stated in column *AMNES Operating Frequency [MHz]* (F_{op}) in Table 3.2. The degradation of the operational frequency is due to two factors. On the one hand, the number of MAC units required for the sum of products grows quadratically with the number of analyzed streams, leading to a larger fanout. The larger the fanout, the larger the time delay, so in order to avoid metastability hazards the frequency of the design has to be reduced. On the other hand, the spatial architecture of the FPGA has to ensure all the wiring that is required between the accumulators, multiply accumulators and the rest of the system. Given the working frequency and the fact that AMNES works at cacheline granularity, we can compute the theoretical upper bound of the throughput ($Th_{theoretical[GB/s]} = F_{op} * 64B/1000$). More specifically, 19.20 GB/s for AMNES's 32-bit data width and 300 MHz operating frequency, 16.00 GB/s for AMNES's 16-bit data width and 250 MHz operating frequency, and finally, 12.16 GB/s for AMNES's 8-bit data width and 190 MHz operating frequency.

Datasets, data types and data distribution. For evaluating AMNES, we use synthetic datasets with arbitrary precision (ap) integer data types representations: ap_uint_8, ap_uint_16 and ap_uint_32 [229], and vary the stream length between 1000 and 2 million items per stream. The number of streams we analyze in parallel is determined by the number of streams that fit into a cacheline (512 bits) for the given data representation, namely 64 streams for ap_uint_8, 32 streams for ap_uint_16, and 16 streams for ap_uint_32. We opt to analyze the unsigned integer data representations (ap_uint_x) due to their ability to stretch the FPGA resource utilization the most out of the fixed-size data representation selection: fixed-point decimal numbers (ap_[u]fixed) and signed (ap_int). The data volume transmitted across PCIe or RDMA networks remains constant (i.e., the bytes number) regardless of the data width representation.

Even if in the evaluation we opt for all the pipelines to have the same data layout (width and type), which can be a practical approach in the context of vector DBs and ML systems, AMNES is not restricted to this. The FPGA can be programmed to associate a different data type (integer or fixed-point decimal representations) and width to each pipeline (stream), e.g., for a 512-bit cacheline, 4 streams could be associated to 32-bit signed integer, 4 streams to 32-bit fixed-point decimal and 16 other streams to be associated to 16-bit unsigned. However, if floating-point values are to be employed, they have to be converted to fixed-point representation beforehand, using a dedicated AMD-Xilinx IP [14].

The normal data distribution condition is intrinsic to PCC's algorithm to guarantee a reliable result. AMNES efficiency is not impacted by data distribution since the system processes input data at every clock cycle and updates the underlying network of accumulators.

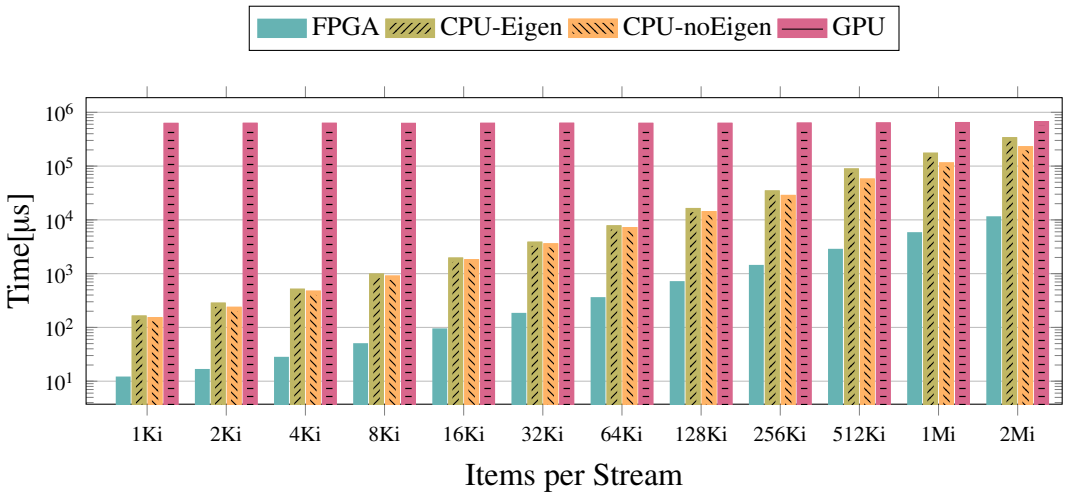


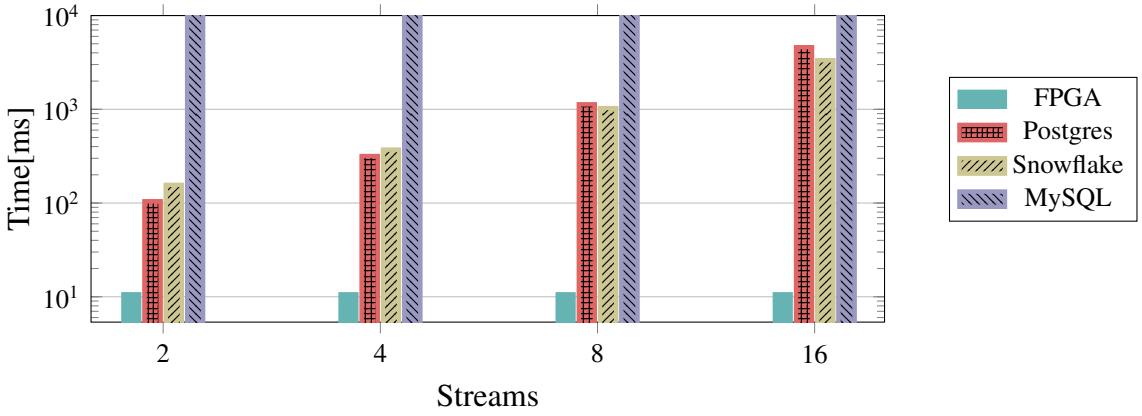
Figure 3.11: Compute time (log scale) of PCC for 16 streams on various platforms (32-bit integer data).

CPUs and GPUs are software-programmable fixed architectures. With the emerge of customized data types and reduced resolution representations, their inherent architectural advantage gets debatable [101]. Data center vendors (Microsoft, Amazon, Baidu) are now focusing on FPGA's programmable data-width capability for ML deployments in the detriment of fixed architectural paths [74].

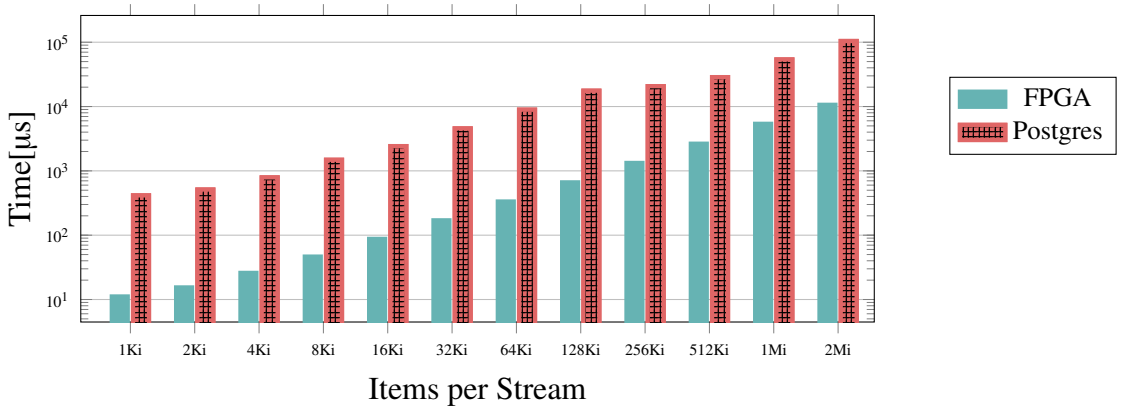
In Figure 3.11, we depict the compute time required on 3 individual platforms (FPGA, CPU and GPU) to obtain the PCC for 16 streams while varying each stream's length from 1 Ki to 2 Mi for 32-bit integer items. For the GPU, the application utilizes the PyTorch library (torch.corrcoef [182]) on a TITAN RTX Nvidia GPU, whereas for the CPU, we utilize the two baselines and allocate only one thread for compute. Although the FPGA operates at a lower clock frequency (MHz range) compared to the CPU and GPU, its spatial pipeline customization allows for parallel processing of multiple items, providing it with an advantage. On the other hand, the GPU's clear advantage is hindered by the data movement overhead.

3.5.3 Comparison with Relational Operators

To provide a performance reference from relational engines supporting PCC computation, we assess 3 engines: PostgreSQL, Snowflake, and MySQL. PostgreSQL and Snowflake offer a *corr(x,y)* aggregate operator (x and y-the two attributes), whereas MySQL queries for all Equation 3.2 items in the *SELECT* statement. Since DBMSs offer 2-by-2 correlations, we examine their efficiency from 2 up to 16 attributes for 2 million 32-bit integers in Figure 3.12a. For Snowflake, we use an *X-Small* deployment (single node, 8 cores) [203] and report only the execution time out of the total time (compilation + execution). The execution time is significantly smaller than the compilation time (2× to 30×). Irrespective of the DBMS, the PCC compute time for two attributes is already one order of magnitude larger than `AMNES`'s compute



(a) PCC for 2, 4, 8 and 16 streams (2 million 32-bit data items per each attribute (stream).



(b) PCC for 2 streams and 32-bit data representation with PostgreSQL.

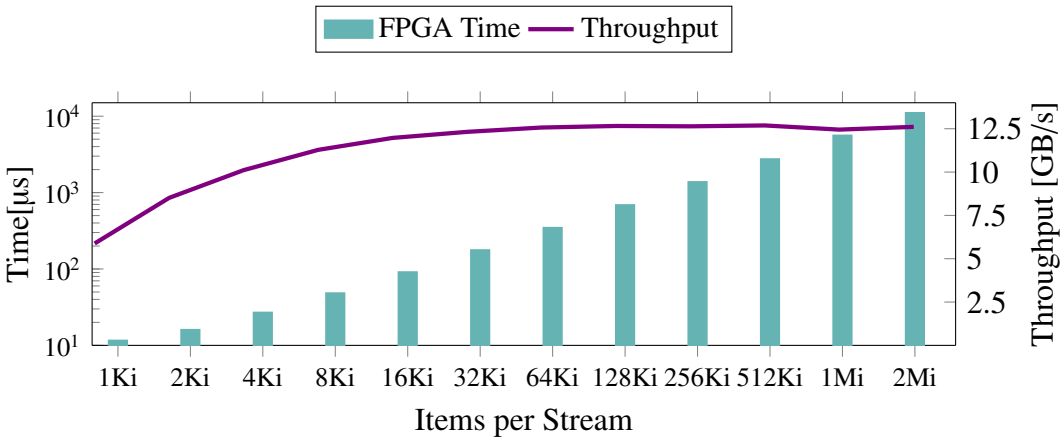
Figure 3.12: PCC compute time (log scale) using PostgreSQL, Snowflake, and MySQL.

time for 2 million items and 16 streams (if we maintain the system in its current state and only calculate two attributes, with the remaining 14 not being used). The increase in FPGA compute time in Figure 3.12b is attributed to the unused slots. The MySQL query complexity makes it not competitive with neither the other two DBs or the FPGA. Since PostgreSQL has the fastest compute time for 2 attributes, we analyze in Figure 3.12b its behavior for down to 1Ki items per attribute and observe that the compute time continues to be 5× larger than for `AMNES`.

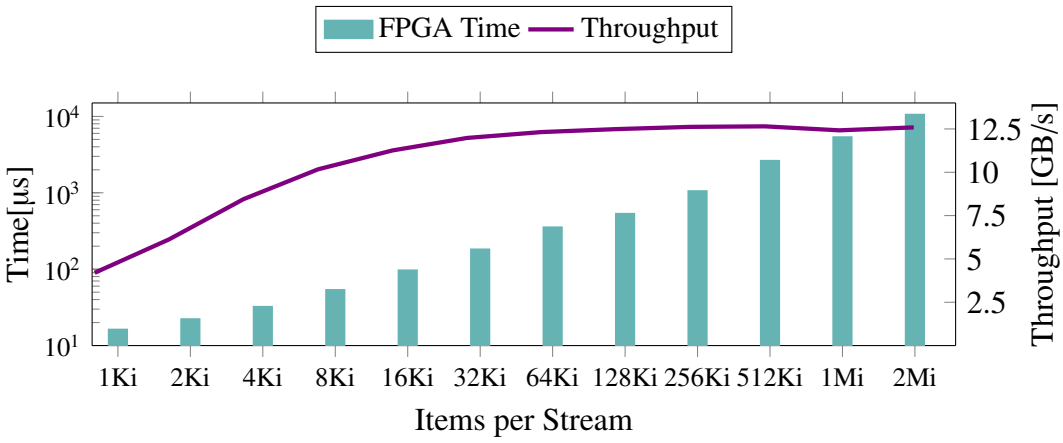
3.5.4 Correlation on a Co-processor

In Figure 3.9a, the data values that are streamed from the host's DDR to AMNES (②->③) via DMA-Write transfers are augmented with a last signal before entering AMNES compute engine, in the pre-processing stage. The last signal serves as a control accompanying the data in the compute engine, indicating the last value of each stream. Once the FPGA computes the PCC values, they are streamed back to the host's DDR (③->④) via DMA-Read transfers from where they can be further used (⑤). The performance measurements for the FPGA as a co-processor include the ②->③->④->⑤ data links and are illustrated in Figure 3.13, where each sub-figure is associated to a different data width and number of streams analyzed in parallel for cacheline granularity: 32 bits (Figure 3.13a), 16 bits (Figure 3.13b) and 8 bits (Figure 3.13c), respectively. For each sub-figure, the measurements include, beside the clock cycles needed to transfer the data over PCIe, the clock cycles required to collect the sufficient statistics and to compute the PCC values for each pair of streams. In Figure 3.13, it can be observed that the throughput saturates for 8-bit data at around 12 GB/s, which is close to AMNES's theoretical upper bound of 12.16 GB/s for this data representation. For 16-bit and 32-bit data, the throughput saturates at values around 12.5 GB/s, which are lower than AMNES's theoretical upper bounds of 16 GB/s and respectively 19.2 GB/s, but are matching the bandwidth exposed by the Coyote framework to the compute kernel. Coyote's setup cost dominates the measurement for small stream lengths but it becomes negligible for streams with more than 32K items, as it is illustrated by the throughput graphs in Figure 3.13. The lower working frequency of the compute kernel impacts latency measurements for 1Ki items per stream. Among the three data representations, the compute kernel operating at 300 MHz exhibits the shortest compute time (Figure 8a), while the one operating at 190 MHz has the longest time for the same data volume.

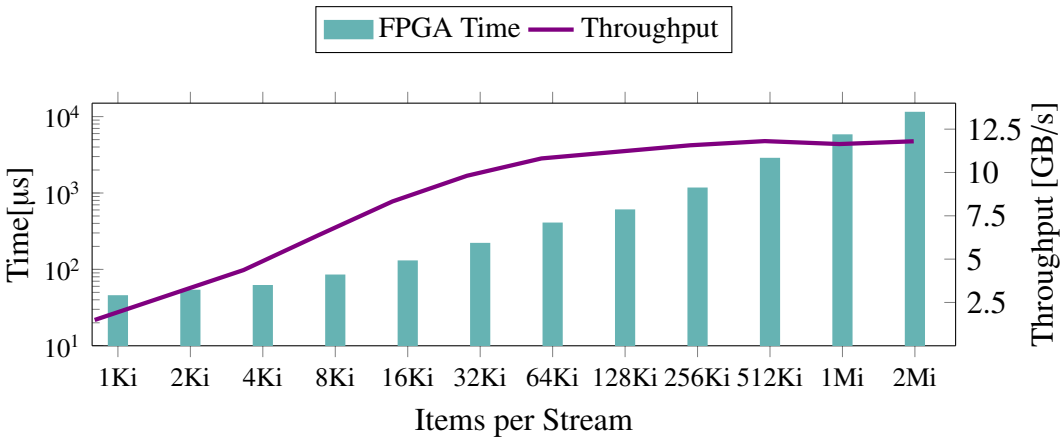
Regardless of the data width representation, the FPGA implementation outperforms the multi-threaded C++ baselines (Eigen or noEigen). The AMNES-like collection baseline (noEigen) performs better than the matrix collection one (Eigen) for single and dual thread deployments, but the Eigen baseline has a better overall performance due to dense matrix-matrix product optimizations, saturating at around 6 GB/s, 5 GB/s and 2.5 GB/s (Figure 3.14) for 16, 32 and 64 streams and 2 million items per stream, respectively. For each baseline, we report the throughput (Figure 3.14-lines) and compute time (Figure 3.14-bar) measurements. Compute time is primarily consumed by collecting the sufficient statistics, whereas the merge and PCC calculation times are much smaller. If the former gets damped when more threads are allocated for the task, saturating for 16 threads and more, the latter two have a similar behavior across all data representations.



(a) 16 streams [32-bit data].

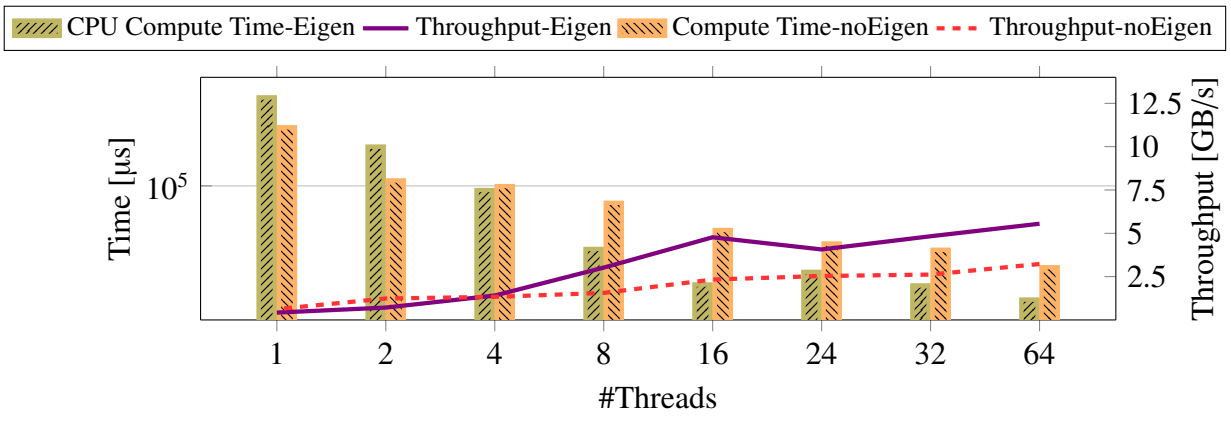


(b) 32 streams [16-bit data].

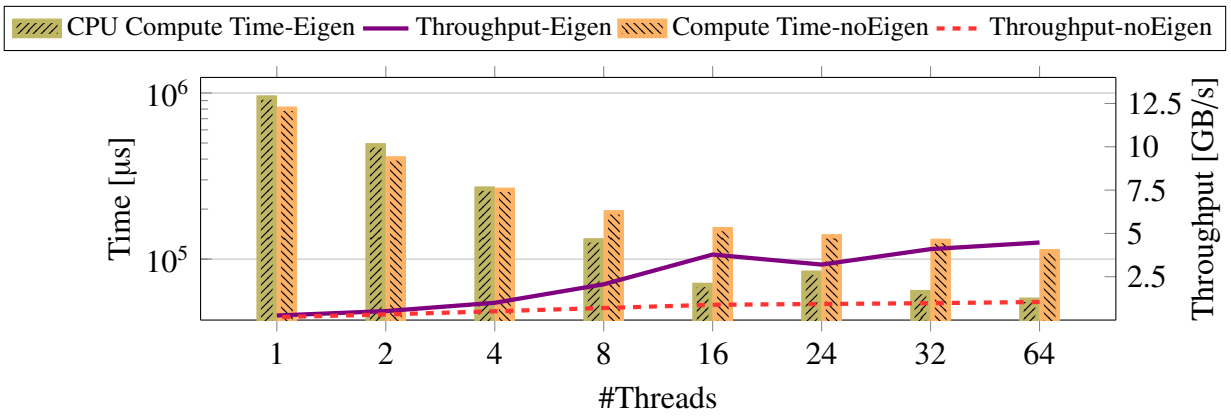


(c) 64 streams [8-bit data].

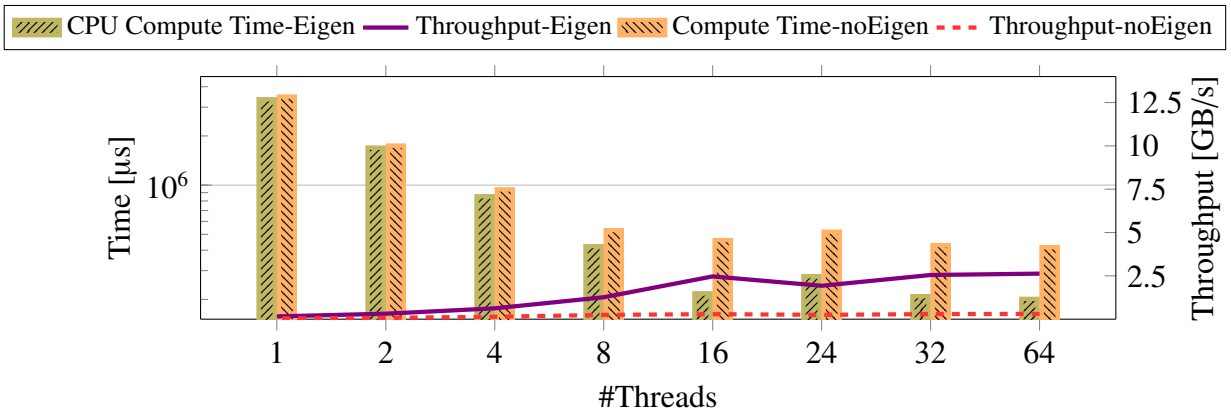
Figure 3.13: Compute time (bar, left y-axis, log scale) and throughput (line, right y-axis, linear scale) for RAM-RAM experiments for 16/32/64 streams analyzed in parallel on the FPGA; including *collecting* the sufficient statistics and computing *pcc* values.



(a) 16 streams [32-bit data].



(b) 32 streams [16-bit data].

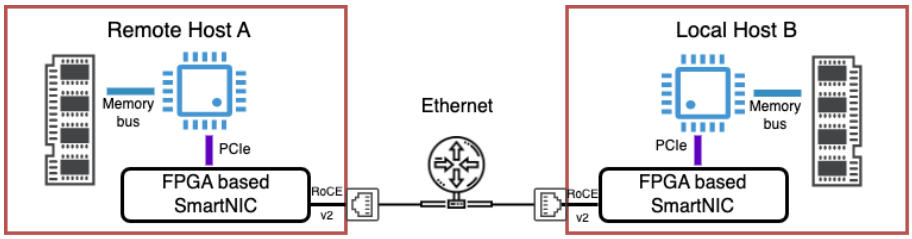


(c) 64 streams [8-bit data].

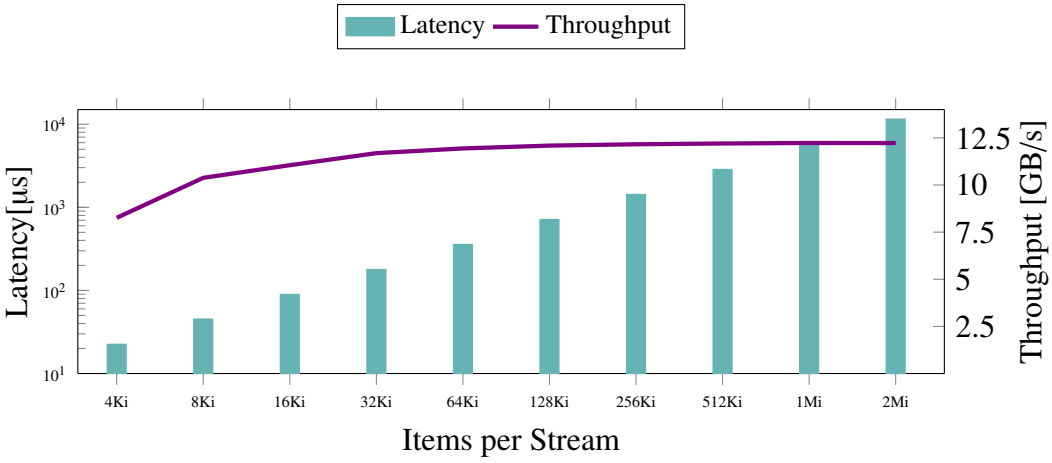
Figure 3.14: Compute time (bars, left y-axis, log scale) and throughput (line, right y-axis, linear scale) measurements for RAM-RAM experiments with 16/32/64 parallel analyzed streams on the CPU, using 2 million items per stream. We differentiate the compute time measurements between sufficient statistics collection by matrix multiplication (Eigen) and `AMNES`-like collection (noEigen). Compute time includes collecting, merging partial values from each thread, and computing the PCC.

3.5.5 Correlation on a SmartNIC

For this use case, we utilize the setup illustrated in Figure 3.15a with two CPUs (remote host A initiates the RDMA-Write transfers while local host B receives them) and two data center class FPGAs connected via a switch for RDMA traffic. The data values coming from the remote host are forwarded by the RDMA network stack simultaneously to the local CPU's DDR(②') and to `AMNES` (②) on the FPGA as illustrated by the links ①->② and ①->②' in Figure 3.9b. When the coefficients results are ready, DMA-Read transfers transfer them from `AMNES` to the local CPU's DDR (③->④). For the software baseline associated with the smartNIC setup, the Eigen-based application starts processing the values as soon as they have been placed in the local CPU's memory (②'->⑤), each thread having associated to it one or multiple slices of the data. The application initiates processing before the entire stream data is transferred, but the first values must reach the CPU's DDR for processing to start. The FPGA performance assessment includes the ①->②->③->④ data links, whereas the CPU baseline for the smartNIC setup includes ①->②'->⑤ data link together with the CPU processing.



(a) RDMA Setup.



(b) Latency (bars, left y-axis, log scale) and throughput (line, right y-axis, linear scale).

Figure 3.15: RDMA communication setup and performance.

We focus on the 32-bit data representation and evaluate the network setup by employing 32 KiB RDMA-Write transfers. To achieve streams length from 4Ki, 16Ki up to 2Mi for 16 streams, multiple RDMA transfers are needed. Considering the RDMA connection in Figure 3.15a, we distinguish between different measurable durations: (1) **network time** - the time from when the first write request is sent from the remote host till the last acknowledgment is received by it from the local CPU; (2) **data reception time** - the time from when the first data byte arrives to the local CPU's memory till the acknowledgment is sent to the remote CPU; (3) **data reception + correlation on CPU (x threads)** - the time from when the first byte arrives to the local CPU's memory till the correlation coefficients are produced (1 to 16 threads are allocated to compute the sufficient statistics using Eigen Library (v3.4.0) and a single thread is allocated for the PCC values computation); (4) **data reception + correlation on FPGA** - the time from when the first byte arrives to the local CPU's memory, until all the coefficients are transferred from the FPGA to the local CPU's memory via PCIe transfers.

In Figure 3.15b, we assess network throughput by adjusting the number of transfers exchanged between the remote and local CPUs. The number of transfers is varied from 8 for 4Ki items per stream, 2048 for 1Mi items per stream, to 4096 for 2Mi items per stream. This serves as a reference point for evaluating network communication overhead. The network throughput saturates when the number of transfers exchanged between the two CPUs exceeds 64. At this point, the throughput stabilizes at approximately 12.5 GB/s, which aligns with the PCIe saturation. This confirms that the Coyote framework does not impose any back-pressure on the network. We present the latency measurements as the *data reception time* instead of the *network time*. The data reception time includes the network time for all transfers except the first one. For 16 streams, the data reception time ranges from 21 µs for 4Ki items per stream to 11 ms for 2Mi items per stream. Both `AMNES` and the CPU baseline start data processing as soon as the data enters the system, in the FPGA and CPU's DDR, respectively, rendering the first network transfer time insignificant. Modifications to the Eigen-baseline allow processing of smaller data chunks than N/thread_count (i.e., 16 items/stream) without waiting for the entire transfer.

On the FPGA, analyzing 16 streams produces 120 coefficient floating-point values. This requires 8 DMA-Read transfers and only one PCIe transfer for a 4096 B payload size. This communication expense, combined with RDMA, incurs an overhead of approximately 1 µs. As shown in Figure 3.16 (rdma and rdma+corr FPGA bars), this overhead results in latency similar to solely receiving the data. We analyze stream lengths of up to 1Mi items per stream (Figure 3.16), which fully utilize the available bandwidth.

Implementing correlation on the host CPU incurs significantly higher time overhead compared to simply receiving the data (baseline). On average, it is 30× larger when a single thread (*rdma+corr CPU: 1 thread* bar in Figure 3.16) is allocated for getting the sufficient statistics and computing the coefficients. Allocating more compute threads reduces this overhead. For 8 threads, it is only 7× larger on average than the baseline (*rdma+corr CPU: 8 threads* bar in Figure 3.16). The compute time on the host CPU decreases on average by 1.62× when the number of threads allocated for the task is doubled. However, even with

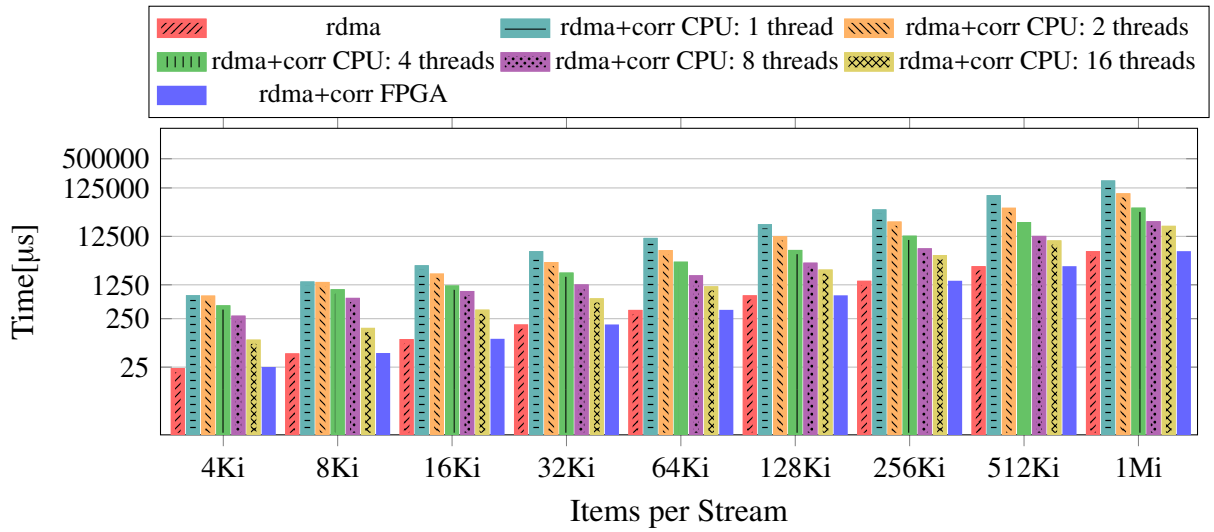


Figure 3.16: RDMA communication w/o correlation computation time on the CPU and FPGA (bars, left y-axis, log scale).

16 threads, the host CPU implementation is on average 4.4× slower than the FPGA implementation (i.e., `AMNES`). The FPGA incurs no back-pressure on the network by processing data at every clock cycle (II=1). Even if the host CPU implementation is slower than the FPGA implementation, it does not have a negative impact on the RDMA network performance since the CPU acts upon data already residing in its DDR memory and transferred there via RDMA-Write transactions that do not involve the CPU.

For a 32-bit data representation, the latency inferred by `AMNES` in the smartNIC setup is larger than the latency inferred in the co-processor setup with a $\Delta < 100\,\mu s$. This difference (Δ) becomes larger as the stream item number increases, since more RDMA-Write transfers are needed to transfer all the data.

3.6 Discussions

In this section, we discuss other systems' stream count and explain how `AMNES` can be generalized to more than 16/32/64 streams, why C++ HLS was the language of choice rather than a classic hardware description language, and how the use of the gathered sufficient statistics can be further extended.

3.6.1 Number of Streams

In our evaluation, we analyze `AMNES`' performance for a maximum of 64 streams in parallel. For reference, in Table 3.5 we summarize the parameters used in related work, covering the number of coefficients computed in parallel, the total number of streams/attributes considered, and the size of each stream.

Table 3.5: Number of attributes (streams).

