Doctoral Thesis

# Specialized Hardware Solutions for In-Database Analytics and Machine Learning

**Author(s):**
Kara, Kaan

**Publication Date:**
2020-02

**Permanent Link:**

https://doi.org/10.3929/ethz-b-000401051 →

DISS. ETH NO. 26581

# Specialized Hardware Solutions for In-Database Analytics and Machine Learning

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZURICH

(Dr. sc. ETH Zurich)

presented by

KAAN KARA

Master of Science, Karlsruhe Institute of Technology

born on 12.07.1990

citizen of Turkey

accepted on the recommendation of

Prof. Dr. Gustavo Alonso (ETH Zurich), examiner
Prof. Dr. Ce Zhang (ETH Zurich), co-examiner
Prof. Dr. Onur Mutlu (ETH Zurich), co-examiner
Dr. Christoph Hagleitner (IBM Research Zurich), co-examiner

2020

# Abstract

The increasing popularity of advanced data analytics workloads combined with the stagnation of transistor scaling has started to direct the evolution of hardware systems towards more specialization. Examples of hardware being specialized include wider vector instructions on central processing units (CPU), dense multiply-accumulate blocks on graphics and tensor processing units (GPU, TPU), and custom architectures deployed on field-programmable gate arrays (FPGA). The specialization efforts aim at improving how vast amounts of data is processed with advanced analytics algorithms in terms of speed and power consumption. The integration of new hardware capabilities into mature software systems raises many challenges.

In this thesis, we take an in-memory database management system (DBMS) and focus on improving online analytical processing (OLAP) and machine learning (ML) workloads within the DBMS by exploring specialization opportunities offered by an FPGA co-processor. An FPGA provides architectural flexibility without the high cost of designing an integrated circuit. We show that thanks to this flexibility, custom computation pipelines can be designed to improve robust hashing and high fan-out data partitioning for OLAP, and generalized linear model training for ML, consuming quantized data. Furthermore, we show that these custom pipelines can include data transformation units such as decompression and decryption to help with integration of accelerators into highly optimized mature software systems such as a DBMS. As a final step, we come up with an architecture, PipeArch, that maintains the deep pipelining advantages of an FPGA while enabling preemptive scheduling capabilities, leading to a tighter integration of FPGA-based specialization efforts with multi-tenant software systems such as a DBMS.

# Zusammenfassung

Die zunehmende Beliebtheit fortschrittlicher Datenanalyseaufgaben in Verbindung mit der Stagnation der Transistorskalierung führt die Entwicklung von Hardwaresystemen zu einer stärkeren Spezialisierung. Beispiele für spezialisierte Hardware sind breitere Vektoranweisungen für Zentraleinheiten (CPU), dichte Multiplikations-Akkumulationsblöcke auf Grafik- und Tensor-Verarbeitungseinheiten (GPU, TPU) und benutzerdefinierte Architekturen eingesetz auf Feldprogrammierbare Gate-Arrays (FPGA). Die Bemühungen der Spezialisierung zielen darauf ab zu verbessern, wie grosse Datenmengen mit fortschrittlichen Analysealgorithmen in Bezug auf Geschwindigkeit und Stromverbrauch verarbeitet werden. Die Integration neuer Hardwarefunktionen in ausgereifte Softwaresysteme wirft viele Herausforderungen auf.

In dieser Doktorarbeit beschäftigen wir uns mit einem In-Memory Datenbankverwaltungssystem (DBMS) und konzentrieren uns auf die Verbesserung der Arbeitslast für die Online-Analyseverarbeitung (OLAP) und das maschinelle Lernen (ML) innerhalb des DBMS, indem wir die Spezialisierungsmöglichkeiten untersuchen die ein FPGA-Coprozessor bietet. FPGAs bieten Flexibilität in der Architektur ohne die hohen Kosten für den Entwurf einer integrierten Schaltung. Wir zeigen, dass dank dieser Flexibilität spezial angefertigte Berechnungs-Pipelines entworfen werden können, um das robuste Hashing und die Datenpartitionierung für OLAP zu verbessern und das Training für verallgemeinerte lineare Modelle zu verbessern, wobei quantisierte Daten verbraucht werden. Darüber hinaus zeigen wir, dass diese benutzerdefinierten Pipelines Datenumwandlungseinheiten wie Dekomprimierung und Entschlüsselung enthalten können, um die Integration von Beschleunigern in hochoptimierte ausgereifte Softwaresysteme wie ein DBMS zu unterstützen. Als letzten Schritt haben wir eine Architektur namens PipeArch entwickelt, die die Vorteile eines FPGA für tiefe Pipelining beibehält und gleichzeitig präventive Planungsfunktionen ermöglicht. Dies führt zu einer engeren Integration von FPGA-basierten Spezialisierungsbemühungen in mandantenfähige Softwaresysteme wie DBMS.