System	Parallel Attributes	Total Number of Attributes	Items per attribute	PCC
AMNES	16-64	16-64	2 Mi	✓
BRAID [195]	2	59	100 Ki	✓
Joglekar et al. [121]	2	5	45 Ki	-
Hermit [225]	2	16 200	4 Mi 15 Ki	-
CORDS [100]	-	18	1 Ki - 64 Ki - 2 Mi	-
Wadjet [194]	-	5000	50 Ki	✓
EXORD$^+$[153]	-	8	90 Ki	-
COCOA [66]	2	22	100 - 356 Ki	rank
Joglekar et al. [121]	-	5	45 Ki	-

AMNES is the only one computing coefficients in parallel (16 to 64 while all others are just between pairs of attributes). Similarly, our experiments consider much larger data sets per stream/attribute than almost in all previous work, especially when compared to those computing PCC instead of simpler or approximated forms of correlation. Nevertheless, none of the systems in Table 3.5 analyzes the parallel computation of correlation, relying on the mere 2 by 2 attribute computation. BRAID [195] computes the lag correlation, analyzing 2 by 2 attributes with up to 100K items. Joglekar et al. [121] exploits correlation for expensive predicate evaluation and looks at up to 5 attributes and 41K items. Liu et al. [153] employ the knowledge of correlation for up to 8 attributes (i.e., a maximum of 28 correlation values) for query optimization. Wu et al. [225] do not mention how they compute correlation, but they design a secondary index mechanism and exploit correlation among 16 attributes with 4 Mi items for their sensor application or build up to 10 indexes from their 210 attribute stock dataset. CORDS [100] employs sampling to discover statistical correlations (150 correlated pairs) between database columns with sample sizes varying from 1K to 64K items and showing that for samples with 10Mi items the registered run-times are in the order of seconds.

3.6.2 Engine Generalization

Engine generalization expands AMNES capabilities to more than 16/32/64 streams by trading latency, resources, or complexity and is characterized by the extended parameters presented in Table 3.6. As before, each cacheline (512 bits) is composed of values from M different streams. Apart from these definitions, we introduce the notion of *batch of streams*. A batch of streams, or simply a *batch*, represents the number of cachelines needed to cover one value from each distinct stream. If the total number of streams to analyze is greater than the number of streams in a cacheline (S > M), then a batch of streams extends over $\lceil S/M \rceil$ cachelines. We focus on generalizing the ACC Engine due to its parallelizability and low latency.

Table 3.6: Symbols for design generalization.

S	Total number of streams ($S \geq 2$).
M	Number of streams in a cacheline ($M \geq 2$).
N	Total number of elements of each stream.
T_{CLK}	Clock period of the design.
L_{ACC}	Latency of the ACC Engine (measured in clock cycles).
T_p	Number of distinct pairs between S streams.
G_p	Number of distinct pairs between M streams.
C_{S_t}	Cumulative streams observed at time t ($C_{S_0} = M, t = 0$).
$State_{ACC}$	Accumulator state to maintain.

The number of accumulators required is given by Equation 3.3

$$ACC_{number} = \frac{S(S+3)}{2} \tag{3.3}$$

Once all the sufficient statistics are gathered, they can be sent to the host CPU and queried for the PCC value. We have identified two approaches to generalize the concept: (1) support a pre-defined total number of streams (S) that are time-multiplexed at the cacheline level as illustrated in Figure 3.17, solution referred to as *timeAMNES* $_{ACC}$, or (2) impose fixed hardware limitations of the engine (M) and run stream values multiple times through it as illustrated in Figure 3.19, solution refered to as *fixedAMNES* $_{ACC}$; the latter solution requires all the streams' values to be temporally stored in the internal or external FPGA's memory. We individually analyze each approach.

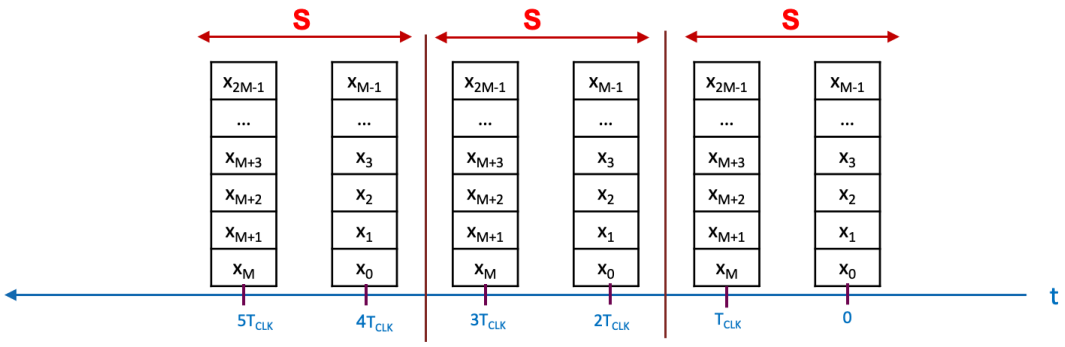
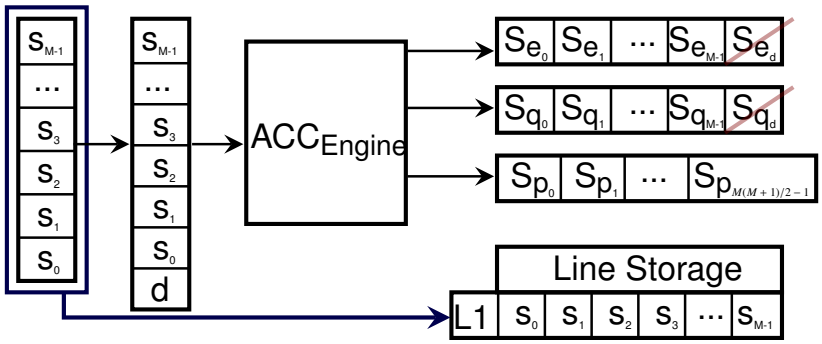


Figure 3.17: timeAMNES $_{ACC}$.

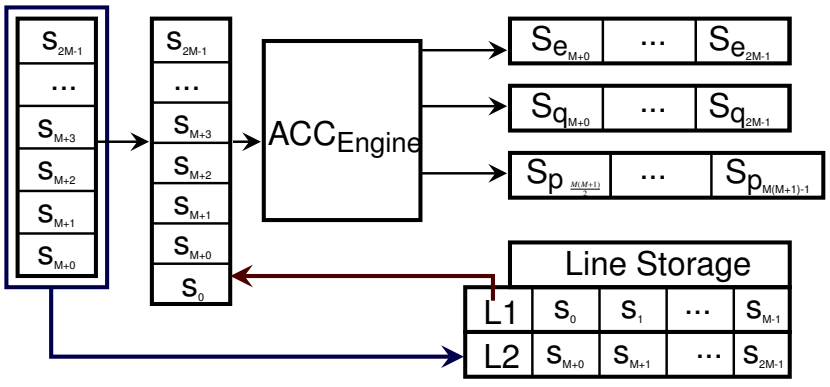
timeAMNES $_{ACC}$-Sufficient Statistics. Figure 3.18 shows the principle for timeAMNES $_{ACC}$ generalization. It differs from the basic engine by supporting (M+1) streams instead of M. The extra one stream is *a dummy stream* (d) at the beginning, when the batch's first cacheline arrives (Figure 3.18a). With the

arrival of subsequent batch cachelines, the dummy stream's place is taken by the values of each previously seen stream (Figure 3.18b).

The procedure is as follows: (1) one cacheline arrives and goes through the ACC Engine; (2) the ACC Engine computes the required sufficient statistics for (M+1) streams, and discards the sums that are part of the dummy component (for the first cacheline) or have already been computed (for the subsequent cachelines); (3) update BRAM storage for sum of elements (S_e), sum of squares (S_{sq}) and sum of products (S_p); (4) update line storage for the later use of line's values; (5) replace the dummy stream with the first element of the line storage for it to be sent together with the new incoming cacheline through the ACC Engine (Figure 3.18b); (6) repeat this process until all the components of the previous line storage have been combined with the new incoming cacheline. The line storage grows in a cumulative way, with cacheline elements being added to it until all streams' values in a batch have been seen. When a new batch of streams arrives, the line storage and the dummy stream are considered empty. We distinguish two actions for a data cacheline observed at a moment t: (1) updates - immediate action (Equation 3.4), and (2) holds - future action (Equation 3.5).



(a) timeAMNES $_{ACC}$, t=0.



(b) timeAMNES $_{ACC}$, t=1.

Figure 3.18: timeAMNES $_{ACC}$.

$$updates = \begin{cases} \frac{M(M+3)}{2} & , t = 0 \\ \frac{M(M+3)}{2} + M * C_{S_t} & , t \geq 1 \end{cases} \tag{3.4}$$

$$holds = \frac{S(S-1)}{2} - \frac{C_{S_t}(C_{S_t}-1)}{2} - \frac{(S-C_{S_t})(S-C_{S_t}-1)}{2} \tag{3.5}$$

For each stream i in a cacheline, the updates include: $S_{e_i}, S_{sq_i}, S_{p_i}$ for the distinct pairs that can be formed between the streams of the cacheline, and S_{p_i} for the pairs that have been on hold and have the second stream (operator) within the given cacheline. The holds are a metric for future actions and include all pair-wise sum of products that can be formed between current cacheline streams and batch streams, but are missing the second stream (operator).

Depending on the total number of streams, the accumulator state $State_{ACC}$ can be maintained on-chip or off-chip. The number of accumulators involved in the computation has a quadratic relationship with the total number of streams, leading to $State_{ACC} = \frac{S(S+3)}{2} * Width_{ACC}$ to have to be stored.

We extrapolate in Table 3.7 how many streams would fit in the internal memory (BRAM or/and URAM) for 64-bit accumulators for AMD data center accelerator cards. We also consider the deployment of the engine in a single FPGA SLR (Super Logic Region-a single FPGA die slice) instead of the entire FPGA in the last table's column. Each data center class FPGA has between 2 to 4 SLRs.

Table 3.7: AMD data center accelerator cards.

Board	BRAM [MB]	URAM [MB]	S_{BRAM}	S_{URAM}	$S_{SLR:BRAM+URAM}$
U50	6.048	23.04	1'228	2'398	1'754
U200	9.72	34.56	1'557	2'937	1'912
U250	12.096	46.08	1'737	3'392	1'853
U280	9.072	34.56	1'504	2'937	1'856
U55C	8.8625	33.75	1'486	2'903	1'883

timeAMNES$_{ACC}$'s latency in Equation 3.6 indicates the back pressure that will be induced for large S values, making this solution inappropriate for high rate communication links. For example for $S = 1000$ and $M = 16$, the total latency for a batch of streams is around 100 µs for a clock period of 3.33 ns.

$$Total_{latency} = L_{ACC} + \left\lceil \frac{S(S-M)}{2M} + 1 \right\rceil * T_{CLK} \tag{3.6}$$

fixed`AMNES` $_{ACC}$-Sufficient Statistics.

As opposed to the previous solution that time-multiplexes stream values, fixed`AMNES` $_{ACC}$ analyzes all the values from the streams covered in one cacheline (M) before moving to another set of streams as illustrated in Figure 3.19. This solution constrains the engine's compute and storage capabilities at M streams. If we have the total number of distinct pairs between S streams, $T_p = S(S-1)/2$, and the number of distinct pairs between M streams, $G_p = M(M-1)/2$, then the lower bound for the number of passes through the engine to obtain all the necessary pairs would be $T_p/G_p = [S(S-1)]/[M(M-1)]$. This lower bound is not a tight bound, and heuristics indicate more passes for full pair coverage are needed [8]. For example, for $S = 6$ and $M = 3$, 6 passes instead of 5 are needed in order to cover all the streams. fixed`AMNES` $_{ACC}$ requires all the streams values to be stored, so that the set of streams of one cacheline can be correlated with the set of streams of another cacheline, after all values of the previous set have been seen by the engine. This offers an upper bound for the supported number of streams that can be stored in the FPGA external memory for 32-bit data width representation: (1) for DRAM, a maximum of 2000 streams of 1 million items each or 250 streams of 8 million items each; (2) for HBM, a maximum of 16000 streams of 1 million items each or 2000 streams of 8 million items each. The latency of the pairing compute will depend on the number of groups that need to be formed in order to create all the pairs that are necessary. The advantages of this solution are that it will not add back pressure on the communication link and will require only the state of one engine to be maintained inside the FPGA. The solution's disadvantages will be: (1) all streams values need to be temporally stored in the memory and passed through the engine, with an increased latency perceived by the receiver of the data; (2) a data structure that tracks formed pairs, potentially solvable through edge clique covering [8, 53]. If we consider the streams being the vertices of a graph and the pair between each two streams being the edges, then all the unique pairs between S streams are characterized by a fully connected graph (complete graph) with S vertices. If the same analogy is applied to ACC Engine capabilities, creating a fully connected graph with M vertices, then the engine becomes a clique of the total number of supported streams, i.e.,

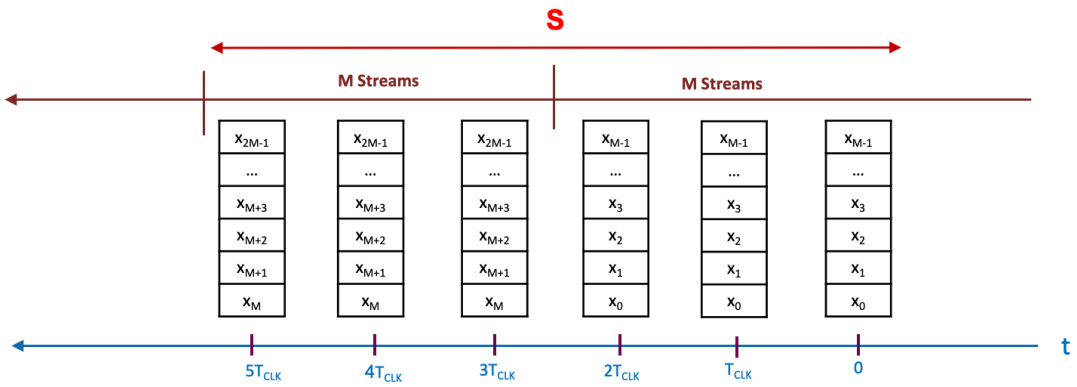


Figure 3.19: fixed`AMNES` $_{ACC}$.

the number of streams that can be correlated at one moment represents a subgraph of the complete graph with edges between every two vertices.

FPGA for AMNES. The FPGA application-oriented parallelism and scalability come at the cost of operating frequency. As it is observed in Table 3.2, the system's operating frequency decreases as the level of parallelism increases, going from 300 MHz (16 streams) to 190 MHz (64 streams), due to the large FPGA SLR's occupancy. By deploying time`AMNES` $_{\text{ACC}}$ that targets the entire FPGA, the operating frequency can decrease even more, making fixed `AMNES` $_{\text{ACC}}$ preferable if I/O bandwidth needs to be saturated. A concluding comparison between the two FPGA generalization solutions is given in Table 3.8, stating the advantages and disadvantages of each solution.

Generalization for the software baselines. We can also analyze the generalization of the two software baselines from the point of view of the number of operations required to gather the sufficient statistics. If we consider the symbols in Table 3.6, and we define the following operations: sum operations ($\sum$), product operation ($\prod$), and all the operations (Φ) the unification of the previous two, then gathering the sufficient statistics requires the following number of operations: (1) for noEigen method, the number of distinct operations is $\frac{S(S+3)}{2} * N\sum + \frac{S(S+1)}{2} * N\prod$, leading to $(S^2 + 2S) * N\Phi$ unified operations; and (2) for Eigen method, the number of distinct operations is $(S+1)^2 * N\sum + (S+1)^2 * N\prod$, leading to $(2S^2 + 4S + 2) * N\Phi$ unified operations. As already seen in Section 3.5.2, the number of operations required by Eigen method is double than for noEigen one.

Table 3.8: Time`AMNES` $_{\text{ACC}}$ vs. fixed`AMNES` $_{\text{ACC}}$.

time`AMNES` $_{\text{ACC}}$	
Advantages	- simple architecture of the control structure; - streams don't have to be stored; - can scale to thousand of streams; - deterministic;
Disadvantage	- back-pressure on the network; - all state needs to be maintained in the internal or external memory of the FPGA; - working clock frequency will decrease with the high occupancy of the FPGA;
fixed`AMNES` $_{\text{ACC}}$	
Advantages	- only the state of one engine is maintained inside the FPGA; - no back-pressure on the network;
Disadvantage	- all stream values need to be maintained on the memory and passed through the engine; - data structure that keep track of all pars formed – complex structure; - the solution of this data structure has only an heuristic approach;

3.6.3 HLS Experience

AMNES leverages high-level synthesis (HLS) technology instead of register transfer level (RTL) using a hardware description language. HLS provides the advantage of a higher abstraction level accessible to software engineers and abstracts away tedious hardware details like a cycle-accurate behavioral design description using explicit clock and reset signals. Instead, the HLS tool automatically generates these details and injects appropriate pipeline stages to reach the targeted clock frequency. Furthermore, it ensures the proper implementation of data stream protocols by generating the necessary logic to maintain correct operation of the processing pipeline in the presence of back pressure or flow disruptions on the streaming interfaces. Finally, the HLS tool is instrumental in generating the complex arithmetic required for the final coefficient computation in the COEFF pipeline. In particular, floating-point arithmetic cannot be synthesized from native HDL descriptions but relies on the structural instantiation of implementing IP cores. Due to floating-point operations' complexity, these IP cores are deeply pipelined and, hence, exhibit a behavior that must be carefully scheduled. In the HLS domain, the designer can just use a behavioral description using standard C floating-point types. The translation into structural IP core instantiations and their correct embedding with the overall processing pipeline is transparently handled by the tool.

3.6.4 Extended Use of the System

The Kendall and Spearman correlation coefficients are non-parametric statistical tests that rely on the data ranks rather than the values themselves, requiring data to be sorted beforehand. The Spearman coefficient (r_S) can be calculated using AMNES when the values are rank values associated with the stream values. Nevertheless, if all ranks are distinct integers, the Spearman coefficient (r_S) is reduced to $r_S = 1 - \frac{6\sum_{t=0}^{N-1} d_t^2}{N(N^2-1)}$, where d_t represents the difference between the ranks of two observations. This formulation is easier to implement on FPGA since it requires fewer accumulators and multipliers than AMNES. The same ranked data can be applied to the computation of the Kendall coefficient (τ), which then reduces to comparisons and unitary additions, namely to $\tau = \frac{n_c - n_d}{\frac{1}{2}N(N-1)}$, where n_c represents the number of observations ordered in the same way, and n_d represents the number of observations ordered differently.

As introduced earlier, AMNES can also be used to compute the Spearman correlation coefficient, with the prior requirement that ranks are associated to the streams and considered for calculation instead of their actual values. Moreover, the sufficient statistics gathered in the ACC Engine can be used to estimate a simple linear regression, a statistical function commonly available in relational engines (e.g., Oracle's *REGR_SLOPE* operator). If we consider the general line equation $y = mx + n$, knowing the stream values x and y, then the slope and the intercept are given by Equation 3.7, using the least squares method for a set of paired data [89].

$$m = \frac{N \sum_{t=0}^{N-1} x_t y_t - \sum_{t=0}^{N-1} x_t \sum_{t=0}^{N-1} y_t}{N \sum_{t=0}^{N-1} x_t^2 - (\sum_{t=0}^{N-1} x_t)^2}$$
$$n = \frac{\sum_{t=0}^{N-1} y_t}{N} - m * \frac{\sum_{t=0}^{N-1} x_t}{N} \tag{3.7}$$

For only $y = mx$, m is calculated with $m = \frac{\sum_{t=0}^{N-1} x_t y_t}{\sum_{t=0}^{N-1} x_t^2}$. Furthermore, the ACC Engine can server cosine similarity computation [175] as illustrated by Equation 3.8 for two streams x and y, in particular the computation of sum of products and sum of squares. In turn, the cosine similarity can derive the cosine distance between vectors in a multi-dimensional space, since $cosine\_distance(X, Y) = (1 - CosSim(X, Y))$ [175].

$$CosSim(X, Y) = \frac{\sum_{t=0}^{N-1} x_t y_t}{\sqrt{\sum_{t=0}^{N-1} x_t^2} \sqrt{\sum_{t=0}^{N-1} y_t^2}} \tag{3.8}$$

3.6.5 Summary

In this chapter we have explored the implementation space of correlation computation, namely the Pearson correlation coefficient, as a co-processor and in-network advanced data analytics compute kernel, leveraging specialized hardware implementations using a stream-based algorithm without affecting either network or CPU performance. The co-processor approach delivers benefits in particular for a large number of items per streams, however data transport via PCIe limits this approach. Maximum advantages emerge when correlation is deployed within the network, as additional and unnecessary data transfers via PCIe are avoided. Additionally to the experimental evaluation, we analyze the generalization capability of the system, deriving two approaches and determine their advantages and disadvantages. We conclude the chapter with an overview of possible implications of the gathered sufficient statistics values into computing multiple correlation coefficients rather than solely the Pearson correlation coefficient along with linear regression and cosine distance.

Chapter 4

Sketch Algorithms - Approximation Compute

In the previous chapter we have analyzed how correlation computation can be offloaded to an FPGA and generalized to thousands of streams. Irrespective of the number of streams, the internal processing capacity of the engine is determined by the bit width of the accumulators and the data representation, generating a dependency between the internal structure and the streams' number of elements.

Today's global digitalization brings with it huge amounts of data to be processed, in the order of billions of items. Therefore, to make the structure of a compute engine independent of the size of analyzed data, we explore the performance characteristics of sketch algorithms on the FPGA.

Sketch algorithms, also known as probabilistic data structures or approximate algorithms, are techniques used to efficiently process and summarize large amounts of data while providing approximate results. They are designed to trade off accuracy for reduced memory usage and faster computation time.

The following chapter is dedicated to how multiple sketch algorithms can run together, analyzing streams of data that either reside in the host CPU memory or come from the TCP/IP network. In particular we are interested into Hyperloglog, Count-Min and Fast-AGMS algorithms.

4.1 Motivation

Many tasks in data processing, both in streaming scenarios as well as in conventional databases, start with the need to summarize and characterize data sets by computing basic metrics: the cardinality (number of their *distinct* elements), the overall frequency distribution, the heavy-hitters (their high-frequency elements), as well as basic statistics such as average, maximum and minimum values, or histogram distributions of the data. Except for the most basic statistics, there is often a trade-off between computing

a given quantity with a certain accuracy and the efficiency of the algorithm used both in time and space. This has given rise to the widespread adoption of sketch algorithms [47, 49, 59] as the canonical approach to compute such metrics over streams and large data sets. For instance, HyperLogLog (HLL) is widely used in systems such as Google's BigQuery to compute the cardinality of data sets with several billion distinct items [91]. Other sketch algorithms are used to, e.g., estimate results for queries of the type `COUNT (DISTINCT _,...)` to predict the size of joins [9] or to compute join orders [148].

When used for data characterization, efficient and useful as they are, sketch algorithms become expensive. First, they require to perform a pass over the entire data set or data stream. Second, summarizing a data stream often involves computing several of its characteristics and not just one. As an example, if a data set has a rather uniform distribution, knowing the frequency of each one of its items is not very informative. Conversely, for a skewed data set, we may want to identify the most common items. Since the data has not been processed before, its characteristics are unknown and, hence, the first step is to compute several such metrics to then decide which ones are useful and provide the most information about the data. In fact, if several metrics are computed, some metrics can be used as quality indicators for the approximations produced by others, an important aspect when using sketches as they produce only approximate results. For this purpose, it would often be beneficial to be able to characterize the data as much as possible in a single pass, for instance, as the data is read from storage. Unfortunately, computing several sketches over a data set using CPUs is expensive and requires significant CPU capacity. For streams, CPUs are unable to match the bandwidth of a 100 Gbps TCP/IP network, which leads to additional inefficiencies and bottlenecks. This represents a problem for the ever growing amount of data that must be processed under stricter and stricter throughput and latency constraints.

To address the problem of efficient data characterization, we take advantage of the growing availability of FPGA accelerators and explore their inherent spatial parallelism to efficiently compute *several sketches and statistics* over a data set *in a single pass*. The resulting system, `SKT`, provides important insights into the design of algorithms for spatial architectures and the use of accelerators for data processing. `SKT` can be run on a PCIe-attached FPGA card but also on a smartNIC where it processes the data as it arrives from (or departs to) the network. This makes `SKT` amenable to deployments such as those in Microsoft's Catapult, where the smartNIC approach would allow to add `SKT` onto any computing node, or in Amazon's AQUA, where `SKT` could be used to compute key statistics (or even parts of queries) over relational data while it is being sent to the query processing nodes.

In this chapter, we present three complementary sketch algorithms commonly used for characterizing data sets and streams, as well as for cost-based query optimization. These sketches are:

a. *HyperLogLog* (HLL), which estimates the number of *distinct elements* in a data set;

b. *Count-Min*, which estimates *the frequency of items in a stream or data set*;

c. *Fast-AGMS*, which estimates the second frequency moment of a distribution.

AGMS has been shown to predict the accuracy of a Count-Min sketch when computed over the same data [192]. We also show that the cardinality estimation given by HLL provides another valuable accuracy indication and the three sketches together can be used to obtain a rather accurate picture of the distribution, size, and nature of the data set (stream). While these sketches have been shown to benefit from hardware acceleration [141, 132], SKT is the first system to combine them into a single design and study the resulting non-trivial resource-accuracy-performance trade-offs, making several novel and timely contributions.

The chapter is structured as follows. The first section introduces an overview of each of the sketch algorithms deployed on the FPGA, followed by a literature review of the software and hardware systems that try to address the architecture of sketch algorithms. Subsequently, in Section 4.3.1, the overall SKT design is presented, exploring how to merge several sketches into a single design both on CPUs and FPGAs. Based on the in-depth analysis of the resource utilization and the accuracy of the algorithms in Section 4.3.4, the chapter moves to the actual FPGA implementation in Section 4.4, showing how the inherent parallelism of FPGAs can be used to implement combined versions of several sketch algorithms without any of them interfering with each other in terms of performance. Next, in Section 4.5.2, it explores the resulting performance with comparisons against CPUs with different processing capacities, considering both the co-processor (Section 4.5.3) and smartNIC (Section 4.5.4) FPGA deployments. The chapter concludes with a summary on the main findings, demonstrating the FPGA performance gains over CPU deployments: 1.75× over a high-end server with 2× Intel®Xeon®Gold 6248 Processors, and 3.5× over a smaller, more conventional Intel Xeon Gold 6234 Processor.

4.2 Sketch Algorithms

Sketch algorithms compute summaries over data streams typically in sub-linear space. Basic scalar summaries include the minimum and maximum values, the stream's size, and its total sum. These four numbers can already characterize the value range, and the average across the whole data stream. More elaborated summaries estimate parameters such as the cardinality or the skew of the data set. There are also sketches that can be queried for item-related estimates, such as individual occurrence counts or item frequencies. Cormode et al. [49] offer an in-depth review of sketches.

Table 4.1: Symbols used for defining the frequency moments.

S	Set of data items contained in the stream
f_i	Occurrence count (frequency) of data item i in the stream
F_q	q-th frequency moment

Table 4.1 lists the symbols used for defining the q-th frequency moments, F_q, of a data stream, a more formal definition of what sketches calculate. They are computed as:

$$F_q = \sum_{i \in S} f_i^q \qquad \text{(using } 0^0 = 0 \text{ for } q = 0\text{)} \tag{4.1}$$

The zero$^{\text{th}}$ frequency moment (F_0) represents the number of *distinct* elements in the stream, i.e., its *cardinality*. The first frequency moment, F_1, represents the total number of elements in the stream or *item count*. The other frequency moments, F_q, with $q \geq 2$ represent degrees of skew in the distribution of the data [10]. The second frequency moment, F_2, is particularly relevant in query optimization as it is used to estimate self join sizes [9].

4.2.1 HyperLogLog

The HyperLogLog (HLL) sketch estimates the number of distinct data items in a data stream (its zero$^{\text{th}}$ frequency moment) [71]. Cardinality estimation is commonly used in databases, e.g., for approximate query processing [39], query optimization (DBMS) [242, 173], and data mining [161]. HLL has become the standard algorithm to estimate cardinalities in very large data sets. It is used by Google [93, 91] in BigQuery for data analytics, by Facebook [25] to estimate their graph's degrees of separation for social network analysis, and in systems such as Amazon's Redshift [185] or Databricks [202].

HLL is a hash-based algorithm which maps each item of the data stream to a hash value and monitors the maximum number of leading zeros observed among the binary representations of these hash values. HLL exploits that a randomized hash with i leading zeros is expected to be seen only once across 2^i distinct elements as it is intuitively illustrated by Table 4.2. Thus, one needs to process around 2^i elements to observe a hash containing i leading zeros. For the same reason, if a maximum of i leading zeros has been seen, one has probably processed 2^i distinct elements. The approach can lead to large errors overestimating the cardinality, for instance, through encountering a value with many leading zeros in a short data stream. To mitigate such anomalies, adjustments are made such as using parallel sub-sketches and correcting the estimated value for particularly small cardinalities.

Table 4.2: Monitoring the leading zeros in the hash value.

Leading '0'(s)	Elements	Cardinality
1	1 / 0	2
2	11 / 10 / 01 / 00	4
3	111 / 110 / 101 / 100 / 011 / 001 / 010 / 000	8
n	...	2^n

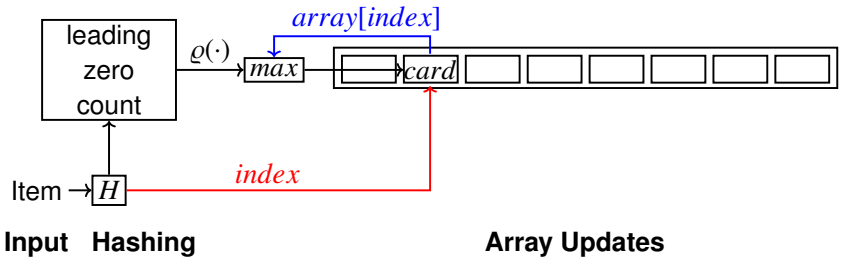


Figure 4.1: The HLL update process.

HLL maintains a one-dimensional, zero-initialized array as its underlying data structure. For an item insertion, part of its hash value indexes into this array to identify an update location. The remaining hash is subjected to a leading-zero count. The resulting number plus one is called the *rank*, $\varrho(\cdot)$. The identified update location is then set to the maximum of its current value and the computed rank. Ultimately, each array element stores the cardinality estimate for the input substream whose hash values produced the corresponding array index. The overall stream cardinality is estimated as the harmonic mean across all these partial estimates normalized by the number of differentiated indexes, since the harmonic mitigates the impact of large outliers, operates as a variance reduction and increases the quality of estimates. Statistic low-cardinality corrections are applied if some array elements remain untouched at a value of zero. Figure 4.1 depicts the HLL update process using a very small sketch array that differentiates eight array indexes (aka. buckets) by using three hash bits as index and the rest for leading zero detection.

Hyperloglog has an expected relative error of $\pm\frac{1.04}{\sqrt{2^P}}$, where 2^P represents the number of array elements [71]. P represents the precision parameter, as it is illustrated in Table 4.3. This means that a stream's cardinality can be estimated within an error margin of 1-2% with a memory footprint of only a few KBytes [47]. An analysis by Databricks on Spark's HLL performance shows that the error margin can be reduced to values below 1% at the expense of memory usage and speed [202]. When keeping the estimation error below 1%, Spark's HLL algorithm is slower than the actual count of distinct elements [202], proving the significant overhead of computing HLL on conventional CPUs. Kulkarni et al. [141] show that the error range can fall below 0.41% if more memory is allocated for the sketch array.

4.2.2 Count-Min

Count-Min [51] computes the approximated occurrence of data items in a data stream. It can be queried for item counts, heavy hitters, or the most popular items in a data stream [241]. It is also used to find the optimal join order for multiple joins [148].

Count-Min is similar to Bloom filters [29], but it differs from them by offering an estimate of the item frequency rather than just a binary set membership. Count-Min is closely related to Counting Bloom

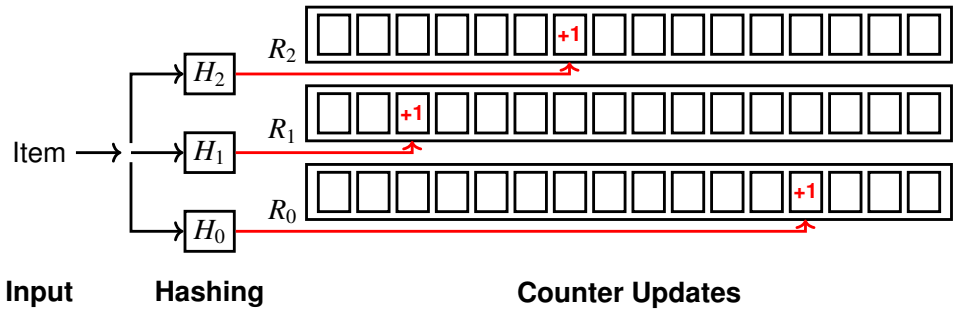


Figure 4.2: Item insertion into a Count-Min sketch.

Filters as used to maintain a cache summary of a Web proxy [67]. Both algorithms maintain a set of zero-initialized counters. The update locations associated with a data item are identified by a fixed number of hash functions. All counters at the identified locations are incremented for an item insertion and decremented for an item deletion. While a Counting Bloom Filter uses a one dimensional array, Count-Min uses a two-dimensional counter matrix with a fixed association of each row with one of the hash functions. Therefore, Count-Min has a structural benefit by performing parallel updates into independent memory regions. We will exploit this feature in the FPGA implementation of Count-Min.

Figure 4.2 illustrates the insertion of an item into a Count-Min sketch, but the algorithm supports deletion as well. This small sketch example maintains $R = 3$ rows with 16 (2^P with $P = 4$) counters each. For each inserted data item, independent hash functions H_i derive 4-bit addresses that identify the counters to be incremented in each of the rows (R_i, $i \in [0, 2]$). For deletions, the corresponding sketch update process is identical, except that the identified counters is decremented rather than incremented. To find out the frequency of an item, a query follows the same procedure to access the counters. However, it only reads the counter values in order to report the minimum of all values read as the estimated item's frequency. Observe that this result will never underestimate the actual item frequency as every counter included in this minimum has been incremented for every occurrence of the queried item. However, overestimation is possible in the presence of hash collisions with other items, which is mitigated by taking the minimum across the R rows. In this way, only the smallest contribution produced by collisions under independent hash functions may affect the reported estimate. The independent hash values can be obtained by slicing a hash value whose number of bits equals the cumulative bit widths of the individual hash values. The collisions' effect is also significantly reduced since two items would need to collide in all hash functions in order to affect the count of each other.

Count-Min estimates an item's frequency with an error of at most εC with a probabilistic confidence $(1 - \delta)$ [49], where $\varepsilon = \frac{2}{2^P}$ with 2^P being the number of counters, and $\delta = \frac{1}{2^R}$ with R being the number of rows, and C representing the expected number of colliding items. The value of C can be assessed by correlating the input stream cardinality with the number of counters. We will take advantage of this feature and use HLL to calibrate Count-Min.

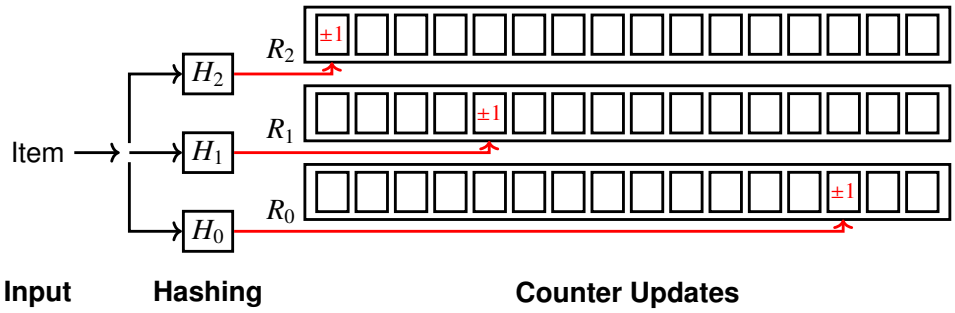


Figure 4.3: Item insertion into a Fast-AGMS sketch.

4.2.3 AGMS

AGMS does not represent an acronym that defines the operational approach of the algorithm but rather consists of the initials of the individuals who introduced it at PODS'99, namely Noga *Alon*, Phillip B. *Gibbons*, Yossi *Matias*, and Mario *Szegedy*.

AGMS estimates the second frequency moment [10] mentioned in Equation 4.1 which is also known as *the repeat rate or Gini's index of homogeneity*. The sketch itself was first introduced for relational databases in order to estimate the size of joins in the context of limited storage [9]. Traffic validation systems [240] also employ AGMS to identify anomalies between the incoming and outgoing traffic and emphasize the scalability of applying AGMS based on its ability to be distributed and its implied small size of the required router state.

Like the Count-Min sketch, AGMS maintains a two-dimensional array of counters where each row has its own independent hash function. When inserting a data item, however, *all* counters in the sketch are updated. Each hash function produces one bit for each counter (column) in the row. This bit decides whether each individual update is an increment or a decrement. Deletions can be supported by reversing the sign of these updates. The result is obtained by extracting the median from the arithmetic means of the squared counter values computed for each row.

Cormode and Garofalakis [48] identify the parallel update of the *whole* AGMS counter matrix as a major performance bottleneck and propose a modified algorithm. Instead of using the hash to choose between incrementing or decrementing each and every counter, they limit the update to a single counter in a row. Both the counter's index and the direction of its update are determined by the hash value of the data item. Reducing the number of updates to a single cell per row makes the new algorithm much faster than AGMS and known under the name *Fast-AGMS*. This modification to the original algorithm makes the update procedure similar to that of the Count-Min sketch. The only difference is that the row hash needs one additional bit to select between an increment or decrement for the performed counter update.

Cormode and Garofalakis prove that this modified sketch, Fast-AGMS, achieves the same error bounds

and confidence as the significantly more expensive original AGMS sketch. The evaluation of the sketch is only changed in that the per-row averages of squared counter values are replaced by per-row sums of squared counter values. Computationally, this even saves a division operation at the end. In SKT, we implement Fast-AGMS, with the counter updat process as illustrated in Figure 4.3.

Fast-AGMS and AGMS offer the same trade-off between space and accuracy. They estimate the second frequency moment of a stream with an error of at most ε with the probabilistic confidence $(1 - \delta)$ [48], where $\varepsilon = \frac{1}{2^P}$ with 2^P being the number of counters, and $\delta = \frac{1}{2^R}$ with R being the number of rows. Note the similarities with Count-Min, although we will show that the error characteristics of these two sketches are different depending on the data set analyzed.

Related Work. Kulkarni et al. [141] implement HLL and use it to process streams received from a 100 Gbps TCP/IP network, while using the entire FPGA for the HLL design. In contrast, SKT adjusts the design of the HyperLogLog sketch computation to still match the network bandwidth but leaving space for the computation of other sketches *in parallel*. Kulkarni et al. also demonstrate empirically the need to use a 64-bit Murmur3 hash to improve the cardinality estimation accuracy for larger data streams. In this chapter, we expand the range of configurations considered and provide a detailed analysis of the throughput-accuracy-resources trade-offs.

Scotch [132] is a VHDL-based code generator for sketch implementations. Similarly to SKT, Scotch also aims at maximizing the use of the available on-chip memory. However, it does so to implement a *single* custom sketch out of the supported sketch classes. While the plain scaling of a single sketch faces diminishing returns, the multi-sketch approach pioneered by SKT uses the resources to deliver extra value. Scotch and SKT also differ in the chosen design methodology. Scotch relies on traditional RTL design. Hand-coded VHDL code is patched by a generator tool to match a custom sketch specification. Scotch explores the largest feasible sketch dimension only during the design implementation, putting the complete synthesis toolchain into the exploration loop. SKT avoids this huge, often multi-day effort by providing a utilization model for the critical BRAM resources. Last but not least, Scotch forgoes an end-to-end in-system evaluation. Its performance results do not include data transmission and result extraction overheads although the integration with the host application pose decisive practical bounds to the performance that can be extracted from a hardware accelerator. A further important difference in terms of design and accuracy is that Scotch uses H_3 hashes while SKT uses Murmur3 hashes.

Tong and Prasanna have proposed an FPGA-based, RTL-designed sketch accelerator [213] for monitoring and detecting network traffic anomalies [214]. Their implementation is a Count-Min sketch, over which they support Count-Min and K-ary queries. In contrast to SKT, which particularly accelerates the sketch constructions, Tong and Prasanna monitor individual sketch updates to identify on-the-fly anomalies, i.e., heavy hitters (Count-Min) or heavy change (K-ary). They never communicate the total accumulated sketch itself. Like Scotch, their implementation is based on H_3 hashes.

In the same way that we use SKT for characterizing streams, sketches can also be used for the structural analysis of multidimensional data. Rouhani et al. [190] describe SSketch as a framework for building streaming analysis accelerators on FPGAs for this purpose. The design works on a 1 Gbps network while SKT targets 100 Gbps, a two orders of magnitude higher throughput.

4.3 Sketch Engine Design

In this section, we discuss SKT's design and its implementation on the FPGA. For the exploration of the design space and as a baseline reference, we use an equally parallelizable SW-SKT targeting high-end multi-core CPUs. We consider configurations that operate (a) on data streams residing in the CPU memory, with the FPGA behaving as a co-processor, and (b) on data streams arriving from the network, with the FPGA behaving like a bump-in-the-wire. We also discuss the system trade-offs between sketch accuracy and the choice of the hash function and sketch sizes.

4.3.1 System Overview

SKT's objective is to compute the three sketches described above in a single pass over the data and characterize a stream or data set. For this purpose, we explore a composite algorithm combining the initial hashing of the input data with three parallel sketching backends, one for each targeted metric, i.e., HLL, Count-Min and Fast-AGMS. As the update operations of these three sketches are all associative, parallel partial sketches can be computed over arbitrarily distributed sub-streams (data pipelines) before their results are merged in the output path. SKT's general architecture is shown in Figure 4.4. The figure details in the foreground the architecture of one data pipeline with the updates associated with each of the three algorithms data structures, followed by the merge of the computed N parallel partial sketches, and finishing with the generation of the output results. The current implementation processes all the values in a data set before emitting the results computed for each sketch. While Count-Min just streams the computed sketch verbosely, HLL and Fast-AGMS perform additional computations to reduce their sketches to their final scalar estimates.

The slicing of a well randomized hash by each of the sketches can be arbitrary. We opted for an approach that results in a simple homogeneous sequence of masking extractions and constant right shifts for the software implementations of the matrix sketches. HLL is implemented according to its customary formulation relying on the *count-leading-zeros* operation after slicing off the index. This can leverage an x86 instruction in the software implementation.

In Figure 4.4 each pipeline consumes 32-bit of data that is hashed in the *Hashing* phase; the hash value is then distributed to each partial sketch, sliced, and used to update the data structure behind the sketch

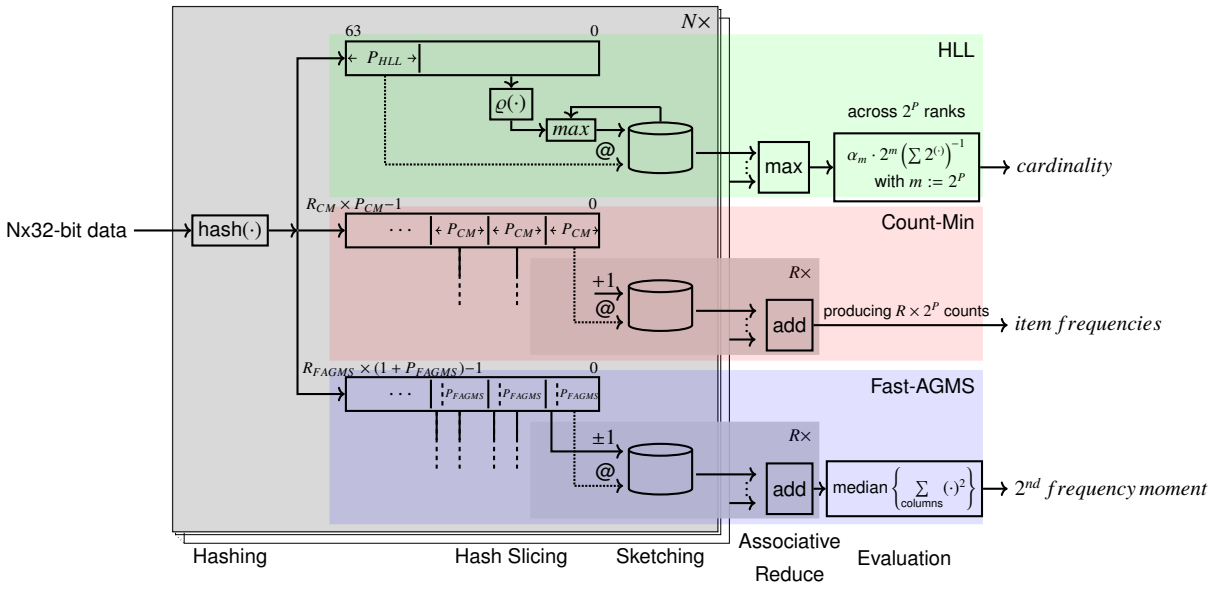


Figure 4.4: SKT architecture dataflow.

in the *Hash Slicing* and *Sketching* phases. After consuming all the input values, SKT triggers the merging of all partial sketches in the *Associative Reduce* phase and generates the output results in the *Evaluation* phase for the HLL and Fast-AGMS sketches.

The degree of parallelism and the size of the data structures is controlled by the parameters listed in Table 4.3. The counter matrices computed by Count-Min and Fast-AGMS are R_* rows times 2^{P_*} columns. HLL uses a linear array with $2^{P_{HLL}}$ buckets. The bit width of the counters is universally chosen to be $W_* = 32$. Compared to a smaller size of $W_* = 16$, this enables all sketches to cope with heavy hitters even in large data streams of a few billion items without an overflow.

The choices for the other key parameters are explored below. Larger sketches can generally be expected to produce more accurate estimates. However, they also incur higher costs both in terms of sketch storage capacity and in terms of hash computation effort.

Table 4.3: Design parameters of the SKT kernel.

N	Input parallelism: threads / data pipelines (partial sketches)
$P_{HLL\|CM\|FAGMS}$	Precision: hash bits used for indexing buckets (HLL) or columns (Count-Min, Fast-AGMS)
$R_{CM\|FAGMS}$	Rows: parallel rows in the matrix sketches
$W_{CM\|FAGMS}$	Bit Width of Count-Min and Fast-AGMS counters

4.3.2 Hash Function and Hash Size

Sketch algorithms typically seek to isolate themselves from the concrete encoding of the input by means of a randomizing hash. After hashing, they can assume that each unique data item is assigned a fixed but seemingly random encoding drawn from the range of the chosen hash function. The statistic input randomization is the main purpose of hashing and dictates what hash function should be used.

Randomization is a stricter requirement than hash uniformity, with the latter being commonly used as a hash metric for the implementations of hashing data containers. For example, inexpensive, and hence popular, H_3 hashes can be perfectly uniform. They map an n-bit key a from the domain $A = \mathbb{Z}_2^n$ to an m-bit hash value b of the range $B = \mathbb{Z}_2^m$ as shown in Equation 4.2, where M defines the random $m \times n$-matrix over $\mathbb{Z}_2 = GF(2)$. The H_3 computation uses `AND` ($\cdot$) as the multiplicative and `XOR` ($\oplus$) as the additive boolean operators. As soon as the number of needed random hash bits m exceeds the number of key bits n, linear dependencies between the matrix rows become unavoidable. When that happens, hash bits become mutually dependent and patterns among input encodings reemerge in the computed hashes. The critical randomization assumption is violated.

$$\begin{aligned} h : A &\to B \\ a &\mapsto M \cdot a \end{aligned} \tag{4.2}$$

MurmurHash is a family of seedable non-cryptographic hash functions introduced by Austin Appleby in 2008. Seedable because it allows the use of a seed value which in return initializes the internal state of the hash function. By varying the value of the seed, one can produce different hash values for the same input data value. Non-cryptographic because it is not suitable for cryptographic purposes, since its design does not specifically aim to not be reverse-engineered. Examples of cryptographic hash functions are SHA-1/2 and MD4/5, which are built through a similar method as the block cipher modes of operation used for encryption.

Richter et al. [186] concluded that Murmur hashing delivers the best trade-off between performance and robustness among four hash functions (multiply-shift, multiply-add-shift, tabulation, and Murmur hashing). Kaan et al. [125] evaluate hash functions and show that simple tabulation hashing is 6.6× faster on FPGA than in software, and Murmur hashing is 1.7× faster, respectively. They also show how expensive hashing can be used in data partitioning on FPGAs without loss of performance [127]. Murmur hashing has gained a lot of popularity, with Murmur3 being widely used in practice, e.g., in Google's BigTable [91] and in research [78], due to its versatile design oriented towards speed, efficiency and portability across different platforms, architectures and programming languages (e.g., C++, Java, Python).

`SKT` uses Murmur3 as the hash function. We use a software `SKT` implementation to validate the suitability of this choice and to determine acceptable lower bounds for the sketch sizes in Section 4.3.4. The number of required hash bits is determined by the maximum across all sketches implemented in `SKT`. Allocating

no less than 64 bits to HLL [91], which leaves $(64 - P_{HLL})$ bits for the leading-zero detection, we derive a formula (Equation 4.3) that determines the number of bits to be used for hashing for all sketches:

$$H = \max\{\ 64,\ \ R_{\text{CM}} \times P_{\text{CM}},\ \ R_{\text{FAGMS}} \times (1 + P_{\text{FAGMS}})\ \} \tag{4.3}$$

4.3.3 Implementation Targets

Besides the SKT design for FPGA acceleration, we have implemented SW-SKT targeting multi-core CPUs. Both versions are implemented in C++ (conventional C++ for SW-SKT on the CPU, and High Level Synthesis [130] – HLS C++ for SKT on the FPGA). The code bases are distinct so as to optimize for each target platform. Parallelism and data reuse are thoroughly exploited in both cases. Each algorithm is parallelized over cores and threads (SW-SKT) or unfolded spatially (SKT). The hashes of incoming data items are re-used to drive the update of all three algorithms.

We use SW-SKT as a baseline, for validating the choice of hash function and hash size, and for exploring the trade-off between accuracy and sketch size. For the sake of a sound design evaluation and platform comparison, we determine the minimum, and hence most effective, sketch size that supports one billion stream cardinality. Note that SW-SKT is a contribution on its own as it can be used in conventional systems without hardware accelerators for achieving performance gains over running each sketch separately.

4.3.4 The Accuracy vs. Size Trade-off

The sketch sizes are determined by the parameters R_* and P_*. In all cases (HLL, Count-Min and Fast-AGMS), the precision P_* determines the number of sketch columns (or buckets). An increase in the value of P_* reduces the chance of hash collisions and, hence, improves accuracy but also requires more space. Count-Min and Fast-AGMS take extra measures to mitigate individual hash collisions by maintaining several parallel rows with *independent* hash functions since mutual collisions in one row are unlikely to reproduce in another. As a result, increasing R_* improves the accuracy of the overall sketch at the price of requiring more space, more memory accesses, and the computation of more hash bits. Since the final goal is to have the three sketches deployed together in a spatial architecture, it is important to understand the implications of the choices for P_* and R_* for each sketch accuracy trade-off.

To evaluate the accuracy vs. performance trade-off, we compute sketches of various dimensions over data sets with known item frequencies and pre-computed frequency moments. We create two classes of data sets of 32-bit integer elements:

- **CLS1**: This class comprises 9 data sets with uniform data distributions. Each data set is defined by a frequency $f \in \{1, 20\}$ where f is the number of times every item occurs. The maximum cardinality in this class is 1 billion (1 B) and the maximum stream length is 20 billions (20 B).

- **CLS2**: This class comprises 21 data sets with Zipfian distributions with an $s \in \{1.5, 2.0, 3.0\}$ skew. The maximum cardinality of this class is 1.4 millions (1.4 M) and the maximum stream length is 1 billion (1 B).

Realistic data is frequently highly skewed [50, 156]. Intuitively, in highly skewed data, a relatively small number of distinct items appear very frequently; whereas in low-skew data, item occurrences are more uniformly distributed. Relating skew and cardinality, observe that a high skew implies a smaller cardinality in relation to a stream's length and that a high cardinality on the order of the stream length implies a low skew. The converse statements, however, do not necessarily hold true.

We studied the impact on the sketch accuracy of the: (1) sketch size, (2) input stream length, and (3) stream cardinality. The accuracy is reported in terms of the observed relative error. We report the *maximum* relative error (Equation 4.4) encountered across our class data sets, unless otherwise specified.

$$relative\_error\ [\%] = \left| \frac{obtained\_value - actual\_value}{actual\_value} \right| * 100\ [\%] \tag{4.4}$$

HLL. Figure 4.5 shows the relative error of HLL as a function of the stream cardinality. HLL is a very robust algorithm scaling well to large input cardinalities and sizes. A choice of $P_{HLL} \geq 14$ results in an error band of 1-2% across all input stream cardinalities. This value is aligned with other implementations and available systems as well as results reported in the literature.

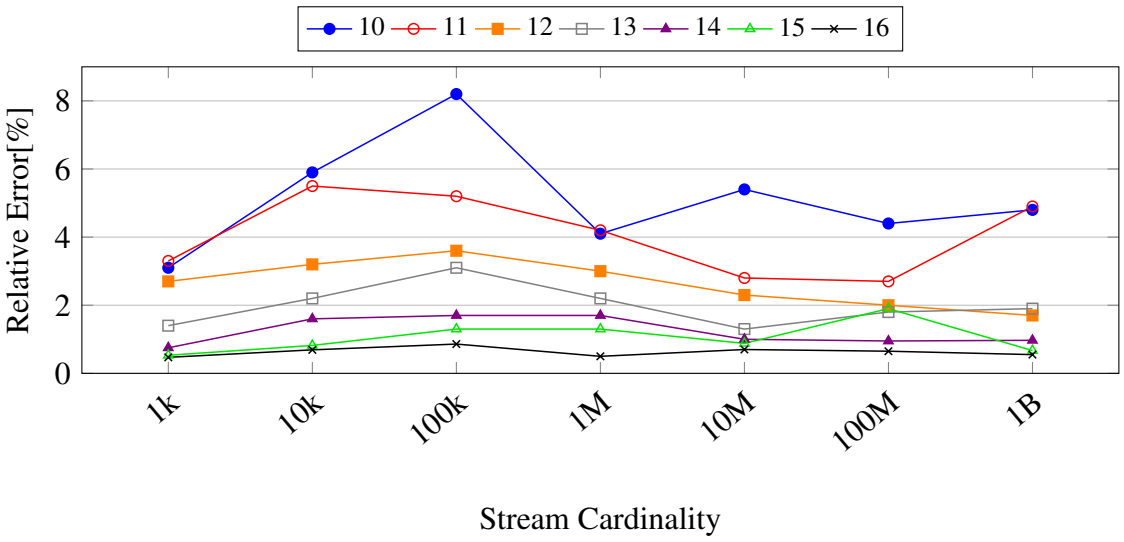
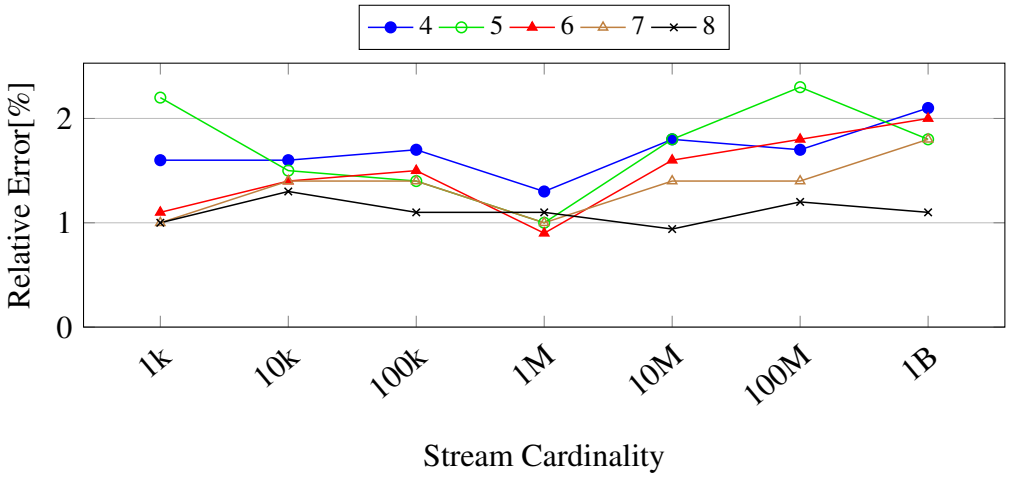
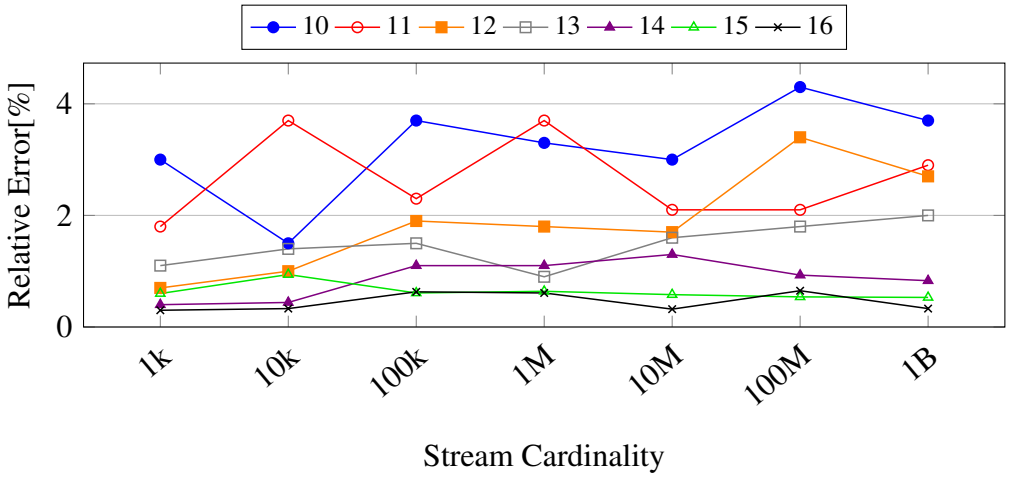


Figure 4.5: HLL-Relative error vs. Stream cardinality for different values of P_{HLL}. Maximum stream length is 20 B.

Fast-AGMS. We evaluate the relative error of Fast-AGMS in two configurations under stream cardinality variation. The first configuration fixes the precision bits ($P_{FAGMS} = 13$) and varies the number of rows (R_{FAGMS}). The results are illustrated in Figure 4.6a and suggest that the row variation has little impact on the relative error of the sketch, with $R_{FAGMS} \geq 6$ keeping the error below 2% for the chosen hash seed value. Higher row counts improve the error only by little at the cost of more memory accesses for each sketch update. For example, for $R_{FAGMS} = 8$, the relative error fluctuates around 1%. The second configuration fixes the number of rows, $R_{FAGMS} = 6$, and varies the precision bits P_{FAGMS}. In Figure 4.6b is observed that by varying P_{FAGMS}, the sketch capacity is scaled faster than by varying R_{FAGMS}, with $P_{FAGMS} = 16$ registering relative errors below 1%. Each increment of 1 for P_{FAGMS} doubles the size of the sketch, allowing a wider spread of the counters and reduces the possibility for hash collisions.

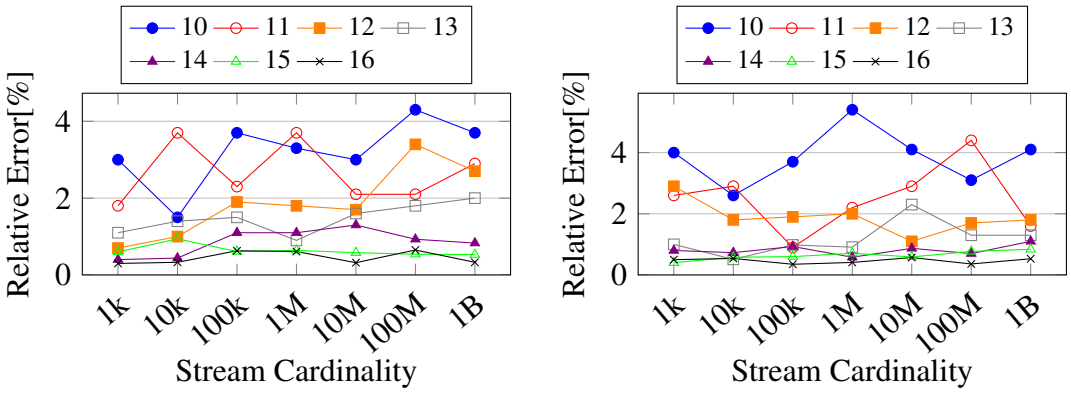


(a) Fixed P_{FAGMS}=13 & different values of R_{FAGMS}



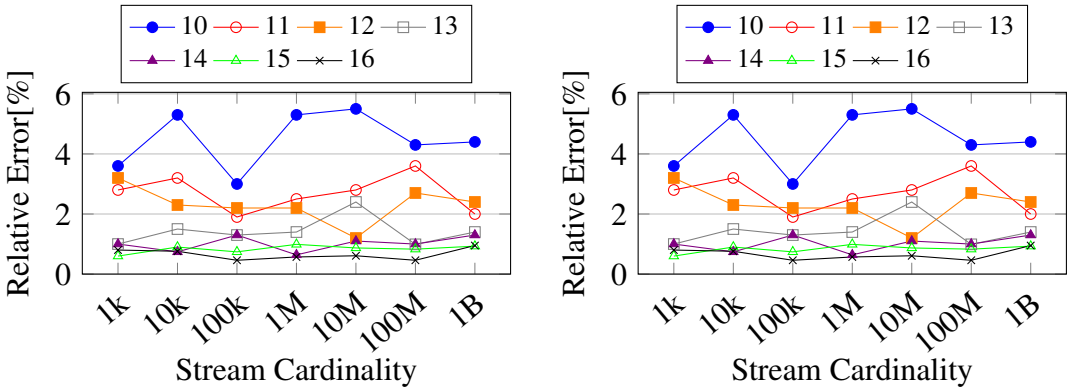
(b) Fixed R_{FAGMS}=6 & different values of P_{FAGMS}

Figure 4.6: Fast-AGMS-Relative error vs. Stream cardinality for maximum stream length of 20 B.



(a) Hash seed 1 **(b)** Hash seed 2

Figure 4.7: Fast-AGMS-Relative error vs. Stream cardinality for maximum stream length of 20 B. Fixed $R_{FAGMS} = 6$ and different hash seeds.



(a) Hash seed 1 **(b)** Hash seed 2

Figure 4.8: Fast-AGMS-Relative error vs. Stream cardinality for maximum stream length of 20 B. Fixed $R_{FAGMS} = 5$ and different hash seeds.

Both results in Figure 4.6 show that accuracy improves by increasing row count or precision bits. However, an additional row increases the sketch size by 20%, whereas an additional precision bit doubles the size of sketch, leading to different memory footprints and resource consumption.

For HLL and Fast-AGMS, the individual upper and lower peaks observed at different cardinalities (e.g., 1 M or 10 M) are an artifact of the interaction between the data and the concrete choice of hash. Changing the seed of the chosen Murmur-3 hash function displaces the jitter observed in the error graphs. In Figure 4.7, we show the effect of the hash seed on the relative error for the configuration $R_{FAGMS} = 6$ and varying P_{FAGMS}, with the second seed generating a relative error a bit over 2%. The two hash seed values are part of an example used by LevelDB [77] for their hash function implementation which is similar

to murmur hash. The same jitters are also observed for a reduction in the row count $R_{FAGMS} = 5$ in Figure 4.8. However, for this configuration, the relative error surpasses our predefined 2% margin in both cases. Based on these results, we establish the $R_{FAGMS} = 6$ and $P_{FAGMS} = 13$ for the Fast-AGMS sketch data structure.

Count-Min. Unlike HLL and Fast-AGMS, Count-Min does not produce a single scalar estimate. Rather, the sketch is queried for individual item frequencies. We track the maximum relative query error to characterize query quality as well as the average relative error for a comprehensive accuracy assessment of Count-Min. Since Count-Min can be queried for items that have not appeared in the input stream, we report: (1) the maximum and average relative errors exclusively for items that appear in the input data stream in Figure 4.9, and (2) the absolute error, calculated as in Equation 4.5, for all members of the 32-bit input domain in Figure 4.10.

$$absolute\_error = |obtained\_value - actual\_value| \tag{4.5}$$

Figure 4.9 shows that the maximum relative error (.a lines) increases rapidly for cardinalities larger than 1'500, whereas the average relative error (.b lines) increases at a slower and steadier pace. The average relative error is calculated as the average of all relative errors with respect to the number of queried items. The maximum relative error demonstrates that Count-Min can fail *individual* queries with a margin larger

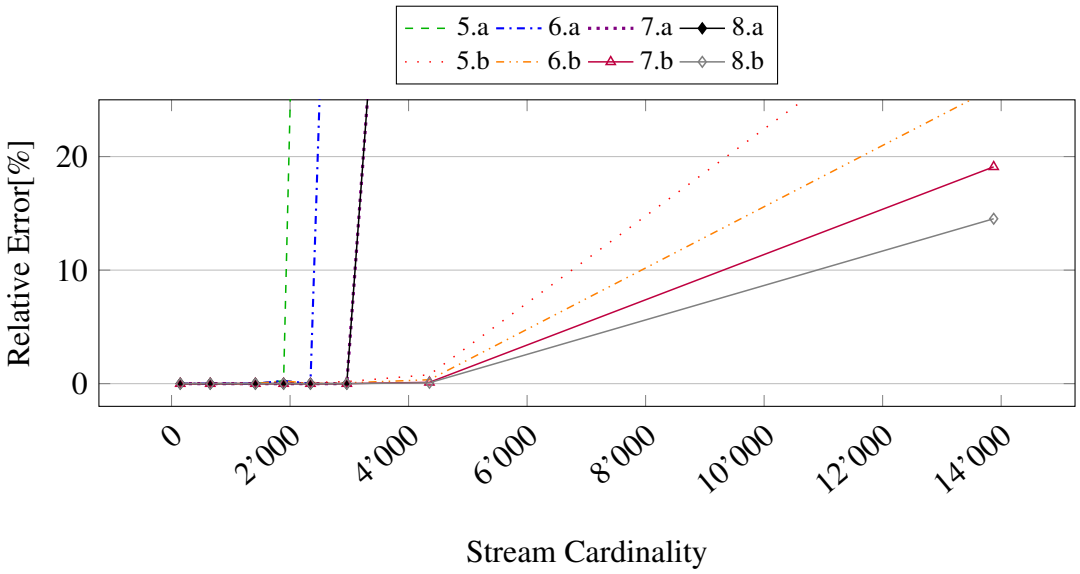


Figure 4.9: Count-Min-Relative error vs. Stream cardinality for fixed P_{CM}=13 & different values of R_{CM}. R_{CM}.a lines represent the maximum relative error, whereas R_{CM}.b lines represent the average of all relative errors with respect to the number of queried items.

than 20%. As collisions accumulate with larger stream cardinalities, there will eventually be an item that is affected across all rows. The overestimation even of a single low-frequency item quickly pushes the observed *maximum* relative error beyond any sensible scale.

However, the *average*, and hence expected, error of sketch queries remains within acceptable bounds for larger stream cardinalities, up to 3× larger cardinalities than the relative error. Increasing the number of rows R_{CM} or precision bits P_{CM}, indeed, makes the sketch suitable for larger stream cardinalities, but it still remains frail. Even computing 128 hash bits to maintain a sketch of 2 MiBytes ($R_{CM} \times P_{CM} = 8 \times 16$) has an average relative query error of 14.22% for a cardinality of 100'000.

A common metric to evaluate Count-Min is the average *absolute* query error as shown by Figure 4.10. Under this metric, the sketch appears to scale significantly better to larger cardinalities. It is important to understand that the averaging hides that error contributions derived from rather few but significant overestimations. This underlines, once more, the accuracy compromise inherent to Count-Min. It tolerates individual significant estimation errors, which appear particularly huge on a relative scale, for an otherwise concise item frequency representation. Therefore, it must be decided at application level whether or not this compromise is acceptable.

All of the quality metrics discussed indicate that the quality of Count-Min degrades as the cardinality increases. This can be explained by the growing number of distinct data items and, hence, column index signatures across the sketch rows, which eventually encounter mutual overlaps in all rows. Observe that stream length alone does not pose a challenge. The fewer index signatures in a long stream with a low cardinality are more likely to avoid a collision in, at least, one of the rows. This is in line with results

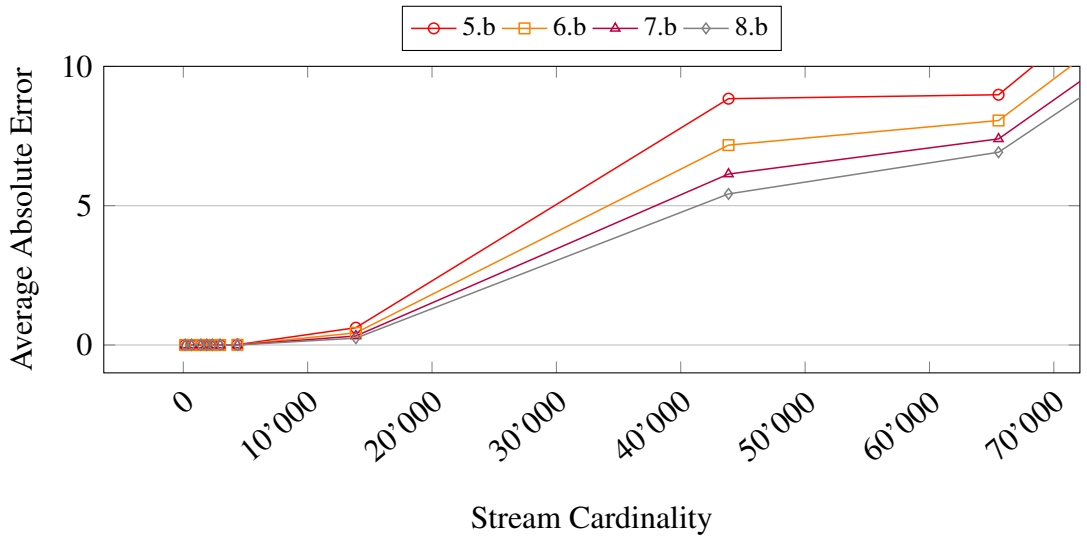


Figure 4.10: Count-Min-Average absolute error vs. Stream cardinality for fixed P_{CM}=13 & different values of R_{CM}. The average is computed with respect to the number of queried items.

reported in the literature, which show that high data skew (i.e., reduced cardinality) is beneficial for the accuracy of Count-Min [50]. Nevertheless, when cardinality is too large and the Count-Min estimate becomes untrustable, the sketch can still be used as a Bloom filter.

4.3.5 Practical Implications

Accuracy requirements depend on the application and the required minimum size of each sketch might differ accordingly. For our specific goal, the characterization of data streams of several billion items, we choose the sketch sizes as follows. We set $P_{\text{HLL}} = 16$ aiming at cardinality (HLL) estimation errors below 0.5%. We set $R_{\text{FAGMS}} = 6$ and $P_{\text{FAGMS}} = 13$, which yields Fast-AGMS errors below 2% for stream lengths on the order of billions. Scaling Count-Min to support large cardinalities requires too many resources on the order of this cardinality regardless of where it is implemented. Thus, we opt to match the Count-Min sketch dimension to that of Fast-AGMS. We then use HLL and Fast-AGMS to assess the trustiness of the results from Count-Min frequency estimates. The intuition is that for large cardinalities and more uniform distributions, Count-Min will produce more errors and be less useful. The results of HLL and Fast-AGMS allow us to identify such situations and discard the results of Count-Min or, at least, indicate the potential for large individual query errors.

As we will show below, the matrix structure in Count-Min and Fast-AGMS is beneficial for FPGA designs as the updates imply R parallel read-modify-write accesses over R *independent* memory regions rather than R accesses competing over a single flat one as it is used for HLL .

4.4 FPGA Implementation

`SKT`[1] is implemented as a customizable Vitis[2] HLS kernel [52] using a streaming architecture operating at an *initiation interval* of one (i.e., $II = 1$). This means that the design consumes and processes inputs with every single clock cycle. `SKT` is completely implemented in C++. Functions and classes are templated in terms of I/O types and behavioral functors to facilitate code reuse among structurally similar modules. The kernel is customized by the parameters identified in Table 4.3. The parameter N is decisive for tuning the design throughput as it defines the number of inputs that are consumed in parallel in each clock cycle by the N structurally unfolded processing pipelines, each consuming one input per cycle.

Note that the underlying high-level synthesis (HLS) technology is itself a compromise. It enables the generation of hardware accelerators from algorithmic descriptions on the abstraction level of systems

[1] `https://github.com/fpgasystems/skt`
[2] Xilinx' High-Level Synthesis Unified Platform Software

software. This results in improved productivity over designs using register transfer level (RTL) languages. However, HLS code, to a large extent, still reflects that it targets a spatial implementation, often for an FPGA, rather than a temporal CPU program. Unlike the C++ code for the CPU which abstracts away the processor architecture, the HLS code written in C++ still reflects the spatial architecture of the FPGA through the pragma directives that need to be used. Moreover, HLS compilers will regularly require some assistance to produce efficient designs, by empirically trial of the pragma directives. Nevertheless, these pragma directives ensure that HLS compilers produce efficient parallel designs. Listing 4.1 illustrates the use of pragmas to spatially unroll a loop.

4.4.1 Overall Design

`SKT`'s architecture is presented in detail in Figure 4.4 and has a unidirectional data flow across a number of modules that are connected via FIFO queues, modeled as `hls::stream` objects. We present a streamlined depiction of `SKT`'s architecture in Figure 4.11.

Throughout the design, the data is augmented with a `last` flag to mark the end of a particular stream and, thus, of a job in the *pre-processing* phase. The same concept of the last signal is used in Chapter 3 and Chapter 5. The last flag traverses the processing pipeline together with the data both through *Hashing* and *Hash Slicing* phases. Its arrival at the *Sketching* memories triggers the switch from consuming sketch updates to outputting their accumulated content and resetting their state. On the output path, the N parallel, partial sketch computations are merged in the *Associative Reduction* phase. The resulting complete sketches are further *processed* in the *Evaluation* phase so as to compute the final results for HLL and Fast-AGMS. Finally, the results of all sketches are concatenated in the *post-processing* phase into a single output stream that is written to a data structure in the memory of the host CPU. The sketch results are complemented by a few trivial metrics maintained during the processing: minimum, maximum, and count of all processed inputs.

The design consumes a customizable number of N parallel inputs in a single clock cycle. This requires that every sketch memory is able to process N updates. However, memory ports are an expensive and

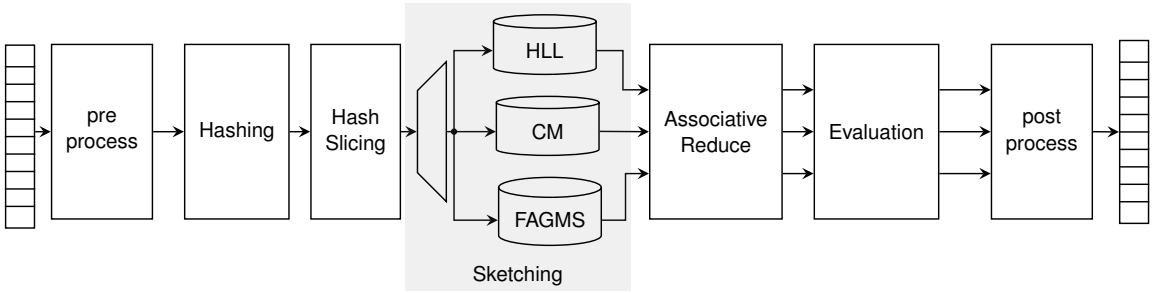


Figure 4.11: `SKT` simplified block diagram.

scarce resource. The on-chip block RAMs (BRAMs) have two such ports. Considering that each update comprises both a read and a write operation, a single BRAM can only sustain the rate of a single update per clock cycle. Therefore, the parallel pipelines must construct partial sketches in *independent* memories that are merged for the final sketch output. For the same reason, SKT ensures to update the rows of the matrix sketches (Count-Min, Fast-AGMS) in parallel and assigns to each row its own independent memory.

In Figure 4.11, for each pipeline the forking of the input path to all the partial parallel sketches occurs after hashing the input. In software, this re-use of computed intermediates has a runtime benefit. In SKT, on the FPGA, parallel identical computations do not improve accelerator performance but increase its resource usage and power consumption footprint.

4.4.2 Hashing & Hash Slicing

Strong hashes, such as Murmur3, involve a series of structurally complex operations like multiplications. In RTL, the design of a such hash function requires manual algorithm decomposition and pipelining in order to meet the throughput goals. By using Vitis HLS tool, the purely functional description of the hash computation is mapped and pipelined *automatically* to meet the targeted clock frequency and throughput goals. As the hashing is an acyclic, stateless computation, there are no systematic limits to the pipelining granularity. The multiply and rotate (logic) operations of the hash function are mapped to dedicated FPGA DSP slices which contain pipeline registers that maintain the speed and efficiency of the design.

After distributing the same wide hash to the individual sketches, each sketch slices it up in different ways to derive appropriate state updates. All elementary state updates are ultimately represented as key-value pairs comprising the address of the memory location to update and the update value. In the case of HLL, the address is simply a prefix slice of the hash and the update value is its ranked tail. The update function applied to the sketch memory is the maximum with its current content. The matrix sketches, Count-Min and Fast-AGMS, feed non-overlapping parallel hash slices to the individual sketch rows. For Count-Min, a single row slice directly identifies the sketch memory location to increment. In the case of Fast-AGMS, the hash slice provides both a counter address and an extra bit determining the sign of the update step.

4.4.3 Sketching

The maintenance of the sketch memories is the most involved part of the design as it must explicitly account for the structural constraints on the FPGA to maximize performance. In addition to the memory port limitation, there is a state-carried data dependency that conflicts with the memory update latency of two cycles. The Vitis HLS tool needs manual assistance to circumvent these dependencies that would otherwise result in a higher initiation interval (i.e., $II > 1$). The sketch memories, therefore, maintain a history of the most recently issued sketch updates in a shift register. If another update to an address

within this history is encountered, it is based on the buffered write-back value rather than a stale read from memory. With this bypass in place, read-after-write hazards are masked and the initiation interval of a single clock cycle can be maintained. A `dependence` pragma explicitly allows Vitis HLS to disregard the data dependency carried through the memory state. These design details are not exposed to the sketch user, they are encapsulated inside a generic `Collect` class that is re-used among all `SKT` sketches.

Listing 4.1: HLL usage example of the collect class.

```
1  // Instantiate Custom Key Space and Storage Type
2  static Collect<ap_uint<P_HLL>, T_RANK> collector[N];
3  #pragma HLS array_partition variable=collector dim=1

5  // Unrolled, i.e. Parallel, Operation
6  for(int i = 0; i < N; i++) {
7  #pragma HLS unroll
8      collector[i].collect(ranked[i], sketch[i],
9          // Lambda-based Update Customization
10         [](T_RANK a, T_RANK b) {
11             return std::max(a, b);
12         }
13     );
14 }
```

Listing 4.1 illustrates how the `Collect` class is used by the HLL sketch inside `SKT`. The specialization of this class and its behavior are achieved through template parameters. They define (a) the dimension and type of the backing memory array (line 2), and (b) the state update function, i.e., maximum operation for HLL, which is specified through a lambda expression (lines 10-12). Note the custom key space, ap_uint<P_HLL>, and storage type, T_RANK. Using the HLS integral type *ap_uint<n>*, the bit width of signals and, in this case, the address space can be controlled precisely. This code instantiates N parallel memories to serve the corresponding number of data inputs (line 2). Vitis HLS must be prevented from flattening this added outer dimension by a pragma (line 3) so that designated memory ports remain available for each instance. The operational behavior is wrapped into the class member function *collect*, which is invoked within an unrolled and, hence, parallel loop (line 6) across all memory instances. Each instance consumes and processes updates from an hls::stream ranked[i] until encountering an update carrying an asserted last flag. Only then the design starts streaming the accumulated partial sketch to the hls::stream sketch[i].

4.5 Experimental Evaluation

In this section we introduce the CPU and FPGA setups used for deploying and testing SKT capabilities.

4.5.1 Setup

SKT has been prototyped and tested on the Xilinx Adaptive Compute Cluster (XACC) [227] at ETH Zurich, later referred to as the Heterogeneous Accelerated Compute Clusters (HACC) [13]. Before transitioning to server class AMD machines (CPUs), the XACC cluster was populated with high-end server class Intel machines (CPUs). The software evaluation of SKT is done using the Intel machines, one host server with 2 Intel Xeon Gold 6248 with 376 GB RAM , and another with one Intel Xeon Gold 6234 with 376 GB RAM. For the FPGA deployment, we use the Alveo U250 and Alveo U280 accelerator cards.

We conduct RAM-to-RAM experiments for establishing performance baselines on both server platforms and for evaluating the SKT hardware acceleration. Finally, we also evaluate the direct sketching of data received over a 100 Gbps TCP/IP network link both in software and on the accelerator. In all experiments, the final sketch results are written back to the CPU main memory.

All measurements are conducted over data streams of 32-bit integers subjected to a randomizing 128-bit Murmur3 hash function. The 32-bit integer datatype is big enough to accommodate large cardinalities that escape their straightforward and concise characterization by histograms. It is compact enough to prevent a squashing dominance of the data I/O bounds over the actual sketching performance. Larger data types make the hash computation iterate over longer input stretches and, thus, yield fewer sketch updates per input volume. A similar reductive effect occurs for extracting fields of interest from structured data. Making this field extraction runtime-programmable is trivial and would add the capability to adjust the processing to changing data schemas. While a software implementation would have to pay for this flexibility with compute time, the accelerator would be able to accommodate this functionality in a pipeline extension while impacting neither throughput nor critical sketch memory resources.

We evaluate the individual performance of each sketch algorithm (HLL, Count-Min and Fast-AGMS) using the same $P_* = 13$ in order to observe individual throughput performances. The reference dimensions for SW-SKT are $P_{\mathrm{HLL}} = 16$ and $R_* \times P_* = 6 \times 13$ for Count-Min and Fast-AGMS.

4.5.2 Software Baseline

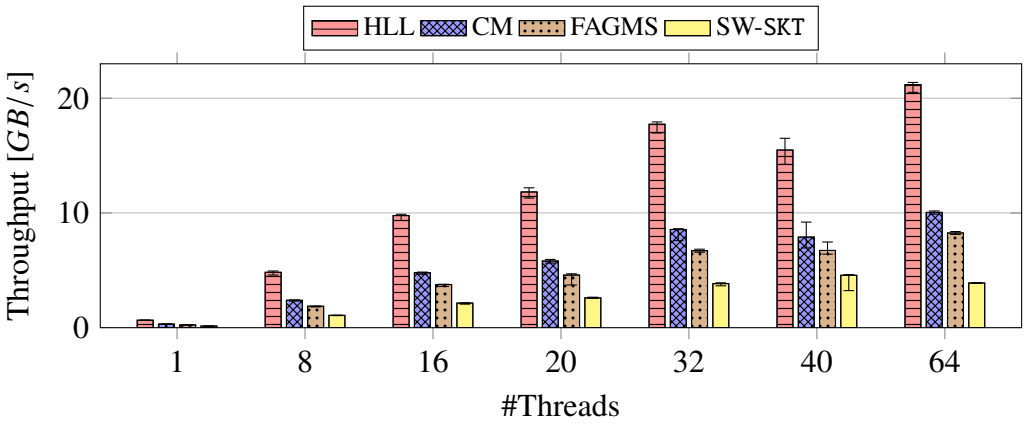
In order to assess the FPGA accelerator, it is important to understand the I/O and compute bounds of the cluster platform. For this we first assess the performance of a multi-threaded software baseline on the cluster's machines.

Methodology. Each of the three algorithms (HLL, Count-Min and Fast-AGMS) benefits from thread parallelism, with each thread maintaining a per-thread data structure (partial sketch) for updates. The partial sketches from all threads are merged once all the input data has been processed. In this multi-threaded CPU implementation, threads can be equated to the parallel pipelines of the hardware design. The data flow of the CPU implementation follows the main idea behind SKT's architecture in Figure 4.11, namely: (1) compute the hash value once for each data input item, (2) update the corresponding partial data structures for each of the three algorithms, and (3) evaluate the sketches after processing the last value.

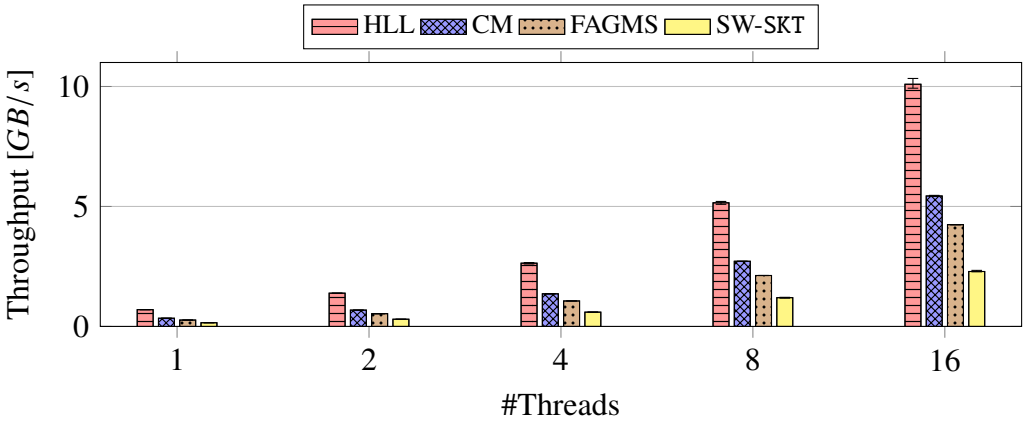
Performance analysis – data samples in CPU's main memory. We analyze the individual performance of each of the algorithms (HLL, Count-Min and Fast-AGMS) running in isolation as well as the performance of all three algorithms running jointly under the name SW-SKT. The experimental platform setup comprises of: (1) 2× Intel® Xeon® Gold 6248 Processors @2.5 GHz with a total of 40 cores and 80 hyper-threads which we label *Alveo0* and present its performance capabilities to compute the sketches in Figure 4.12a, and (2) a single Intel Xeon Gold 6234 Processor @3.3 GHz with 8 cores and 16 hyper-threads, labeled *Alveo3b* and having its compute capabilities assessed in Figure 4.12b. For the remaining of the chapter we are going to address these two machines by Alveo0 and Alveo3b, respectively.

The observed sketch throughput scales with the allocation of CPU threads on each of the two machines. The first three bars (HLL, CM, FAGMS) show similar throughput, which is roughly twice the throughput of the fourth bar (SW-SKT). This can be explained by the fact that HLL, Count-Min and Fast-AGMS have each small data structures that fit into the cache, whereas SW-SKT holds three times as much data as the individual algorithms and thus it does not fit into cache. The observed sketch throughput scales with the allocation of CPU threads on each of the two machines. Figures 4.12a and 4.12b show that each algorithm's performance is correlated with the complexity of its conducted update operation. HLL performs a single update into a linear data structure. This results in the best individual performance. Count-Min and Fast-AGMS each perform $R_*\times$ more elementary counter updates than HLL. This reduces their individual performances considerably. The further difference between Count-Min and Fast-AGMS is solely due to their different update operations. While Count-Min performs *unconditional* increments on the counters identified for every row, Fast-AGMS extracts another hash bit for each counter update to *select* the sign of the increment.

When fusing all three algorithms, SW-SKT combines their compute and I/O bounds and attains a peak throughput of 4.56 GB/s for 40 compute threads, i.e., the physical core count of the Alveo0 machine. SW-SKT's compute intensity puts a lot of strain on one CPU thread, registering a performance of only 0.3 GB/s. The thread increase efficiency peaks at 8 threads, with 62.5% efficiency. Even if more threads are allocated, the efficiency decreases, reaching 60% for 16 and 20 threads, and 56% for 32 threads. While all individual sketches and their combination in SW-SKT compute the same input hash, their throughput are clearly differentiated by the performed sketch updates. The observed performance decreases with the number of required counter updates and with the complexity of the increment operations to perform.



(a) Host server: Alveo0



(b) Host server: Alveo3b

Figure 4.12: CPU baseline on the host servers. Median throughput for HLL, Count-Min (CM), Fast-AGMS (FAGMS) and SW-SKT. Error bars indicate the 5th and 95th percentile. Stream length of 1B.

Performance analysis – data samples from the TCP/IP network. We illustrate in Figure 4.13 the performance of SW-SKT when the data samples are received from the TCP/IP network compared to the available network bandwidth. The experimental setup consists of the Alveo3b machine receiving data from multiple TCP/IP clients. SW-SKT throughput performance scales with the number of threads allocated for receiving the data and computing the sketches. A maximum throughput of 2.28 GB/s is reached for 16 threads. This result is similar to the in-memory performance of this machine but significantly lower than the bandwidth the network can deliver without any subsequent data processing. In Section 4.5.4 we show how Alveo0 can deliver better SW-SKT throughput results since multiple thread can be allocated independently for receiving the data and computing the sketches.

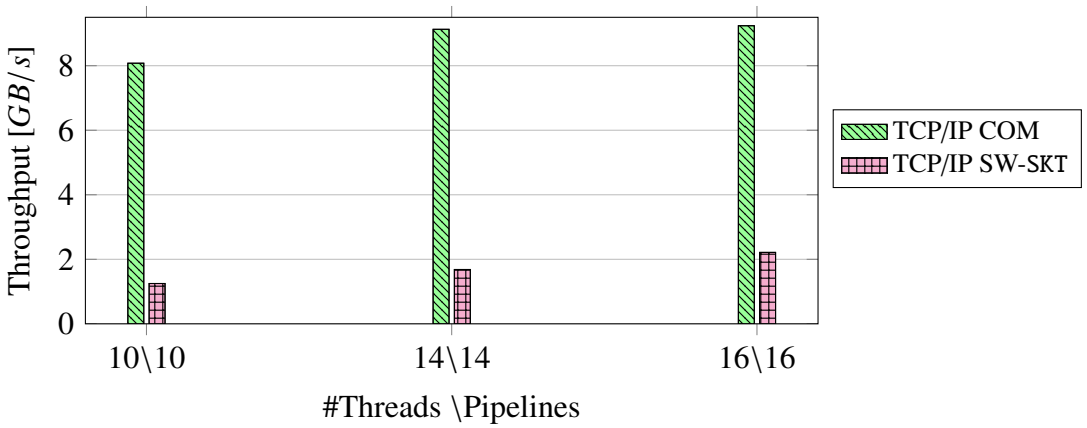


Figure 4.13: Mean throughput for TCP/IP communication and SW-SKT with samples coming from the TCP/IP network. Stream length of 1B.

4.5.3 Sketches on a Co-processor

For the first evaluation on the FPGA, the SKT compute kernel is interfaced directly from the host as a free-running OpenCL kernel within the AMD-Xilinx QDMA shell on an Alveo U250 accelerator card operated under Vitis 2020.2. The Alveo U250 deployment environment together with the software framework running on the host server is illustrated in Figure 4.14. The compute kernel is configured to process streams comprising 32-bit data items, which are hashed by a strong 128-bit Murmur3 hash. The design operates at the default platform clock of 300 MHz. The input parallelism can be scaled up to 16 parallel data lanes fit in by the 512-bit user kernel interface. This parallelism and the sketch dimensions are explored and evaluated experimentally.

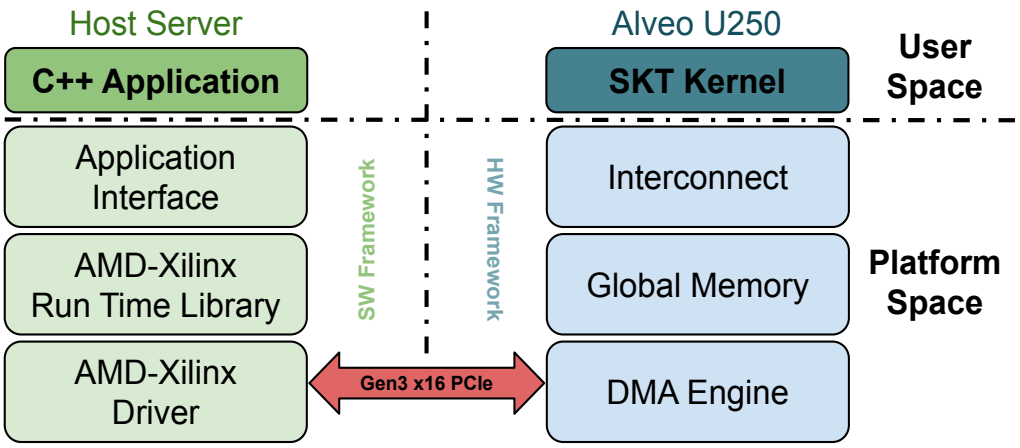


Figure 4.14: FPGA connected as a co-processor [Alveo U250]-QDMA shell platform overview.

Throughput Scaling. A single data pipeline processing 32-bit data inputs at 300 MHz requires a bandwidth of 1.2 GB/s. This figure is independent of the computed sketches. We aim at matching the number of parallel pipelines (N) with the maximum kernel interface bandwidth. The maximum theoretical bandwidth for the PCIe Gen3 ×16 connection on the Alveo U250 card is given by Equation 4.6. For our PCIe connection, the formula reduces to 15.62 GB/s =(8G * 16 * (1 - 2/130) - 1G)/8, a physical bandwidth limit just below 16 GB/s.

$$PCIe_{max}[Gbps] = speed * width * (1 - encoding) - 1Gbps \tag{4.6}$$

Bus interference and protocol overhead imposed by both PCIe and the QDMA shell reduce the actually achievable figure. Starting the exploration with $N = 16$ parallel pipelines ensures that we identify the actual limit of the platform rather than a bottleneck in SKT implementation. Experiments are run from small stream sizes of 1000 data items up to streams of 500 million data items. The job size limit imposed by the QDMA shell is reached for 16 streams, each containing 500 million items. Currently, the shell driver truncates any larger job *silently*. In order to mitigate this drawback of the shell, one would have to add a scalar kernel parameter or a custom streaming protocol layer to aggregate multiple QDMA transmissions before evaluating the collected sketches. Then, larger jobs could be partitioned and handled by multiple QDMA invocations. Yet another service management layer would have to ensure the integrity and atomicity of these partitioned jobs in contended use contexts. This modifications would conceal the real benefit of streaming interface on the FPGA.

Figure 4.15 shows the data throughput achieved for these experiments as measured in the host application from initiating the streaming to the accelerator to completing the reception of the concatenated computed sketches. Hence, this measurement includes the communication initiation overhead as well as the sketch

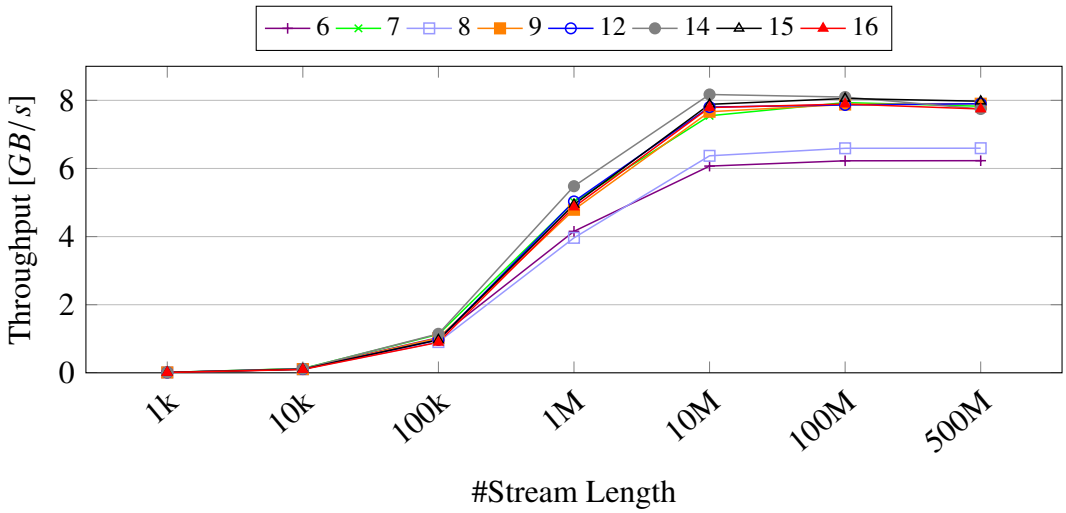


Figure 4.15: SKT on QDMA Shell-FPGA as a co-processor. Mean throughput observed from the CPU.

evaluation and transmission times, which are all independent of the size of the input stream. Actual observations experience a variation of ±5% around the shown means.

Note that the Count-Min sketch, which is transmitted verbosely, already has a size of $R_{CM} \times 2^{P_{CM}}$ 32-bit words. Its write back completely dominates the observed accelerator latency for inputs up to 10^5 items. For larger stream sizes, the impact such one-time costs diminishes. The throughput is observed to saturate around 8 GB/s even for $N = 16$. In conclusion, only a reduction down to $N = 6$ parallel processing pipelines would be expected to establish a processing bottleneck. However, more reductions of parallelism were causing notable performance degradations. Apparently, the assertion of back-pressure across this boundary causes disadvantageous interactions with the flow control of the QDMA infrastructure. While the FIFOs eliminated this effect completely for most settings, a decrease in performance remained reproducible for $N = 8$ parallel processing pipelines. The degradation of the performance for $N = 6$ is expected, since the 6 pipelines can only consume 7.2 GB/s, causing the shell to experience back-pressure. Nevertheless, the throughput of 8 GB/s that is achieved by 9 or more pipelines is 1.75× higher than what the 40 cores of the dual-processor *Alveo0* system were able to attain.

Sketch Optimization. Table 4.4 shows the utilization of the FPGA hardware resources, which are available to the user kernel, by selected `SKT` configurations. Note that `SKT` consumes a small fraction of the general-purpose combinatorial and sequential fabric logic, i.e., lookup tables (LUTs) and registers (FFs), respectively. Also, the utilization of arithmetic DSP slices is moderate (<10%), with less than 1% being used for sketch evaluations, while the rest is used for hash computation. Hence, there is the correlation with the number N of input lanes. The critical and limiting resources type is the on-chip memory provided by Block RAM (BRAM) tiles. As shown in Table 4.4, the QDMA integration of `SKT` can scale beyond the fixed sketch size of $P_{\text{HLL}} = 16$ and $R_* \times P_* = 6 \times 13$ even for $N = 16$ parallel pipelines.

To utilize memory resources optimally, a good understanding of their demand in relation to the kernel parameters is needed. Individual BRAM tiles serve a 10-bit address space with 36-bit data. Larger address spaces are assembled from multiple memory tiles. In an RTL design a traditional manual memory allocation would use one memory location for each of the 32-bit counters for Count-Min and Fast-AGMS, leading to a higher resource consumption. In HLS, the memory allocation is done automatically and can

Table 4.4: User budget resource utilization on Alveo U250.

$N \times R_* \times P_*$	**LUT**	**FF**	**BRAM**	λ	**DSP**
$6 \times 6 \times 13$	43082 (2.8%)	63279 (2.0%)	624 (25.1%)	0.93	476 (3.9%)
$9 \times 6 \times 13$	59614 (3.9%)	86216 (2.7%)	936 (37.7%)	0.93	662 (5.4%)
$16 \times 6 \times 13$	91530 (6.0%)	138945 (4.4%)	1664 (67.0%)	0.93	1096 (8.9%)
$16 \times 8 \times 13$	115851 (7.7%)	172030 (5.4%)	2144 (86.4%)	0.93	1104 (9.0%)

be optimized by the tools. Instead of allocating one memory location for each counter, the tools avoid memory fragmentation, claiming storage space more optimally. This is seen especially for the matrix sketches whose 32-bit counters are dominating the memory resource consumption figure. For the HLL sketch, the memory geometry of each BRAM tile is reshaped into a 12-bit address space of 9-bit data. This suffices to accommodate the ranks whose range is bounded by the bit width of the consumed hash value. This layout transformation is directly supported by the physical BRAM structures. All these considerations give rise to the BRAM utilization model in Equation 4.7 for sketch parameter settings with $P_{\text{HLL}} \geq 12$ and $P_{\text{CM}}, P_{\text{AGMS}} \geq 10$:

$$U_{\text{BRAM}} = \lambda \cdot N \cdot \left(R_{AGMS} \cdot 2^{P_{AGMS}-10} + R_{CM} \cdot 2^{P_{CM}-10} + 2^{P_{HLL}-12} \right) \tag{4.7}$$

with $\frac{W}{36} \leq \lambda \leq 1$, with W representing the counter's width. The model introduces an extra coefficient λ that reflects the automated memory layout optimization of the HLS. An optimal recycling of all 32-bit counters memory fragmentation would, at best, allow the reduction of the memory footprint to $\lambda_{\text{min}} = \frac{32}{36} \sim 0.89$.

The λ(s) are tabulated with the designs in Table 4.4. They demonstrate that Vitis HLS is able to assemble BRAM resources for the sketch storage very efficiently, close to the projected optimum. However, it must also be noted that Vitis was unable to complete the hardware implementation for any sketch design with a projected BRAM tile utilization above 90%. This limit is due to signal routing challenges inside the FPGA that put the tools into a boundless optimisation problem. A way out is to resort to the optimization techniques available in RTL but this would defeat the aim of implementing the system as an HLS design.

In summary, a SKT implementation that serves the full 512-bit kernel interface with sketch dimensions of $P_{\text{HLL}} = 16$ and $R_* \times P_* = 6 \times 13$ is feasible. The FPGA resource utilization permits to directly increase the row count to $R_* = 8$. Even larger sketches are attainable by reducing the number of parallel pipelines N. There are no observable throughput penalties as long as $N \geq 9$. The developed utilization model for the critical BRAM resources is valid across the data center AMD-Xilinx platforms and also guides the assessment of the network integration.

4.5.4 Sketches on a SmartNIC

For the TCP/IP network experiment, SKT is used as a free-running OpenCL kernel within the AMD-Xilinx XDMA shell on the Alveo U280 accelerator card. The shell and accelerator card differ from the ones used for SKT's stand-alone evaluation with data residing in the CPU memory. At the time this experiment was run, the ability to stream data from the CPU was only available in the QDMA shell on Alveo U250 and the 100G TCP/IP network stack [191] was only available under the XDMA shell on the Alveo U280 platform [85, 86]. Hence, for the network integration experiments, we used the XDMA shell. At the time of writing the thesis, the QDMA shell has been disrupted and EasyNet has been extended to all the Alveo boards in the HACC cluster, i.e., U280, U250, U50 and U55c.

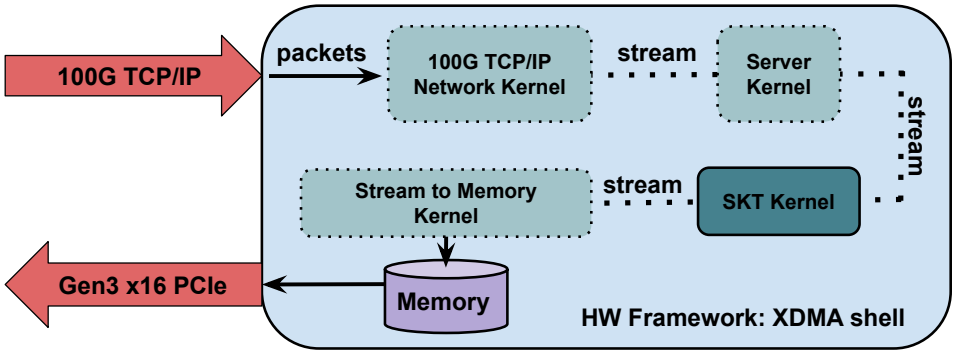


Figure 4.16: FPGA with TCP/IP network [Alveo U280]-XDMA shell instantiation of user space kernels.

Feeding the data from the network avoided the job size limit imposed by the QDMA shell. As the two shells ran on different boards, U250 for QDMA and U280 for XDMA, SKT faced different operating conditions. First, the kernel clock frequency of 250 MHz is lower within the XDMA shell. Second, there are fewer available BRAM tiles. In order to fit onto the U280, SKT had to shrink. To implement the 16 parallel pipelines needed to sustain the 100 Gbps line rate, we needed to reduce the number of rows for the matrix sketches to $R_* = 5$. We did so for the purposes of the experimental evaluation here.

Methodology. Figure 4.16 illustrates the chain of kernels used by our system. It processes the data received from the 100G TCP/IP network and passes the results to the server through a Gen3 ×16 PCIe connection. The *100G TCP/IP Network Kernel* instantiates the 100G Ethernet subsystem, provides TCP/IP functionality, and converts the data packets to a flat data stream. The *Server Kernel* listens on the network, accepts TCP/IP connections, and forwards the incoming data to the SKT kernel. When all data has arrived, the server kernel asserts the last signal communicating the completion of a job so as to trigger the result generation. The *SKT Kernel* consumes the received data stream and generates a sketch stream upon job completion. The output is sent to the host CPU memory by the *Stream to Memory Kernel*, from where the results are read by the host server via the OpenCL API. This last kernel is needed as an adapter to the XDMA shell, which does not support streaming interfaces natively. Except for the free-running SKT kernel, all the other three kernels are explicitly controlled by the application running on the host server.

Performance analysis – data samples from the TCP/IP connections. Our network experiments are summarized in Figure 4.17. They show that SKT needs 16 pipelines in order to support the 100 Gbps network line-rate. Whenever fewer pipelines are allocated, e.g., 14 or 10, the sustained line-rates drop sharply to 3.35 GB/s and 2.77 GB/s, respectively. This drop is caused by the way the 100G TCP/IP network kernel expects its output to be consumed. More specifically, the network kernel expects the following kernel to consume 512-bit data for a 100 Gbps line-rate. This amounts to 16 parallel lanes

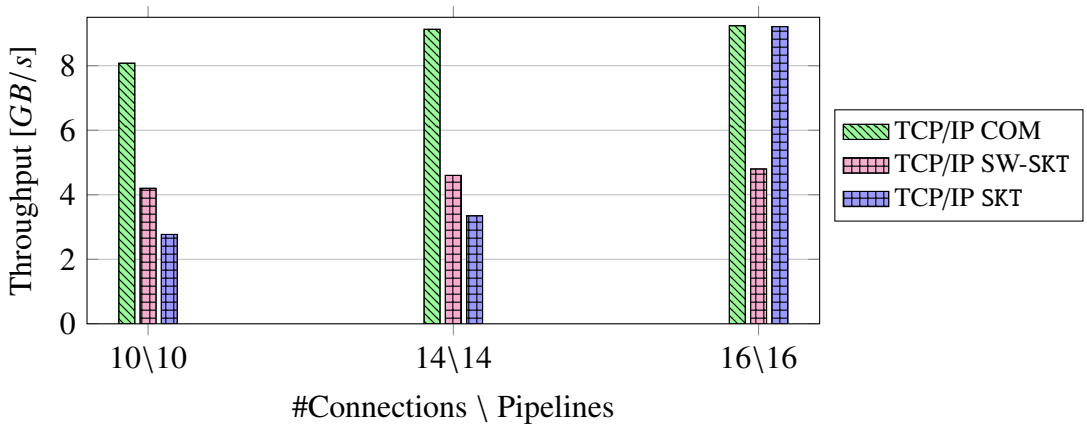


Figure 4.17: Mean TCP/IP throughput for remote communication (COM) and remote sketching over 1B samples by SW-SKT (Alveo0) and by SKT inside XDMA shell (Alveo U280 FPGA connected to Alveo3b).

of 32-bit data. If SKT, for example, only implements 14 parallel pipelines, it consumes 448-bit in every cycle, postponing the remaining 64 bits in a buffer. At 100 Gbps, the accumulation of the buffered data eventually triggers the flow control of the network kernel. This mechanism relies on external memory whose throughput within the XDMA shell is much lower than 100 Gbps.

For comparison, SW-SKT is used to sketch the same data received over the network on the 40-core Alveo0 machine. So as to hide the TCP/IP stack of the OS as a bottleneck, the traffic is split across parallel TCP/IP connections. A designated receiver thread is forked to service each of these connections. Without any processing of the received data, this setup is able to sustain a data throughput of almost 10 GB/s as shown by the columns labeled 'COM' in Figure 4.17. The actual sketching is challenging enough to warrant another level of workload distribution. We show the highest throughput figures, which are obtained by backing each receiver thread by four compute threads. The receiver only delegates data blocks through a job queue and guarantees a steady consumption of data from the TCP/IP socket. As in the FPGA-accelerated system, a throttling of the data input by back-pressure proved to impact the achievable system performance negatively. An overprovisioning of TCP/IP connections was not able to mitigate this effect. In the background, each compute thread maintains its own partial sketch so as to confine the synchronization overhead to the conclusive sketch merger. As shown by the SW-SKT in Figure 4.12, even the parallelization of the sketch computation across all cores fall short of attaining a processing throughput of 5 GB/s. This is consistent with the performance observed for the RAM-to-RAM deployment of SW-SKT but only half of the performance achieved by the network-facing FPGA accelerator.

4.6 Discussions

Before concluding the chapter, we share our HLS experience and compare the performance of SKT and SW-SKT with Apache Spark™, an open-source analytics engine for both single-node and large-scale data deployment and processing.

4.6.1 HLS Experience

We evaluate our experience from using high-level synthesis with a C++ hardware design entry. This approach comes with clear productivity benefits, in particular:

- An easier *accessibility* for software engineers.
- A ready application *integration* stack.
- Automated *pipelining* for high accelerator performance.
- Automated *memory layout optimization* beyond what would be reasonably maintainable in a manual RTL design.

The price paid for these gains is best illustrated by the device resource utilization figures achieved. Sketch sizes that would demand more than 90% of the user budget of BRAM tiles failed consistently to implement. Mutually amplifying utilization and timing challenges are not uncommon in classic hardware design in RTL. However, the added layers of abstraction in HLS severely limit the ability to resolve such situations. Engaging in floorplanning (where each part of the design goes), manual placement, or even manual signal routing would clearly defeat the purpose of using HLS.

Finally, the use of FPGAs is always a question mark due to the complexity added over software programming. SKT has been programmed entirely in High Level Synthesis (HLS), using a version of C++ that is accessible to software programmers as many of the details of the FPGA architecture are hidden behind pragmas and high-level abstractions provided by the development environment.

4.6.2 Cardinality Computation using Apache Spark

We have performed an end-to-end comparison between: (1) Apache Spark running locally (i.e., with data located on the CPU's memory) and computing approxCountDistinct function (Apache Spark's internal HLL) over the data; (2) the software implementation (SW-SKT), and (3) the FPGA implementation (SKT), in order to contrast the performance measurements.

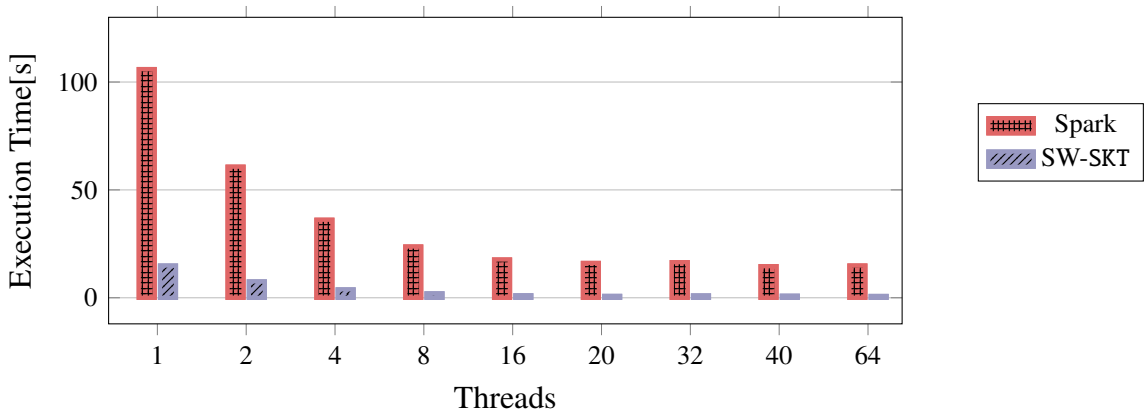


Figure 4.18: Cardinality computation using SW-SKT vs. Apache Spark.

For each of the three experiments, we read data from a binary file stored on the local disk and compute (1) approxCountDistinct with Spark; (2) all three sketches (HLL, Fast-AGMS and Count-Min) with SW-SKT and (3) SKT on FPGA. Finally, we query the results for the HLL estimate in each of the three cases. We run the experiments on the Alveo0 machine and include the I/O times in the numbers presented in Figure 4.18. We do not include the FPGA numbers into the figure due to large difference in performance between Apache Spark and SKT. Nevertheless, for the FPGA implementation, SKT performance is 0.5s (8 to 16 Pipelines) and 0.55s (6 Pipelines) for end-to-end data characterization of 500 million items.

We run Apache Spark in local mode since our current implementations of SW-SKT and SKT target a single machine rather than a distributed deployment. The results show that SW-SKT and SKT, which both compute 3 sketches in parallel, have an end-to-end execution time that is one order of magnitude faster than Apache Spark, which only computes HLL. The larger execution time of Apache Spark can be explained by the series of transformations data needs to pass through before entering the approxCountDistinct function. After being read from the binary file, individual data items need to be converted to integer values that are represented as a row of values. Before all these values are passed to the approxCountDistinct function, they need to be transformed to a columnar representation.

Apache Spark is a trade-off between performance and flexibility. For instance, doing the same experiment with text files would add even more overhead to its execution time. Based on this, we demonstrates that SW-SKT is a fair and competitive baseline for evaluating the hardware acceleration.

4.6.3 Summary

In this chapter, we show that characterizing data sets and streams (i.e., obtaining information over the data such as cardinality and data distribution) can be an expensive operation on conventional CPUs, even when computation is parallelized across many cores and processors, both when the data is in the host CPU's

memory or is streamed over the network. With SKT, on the FPGA, data characterization information can be obtained with one-pass over the data and used as meta-data by further processes.

SKT computes three widely used sketch algorithms, HyperLogLog, Count-Min and Fast-AGMS, that complement each other in terms of the information they provide and two of which (HyperLogLog and Fast-AGMS) can be used to identify data distributions where the accuracy of Count-Min is compromised.

SKT architecture is based on an extensive design space exploration to identify the interesting trade-offs in terms of performance, resource utilization, and accuracy. The experimental evaluation demonstrates that SKT running on a single FPGA can match the performance of 70 CPU cores. Furthermore, SKT is deployed on a smartNIC to demonstrate its capability to directly process data streams at 100 Gbps from the network. The technological evolution of FPGAs has enabled the integration of more and more resources on FPGA devices. This has opened the door for deploying increasingly complex designs, such as SKT, on FPGA accelerators. SKT is a complete data center solution that has been evaluated in an integrated end-to-end system context.

In addition to an effective solution to the data characterization problem, we also provide general guidelines for designers interested in taking advantage of the heterogeneous, special-purpose hardware acceleration that is defining today's advances in massive data processing.

Chapter 5

Offloading I/O operations to the FPGA - Security

In the previous two chapters (Chapter 3 and Chapter 4) the attention was primarily directed towards the analysis of FPGA offloading of complex and compute-intensive data analytics algorithms. Before transitioning database engines to heterogeneous computing models [140], it is crucial to examine if tasks that are presently performed by the database engine but hold secondary importance in terms of the engine's efficiency, can be offloaded to accelerators.

This chapter explores the use of an FPGA-based accelerator placed on the I/O path, within the context of SAP HANA being deployed as a cloud service. More specifically, it focuses on the analysis of how the 256-bit key AES (Advanced Encryption Standard) encryption/decryption can be offloaded to the FPGA, namely on the architecture, implementation and deployment of AES.

The work presented here is part of a broader effort that encompasses the offloading of I/O path operations such as compression and encryption to an FPGA-based accelerator [42].

This chapter is extended by a Master thesis project that automates the analyzes and testing of the AES module, comprising also the 128-bit and 192-bit key sizes for the block cipher modes discussed here.

5.1 Motivation

The transition from traditional in-house database deployments to a Software-as-a-Service (SaaS) model has introduced architectural challenges and opportunities for the database ecosystem. While computing nodes can be time-shared by a much larger number of client instances, multi-tenancy often raises concerns and requires data to be secured at any given state, namely when it is at rest (i.e., it is not accessed and stored on a physical or logical medium), in transit (i.e., it is moved from one point to another, or in use (i.e., it is consumed by applications or accessed by the users). SaaS databases need to ensure that data is encrypted while in transit or at rest, with cloud vendors providing this option as a key element of cloud databases [20, 21]. If data is encrypted on its way out, that means it has to be decrypted on the way in, demanding even more resources to deal with I/O operations.

Computing resources in the cloud are a commodity affecting the costs of the service. Even if encryption and decryption are external to the database system, they compete with the engine for computing resources. Moreover, they occur on the latency-sensitive I/O path. I/O has been a crucial aspect of the design of database engines from their inception. While the reasons to optimize the I/O path have varied over time, it remains a critical aspect of a well functioning engine and plays a large role in its overall performance. In the cloud, storage disaggregation makes the I/O path even more critical, due to the physical separation of compute and storage resources. Therefore, any performance bottlenecks in the I/O path can significantly degrade the ability of the system to process data in a timely manner.

While posing these challenges, the cloud also offers architectural opportunities over conventional local deployments through the variety of boards and chips that coexist with traditional compute nodes, namely hardware accelerators (i.e., GPUs, FPGAs) and specialized architectures (i.e., ASICs, TPUs). This diverse environment allows different algorithms to be processed at different points in the architecture, with an increasing number of operations being offloaded away from the CPU. To take advantage of this heterogeneous setting, this chapter focuses on offloading the I/O path encryption/decryption operations to an FPGA-based accelerator, and analyses the encryption behavior when coupled to the DEFLATE compression algorithm. The I/O traffic traces used for testing are extracted while running SAP HANA on a common loading operation that entails substantial traffic.

The obtained results demonstrate the advantages of the design: data can be efficiently encrypted on its way to storage (at up to 15 GB/s, and at around 4 GB/s when combined with compression [42]). On its way from storage, data can be decrypted at up to 15 GB/s. For certain data loads, data can be processed at a higher rate and more efficiently than using the CPU, which can be significantly slower (i.e., up to one order of magnitude) for certain loads. In addition to these findings, when combined with a reduction algorithm such as compression, a non-parallelizable algorithm deployed on the FPGA demonstrates better throughput performance when compared to the CPU.

Nevertheless, the picture that emerges is not a clear win for hardware acceleration. The gains are not uniform across all I/O sizes, with the biggest gains being observed for larger data transfers. After analyzing some of the I/O patterns produced by SAP HANA, the conclusion is that there is a solid case in favor of using the accelerator from the perspective of the overall I/O traffic, even if the accelerator does not gain much when I/O requests are small. The emphasis on analytical workloads proves beneficial in this context, since I/O is mostly asynchronous and I/O requests can potentially be batched (SAP HANA already does it to a certain extent). Our results show that the approach we have explored is unlikely to provide a huge advantage in transactional workloads due to the need to implement synchronous I/O and the small I/O sizes involved. Nevertheless, actual deployments can opt for encrypting data on-the-fly.

The subsequent sections of the chapter are organized as follows. In Section 4.2, we give an overview of the SAP HANA engine, the encryption algorithm we analyze, and of the related work. In Section 4.3, we present the design of our stand alone encryption module and how it connects to the compression module. Before concluding the chapter with a discussion on our findings in Section 5.6, we evaluate the throughout performance of our design in Section 5.5.

5.2 SAP HANA, Encryption and Hardware Acceleration for Databases

In this section, we briefly describe SAP HANA, the encryption algorithms we analyze and the potential impact of hardware acceleration on database operations, together with relevant prior research.

5.2.1 SAP HANA

SAP HANA [68] is a column-oriented in-memory database specifically built to integrate both analytical and transactional workloads into a single engine. It can be deployed on-premise or as part of SAP HANA Cloud[1] as a SaaS. Together with SAP HANA, the cloud platform also provides SAP IQ, a robust, traditional disk-based database. This work is part of recent efforts from SAP to enhance their on-premise products with the opportunities offered by cloud deployments. For instance, Abouzour et al. [2] detail the process of porting a disk-based data management system to cloud object-storage. In addition to these efforts, we focus on leveraging the latest hardware options in the cloud to address data encryption on the I/O path.

[1] `https://www.sap.com/products/hana/cloud.html`

5.2.2 Encryption - Decryption

The Advanced Encryption Standard (AES) [54] is the most widely used block cipher encryption standard, having its own set of dedicated instructions in both standard CPU [102] and embedded [22] architectures, and being generally adopted also by database engines. AES is a symmetric key algorithm with three possible *initial key sizes*: 128 bits, 192 bits, and 256 bits. We focus on the 256-bit initial key size implementation and define the process of encryption or decryption of a cacheline (512 bits) as AES-256.

The algorithm design is based on a chain of substitutions and permutations, known as *transformation rounds*, which makes it suitable for both efficient software and hardware implementations. However, its block cipher modes of operation cause significant performance differences due to the resulting implementations. We show these differences by implementing three AES block cipher modes: Electronic Code Book (ECB), Counter (CTR), and Cipher Block Chaining (CBC). The data amount that has to be encrypted/decrypted together represents the transformation granularity of a block. For CTR and CBC modes it is given by the data block size, whereas for the ECB mode, the granularity is the 128-bit data word.

Figure 5.1 and Figure 5.2 illustrate the processing of a 512-bit cacheline, with each AES module consuming one 128-bit data word at a time. We differentiate between the parallelizable modes in Figure 5.1 (ECB and CTR for both encryption and decryption and CBC only for decryption), and the non-parallelizable mode in Figure 5.2 (CBC for encryption). ECB is the simplest of the modes, requiring only the transformation rounds for both encryption and decryption, whereas CTR and CBC modes require pre- or post-transformations of the data word before entering or after exiting the transformation rounds. For its pre-transformation, CBC encryption needs the previous encrypted word, thereby creating a data dependency between consecutive iterations. As a result, while the number of transformation rounds does not have an impact on the performance of the parallelizable modes (ECB and CTR), the encryption perfor-

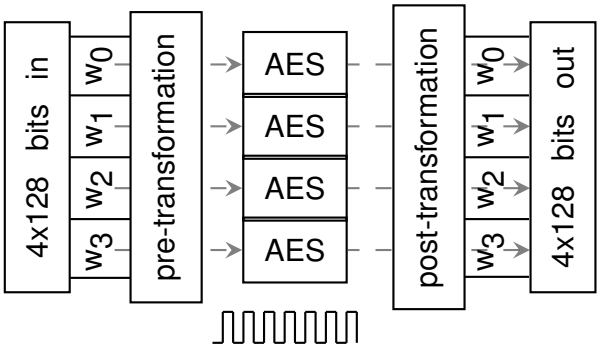


Figure 5.1: Block diagram of one AES-256 module operating in parallelizable mode while processing a 512-bit cacheline (64 bytes).

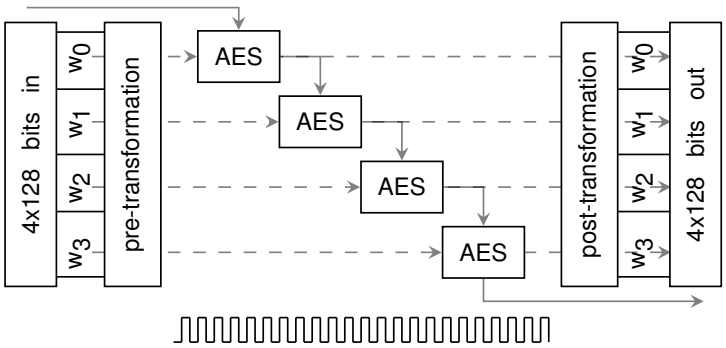


Figure 5.2: Block diagram of one AES-256 module operating in non-parallelizable mode while processing a 512-bit cacheline (64 bytes).

mance in CBC mode is significantly affected by it. The number of transformation rounds a 128-bit data word has to pass through to get encrypted or decrypted is determined by the *initial key size*. Currently three initial key sizes are supported: 128 bits, 192 bits and 256 bits, respectively. In this chapter, we focus on the 256-bit key size and address the process of encryption or decryption of a cacheline (512 bits) as AES-256. In our case, for an initial short 256-bit key size, 14 transformation rounds are needed. In Table 5.1, we distinguish between each initial key size and the number of required transformations rounds associated to it. Each transformation round requires for its internal bitwise XOR operations a dedicated 128-bit *round key*. In column 'No. of Round Keys' of Table 5.1, we add the number of 128-bit words that need to be extracted from the initial key for each transformation round, with one more round key being needed for the initial transformation round. All the required round keys are obtained from the initial 256-bit key through a key expansion process that is independent of the datapath.

Many database systems have software implementations of encryption as well. Oracle uses all three initial key sizes (128, 192 or 256 bits) of the AES algorithm for its Transparent Data Encryption (TDE) [179], employing the 256-bit initial key only for highly sensitive data [180], and relies on Intel's intrinsic instruction set to boost encryption and decryption performance for ECB, CTR and CBC modes. MySQL (v8.0) supports AES encryption in ECB and CBC modes using all three initial key sizes, with ECB and 128-bit initial key being the default block cipher mode encryption option [170]. The same encryption algorithm, AES and 256-bit initial key, is also used by Azure's TDE for its SQL Server, Azure SQL Database, and

Table 5.1: Initial key size and associated number of rounds.

Initial Key Size	Transformation Rounds	No. of Round Keys
128	10	11
192	12	13
256	14	15

Azure Synapse Analytics services [165]. SQL Server relies on the AES modes supported by the Microsoft encryption library: ECB, CBC, CFB (cipher feedback) and OFB (output feedback).

From data security point of view, out of the three modes, ECB is the most prone to failures under attacks due to its simple architecture that can map properties of the input word into the output word. CBC mode is the most robust of them, but at the cost of reduced parallelism. CTR mode is the intermediate point between security and performance, providing a possible solution for database encryption, as suggested by HighGo for their PostgreSQL solution [221].

SAP HANA has the option to encrypt data at rest using CBC mode on the CPU, with at most two threads allocated to the task. The number of allocated threads is such that the encryption does not interfere with query processing. Through this chapter, we also consider modes that are more amenable to parallelization than CBC mode, such as ECB and CTR.

Related Work. Much attention has been given to AES implementations on FPGAs. AMD-Xilinx [226] proposes their own proprietary AES module focusing only on the parallelizable modes. Our open source design[2] achieves the same order of magnitude in terms of throughput performance as AMD-Xilinx's custom module. Other efficient AES implementations focus mainly on enhancing the encryption/decryption of a single 128-bit word for FPGAs [41, 44, 118, 160], or ASICs [4, 84], and put less consideration to the block cipher modes, the non-parallelizable modes and the interaction of the AES module with other computing modules. Kara et al. [126] show that AES-256 CBC decryption placed prior to a stochastic coordinate descent (SCD) computation increases the processing time by 3.6×, despite the existence of dedicated CPU instructions set for decryption, and propose the offloading of AES-256 CBC decryption transformation rounds to the FPGA, while keeping the key expansion on the CPU. In contrast to this work, we develop a complete solution for AES-256 CBC encryption and decryption, while offloading the entire computation (key expansion and transformation rounds) to the FPGA. Moreover, beside the CBC mode, we also analyze the performances of two additional modes, ECB and CTR, and we distinguish between the block sizes performance.

5.2.3 Hardware Acceleration for Databases

FPGAs have emerged as one type of accelerator for data processing [212]. Having a spatial architecture, FPGAs allow algorithms to be executed with a great degree of parallelism at line-rate, when attached to the network as smartNICs. Cloud computing made FPGAs not only affordable, but also convenient to deploy close to traditional systems as they are migrated to the cloud [112].

[2] `https://github.com/fpgasystems/hw-acceleration-of-compression-and-crypto`

Related Work. Ringlein et al. [187] show the advantages (cost reduction, tail-latency minimization, execution efficiency) of using FPGA-based Function-as-a-Service into disaggregated cloud infrastructures, and Zha and Li [234] build an abstraction framework for virtualizing heterogeneous cloud FPGAs. Key-value stores also benefit from FPGAs with new memcached architectures [30], native computational storage [219], and hardware-managed transactions [111]. Kara et al. [126] use FPGAs to combine column-store ML algorithms with on-the-fly data transformation, such as decryption and delta-encoding decompression, while Lasch et al. [145] accelerate Re-Pair compression algorithm using FPGAs.

Zheng et al. [243] propose that by offloading the physical storage utilization efficiency to the modern SSDs (with built-in transparent compression), a DBMS can obtain higher performance and simpler data structures and algorithm complexity. Zuck et al. [245] study various approaches of implementing transparent compression in the firmware of SSD devices, either in the context of database systems or in that of filesystems. Complementary to them, we show how encryption can be added on top of compression, and offloaded to an intermediate device that can be positioned between compute and storage, leading to a more modular and flexible system.

Lee et al. [146] show how offloading online analytical processing (OLAP) operations to a near-memory accelerator (an FPGA board with attached DIMMs) that sits between CPU and main memory eliminates the performance degradation of online transactional processing (OLTP) workloads when co-existing with OLAP workloads. While Lepers et al. [149] show that the CPU is the bottleneck and not the storage device for tree compaction in an Log-Structured Merge (LSM) key-value store (KVS) on NVMs SSDs, Huang et al. [96] address this issue by accelerating compaction computation on the FPGA.

In the cloud, Microsoft Azure uses FPGAs to offload network management functionality [70] and accelerate key-value store applications [150], and Microsoft's Cipherbase project [20, 21] analyzed the use of an FPGA as a hardware trusted module to process data inside the database engine in an encrypted form, such that parts of the data would remain encrypted while in memory. Intel's QuickAssist Technology [109] offer acceleration solutions for lossless data compression and symmetric and asymmetric data encryption, but they do not give details about design, implementation, or performance.

5.3 Security Engine Design

Through this encryption and decryption implementation, we present a comparison between three different AES block cipher modes, parallelizable and non-parallelizable, and analyze how encryption interacts with a compression module inside an FPGA processing pipeline.

5.3.1 Encryption/Decryption

Our encryption/decryption FPGA design aims to achieve the following properties: (1) reusability and modularity of the code, irrespective of the AES block cipher mode and initial key size, thereby making the code base less error-prone and more tractable, (2) control over the operations happening at each pipeline stage of the design to maximize the overall throughput, and (3) flexibility for the initial key size and the number of 128-bit input data words that can be processed in a transaction to ensure wider applicability.

AES processing involves a sequence of transformations: substitution, shift, bitwise operation, and polynomial multiplication (known as a transformation round), that are consecutively applied to each 128-bit input data word. The initial key size determines the number of transformation rounds needed to encrypt or decrypt an input data word. For a 256-bit initial key size, 14 transformation rounds are needed, and each transformation round requires a dedicated 128-bit *round key* for its internal bitwise XOR operations as illustrated in Figure 5.3, plus one more for the initial round. Together with the pre-/post- data transformation operations, the 14 transformation rounds represent the processing *datapath* of encryption/decryption.

All the required round keys (i.e., in total 15 for the 256-bit initial key) are obtained from the initial 256-bit key through the key expansion process that requires that the initial key is split into 32-bit words and used to derive all the 60 32-bit words needed for the key expansion. Each *add round key* module in Figure 5.3 requires 4 32-bit words (i.e., 128-bits).

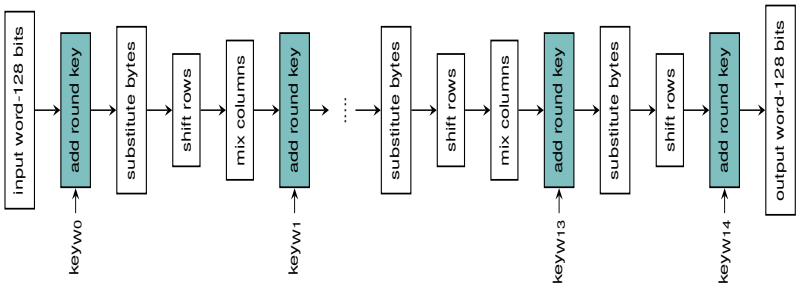


Figure 5.3: Transformation rounds and associated keys for 128-bit input and 256-bit initial key.

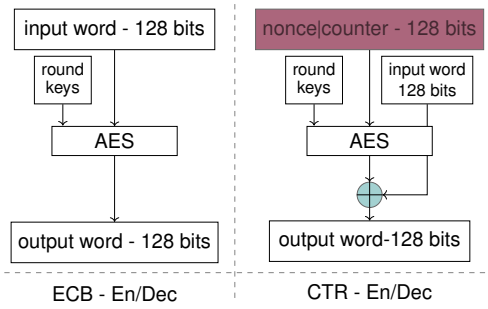


Figure 5.4: AES block cipher modes for Encryption and Decryption - ECB & CTR modes.

On the FPGA, we implement each AES module shown in Figures 5.1 and 5.2 in a pipeline fashion, with each stage of the pipeline applying a transformation round to the 128-bit data word. Therefore, the depth of the pipeline equals the total number of transformation rounds, namely 14 stages. In Figures 5.4 and 5.5, we illustrate the block diagrams for the three AES block cipher modes we implement (ECB, CTR, CBC) and emphasis their common points: (1) transformation rounds (grouped under the AES name) and (2) round keys (generated by the key expansion); and differences (presence or absence of the pre- or post-processing data transformation operations – XOR operations, the initialization vector or the nonce). For CBC mode in Figure 5.5, the 128-bit encrypted word of one AES module is used as an IV by the subsequent AES module, except for the first module, until the block being processed is completed.

We offload the entire encryption and decryption computation to the FPGA, namely both transformation rounds and key expansion, and design their corresponding modules as parameterizable modules in RTL (Register Transfer Level). We use VHDL as modeling language to maximize the throughput [166] for the transformation rounds. All the modules, transformation rounds for encryption and/or decryption and the key expansion for encryption and/or decryption, are part of an OpenCL library and can be instantiated into OpenCL FPGA kernels as value-returning function calls, abstracting away the VHDL implementation from the user. The corresponding function calls are illustrated in Listing 5.1 where *aes_256* is the function call for encryption and *aes_256_decrypt* is the one for decryption. Their associated key

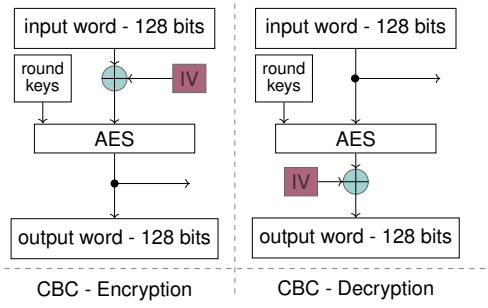


Figure 5.5: AES block cipher modes for Encryption and Decryption - CBC mode.

expansion generation function calls are *aes_key_256* and *aes_key_256_decrypt*, respectively.

For the encryption/decryption functions, the returned values correspond to the encrypted/decrypted cachelines, and the parameter list includes: (1) (long8) x, representing the cacheline to be processed (512 bits); (2) (int8) config, integrating the configurations sent to the VHDL module from the user application running on the host CPU, such as the number of 128-bit words that need to be processed, the *nonce* value required by the CTR mode, and the *initialization vector* (IV) required by the CBC mode to randomize the encryption; (3) (long16) key_lsb and (long16) key_msb, representing the least and most significant components of the expanded key. The number of bits required to represent the expanded key for a 256-bit initial key exceeds the vector bit representation capabilities of OpenCL, therefore two function parameter values are necessary to represent it. These two parameters (key_lsb and key_msb) make the connection between the key expansion module and the transformation rounds module. For the key expansion function calls parameters for encrytion/decryption, (int8) x represents the initial key, and (char) flax acts as a flag choosing if the most significant part or least significant one of the expanded key is returned. Even if the expanded key for decryption is the reverse word order of the expanded key for encryption, we choose to keep two different functions calls for a modular design and do the word reversing inside the VHDL module associated to key expansion.

Listing 5.1: OpenCL function calls associated to the VHDL modules.

```
1  // encryption
2  long8  aes_256(long8 x, int8 config, long16 key_lsb, long16 key_msb);
3  long16 aes_key_256(int8 x, char flax);
4
5  // decryption
6  long8 aes_256_decrypt(long8 x, int8 config, long16 key_lsb, long16 key_msb);
7  long16 aes_key_256_decrypt(int8 x, char flax);
```

The user can set at FPGA compilation time the following programmable parameters: (1) $OPERATION$ (0-encryption, 1-decryption); (2) N_WORDS (number of 128-bit words processed in parallel), four 128-bit words (512-bits) represent our default choice and the granularity for FPGA external memory read and write operation; (3) KEY_WIDTH (128, 192, or 256 bits), for the purpose of this work we use only the 256-bit initial key size; and (4) $MODE$ (0-ECB, 1-CTR, 2-CBC), for choosing the block cipher mode.

Listing 5.2 illustrates how the connection between the OpenCL function call and the VHDL module is done through an XML kernel description file. *The <FUNCTION>...</FUNCTION>* element defines the scope of each function, transformation rounds or key expansion for encryption or decryption. Within the opening tag, the *name* attribute identifies the function call used in the OpenCL kernel (Listing 5.1) and the *module* attribute identifies the VHDL module that describes the function implementation (e.g., *name = aes_256* and *module = aes_user_intel* represent the transformation rounds function call and the VHDL module, respectively, for encryption). Through Listing 5.2 we want to emphasize the programmability of the FPGA design. Note that for different OpenCL function call names we use the

same VHDL module name with different settings for the programmable parameters, e.g., for decryption we use the same *module = aes_user_intel* and a different function call *name = aes_256_decrypt*.

Listing 5.2: Connection between OpenCL function call and the corresponding VHDL module.

```
<RTL_SPEC>
    <FUNCTION name="aes_256" module="aes_user_intel">
        ...
        <PARAMETER name="OPERATION" value="0" />
        <PARAMETER name="MODE" value="2" />
    </FUNCTION>
    <FUNCTION name="aes_key_256" module="key_user_intel">
        ...
        <PARAMETER name="OPERATION" value="0" />
    </FUNCTION>
    <FUNCTION name="aes_256_decrypt" module="aes_user_intel">
        ...
        <PARAMETER name="OPERATION" value="1" />
        <PARAMETER name="MODE" value="2" />
    </FUNCTION>
    <FUNCTION name="aes_key_256_decrypt" module="key_user_intel">
        ...
        <PARAMETER name="OPERATION" value="1" />
    </FUNCTION>
</RTL_SPEC>
```

Beside the programmable parameters that can be set using the *PARAMETER* element, the XML file also exposes the characteristics of each VHDL module: expected latency, stall free or not, stateful or stateless, the communication interface between the VHDL module itself and the other operations inside the OpenCL kernel [108], and the list of all RTL files that are describing the module's behavior. Since the OpenCL kernel is implemented into hardware as a pipeline similar to the instruction pipeline of processors, all these VHDL module details are necessary such that the module's pipeline itself is integrated into the OpenCL kernel pipeline.

In Figure 5.6 we illustrate the two streaming flows created between the FPGA DDR4 memory buffers that are allocated from the host CPU application through the OpenCL API function that creates the global buffers. These buffers are the means through which data is communicated from the host application to/from the OpenCL kernels, and can be used to: (a) *write* the user values from the host application and *read* them by the encryption kernel (Read Buffer), (b) *write* the encrypted values obtained from the encryption kernel and *read* them by the decryption kernel or by the host (Write-Read Buffer), (c) *write* the decrypted values obtained from the decryption kernel and *read* them by the host application (Write Buffer). In addition to the Read and Write Buffers, the host application has also access to the Write-Read Buffer. Through this two flow setup we can check and guarantee the correctness of the values obtained.

The datapath for AES-256 is stateless, input data being streamed through the encryption or decryption pipelines. For each stage of the pipeline, two stateful objects are needed: an 8-bit substitution box for

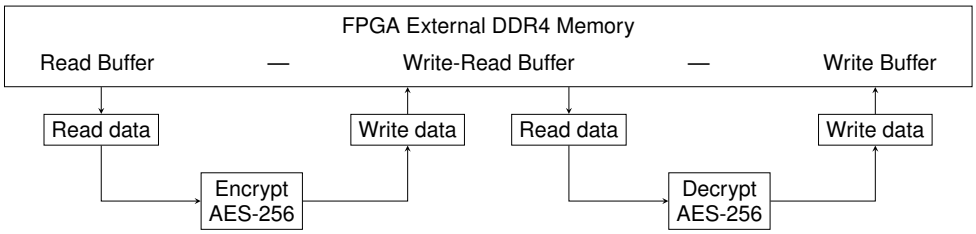


Figure 5.6: Encryption/decryption block diagram within the OpenCL framework.

adding randomness to the transformation round, and a round key from the key expansion. Round keys data is stored in registers, whereas the 8-bit substitution box keeps its state using lookup tables.

The latency of the AES-256 FPGA module is mainly determined by whether a parallelizable or a non-parallelizable block cipher mode is used. ECB and CTR modes have 15 clock cycles of latency for processing one cacheline (4x128-bit input data words), for both encryption and decryption. For the same cacheline, the CBC mode needs 15 clock cycles for decryption, which is parallelizable, and 57 clock cycles for encryption, which is non-parallelizable. The latency for one AES module can be manually computed, one clock cycle for each transformation round plus one extra clock cycle for delivering the results. The key expansion computation takes 11 clock cycles on hardware, but since it is independent of the datapath, it can be ignored when computing the latency of the results. Seen from the host, the key expansion latency is 62.90 µs and includes beside the 11 clock cycles of hardware latency, also the invocation of the kernel from the CPU when an I/O request is made.

5.3.2 Compression and Encryption

We take advantage of the OpenCL environment versatility to combine modules written in different programming languages. The DEFLATE compression module presented in detail in the paper associated to this chapter [43] is implemented as an OpenCL kernel, whereas encryption is implemented in VHDL, but integrated in OpenCL as a library and exposed to the system kernels as a function call. The result is a combined operator that compresses data and then encrypts it when invoked by the database. The block diagram of the two combined modules is illustrated in Figure 5.7, where LZ77 compression, Huffman encoding and Load Huffman tree components are part of the compression module.

From OpenCL integration perspective, i.e., FPGA resource consumption and compilation time, it is recommended to keep compression and encryption as separate kernels that are instantiated on the FPGA than to have a single kernel that unites them. The modularity achieved on the FPGA results in a more complex control flow for the host CPU application which is responsible for managing those kernel instances.

When the compression module is plugged upstream the encryption one, the compressed transactions are forwarded via FIFO buffers to the AES-256 module(s). The encryption module uses the *last* flag that goes

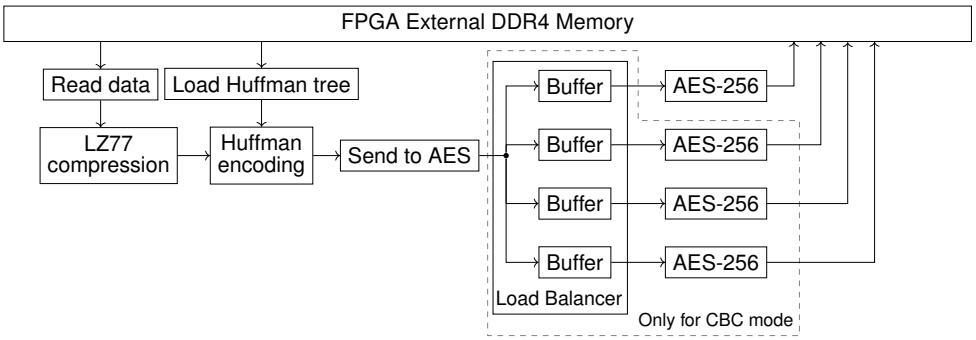


Figure 5.7: Compression and encryption pipeline block diagram for the three block cipher modes.

along with the compressed transactions to determine the granularity of the operation, i.e., a compressed payload is encrypted as a single unit until the last flag is met. When the last flag is met, the encryption module resets its nonce and counter for CTR mode, and restarts from the initialization vector for CBC mode. The last flag has no reset effects on ECB mode, since its granularity is always 128-bit word size.

For the two parallelizable encryption modes, ECB and CTR, a single AES-256 module suffice to consume data coming from the compression module without putting back-pressure on the upstream module, as it is later shown in Section 5.5.4. The sequential nature of the encryption in CBC mode creates back-pressure on the compression module, that is, the encryption — the second stage in the pipeline — is slower than the compression — the first stage in the pipeline. We minimize this back-pressure by using multiple AES-256 modules and a load balancer that round-robins the input between them. The goal is to eliminate the back-pressure by matching the compression throughput with several AES-256 modules. However, as a spatial architecture, we can only put several of them into a given FPGA. For the FPGA we use, we are able to deploy up to four parallel AES-256 modules in CBC mode.

5.4 FPGA Implementation

We implement our designs on the Intel FPGA PAC D5005 Acceleration Card. Table 5.2 summarizes the FPGA resource utilization, including the resources allocated for the static part of the FPGA, called Board Support Package (BSP), and the dynamic part of the FPGA, called User Kernel System (where our design(s) reside). For the 'FPGA Design' column, the CBC/CTR/ECB AES-256 design include both the encryption and the decryption kernels, together with their associated key expansion kernels. The interaction between these two parts of the FPGA (static and dynamic) is illustrated in Figure 5.8.

The BSP represents the FPGA logic dedicated to the FPGA interaction with external elements, host processor and DDR4 memory, and consists of the host interface, the external memory controller, and the global memory interconnect. The User Kernel System part contains the logic dedicated to the compute

Table 5.2: FPGA resource consumption.

FPGA Design	ALUT	Logic Utilization	RAM	Op. Freq.
CBC AES-256	259,353	256,438 (27%)	667 (6%)	248 MHz
CTR AES-256	236,211	252,822 (27%)	899 (8%)	248 MHz
ECB AES-256	252,587	258,754 (28%)	899 (8%)	232 MHz
DEFLATE + 1 CBC	458,061	428,244 (46%)	2,298 (20%)	243 MHz
DEFLATE + 4 CBC	712,081	678,698 (73%)	2,460 (21%)	243 MHz
DEFLATE + 1 CTR	435,919	409,742 (44%)	2,298 (20%)	244 MHz
DEFLATE + 1 ECB	451,274	415,508 (45%)	2,298 (20%)	236 MHz

kernels, e.g., AES-256, LZ77 compression or Huffman encoding, and to the kernel communication with the on-chip FPGA memory.

In Table 5.2 we report the working frequency for each standalone and combined implementations. Unlike the CPU that operates in a GHz frequency range, the FPGA operates in the MHz frequency range. Our designs have a working clock frequency around 240 MHz. Despite this large frequency range gap, the FPGA is able to achieve better performance than the CPU when the FPGA is implementing algorithms that can benefit from its spatial, parallel architecture and low-latency on-chip memory accesses, providing a MIMD (Multiple Instructions, Multiple Data) analogy. The CPU, however, must operate at a much higher frequency to overcome several overheads imposed by its architecture [26], and the best parallelism it can achieve is restricted to SIMD (Single Instruction, Multiple Data).

The OpenCL compiler determines the highest possible operating frequency based on (i) the design characteristics, and (ii) the placement as physical components (the non-deterministic operation called *place and route*). For example, even if different block cipher modes of the AES-256 modules have similar designs, their final resource allocation, and therefore placing and routing, is unlikely to be equivalent, leading to different working frequencies. The highest working frequency of the encryption module is 248 MHz and is achieved when the module is configured to work in the CBC and CTR modes.

The logic utilization in Table 5.2 represents the number of adaptive logic modules the design needs, with

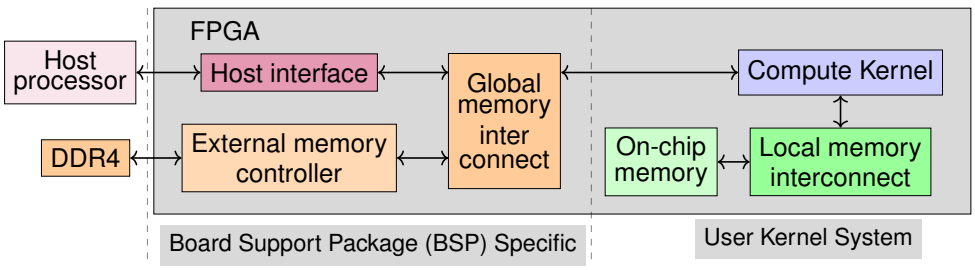


Figure 5.8: OpenCL FPGA framework overview.

each adaptive logic module being used either as a 2-combinational Adaptive Lookup Table (ALUT), as a 2-bits full adder or as four registers [104]. In all cases, the resources used leave enough room for additional logic (e.g., a network stack [191] to send the data to cloud storage through the network), even when using 4 AES-256 CBC encryption modules.

5.5 Experimental Evaluation

We evaluate the performance of the encryption module in isolation (Figure 5.6) and when combined with compression (Figure 5.7). As baseline, we run their equivalent software implementations using typical configurations currently used in SAP HANA, i.e., up to two threads being allocated for the tasks. For the modules on the write path of the I/O request (i.e., encryption and compression combined with encryption), we focus on blocks ranging from 4 KiB to 16 MiB. For the modules placed on the read path of the I/O request (i.e., decryption), we focus on blocks ranging from 4 KiB to 64 MiB. The block sizes for writing and reading emerge from an analysis of a I/O trace obtained from SAP HANA and presented in Section 5.5.1.

Our observations represent an evolution towards larger logical pages compared to a study of Chen et al. [40] from 2010, who observe that "a commercial DBMS" accesses almost exclusively logical pages of size 8 KiB for both reads and writes, or MySQL's default page size of 16 KiB [171]. The same direction towards larger logical pages is also reported by Umbra [172], with the smallest page size being 64 KiB, by Snowflake [205], with 100 MB to 250 MB page sizes, or by Vertica, with Hadoop FS Block Size being set to 64 MB. Antonopoulos et al. [19] report initial log sizes between 3 GB-38 GB as they attempt to reach constant time recovery in Azure SQL Databases by truncating the initial log sizes to 200 MB.

5.5.1 Real-world I/O Trace

SAP HANA supports a wide variety of workloads. We focus on a concrete use case that generates significant I/O traffic and which is especially common in the cloud: loading data into the system. During this and other write-intensive workloads, SAP HANA initially applies updates to the write-optimized store of each table residing in memory. For durability, changes are also persisted in the transaction log on storage. Depending on the configuration of the system, the write-optimized store is merged into the read-optimized store in order to maintain high read performance. When this process completes, the new read-optimized store (i.e., the snapshot of the table at the point of the merge) is persisted on disk and outdated snapshots are deleted [17]. The snapshots and the log entries caused by data import, which is typically committed in batches of a few thousand rows, result in a regular stream of relatively large writes to storage, and are thus potentially well-suited for block-based compression.

The payload of our compression and encryption modules concerns data from (1) the transaction log, and (2) from current snapshots of the read-optimized partition of tables. Since such snapshots are only read at column granularity, block-based schemes are suitable for this use case.

To understand the granularity of the I/O transfers in a real system, we analyzed an I/O trace using SAP HANA 2.0 SPS 04 Database Revision 045. This version has a similar I/O behavior as the newer SAP HANA releases, e.g., SAP HANA 2.0 SPS 06 Database Revision 061. SAP HANA uses Linux asynchronous I/O subsystem for logs and data files, so we intercept calls to that subsystem (`io_submit` and `io_getevents`) by attaching a small tracing library using `LD_PRELOAD` and trace the content and metadata of all I/O blocks read and written by the database. Since queries to loaded tables are answered from their in-memory copy, they do not incur any I/O themselves.

We use a representative workload that captures common operations in cloud deployments to generate the trace. We import the CSV files from the TPC-H data generator for Scale Factor 10, which take about 5.5 GB on disk. For each table in the schema, we run a sequence of five operations: (1) import the file from CSV; (2) unload the table from main memory; (3) load the table again; (4) run `COUNT(*)`; and (5) finally unload again the table. Such a workload emphasizes data loading, a critical operation in replication tasks, e.g., regularly replicating data from an Enterprise Resource Planning (ERP) system into an analytical warehouse. SAP BWH benchmark[3], as well as the TPC-H and TPC-DI benchmarks, report this type of workload as being a crucial element in performance. In the cloud, it captures the constant ETL and loading of data into an analytical system from transactional engines and other sources. Such a workload stresses the I/O subsystem with writes to the write-ahead log, and reads from and writes to data files when tables are loaded into main memory. The same pattern is observed when the read-optimized storage is snapshot to storage after being merged with the newly loaded records from the write-optimized store.

As part of the workload, which includes the startup of the system, SAP HANA reads and writes a total of 7.0 GB and 21.7 GB from and to the storage. Most of these reads are caused by loading the tables in step (3), while most of the writes are caused by the repeating merges during the import in step (1). At the same time, the system reads and writes a total of 64.0 MiB and 10.1 GB from and to the log, respectively. The reading happens in a single access directly after start-up. Presumably, the system batch-loads the tail of the log and simply determines that the previous shutdown was clean. Most of the data written to the log is caused by the import in step (1). In addition to the described system activities, we also see I/O traffic caused by background activities of the system that we could not identify further but that correspond to less than 10 % of the overall traffic.

[3] `https://www.sap.com/dmc/exp/2018-benchmark-directory/#/bwh`

Figure 5.9 shows the distribution of the block sizes and Table 5.3 summarizes the percentage occurrence of relevant block sizes for each page access type. The plot shows that the system accesses most data in blocks larger than the traditional 4 KiB page size. On the data file, most accesses are 64 KiB and some up to 64 MiB. This is not surprising since the snapshots stored in the data file consist of large column vectors and dictionaries that are always accessed in their entirety (namely when flushed after a merge or when loaded back to main memory). The reads from the data file always access blocks whose size is an even power of two and the writes mostly do the same. However, interestingly, the system occasionally also writes blocks with a non-power-of-two size (but still multiples of 4 KiB). The workload only contains a single read from the log (64 MiB at system start-up) while writes occur with a large number of different block sizes. A few of the write block sizes are much more frequent than others. Most notably, about 60 % of all writes are of 118 784 B, precisely 116 KiB (i.e., 29 physical disk pages).

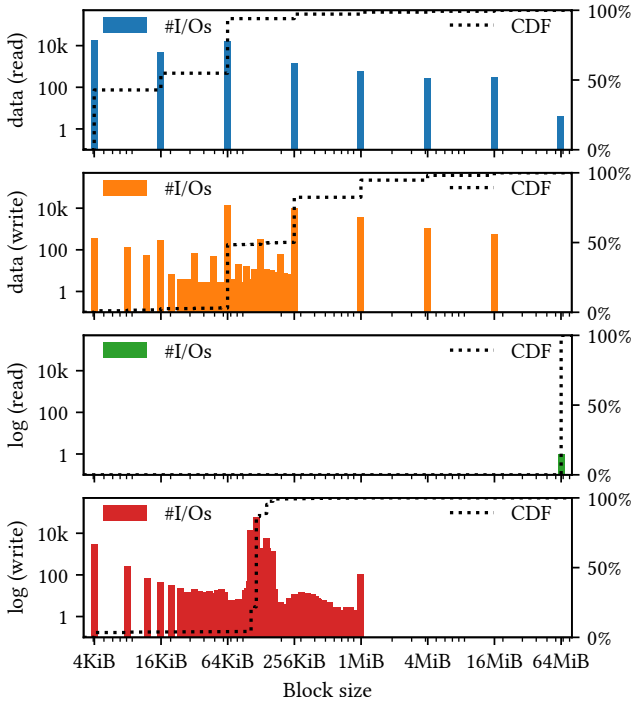


Figure 5.9: Cumulative distribution function (CDF) and histogram of I/O sizes in I/O trace.

Table 5.3: Page type accesses [%].

Block size [B]	4 Ki	16 Ki	64 Ki	116 Ki	256 Ki	1 Mi	4 Mi	16 Mi	64 Mi
data read	42.87	12.01	39.13	-	3.24	1.41	0.64	0.70	0.01
data write	1.08	0.92	45.38	-	32.33	12.28	3.44	1.82	-
log read	-	-	-		-	-	-	-	100
log write	3.46	0.05	0.01	64.64	0.01	0.11	-	-	-

5.5.2 Setup

Software. For the software baseline we use an Intel® Xeon® Gold 6234 Processor 3.3 GHz machine with 8 cores and 16 threads featuring: 512 kB (L1 cache), 8 MB (L2 cache), and 24.75 MB (L3 cache). The level of parallelism set for our encryption/decryption and compression combined with encryption baselines is consistent with the number of threads SAP HANA allocates for these background tasks, namely 1-2 threads. SAP HANA does not do intra-page parallelism in software, and only inter-page parallelism, meaning that each thread processes one block in its entirety. We use gcc 4.8 and set the optimization flags for `-O3 -march=native -pthread` for our baselines.

Hardware. Our target platform consists of the Intel Programmable Accelerator Card (PAC) for data centers, Intel FPGA PAC D5005 [106], connected to the CPU via a PCIe Gen3x16 link. The card features a Stratix 10 SX FPGA, two QSFP+ connectors with up to 100 Gbps support, and 32 GB of on-board DDR4-2400 memory, with a peak transfer rate of 19.2 GB/s. We use OpenCL for Intel FPGA SDK (OpenCL RTE version 19.2.0.57) [107] to implement and instantiate the FPGA compute kernels that are interfacing with the on-board DDR4 memory at 512 bits cacheline granularity for both read and write operations. The CPU (host processor) allocates memory for the FPGA computing kernels, and a memory management library handles the address translation between CPU main memory and FPGA external memory.

Figure 5.8 illustrates the OpenCL FPGA framework diagram and differentiates between three memory types in the OpenCL memory model: (1) the device global memory, FPGA external DDR4 memory; (2) the local memory, FPGA on-chip memory like Block RAM (BRAM); and (3) the private memory, FPGA on-chip registers.

Global memory represents a major component of the OpenCL compute model, being used to transfer data between the host processor and the FPGA via input/output buffers mapped inside the global memory. The host writes the data to be processed into the input buffer from where the computing kernel reads it, processes it, and writes the obtained results into the output buffer. The results are then transferred back by a memory access generated by the host. This OpenCL computation model is supported by the Board Support Package (BSP) for our Intel FPGA PAC D5005 board, and we use it to evaluate our standalone encryption and pipelined compression and encryption kernels.

For our performance evaluation, we assume data to be already in the FPGA's global memory, and the computing kernel reads it from there at cacheline granularity (512 bits–64 B). At this granularity, the global memory latency is higher than the compute time, which affects the performance measured for small block sizes (e.g., 4 KiB). To partly mask this overhead, we use prefetching inside the compute kernel. Note that, in the cloud, the CPU needs to move the data after compression and encryption to a NIC, thereby also paying a data transfer overhead. Our design is intended to be deployed on an FPGA with direct network access, so no further data transfer overhead occurs. Thus, we focus on the performance inside the FPGA compared to that of the CPU, since transfer overheads will be either similar in both

cases or much less on the FPGA by skipping a transfer to the NIC. Also note that the FPGA board we use has conventional DDR4 memory. With the emerge of CLX cache protocols, the overhead for small data transfers should be eliminated.

Before proceeding with the actual evaluation, we would like to introduce the *OpenCL Application Programming Interface (API)* which comprises: (1) the Platform Layer API, which manages the discovery of the hardware system, device (i.e., FPGA) capabilities and device memory, and sets up the execution environment through a construct named *context*, and (2) the Runtime Layer API, which manages the write/read actions to the memory and the execution of the compute kernels on the device through constructs named *command queues*. The context coordinates the interaction between the host CPU and the device, whereas the command queue represents a mechanism for the host to request action from the device (the commands execute in-order if not otherwise specified). For the encryption/decryption implementation, the context includes the three memory buffers illustrated in Figure 5.6 and the D5005 FPGA board, with only one command queue for executing encryption and decryption, since they are sequential operations. For the compression and encryption operator setup, the memory buffers are allocated as illustrated in Figure 5.7, and two command queues are instantiate (i.e., one command queue for each operation), since the encryption kernel starts processing the data as the compression kernel generates it.

5.5.3 Encryption/Decryption

As a baseline, we build a library on top of Intel AES intrinsic instruction set for the three AES modes (ECB, CTR, CBC) [102]. Each block cipher mode receives for encryption/decryption the same block sizes as the ones traced in SAP HANA (Section 5.5.1).

Our results yield performance for a peak clock frequency of 3.3 GHz consistent with the results reported by Intel [102] for single threaded implementations: 1.76 cycles/byte for ECB encryption/decryption; 1.88 cycles/byte for CTR encryption/decryption; 1.78 cycles/byte for CBC decryption, and 5.65 cycles/byte for CBC encryption. Both AES-256 software operations, encryption (Figure 5.10) and decryption (Figure 5.11), are compute-bound, with the individual performance of each mode scaling proportionally with the doubling of allocated threads.

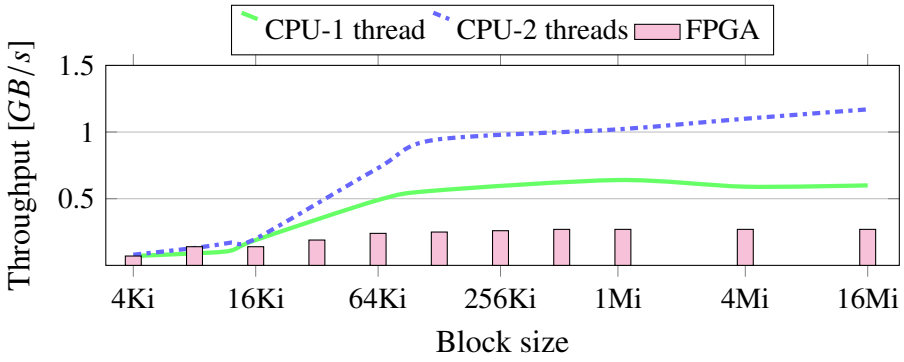
The difference in the software performance of the three modes comes from their compute complexities. ECB mode is the simplest of them with only 14 transformation rounds on the datapath, resulting in the best software performance of a maximum of 4 GB/s for both encryption and decryption. CTR mode adds-in complexity by requiring a 128-bit XOR operation on the datapath after the transformation rounds are performed. Its software performance gets penalized, reaching a maximum throughput of 2.5 GB/s. Figure 5.11a shows that the software performance of decryption in CBC mode is similar to the one in CTR mode due to the required 128-bit XOR operation. Encryption in CBC mode requires each 128-bit input data word to be XOR-ed with the previously obtained 128-bit output data word (the first one is XOR-ed

with an initialization vector), before entering the transformation rounds. The cost of the data dependency adds to the cost of the XOR operation, leading to a maximum throughput of 1.2 GB/s. The performance of both the CPU and FPGA is impacted by data dependency, resulting in a decrease in performance. However, the CPU outperforms the FPGA in terms of overall performance for encryption in CBC mode due to its higher operating frequency in GHz.

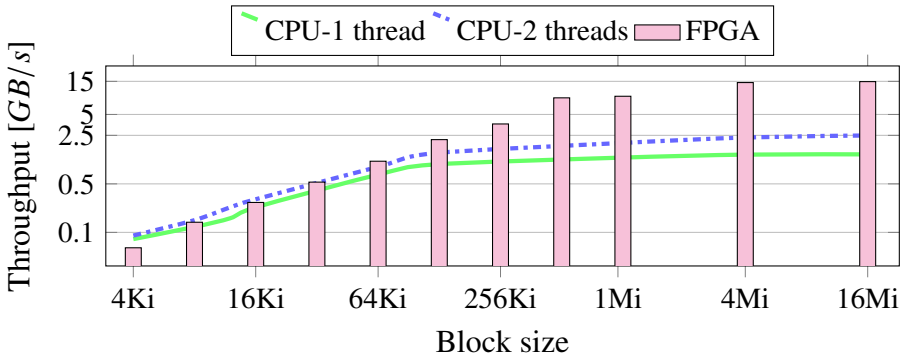
In the master thesis that automates the testing of the encryption/decryption module the sw baselines are run for up to 24 threads. For 24 threads, the registered throughput performances for encryption (en) and decrytion (de) operations are: (1) for ECB mode–around 25 GB/s (en/de) for both operations, (2) for CTR mode–15 GB/s (en) and 12 GB/s (de), and (3) for CBC mode–10 GB/s (en) and 8 GB/s (de).

In Figure 5.10a, we show the limitations of the MHz operational clock range of the FPGA. Even if for the FPGA the XOR operation comes at no performance cost, the data dependency translated into the sequential nature of the CBC mode implementation limits the FPGA CBC encryption throughput performance to 0.27 GB/s. At block size granularity, CBC encryption cannot take advantage of the parallelization potential of the FPGA, whereas the CTR and ECB modes benefit from it, reaching a maximum throughput performance of 15 GB/s. By exploiting the spatial parallelism available on the FPGA, CTR and ECB encryption modes exceed by up to seven times their corresponding CPU performance. Irrespective of the mode used for encryption, the transfer latency from the global memory is visible for small block sizes (up to 16 KiB), a common situation for PCIe-attached accelerators [119]. The Figures in 5.10 (a-c) show that this overhead has a larger impact on performance than the algorithm's complexity itself, making all three FPGA encryption modes perform similarly when compared to their corresponding CPU implementations.

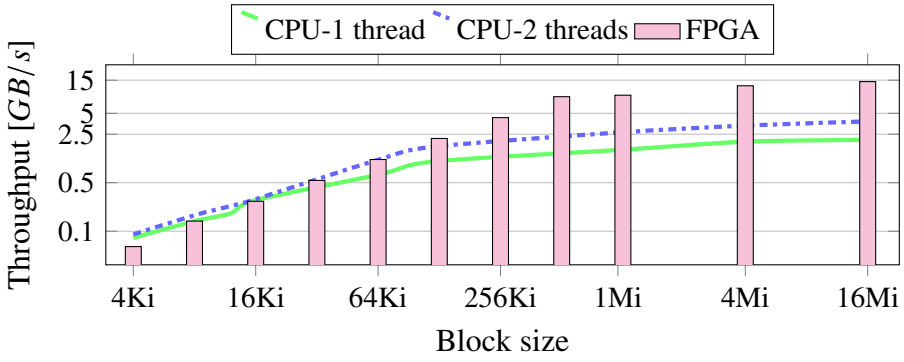
The Figures in 5.11 (a-c) illustrate the parallelization potential of the FPGA since decryption is parallelizable for all three modes. While the CPU performance saturates at around 2.5 GB/s for the CBC and CTR modes and at around 4.5 GB/s for the ECB mode, the FPGA implementation can reach up to 15 GB/s, irrespective of the mode employed for decryption. The transfer latency from the global memory for small block sizes (up to 16 KiB) impacts decryption throughput performance as it did for encryption. For small block sizes, the CPU and the FPGA have comparable performance.



(a) AES-CBC

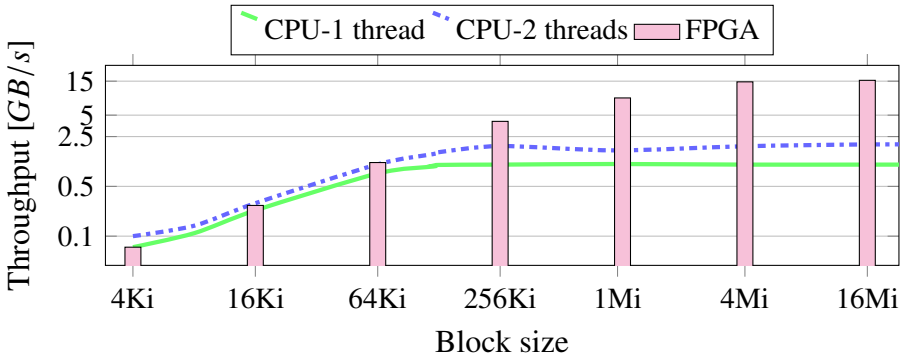


(b) AES-CTR

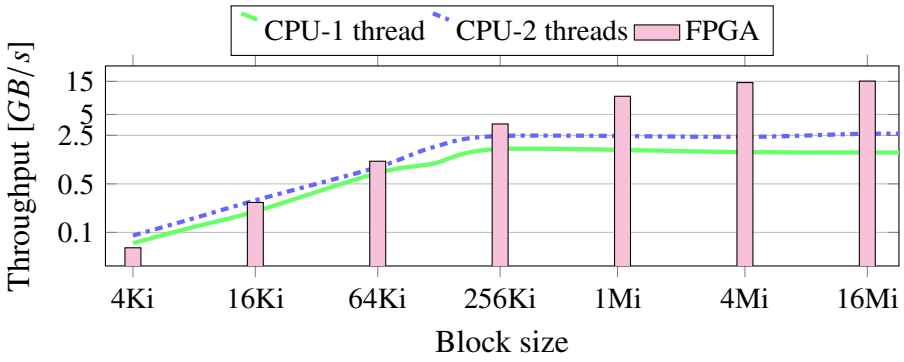


(c) AES-ECB

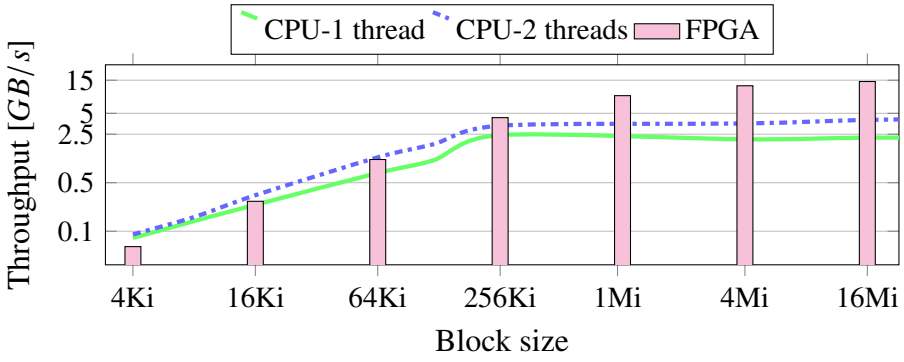
Figure 5.10: AES Encryption - with 1 and 2 threads on CPU vs. FPGA design. Note the logarithmic scale of the y axis for (b) & (c).



(a) AES-CBC



(b) AES-CTR



(c) AES-ECB

Figure 5.11: AES Decryption - with 1 and 2 threads on CPU vs. FPGA design. Note the logarithmic scale of the y axis.

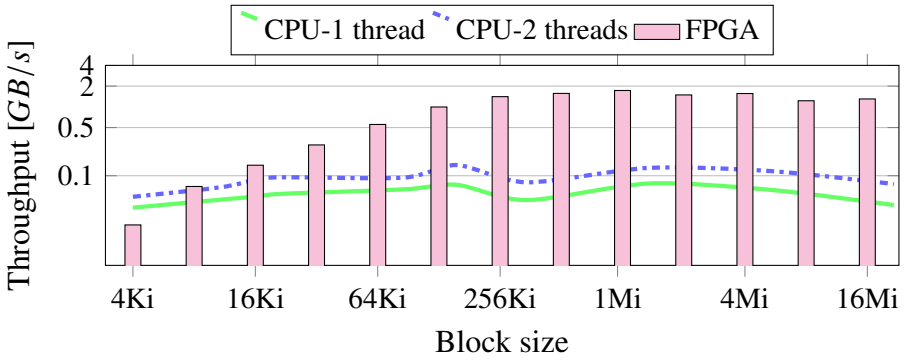
5.5.4 Compression and Encryption

For the compression and encryption pipeline deployment on the FPGA as illustrated in Figure 5.7, we have to take into consideration that each module, compression and encryption, have different maximum throughout performances. As presented by Chiosa et al. [42], the saturation throughput delivered by the compression module is 4 GB/s. Compared to this performance, the CBC mode (0.27 GB/s) would impose back-pressure and limit the overall performance, whereas both CTR and ECB modes (15 GB/s) would turn compression into the bottleneck. Therefore, the choice of which AES-256 block cipher mode to use plays a big role in determining the overall throughput performance of the pipeline.

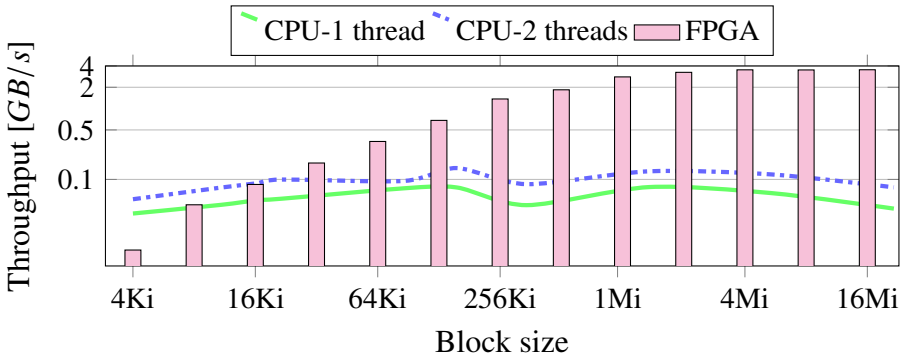
In Figure 5.12 we analyze the throughput performance of the full compression and encryption pipeline. The pipeline with encryption in CBC mode reaches a maximum throughput of 1.72 GB/s, while for encryption in CTR or ECB mode, we achieve a comparable throughput as imposed by the compression module alone, namely 4 GB/s.

For the software baseline, we extend compression with encryption functionality for the three encryption block cipher modes. Each compressed block is encrypted using Intel's AES intrinsic instruction set and placed at a contiguous memory location from where it can be read. The performance of the software pipeline is limited by compression, and even if its performance is better or marginally comparable to the FPGA (for block sizes of 4 KiB and up to 16 KiB, respectively), the full pipeline is accelerated on the FPGA for block sizes bigger than 16 KiB in all three cases, as shown in Figure 5.12.

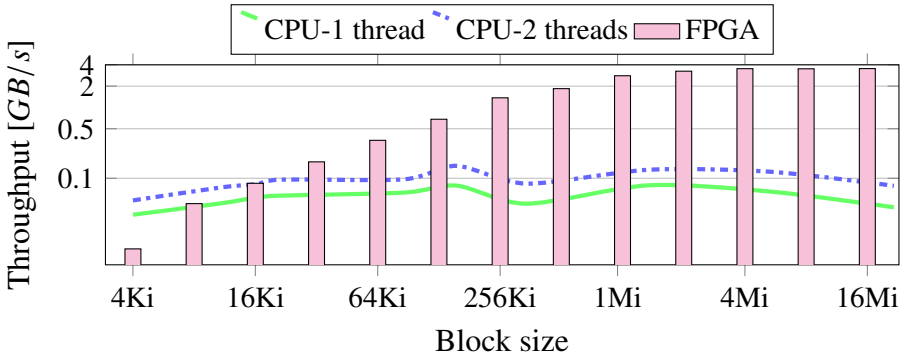
To accurately evaluate the performance of the integrated design, it is important to understand the implications of compressing the data prior to encrypting it. The intrinsic purpose of the compression module is to reduce the size of the data it processes (i.e., consumes). Therefore, the rate at which the compression module produces the data is equal or smaller than the rate at which it consumes the data. Furthermore, the higher the compression ratio, which is determined by the ratio of uncompressed file size to compressed file size, the lower the rate at which the compression module produces data. This means that as the compression ratio improves, the module is more efficient, generating less data. While this observation does not play a crucial role when evaluating the compression module in isolation, it becomes a key factor when the compression module is combined with the encryption one. Notably, the maximum throughput observed for AES-256 in CBC mode (Figure 5.10a) does not match the maximum throughput when the module is placed *after* compression (Figure 5.12a) in the integraded design. Since AES-256 CBC encryption works on compressed data, the overall throughput of the combined design is higher than that of the AES-256 module in CBC mode alone.



(a) Compression + AES-CBC



(b) Compression + AES-CTR



(c) Compression + AES-ECB

Figure 5.12: Full pipeline - with 1 and 2 threads on CPU vs. FPGA design. Note the logarithmic scale of the y axis.

5.6 Discussions

Based on the results in Section 5.5, we can now put into perspective the potential of using an accelerator on the data path of SAP HANA.

5.6.1 An Accelerator on the Data Path of SAP HANA

The results indicate that, for larger data transfers, offloading encryption and compression coupled with encryption to an FPGA instead of performing them on the CPU has significant advantages. On the one hand, the process is faster on the FPGA, which makes the option of both compressing and encrypting data on the I/O path for cloud deployments feasible. This has many advantages: (1) data is secured while at rest and in transit, (2) the encryption keys remain in control of the database engine and not of the cloud provider (i.e., it is the database the one compressing and encrypting, not the storage layer), (3) the space needed on storage is reduced and also makes the network, a scarce resource in the cloud, more efficient. On the other hand, doing so frees up valuable CPU cycles that can be either used for other purposes or not used at all, with the engine requiring less virtual CPUs to run on the cloud for a similar performance.

The results also show that while there are gains when compressing and encrypting, the potential gains when encrypting alone would be much bigger. Encryption for large block sizes operates at several GB/s, a rate that matches or surpasses the available network bandwidth in conventional cloud deployments and, thus, making the encryption free in terms of I/O latency. From a business perspective, several commercial strategies could be envisaged. First, for users prioritizing storage price over performance, the engine could compress and encrypt to reduce storage cost. Second, for high-end users who prioritize performance, encryption can be provided without compression, thus maximizing throughput and freeing up CPU resources for query processing. Third, the engine can accommodate different security levels. Finally, with compression placed before encryption, the security gap between different AES encryption block cipher modes is minimized since compression aims to eliminate redundancies within a given data block.

Introducing an accelerator like the one proposed on the I/O path also opens up several interesting possibilities for cloud database engines. Databases are full of design decisions driven by the assumption of a slow I/O path and the overhead associated with compressing and encrypting the data. Such design decisions would change with an accelerator. In fact, cloud native databases already use block sizes that are typically larger than those of conventional databases [55]. The accelerator would benefit from these larger transfer sizes and motivate such change even further.

5.6.2 I/O Transfer Sizes

The experimental results indicate that offloading compression and encryption to an accelerator becomes attractive for block sizes larger than 64 KiB. Compression and encryption are relevant when writing data or the log to storage. As the traces show, this size range corresponds to more than 95 % of the I/O requests observed in the traces (observe the jump in the CDF in Figure 5.9 at 64 KiB for data and at slightly larger sizes for the log). The same observation can be made for decryption: there are many requests of sizes large enough for the accelerator to be a better choice overall. From this analysis we conclude that the disadvantage of processing small block sizes on the FPGA is mitigated by the small frequency of such requests. Even if for some small sizes the accelerator is slightly slower, for the entire workload it brings a clear advantage. This compromise across a workload is very common in database engines and the data-write path of SAP HANA can afford such a compromise, as is probably the case for most analytical databases.

5.6.3 Summary

In this chapter, we have explored the implementation of different AES encryption modes and how they couple with a compression module on the I/O path of a relational, analytical engine (SAP HANA). These two operations (encryption and compression) play a key role in cloud deployments: in the case of compression, to reduce the cost of storage as well as to maximize the available network bandwidth; in the case of encryption, to protect the data both while at rest and while it is being transmitted over the network. Our results show that offloading these operations to an FPGA accelerator offer significant advantages both in terms of performance as well as freeing up valuable CPU resources in cloud settings as long as there is enough data being transferred. As we demonstrate with SAP HANA as an example, in analytical engines the data transfer sizes are large enough for the accelerator to offer a clear advantage. Conversely, the approach we have explored does not seem to be suitable for transactional engines, as the amount of data involved in every transfer tends to be too small and the synchronous I/O excludes batching.

CHAPTER 6

CONCLUSIONS

This chapter concludes the thesis by summarizing its main findings and their implications and outlines the directions for future work that are opened by it.

6.1 Summary and Implications

This thesis takes a step toward understanding and shaping the position of the FPGA in the heterogeneous landscape of the current data center ecosystem. On the one hand, it shows how compute-intensive data characterization algorithms can be offloaded to the FPGA and what is their behavior when attached to an FPGA-based smartNIC. On the other hand, it demonstrates that the I/O operations associated with a database engine can be offloaded to the FPGAs.

Chapter 3 shows that the FPGA can be successfully deployed to accelerate data correlation on large streams of data, thus freeing valuable host-side resources (CPU, RAM) for other tasks. The evaluation shows excellent outcomes for both FPGA deployments, as a co-processor or as a bump-in-the-wire, with the latter deployment alongside an RDMA network interface achieving the maximum advantage while concealing the PCIe limitations for small-size data transfers. Furthermore, `AMNES` can also be extended to computing the Spearman correlation coefficient, whereas `AMNES`' backend can serve in estimating a simple linear regression or computing the cosine distance between vectors in a multi-dimensional space.

The advantage of deploying a compute kernel as a bump-in-the-wire is also shown in Chapter 4. Using a TCP/IP network interface in this instance, three different sketch algorithms (HLL, Fast-AGMS, Count-Min) are offloaded to the FPGA under a unified design (`SKT`). `SKT` does a single pass over the data, extracting information such as cardinality (HLL), second frequency moment (Fast-AGMS) and frequency distribution (Count-Min). Besides a better throughput performance over the CPU-based baseline, the

results also show that an over-provisioning of the SKT's number of pipelines is needed to saturate the network bandwidth, demonstrating that the compute kernel cannot be independent of the capacity at which the network expects data to be consumed, even if the same data rate is maintained. In addition to this architectural design aspect, SKT finds that a stand-alone Count-Min deployment is suitable for low cardinalities, whereas for large cardinalities Count-Min's estimates have a large error.

In each of the two chapters (Chapter 3 & Chapter 4), the compute kernels do compute-intensive operations without incurring back-pressure to the network, thus imposing very low overhead over regular, unaugmented node-to-node communication.

Chapter 5 demonstrates that encryption and decryption can be successfully implemented on the FPGA even when a non-parallelizable block cipher mode is deployed. The experimental evaluation shows that analytical engines benefit more from the FPGA offloading of the data transformation operations on the I/O path than transactional engines. This is attributable to the larger size workloads of the analytical engines. The proposed system provides three tiers of security, capable of meeting various data security requirements in the cloud. In addition to this, the system shows how a non-parallelizable algorithm can achieve a better performance than its stand-alone deployment when placed after a data reduction algorithm such as compression. The combined pipeline exhibits higher throughput performance compared to its CPU-based baseline.

Overall, this thesis adds a building block to the concept of augmenting data movement with compute capabilities with low overhead.

6.2 Future Directions

Direct course. A direct forward step that emerges from Chapter 3 is to generalize the engine and to compare its streaming-based implementation with a matrix-based implementation that utilizes the AI Engine tiles on the ACAP (Adaptive Compute Acceleration Platform) Versal [15] board from AMD-Xilinx. The generalization should target maintaining the network advantage of the system, and obtain a high working clock frequency for the compute kernel while dealing with a large FPGA occupancy.

Pre-processing on the FPGA. Traditionally, many frameworks require data to undergo pre-processing stages before absorbing it into the framework's main task. For example, for machine learning (ML) frameworks, the working flow includes two tasks: processing the data and training the model. Even if processing the data consumes significant computing resources for on-the-fly transformations such as decompression, sampling, and filtering, the training task benefits from most optimizations [79].

Another example comes from the query optimizer of a database engine that needs to gather information about the data (i.e., cardinality or frequency distribution) before deriving an efficient query plan. As has

been shown in Chapter 4, the FPGA is efficient at implementing three sketches in parallel, with cardinality estimation being one of them and of major importance for the query optimizer in a database engine.

With the current trend of disaggregating data storage and processing and considering FPGA capabilities for efficiently implementing on-the-fly data transformation operations and compute-intensive data characterization algorithms, an interesting direction for future research is designing, implementing and evaluating end-to-end data center systems where the FPGA is responsible for the pre-processing stage.

Applying I/O offloading to data analytics engines. In Chapter 5, it is shown how SAP HANA can benefit from having its I/O operations (encryption, decryption and compression plus encryption) offloaded to an accelerator.

The question that directly emerges from this work is to understand if the FPGA would be suitable to implement other types of encryption offloading, such as Fully Homomorphic Encryption (FHE), and integrate them as I/O functionality.

In addition to this, the open question that arises is if stream data analytics engines (e.g., Apache Spark, Apache Kafka) could benefit from having their I/O operations offloaded to the FPGA. Ultimately, the objective would be to generalize these findings and designate the FPGA as the I/O processing unit within the system of a data center.

Unified compute fabric and node. The objective of domain-specific accelerators is to reduce the cost of processing when compared to general architectures. Nevertheless, most algorithms are built and have a cost model that reflects the CPU's architecture [58]. Through its new series, Versal ACAP [15], AMD-Xilinx opens the exploration space of a heterogeneous architecture that comprises a CPU, an FPGA and a vector processor. Besides offering a unified programming tool, this architecture enables the possibility of creating mathematical and cost models that map algorithms to heterogeneity.

However, the compute capabilities limits of the chip are imposed by the physical limits of thermal dissipation efficiency (TDE) and they may not satisfy the compute requirements of data-intensive systems, such as databases, machine learning or distributed frameworks. The questions that arise from this remark are how a unified heterogeneous system node, which architecture comprises of separate boards, each equipped with either a CPU, GPU, or FPGA, would serve this purpose and what are the mathematical models that would derive from it. The cluster instances offered by Intel (CPU and FPGA boards) [105] and by AMD (CPU,FPGA and GPU boards) [13], together with the FPGA positioned as a pre-processing and I/O offloading module could serve as a starting point for exploring this research direction.

List of Figures

List of Tables

List of Listings

Bibliography

[1] 2021. Correlation Does Not Imply Causation: 5 Real-World Examples.

[2] Mohammed Abouzour, Günes Aluç, Ivan T. Bowman, Xi Deng, Nandan Marathe, Sagar Ranadive, Muhammed Sharique, and John C. Smirnios. 2021. Bringing Cloud-Native Storage to SAP IQ. In *The International Conference on Management of Data, SIGMOD 2021*, pages 2410–2422.

[3] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Antoine Murat, Athanasios Xygkis, and Igor Zablotchi. 2023. uBFT: Microsecond-Scale BFT using Disaggregated Memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023*, pages 862–877.

[4] Nabihah Ahmad and S. M. Rezaul Hasan. 2021. A new ASIC implementation of an advanced encryption standard (AES) crypto-hardware accelerator. *Microelectron. J.*, 117:105255.

[5] InfiniBand Trade Association et al. 2000. The RoCE Initiative.

[6] InfiniBand Trade Association et al. 2007. InfiniBand Architecture Specification Volume 1.Release 1.2.1.

[7] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. 2017. QSGD: communication-efficient SGD via gradient quantization and encoding. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017*, pages 1709–1720.

[8] Noga Alon, Yair Caro, and Raphael Yuster. 1998. Packing and covering dense graphs. *Journal of Combinatorial Designs*, 6(6):451–472.

[9] Noga Alon, Phillip B. Gibbons, Yossi Matias, and Mario Szegedy. 1999. Tracking Join and Self-Join Sizes in Limited Storage. In *Proceedings of the Eighteenth ACM Symposium on Principles of Database Systems, SIGACT-SIGMOD-SIGART 1999*, pages 10–20.

[10] Noga Alon, Yossi Matias, and Mario Szegedy. 1996. The Space Complexity of Approximating the Frequency Moments. In *Proceedings of the 28th Annual ACM Symposium on the Theory of Computing 1996*, pages 20–29.

[11] Meteb M. Altaf, Eball H. Ahmad, Wei Li, Houxiang Zhang, Guoyuan Li, and Changshun Yuan. 2015. An ultra-high-speed FPGA based digital correlation processor. *IEICE Electron. Express*, 12(8):20150214.

[12] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. 2020. Can far memory improve job throughput? In *The 15th EuroSys Conference 2020, EuroSys 2020*, pages 14:1–14:16.

[13] AMD. 2022. Heterogeneous Accelerated Compute Clusters.

[14] AMD-Xilinx. Floating-Point Operator v7.1 Product Guide (PG060).

[15] AMD-Xilinx. 2020. Versal: The First Adaptive Compute Acceleration Platform (ACAP)(WP505).

[16] AMD-Xilinx. 2022. Defense and Space Grade Products.

[17] Mihnea Andrei, Christian Lemke, Günter Radestock, Robert Schulze, Carsten Thiel, Rolando Blanco, Akanksha Meghlan, Muhammad Sharique, Sebastian Seifert, Surendra Vishnoi, Daniel Booss, Thomas Peh, Ivan Schreter, Werner Thesing, Mehul Wagle, and Thomas Willhalm. 2017. SAP HANA adoption of non-volatile memory. *Proc. VLDB Endow.*, 10(12):1754–1765.

[18] Buda Andrzej and Jarynowski Andrzej. 2010. Life-time of correlations and its applications, vol. 1.

[19] Panagiotis Antonopoulos, Peter Byrne, Wayne Chen, Cristian Diaconu, Raghavendra Thallam Kodandaramaih, Hanuma Kodavalla, Prashanth Purnananda, Adrian-Leonard Radu, Chaitanya Sreenivas Ravella, and Girish Mittur Venkataramanappa. 2019. Constant Time Recovery in Azure SQL Database. *Proc. VLDB Endow.*, 12(12):2143–2154.

[20] Arvind Arasu, Spyros Blanas, Ken Eguro, Manas Joglekar, Raghav Kaushik, Donald Kossmann, Ravishankar Ramamurthy, Prasang Upadhyaya, and Ramarathnam Venkatesan. 2013. Secure database-as-a-service with Cipherbase. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD 2013*, pages 1033–1036.

[21] Arvind Arasu, Spyros Blanas, Ken Eguro, Raghav Kaushik, Donald Kossmann, Ravishankar Ramamurthy, and Ramarathnam Venkatesan. 2013. Orthogonal Security with Cipherbase. In *The 6th Biennial Conference on Innovative Data Systems Research, CIDR 2013*.

[22] Kubilay Atasu, Luca Breveglieri, and Marco Macchetti. 2004. Efficient AES implementations for ARM based platforms. In *Proceedings of the ACM Symposium on Applied Computing, (SAC) 2004*, pages 841–845.

[23] AWSCloud. 2020. AQUA (Advanced Query Accelerator) for Amazon Redshift.

[24] Microsoft Azure. 2020. What is Azure Synapse Analytics?

[25] Lars Backstrom, Paolo Boldi, Marco Rosa, Johan Ugander, and Sebastiano Vigna. 2012. Four degrees of separation. In *Web Science, WebSci 2012*, pages 33–42.

[26] Luiz André Barroso, Mike Marty, David A. Patterson, and Parthasarathy Ranganathan. 2017. Attack of the killer microseconds. *Commun. ACM*, 60(4):48–54.

[27] Mark Bingeman. 2019. HBM and FPGAs.

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

[29] Burton H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426.

[30] Michaela Blott, Kimon Karras, Ling Liu, Kees A. Vissers, Jeremia Bär, and Zsolt István. 2013. Achieving 10Gbps Line-rate Key-value Stores with FPGAs. In *The 5th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2013*.

[31] Tim Bock. What is a Correlation Matrix?

[32] Sebastian Breß, Felix Beier, Hannes Rauhe, Kai-Uwe Sattler, Eike Schallehn, and Gunter Saake. 2013. Efficient co-processor utilization in database query processing. *Inf. Syst.*, 38(8):1084–1096.

[33] Paul Brown and Peter J. Haas. 2003. BHUNT: Automatic Discovery of Fuzzy Algebraic Constraints in Relational Data. In *Proceedings of 29th International Conference on Very Large Data Bases, VLDB 2003*, pages 668–679.

[34] John Burgess. 2020. RTX on - the NVIDIA turing GPU. *IEEE Micro*, 40(2):36–44.

[35] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz S. Czajkowski, Stephen Dean Brown, and Jason Helge Anderson. 2013. LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems. *ACM Trans. Embed. Comput. Syst.*, 13(2):24:1–24:27.

[36] Lei Cao and Elke A. Rundensteiner. 2013. High Performance Stream Query Processing With Correlation-Aware Partitioning. *Proc. VLDB Endow.*, 7(4):265–276.

[37] Dar-Jen Chang, Ahmed H. Desoky, Ming Ouyang, and Eric C. Rouchka. 2009. Compute Pairwise Manhattan Distance and Pearson Correlation Coefficient of Data Points with GPU. In *The 10th ACIS International Conference on Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing, SNPD 2009*, pages 501–506.

[38] Jeffrey S. Chase, Andrew J. Gallatin, and Kenneth Yocum. 2001. End system optimizations for high-speed TCP. *IEEE Commun. Mag.*, 39(4):68–74.

[39] Surajit Chaudhuri, Bolin Ding, and Srikanth Kandula. 2017. Approximate Query Processing: No Silver Bullet. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD 2017*, pages 511–519.

[40] Shimin Chen, Anastasia Ailamaki, Manos Athanassoulis, Phillip B. Gibbons, Ryan Johnson, Ippokratis Pandis, and Radu Stoica. 2010. TPC-E vs. TPC-C: characterizing the new TPC-E benchmark via an I/O comparison study. *SIGMOD Rec.*, 39(3):5–10.

[41] Shuang Chen, Wei Hu, and Zhenhao Li. 2019. High performance data encryption with AES implementation on FPGA. In *The 5th IEEE International Conference on Big Data Security on Cloud, IEEE International Conference on High Performance and Smart Computing, and IEEE International Conference on Intelligent Data and Security BigDataSecurity/HPSC/IDS 2019*, pages 149–153.

[42] Monica Chiosa, Fabio Maschi, Ingo Müller, Gustavo Alonso, and Norman May. 2022. Hardware acceleration of compression and encryption in SAP HANA. *Proc. VLDB Endow.*, 15(12):3277–3291.

[43] Monica Chiosa, Thomas Preußer, and Gustavo Alonso. 2021. SKT: A One-Pass Multi-Sketch Data Analytics Accelerator. *Proc. VLDB Endow.*, 14(11):2369–2382.

[44] Pawel Chodowiec and Kris Gaj. 2003. Very compact FPGA implementation of the AES algorithm. In *The Proceedings of the 5th International Workshop on Cryptographic Hardware and Embedded Systems, CHES 2003*, volume 2779 of *Lecture Notes in Computer Science*, pages 319–333.

[45] Stavros Christodoulakis. 1983. Estimating record selectivities. *Inf. Syst.*, 8(2):105–115.

[46] Eric S. Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian M. Caulfield, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, et al. 2018. Serving DNNs in Real Time at Datacenter Scale with Project Brainwave. *IEEE Micro*, 38(2):8–20.

[47] Graham Cormode. 2017. Data sketching. *Commun. ACM*, 60(9):48–55.

[48] Graham Cormode and Minos N. Garofalakis. 2005. Sketching Streams Through the Net: Distributed Approximate Query Tracking. In *Proceedings of the 31st International Conference on Very Large Data Bases, VLDB 2005*, pages 13–24.

[49] Graham Cormode, Minos N. Garofalakis, Peter J. Haas, and Chris Jermaine. 2012. Synopses for massive data: Samples, histograms, wavelets, sketches. *Found. Trends Databases*, 4(1-3):1–294.

[50] Graham Cormode and S. Muthukrishnan. 2005. Summarizing and Mining Skewed Data Streams. In *Proceedings of the SIAM International Conference on Data Mining, SDM 2005*, pages 44–55.

[51] Graham Cormode and Shan Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms*, 55(1):58–75.

[52] L.H. Crockett, S.A. Elliot, M.A. Enderwitz, and R.W. Stewart. 2014. *The Zynq Book*, 1. edition. Strathclyde Academic Media, UK.

[53] Marek Cygan, Marcin Pilipczuk, and Michal Pilipczuk. 2016. Known Algorithms for Edge Clique Cover are Probably Optimal. *SIAM J. Comput.*, 45(1):67–83.

[54] Joan Daemen and Vincent Rijmen. 1999. *AES proposal: Rijndael*.

[55] Benoô.0t Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *Proceedings of the International Conference on Management of Data, SIGMOD 2016*, pages 215–226.

[56] Ruslana Dalinina. 2017. Introduction to Correlation.

[57] Bill Dally. 2011. Power, programmability, and granularity: The challenges of ExaScale computing. In *The IEEE International Test Conference, ITC 2011*, page 12.

[58] William J. Dally, Yatish Turakhia, and Song Han. 2020. Domain-specific hardware accelerators. *Commun. ACM*, 63(7):48–57.

[59] Apache DataSketches. 2021. The business challenge: Analyzing big data quickly.

[60] Systems Group David Sidler, ETH Zurich. Distributed accelerator OS.

[61] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. 2020. Tsunami: A Learned Multi-dimensional Index for Correlated Data and Skewed Workloads. *Proc. VLDB Endow.*, 14(2):74–86.

[62] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014*, pages 401–414.

[63] Anshuman Dutt, Chi Wang, Azade Nazi, Srikanth Kandula, Vivek R. Narasayya, and Surajit Chaudhuri. 2019. Selectivity Estimation for Range Predicates using Lightweight Models. *Proc. VLDB Endow.*, 12(9):1044–1057.

[64] Lieven Eeckhout. 2017. Is Moore's Law Slowing Down? What's Next? *IEEE Micro*, 37(4):4–5.

[65] Hadi Esmaeilzadeh, Emily R. Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. 2012. Dark silicon and the end of multicore scaling. *IEEE Micro*, 32(3):122–134.

[66] Mahdi Esmailoghli, Jorge-Arnulfo Quiané-Ruiz, and Ziawasch Abedjan. 2021. COCOA: COrrelation COefficient-Aware Data Augmentation. In *Proceedings of the 24th International Conference on Extending Database Technology, EDBT 2021*, pages 331–336.

[67] Li Fan, Pei Cao, Jussara M. Almeida, and Andrei Z. Broder. 1998. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. In *Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, SIGCOMM 1998*, pages 254–265.

[68] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. 2012. The SAP HANA Database – An Architecture Overview. *IEEE Data Eng. Bull.*, 35(1):28–33.

[69] Arash Fard. 2019. How to Calculate a Correlation Matrix – Data Exploration for Machine Learning.

[70] Daniel Firestone, Andrew Putnam, Sambrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian M. Caulfield, Eric S. Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert G. Greenberg. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *The 15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018*, pages 51–66.

[71] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. 2007. HyperLogLog: The analysis of a near-optimal cardinality estimation algorithm. In *Discrete Mathematics and Theoretical Computer Science*, DMTCS, pages 137–156.

[72] Annie P. Foong, Thomas R. Huff, Herbert H. Hum, Jaidev R. Patwardhan, and Greg J. Regnier. 2003. TCP performance re-visited. In *The IEEE International Symposium on Performance Analysis of Systems and Software 2003*, pages 70–79.

[73] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. 2018. A Configurable Cloud-Scale DNN Processor for Real-Time AI. In *The 45th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2018*, pages 1–14.

[74] Karl Freund. Microsoft: FPGA Wins Versus Google TPUs For AI.

[75] Peter Xiang Gao, Akshay Narayan, Sagar Karandikar, João Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Network Requirements for Resource Disaggregation. In *The 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016*, pages 249–264.

[76] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, Fei Feng, Yan Zhuang, Fan Liu, Pan Liu, Xingkui Liu, Zhongjie Wu, Junping Wu, Zheng Cao, Chen Tian, Jinbo Wu, Jiaji Zhu, Haiyong Wang, Dennis Cai, and Jiesheng Wu. 2021. When Cloud Storage Meets RDMA. In *The 18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021*, pages 519–533.

[77] Google. 2021. LevelDB.

[78] Marc Antoine Gosselin-Lavigne, Hugo Gonzalez, Natalia Stakhanova, and Ali A. Ghorbani. 2015. A Performance Evaluation of Hash Functions for IP Reputation Lookup Using Bloom Filters. In *The 10th International Conference on Availability, Reliability and Security, ARES 2015*, pages 516–521.

[79] Dan Graur, Damien Aymon, Dan Kluser, Tanguy Albrici, Chandramohan A. Thekkath, and Ana Klimovic. 2022. Cachew: Machine learning input data processing as a service. In *The USENIX Annual Technical Conference, ATC 2022*, pages 689–706.

[80] Systems Group. 2021. Scalable Network Stack for FPGAs (TCP/IP, RoCEv2).

[81] Gaël Guennebaud, Benôit Jacob, et al. 2010. Eigen v3. http://eigen.tuxfamily.org.

[82] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. 2016. RDMA over Commodity Ethernet at Scale. In *Proceedings of the ACM Conference, SIGCOMM 2016*, pages 202–215.

[83] Mark A. Hall. 2000. Correlation-based Feature Selection for Discrete and Numeric Class Machine Learning. In *Proceedings of the 7th International Conference on Machine Learning, ICML 2000*, pages 359–366.

[84] Panu Hämäläinen, Timo Alho, Marko Hännikäinen, and Timo D. Hämäläinen. 2006. Design and implementation of low-area and low-power AES encryption hardware core. In *The 9th Euromicro Conference on Digital System Design: Architectures, Methods and Tools (DSD 2006)*, pages 577–583.

[85] Zhenhao He. 2020. Vitis with 100 Gbps TCP/IP.

[86] Zhenhao He, Dario Korolija, and Gustavo Alonso. 2021. EasyNet: 100 Gbps Network for HLS. In *The 31st International Conference on Field-Programmable Logic and Applications, FPL 2021*, pages 197–203.

[87] Max Heimel and Volker Markl. 2012. A First Step Towards GPU-assisted Query Optimization. In *The International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures, ADMS 2012*, pages 33–44.

[88] Carsten Heinz, Jaco A. Hofmann, Jens Korinth, Lukas Sommer, Lukas Weber, and Andreas Koch. 2021. The TaPaSCo Open-Source Toolflow. *J. Signal Process. Syst.*, 93(5):545–563.

[89] Cynthia Helzner. 2022. Least Squares Regression: Formula, Method, and Examples.

[90] John L. Hennessy and David A. Patterson. 2019. A new golden age for computer architecture. *Commun. ACM*, 62(2):48–60.

[91] Stefan Heule, Marc Nunkesser, and Alexander Hall. 2013. HyperLogLog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm. In *Joint EDBT/ICDT Conferences Proceedings, EDBT 2013*, pages 683–692.

[92] Torsten Hoefler, Ariel Hendel, and Duncan Roweth. 2022. The Convergence of Hyperscale Data Center and High-Performance Computing Networks. *Computer*, 55(7):29–37.

[93] Felipe Hoffa. 2017. Counting uniques faster in BigQuery with HyperLogLog++.

[94] David Horn. 2020. Panda.

[95] David Horn. 2023. Digilent Blog - What's different between Vivado and Vitis?

[96] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. 2019. X-engine: An optimized storage engine for large-scale e-commerce transaction processing. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD 2019*, pages 651–665.

[97] Ram Huggahalli, Ravi R. Iyer, and Scott Tetrick. 2005. Direct Cache Access for High Bandwidth Network I/O. In *The 32st International Symposium on Computer Architecture, ISCA 2005*, pages 50–59.

[98] IBM. 2022. What are IaaS, PaaS and SaaS?

[99] IBM. 2023. Maintenance for virtual machines in Azure.

[100] Ihab F. Ilyas, Volker Markl, Peter J. Haas, Paul Brown, and Ashraf Aboulnaga. 2004. CORDS: Automatic Discovery of Correlations and Soft Functional Dependencies. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD 2004*, pages 647–658.

[101] Intel. Compare Benefits of CPUs, GPUs, and FPGAs for Different oneAPI Compute Workloads.

[102] Intel. 2010. Intel Advanced Encryption Standard (AES) New Instructions Set.

[103] Intel. 2017. Open Programmable Acceleration Engine.

[104] Intel. 2020. Intel Stratix 10 Logic Array Blocks and Adaptive Logic Modules User Guide.

[105] Intel. 2021. IL Academic Compute Environment.

[106] Intel. 2021. Intel FPGA Programmable Acceleration Card D5005.

[107] Intel. 2021. Intel FPGA SDK for OpenCL Pro Edition: Getting Started Guide.

[108] Intel. 2021. Intel® FPGA SDK for OpenCL™ Pro Edition: Programming Guide.

[109] Intel. 2021. Intel® QuickAssist Technology.

[110] Intel. 2022. FPGA Medical - FPGAs in Healthcare.

[111] Zsolt István. 2020. Let's add transactions to FPGA-based key-value stores! In *The 16th International Workshop on Data Management on New Hardware, DaMoN 2020*, pages 13:1–13:3.

[112] Zsolt István, Kaan Kara, and David Sidler. 2020. FPGA-Accelerated Analytics: From Single Nodes to Clusters. *Found. Trends Databases*, 9(2):101–208.

[113] Zsolt István, David Sidler, and Gustavo Alonso. 2017. Caribou: Intelligent distributed storage. *Proc. VLDB Endow.*, 10(11):1202–1213.

[114] Zsolt István, Louis Woods, and Gustavo Alonso. 2014. Histograms as a Side Effect of Data Movement for Big Data. In *The International Conference on Management of Data, SIGMOD 2014*, pages 1567–1578.

[115] Andrei Ivanov, Nikoli Dryden, Tal Ben-Nun, Shigang Li, and Torsten Hoefler. 2021. Data Movement Is All You Need: A Case Study on Optimizing Transformers. In *Proceedings of Machine Learning and Systems 2021, MLSys 2021*.

[116] Vladimir Ivanov and Todor Stoilov. 2019. Design and Implementation of Moving Average Calculations with Hardware FPGA Device. In *The 12th Annual Meeting of the Bulgarian Section of Advanced Computing in Industrial Mathematics, SIAM 2017*, pages 189–197. Springer.

[117] Carlos M. Jarque. 2011. Jarque-Bera Test. In Miodrag Lovric, editor, *International Encyclopedia of Statistical Science*, pages 701–702. Springer.

[118] Kimmo U. Järvinen, Matti Tommiska, and Jorma Skyttä. 2003. A fully pipelined memoryless 17.8 Gbps AES-128 encryptor. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA 2003*, pages 207–215.

[119] Jiantong Jiang, Zeke Wang, Xue Liu, Juan Gómez-Luna, Nan Guan, Qingxu Deng, Wei Zhang, and Onur Mutlu. 2020. Boyi: A Systematic Framework for Automatically Deciding the Right Execution Model of OpenCL Applications on FPGAs. In *The ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2020*, pages 299–309.

[120] Insoon Jo, Duck-Ho Bae, Andre S. Yoon, Jeong-Uk Kang, Sangyeun Cho, Daniel D. G. Lee, and Jaeheon Jeong. 2016. Yoursql: A high-performance database system leveraging in-storage computing. *Proc. VLDB Endow.*, 9(12):924–935.

[121] Manas Joglekar, Hector Garcia-Molina, Aditya G. Parameswaran, and Christopher Ré. 2015. Exploiting Correlations for Expensive Predicate Evaluation. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD 2015*, pages 1183–1198.

[122] Lana Josipovic, Andrea Guerrieri, and Paolo Ienne. 2022. From C/C++ Code to High-Performance Dataflow Circuits. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 41(7):2142–2155.

[123] Norman P. Jouppi, Cliff Young, Nishant Patil, David A. Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017*, pages 1–12.

[124] Glenn Judd. 2015. Attaining the Promise and Avoiding the Pitfalls of TCP in the Datacenter. In *The 12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2015*, pages 145–157.

[125] Kaan Kara and Gustavo Alonso. 2016. Fast and robust hashing for database operators. In *The 26th International Conference on Field Programmable Logic and Applications, FPL 2016*, pages 1–4.

[126] Kaan Kara, Ken Eguro, Ce Zhang, and Gustavo Alonso. 2018. Columnml: Column-store machine learning with on-the-fly data transformation. *Proc. VLDB Endow.*, 12(4):348–361.

[127] Kaan Kara, Jana Giceva, and Gustavo Alonso. 2017. FPGA-based Data Partitioning. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD 2017*, pages 433–445.

[128] Kaan Kara, Christoph Hagleitner, Dionysios Diamantopoulos, Dimitris Syrivelis, and Gustavo Alonso. 2020. High Bandwidth Memory on FPGAs: A Data Analytics Perspective. In *The 30th International Conference on Field-Programmable Logic and Applications, FPL 2020*, pages 1–8.

[129] Fernanda Kastensmidt and Paolo Rech. 2016. FPGAs and parallel architectures for aerospace applications. *Soft Errors and Fault-Tolerant Design*.

[130] Ryan Kastner, Janarbek Matai, and Stephen Neuendorffer. 2018. Parallel Programming for FPGAs. *CoRR*, abs/1805.03648.

[131] Yiping Ke, James Cheng, and Wilfred Ng. 2007. Correlation search in graph databases. In *Proceedings of the 13th ACM International Conference on Knowledge Discovery and Data Mining, SIGKDD 2007*, pages 390–399.

[132] Martin Kiefer, Ilias Poulakis, Sebastian Breß, and Volker Markl. 2020. Scotch: Generating FPGA-accelerators for sketching at line rate. *Proc. VLDB Endow.*, 14(3):281–293.

[133] Hideaki Kimura, George Huo, Alexander Rasin, Samuel Madden, and Stanley B. Zdonik. 2009. Correlation Maps: A Compressed Access Method for Exploiting Soft Functional Dependencies. *Proc. VLDB Endow.*, 2(1):1222–1233.

[134] Hideaki Kimura, George Huo, Alexander Rasin, Samuel Madden, and Stanley B. Zdonik. 2010. CORADD: Correlation Aware Database Designer for Materialized Views and Indexes. *Proc. VLDB Endow.*, 3(1):1103–1113.

[135] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. 2018. Selecta: Heterogeneous Cloud Storage Configuration for Data Analytics. In *The USENIX Annual Technical Conference, ATC 2018*, pages 759–773.

[136] Nicolas Klodt, Lars Seifert, Arthur Zahn, Katrin Casel, Davis Issac, and Tobias Friedrich. 2021. A Color-blind 3-Approximation for Chromatic Correlation Clustering and Improved Heuristics. In *The 27th ACM Conference on Knowledge Discovery and Data Mining, SIGKDD 2021*, pages 882–891.

[137] Michael A. Koets and Peter W. A. Roming. 2021. Computationally Efficient Image Correlation for De-blurring with Photon-Counting Instruments. In *2021 IEEE Aerospace Conference (50100)*, pages 1–8.

[138] Dario Korolija, Dimitrios Koutsoukos, Kimberly Keeton, Konstantin Taranov, Dejan S. Milojicic, and Gustavo Alonso. 2022. Farview: Disaggregated Memory with Operator Off-loading for Database Engines. In *The 12th Conference on Innovative Data Systems Research, CIDR 2022*.

[139] Dario Korolija, Timothy Roscoe, and Gustavo Alonso. 2020. Do OS abstractions make sense on FPGAs? In *The 14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020*, pages 991–1010.

[140] Dimitrios Koutsoukos, Ingo Müller, Renato Marroqu6́.0n, Ana Klimovic, and Gustavo Alonso. 2021. Modularis: Modular Relational Analytics over Heterogeneous Distributed Platforms. *Proc. VLDB Endow.*, 14(13):3308–3321.

[141] Amit Kulkarni, Monica Chiosa, Thomas B. Preußer, Kaan Kara, David Sidler, and Gustavo Alonso. 2020. HyperLogLog sketch acceleration on FPGA. In *The 30th International Conference on Field-Programmable Logic and Applications, FPL 2020*, pages 47–56.

[142] Alexander Kumaigorodski, Clemens Lutz, and Volker Markl. 2021. Fast CSV Loading Using GPUs and RDMA for In-Memory Data Processing. In *Datenbanksysteme für Business, Technologie und Web, BTW 2021*, volume P-311 of *LNI*, pages 19–38. Gesellschaft für Informatik, Bonn.

[143] AMD/Xilinx Research Labs. 2017. FINN. https://github.com/Xilinx/finn.

[144] Lak Lakshmanan. 2019. Simplified data transformations for machine learning in BigQuery.

[145] Robert Lasch, Süleyman Sirri Demirsoy, Norman May, Veeraraghavan Ramamurthy, Christian Färber, and Kai-Uwe Sattler. 2020. Accelerating re-pair compression using FPGAs. In *The 16th International Workshop on Data Management on New Hardware, DaMoN 2020*, pages 8:1–8:8.

[146] Donghun Lee, Andrew Chang, Minseon Ahn, Jongmin Gim, Jungmin Kim, Jaemin Jung, Kang-Woo Choi, Vincent Pham, Oliver Rebholz, Krishna Malladi, and Yang-Seok Ki. 2020. Optimizing data movement with near-memory acceleration of in-memory DBMS. In *Proceedings of the 23rd International Conference on Extending Database Technology, EDBT 2020*, pages 371–374.

[147] Se Kwon Lee, Soujanya Ponnapalli, Sharad Singhal, Marcos K. Aguilera, Kimberly Keeton, and Vijay Chidambaram. 2022. DINOMO: An Elastic, Scalable, High-Performance Key-Value Store for Disaggregated Persistent Memory (Extended Version). *CoRR*, abs/2209.08743.

[148] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proc. VLDB Endow.*, 9(3):204–215.

[149] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2019. Kvell: the design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019*, pages 447–461.

[150] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. 2017. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles, SOSP 2017*, pages 137–152.

[151] Yuhong Li, Leong Hou U, Man Lung Yiu, and Zhiguo Gong. 2016. Efficient discovery of longest-lasting correlation in sequence databases. *VLDB J.*, 25(6):767–790.

[152] Allison Linn. 2018. Real-time AI: Microsoft announces preview of Project Brainwave.

[153] Yuchen Liu, Hai Liu, Dongqing Xiao, and Mohamed Y. Eltabakh. 2018. Adaptive correlation exploitation in big data query optimization. *VLDB J.*, 27(6):873–898.

[154] Zhenhua Liu, Yuan Chen, Cullen E. Bash, Adam Wierman, Daniel Gmach, Zhikui Wang, Manish Marwah, and Chris Hyser. 2012. Renewable and cooling aware workload management for sustainable data centers. In *ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS 2012*, pages 175–186.

[155] Guy Lohman. 2014. Is Query Optimization a "Solved" Problem?

[156] Nishad Manerikar and Themis Palpanas. 2009. Frequent items in streaming data: An experimental evaluation of the state-of-the-art. *Data Knowl. Eng.*, 68(4):415–430.

[157] Bernard Marr. 2016. *Key Business Analytics: The 60+ business analysis tools every manager needs to know*. Pearson UK.

[158] Toni Mastelic, Ariel Oleksiak, Holger Claussen, Ivona Brandic, Jean-Marc Pierson, and Athanasios V. Vasilakos. 2014. Cloud Computing: Survey on Energy Efficiency. *ACM Comput. Surv.*, 47(2):33:1–33:36.

[159] Matt McGee. 2014. Microsoft's Catapult Project Aims To Speed Bing Search, Improve Relevancy.

[160] Máire McLoone and John V. McCanny. 2001. High Performance Single-Chip FPGA Rijndael Algorithm Implementations. In *The Proceedings of the 3rd International Workshop on Cryptographic Hardware and Embedded Systems, CHES 2001*, volume 2162 of *Lecture Notes in Computer Science*, pages 65–76.

[161] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. 2008. Why go logarithmic if we can go linear?: Towards effective distinct counting of search traffic. In *The Proceedings of the 11th International Conference on Extending Database Technology, EDBT 2008*, volume 261, pages 618–629.

[162] Microchip. 2022. Aviation With FPGAs.

[163] Microsoft. Empowering Azure Storage with RDMA.

[164] Microsoft. 2020. Deploy ML models to field-programmable gate arrays (FPGAs) with Azure Machine Learning.

[165] Microsoft. 2021. Transparent Data Encryption (TDE).

[166] Mehdi Moghaddamfar, Christian Färber, Wolfgang Lehner, Norman May, and Akash Kumar. 2021. Resource-Efficient Database Query Processing on FPGAs. In *Proceedings of the 17th International Workshop on Data Management on New Hardware, DaMoN 2021*, pages 4:1–4:8.

[167] Bonaventura Del Monte, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2022. Rethinking Stateful Stream Processing with RDMA. In *The International Conference on Management of Data, SIGMOD 2022*, pages 1078–1092.

[168] Timothy Prickett Morgan. 2020. The inevitability of FPGAs in the datacenter.

[169] Abdullah Mueen, Suman Nath, and Jie Liu. 2010. Fast approximate correlation for massive time-series data. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD 2010*, pages 171–182.

[170] MySQL. 2020. Encryption and compression functions.

[171] MySQL. 2021. MySQL 8.0 Reference Manual :: 15.11.2 File Space Management.

[172] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A disk-based system with in-memory performance. In *The 10th Conference on Innovative Data Systems Research, CIDR 2020*. www.cidrdb.org.

[173] Thomas Neumann and Guido Moerkotte. 2011. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011*, pages 984–994.

[174] NVidia. 2017. RoCE vs. iWARP Competitive Analysis.

[175] Brendan O'Connor. 2012. Cosine similarity, Pearson correlation, and OLS coefficients.

[176] Shingo Okuno, Fumi Iikura, and Yukihiro Watanabe. 2019. Maintenance Scheduling for Cloud Infrastructure with Timing Constraints of Live Migration. In *IEEE International Conference on Cloud Engineering, IC2E 2019*, pages 179–189.

[177] Vladimir Andrei Olteanu, Haggai Eran, Dragos Dumitrescu, Adrian Popa, Cristi Baciu, Mark Silberstein, Georgios Nikolaidis, Mark Handley, and Costin Raiciu. 2022. An edge-queued datagram service for all datacenter traffic. In *The 19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2022*, pages 761–777.

[178] Oracle. 2017. Oracle Exadata Architecture.

[179] Oracle. 2019. Encryption and redaction with oracle advanced security.

[180] Oracle. 2021. General Considerations of Using Transparent Data Encryption.

[181] Thinh Hung Pham, Suhaib A. Fahmy, and Ian Vince McLoughlin. 2013. Low-Power Correlation for IEEE 802.16 OFDM Synchronization on FPGA. *IEEE Trans. Very Large Scale Integr. Syst.*, 21(8):1549–1553.

[182] PyTorch. TORCH.CORRCOEF.

[183] Aditya Rajagopal, Diederik Adriaan Vink, Stylianos I. Venieris, and Christos-Savvas Bouganis. 2020. Multi-precision policy enforced training (muppet) : A precision-switching strategy for quantised fixed-point training of cnns. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020*, volume 119 of *Proceedings of Machine Learning Research*, pages 7943–7952. PMLR.

[184] Waleed Reda, Marco Canini, Dejan Kostic, and Simon Peter. 2022. RDMA is Turing complete, we just did not know it yet! In *The 19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2022*, pages 71–85.

[185] Amazon Redshift. . Using HyperLogLog sketches in Amazon Redshift.

[186] Stefan Richter, Victor Alvarez, and Jens Dittrich. 2015. A seven-dimensional analysis of hashing methods and its implications on query processing. *Proc. VLDB Endow.*, 9(3):96–107.

[187] Burkhard Ringlein, François Abel, Dionysios Diamantopoulos, Beat Weiss, Christoph Hagleitner, Marc Reichenbach, and Dietmar Fey. 2021. A Case for Function-as-a-Service with Disaggregated FPGAs. In *The 14th IEEE International Conference on Cloud Computing, CLOUD 2021*, pages 333–344.

[188] David P. Rodgers. 1985. Improvements in Multiprocessor System Design. In *Proceedings of the 12th Annual Symposium on Computer Architecture, 1985*, pages 225–231.

[189] Joseph Lee Rodgers and W Alan Nicewander. 1988. Thirteen ways to look at the correlation coefficient. *American statistician*, 42:59–66.

[190] Bita Darvish Rouhani, Ebrahim M. Songhori, Azalia Mirhoseini, and Farinaz Koushanfar. 2015. Ssketch: An automated framework for streaming sketch-based analysis of big data on FPGA. In *The 23rd IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2015*, pages 187–194.

[191] Mario Ruiz, David Sidler, Gustavo Sutter, Gustavo Alonso, and Sergio López-Buedo. 2019. Limago: An FPGA-Based Open-Source 100 GbE TCP/IP Stack. In *The 29th International Conference on Field Programmable Logic and Applications, FPL 2019*, pages 286–292.

[192] Florin Rusu and Alin Dobra. 2007. Statistical analysis of sketch estimators. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD 2007*, pages 187–198.

[193] André Ryser, Alberto Lerner, Alex Forencich, and Philippe Cudré-Mauroux. 2022. D-RDMA: Bringing Zero-Copy RDMA to Database Systems. In *The 12th Conference on Innovative Data Systems Research, CIDR 2022*. www.cidrdb.org.

[194] Md. Shiblee Sadik, Le Gruenwald, and Eleazar Leal. 2018. Wadjet: Finding Outliers in Multiple Multi-Dimensional Heterogeneous Data Streams. In *The 34th IEEE International Conference on Data Engineering, ICDE 2018*, pages 1232–1235.

[195] Yasushi Sakurai, Spiros Papadimitriou, and Christos Faloutsos. 2005. BRAID: Stream Mining through Group Lag Correlations. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD 2005*, pages 599–610.

[196] Vivek Seshadri and Onur Mutlu. 2017. Chapter Four - Simple Operations in Memory to Reduce Data Movement. *Adv. Comput.*, 106:107–166.

[197] David Sidler. 2019. *In-Network Data Processing using FPGAs*. Ph.D. thesis, ETH Zurich, Zürich, Switzerland.

[198] David Sidler, Gustavo Alonso, Michaela Blott, Kimon Karras, Kees A. Vissers, and Raymond Carley. 2015. Scalable 10Gbps TCP/IP Stack Architecture for Reconfigurable Hardware. In *The 23rd IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2015*, pages 36–43.

[199] David Sidler, Zsolt István, and Gustavo Alonso. 2016. Low-latency TCP/IP stack for data center applications. In *The 26th International Conference on Field Programmable Logic and Applications, FPL 2016*, pages 1–4.

[200] David Sidler, Zsolt István, Muhsen Owaida, and Gustavo Alonso. 2017. Accelerating Pattern Matching Queries in Hybrid CPU-FPGA Architectures. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD 2017*, pages 403–415.

[201] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, and Gustavo Alonso. 2020. StRoM: Smart Remote Memory. In *The 15th EuroSys Conference, EuroSys 2020*, pages 29:1–29:16.

[202] Sim Simeonov. 2019. Advanced analytics with hyperloglog functions in apache spark.

[203] Snowflake. Overview of Warehouses.

[204] Snowflake. 2014. Understanding Snowflake Table Structures.

[205] Snowflake. 2022. Continuous data loads and file sizing.

[206] NCSS Statistical Software. Correlation Matrix.

[207] Statistics Solutions. Correlation (Pearson, Kendall, Spearman).

[208] Yanwei Song and Engin Ipek. 2015. More is less: improving the energy efficiency of data movement via opportunistic use of sparse codes. In *The Proceedings of the 48th International Symposium on Microarchitecture, MICRO 2015*, pages 242–254.

[209] Laerd Statistics. 2018. Pearson Product-Moment Correlation.

[210] Xiao Sun, Naigang Wang, Chia-Yu Chen, Jiamin Ni, Ankur Agrawal, Xiaodong Cui, Swagath Venkataramani, Kaoutar El Maghraoui, Vijayalakshmi Srinivasan, and Kailash Gopalakrishnan. 2020. Ultra-low precision 4-bit training of deep neural networks. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems, NeurIPS 2020*.

[211] Mellanox Technologies. 2017. RoCE vs. iWARP Competitive Analysis.

[212] Jens Teubner and Louis Woods. 2013. Synthesis Lectures on Data Management. Morgan & Claypool Publishers. [link].

[213] Da Tong and Viktor K. Prasanna. 2015. High throughput sketch based online heavy hitter detection on FPGA. *SIGARCH Comput. Archit. News*, 43(4):70–75.

[214] Da Tong and Viktor K. Prasanna. 2018. Sketch acceleration on FPGA and its applications in network anomaly detection. *IEEE Trans. Parallel Distributed Syst.*, 29(4):929–942.

[215] Martin A. Trefzer and Andy M. Tyrrell, editors. 2015. *Evolvable Hardware - From Practice to Application*. Natural Computing Series. Springer.

[216] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. 2020. Disaggregating Persistent Memory and Controlling Them Remotely: An Exploration of Passive Disaggregated Key-Value Stores. In *The USENIX Annual Technical Conference, ATC 2020*, pages 33–48.

[217] Akihiro Tsutsui and Toshiaki Miyazaki. 1997. YARDS: FPGA/MPU Hybrid Architecture for Telecommunication Data Processing. In *Proceedings of the ACM/SIGDA Fifth International Symposium on Field Programmable Gate Arrays, FPGA 1997*, pages 93–100.

[218] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Heng Wai Leong, Magnus Jahre, and Kees A. Vissers. 2017. FINN: A framework for fast, scalable binarized neural network inference. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2017*, pages 65–74.

[219] Tobias Vinçon, Arthur Bernhardt, Ilia Petrov, Lukas Weber, and Andreas Koch. 2020. nKV: Near-Data Processing with KV-Stores on Native Computational Storage. In *The 16th International Workshop on Data Management on New Hardware, DaMoN 2020*, pages 10:1–10:11.

[220] Qing Wang, Youyou Lu, Jing Wang, and Jiwu Shu. 2022. Replicating Persistent Memory Key-Value Stores with Efficient RDMA Abstraction. *CoRR*, abs/2209.09459.

[221] Shawn Wang. 2019. The difference in five modes in the AES encryption algorithm.

[222] Shibo Wang and Engin Ipek. 2016. Reducing data movement energy via online data clustering and encoding. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016*, pages 1–13.

[223] Theophilus Wellem, Yu-Kuen Lai, and Wen-Yaw Chung. 2015. A Software Defined Sketch System for Traffic Monitoring. In *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for networking and communications systems, ANCS 2015*, pages 197–198.

[224] Louis Woods, Zsolt István, and Gustavo Alonso. 2014. Ibex - an intelligent storage engine with support for advanced SQL off-loading. *Proc. VLDB Endow.*, 7(11):963–974.

[225] Yingjun Wu, Jia Yu, Yuanyuan Tian, Richard Sidle, and Ronald Barber. 2019. Designing Succinct Secondary Indexing Mechanism by Exploiting Column Correlations. In *Proceedings of the International Conference on Management of Data, SIGMOD 2019*, pages 1223–1240.

[226] Xilinx. 2020. Advanced encryption standard (aes) engine v1.1.

[227] Xilinx. . Xilinx adaptive compute cluster (xacc) program.

[228] AMD Xilinx. 2022. DMA/Bridge Subsystem for PCI Express Product Guide (PG195).

[229] AMD Xilinx. 2022. Overview of Arbitrary Precision Integer Data Types (UG1399).

[230] AMD Xilinx. 2022. Vitis Unified Software Platform Documentation: Application Acceleration Development (UG1393).

[231] Hui Xiong, Shashi Shekhar, Pang-Ning Tan, and Vipin Kumar. 2006. TAPER: A Two-Step Approach for All-Strong-Pairs Correlation Query in Large Databases. *IEEE Trans. Knowl. Data Eng.*, 18(4):493–508.

[232] Lei Yu and Huan Liu. 2003. Feature Selection for High-Dimensional Data: A Fast Correlation-Based Filter Solution. In *Proceedings of the Twentieth International Conference of Machine Learning, ICML 2003*, pages 856–863.

[233] Erfan Zamanian, Xiangyao Yu, Michael Stonebraker, and Tim Kraska. 2019. Rethinking Database High Availability with RDMA Networks. *Proc. VLDB Endow.*, 12(11):1637–1650.

[234] Yue Zha and Jing Li. 2021. When application-specific ISA meets FPGAs: a multi-layer virtualization framework for heterogeneous cloud FPGAs. In *The 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2021*, pages 123–134.

[235] Jian Zhang and Joan Feigenbaum. 2006. Finding highly correlated pairs efficiently with powerful pruning. In *Proceedings of the ACM International Conference on Information and Knowledge Management, CIKM 2006*, pages 152–161.

[236] Kuo Zhang, Peijian Wang, Ning Gu, and Thu D. Nguyen. 2022. GreenDRL: managing green datacenters using deep reinforcement learning. In *Proceedings of the 13th Symposium on Cloud Computing, SoCC 2022*, pages 445–460.

[237] Mary Zhang. 2023. Top 10 cloud service providers globally in 2023.

[238] Shuhao Zhang, Feng Zhang, Yingjun Wu, Bingsheng He, and Paul Johns. 2019. Hardware-Conscious Stream Processing: A Survey. *SIGMOD Rec.*, 48(4):18–29.

[239] Tao Zhang, Tianqing Zhu, Ping Xiong, Huan Huo, Zahir Tari, and Wanlei Zhou. 2020. Correlated Differential Privacy: Feature Selection in Machine Learning. *IEEE Trans. Ind. Informatics*, 16(3):2115–2124.

[240] Xin Zhang, Chang Lan, and Adrian Perrig. 2012. Secure and Scalable Fault Localization under Dynamic Traffic Patterns. In *IEEE Symposium on Security and Privacy, SP 2012*, pages 317–331.

[241] Yin Zhang, Sumeet Singh, Subhabrata Sen, Nick G. Duffield, and Carsten Lund. 2004. Online identification of hierarchical heavy hitters: algorithms, evaluation, and applications. In *Proceedings of the 4th ACM SIGCOMM Internet Measurement Conference, IMC 2004*, pages 101–114.

[242] Zhenjie Zhang, Yin Yang, Ruichu Cai, Dimitris Papadias, and Anthony K. H. Tung. 2009. Kernel-based skyline cardinality estimation. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD 2009*, pages 509–522.

[243] Ning Zheng, Xubin Chen, Jiangpeng Li, Qi Wu, Yang Liu, Yong Peng, Fei Sun, Hao Zhong, and Tong Zhang. 2020. Re-think Data Management Software Design Upon the Arrival of Storage Hardware with Built-in Transparent Compression. In *The 12th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2020*.

[244] Yunyue Zhu and Dennis E. Shasha. 2002. StatStream: Statistical Monitoring of Thousands of Data Streams in Real Time. In *Proceedings of the 28th International Conference on Very Large Data Bases, VLDB 2002*, pages 358–369.

[245] Aviad Zuck, Sivan Toledo, Dmitry Sotnikov, and Danny Harnik. 2014. Compression and SSDs: Where and How? In *The 2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads, INFLOW 2014*.