# Acknowledgments

First and foremost, I would like to thank my advisor Gustavo Alonso for providing constant support and constructive criticism throughout my doctoral studies, always guiding my research with his expertise. Second, I would like to thank my co-advisor Ce Zhang for supporting my research with great ideas and for many intellectually stimulating discussions. I thank Onur Mutlu for his valuable feedback in our collaborations and for being in my thesis committee. I thank Christoph Hagleitner for being in my thesis committee and for his support during my time at IBM Research Zürich.

I would like to thank Michaela Blott and Giulio Gambardella for a productive and fun time at Xilinx Dublin. I would like to thank Ken Eguro, Blake Pelton, and Haohai Yu for a great collaboration and enjoyable time at Microsoft Redmond.

During my 4 years at Systems Group I gathered countless joyful memories besides scholarly and professionally stimulating conversations. For these great times, I thank my colleagues and friends: David, Zsolt, Johannes, Muhsen, Lefteris, Zeke, Hantian, Jana, Darko, Ingo, Renato, Dario, Fabio, Zhenhao, Monica, Dimitris. I also thank my friends at Systems Group for making my life much more enjoyable during this time: Merve, Giray, Can, Simon, Hasan, Minesh, Cedric, Bojan, Johannes, Andrea, and many others.

I would like to thank my parents Gamze and Selim, my brother Kerem for always supporting me in life. Finally, thanks and love to my partner Irem for always being by my side on this journey since we were teenagers.

# Contents

# 1 Introduction

## 1.1 Trends in Modern Computing

There are distinct characteristics in modern computing trends motivating this work:

**(1) Modern Data Processing Trends**

In the past decade, the demands of data processing workloads have been increasing rapidly. On the one hand, the amount of data generated is increasing exponentially. There are many untapped potentials in so-called big data with possibly society-changing impact, such as autonomous drug discovery [CEW+18], personalized health care [MD13], support for renewable energy via better forecasts [SSIS11], and many more. A cornerstone tool to extract and aggregate valuable information from big data is a database management system (DBMS) optimized for online analytical processing (OLAP). Since low query response times are important, these systems are highly optimized to take advantage of the resources of the underlying hardware [BKG+18, FML+12, ZVdWB12, IGN+12]. Thus, their performance depends heavily on advancements in hardware development.

On the other hand, the algorithms to process big data are getting more complex with higher computational demands, mainly driven by the success of machine learning. Learning from data has been especially successful in the fields of computer vision [KSH12] and natural language processing [YHPC18]. Furthermore, the variety of fields where machine learning is applied increases rapidly, with examples such as design automation for chip manufacturing [BP18] to proteome analysis in biology [HZK+19]. A common property of learning based algorithms is their high computational intensity. Combined with the fact that these algorithms need to consume large amounts of data to be successful, increasing their data processing efficiency becomes highly important, which can be achieved mainly

by optimizing the underlying hardware where these algorithms are performed.

**(2) The Slowdown of Moore's Law**

Driven by the requirements of modern data and compute intensive workloads, the demand for efficient and high performance computation is increasing. Since the end of Dennard Scaling [EBA+11] more than a decade ago, the single core performance in processors has been stagnating due to limited clock frequencies. However, the continuation of Moore's Law allowed putting more transistors per chip area, thus leading to the development of CPUs with multiple cores, vectorized instructions, and larger caches. Although increasing the software performance on these processors is not as straightforward as just using a CPU with higher clock frequency as Dennard Scaling enabled in the past, it is still possible for most algorithms by tuning them to the underlying hardware to use multi-core parallelism [BMS+17, FLP+18, PR14], vectorized instructions [WZZY13, PRR15, SKC+10], and the cache hierarchy [BTAÖ13, RKRS07] more efficiently.

However, the advancements in integrated circuit manufacturing has been slowing down in recent years due to physical limitations reached by shrinking transistor sizes, leading to increasing costs for each new integrated circuit process generation [Mac15]. As a result, putting more resources per chip area is becoming highly difficult and is described by a slowdown in Moore's Law [Eec17]. This raises questions about the evolution of future processor architectures. Specialization by means of using available transistors to perform a specific task becomes an alternative to further increase performance and efficiency.

**(3) The Rise of Specialized Hardware**

As a response to these trends, specializing hardware has become important to meet modern computational demands. Broadly defined, hardware specialization is about using the available chip area to perform a specific task or a narrow set of tasks, rather than implementing a general purpose processor to cover a large set of tasks.

Application-Specific Integrated Circuits (ASIC) have been prevalent since the introduction of integrated circuits, however developing and manufacturing an ASIC is very expensive compared to using off-the-shelf processors. Thus, an ASIC is only used if it will be deployed in very large volumes, in the order of millions of units. Furthermore, by definition, ASICs are limited in functionality and only support a narrow set of functions, such as a network interface controller (NIC), a network switch, or a Bluetooth chipset.

However, today's computational demands require flexibility as well as high performance. Some families of workloads have benefited much from already existing specialized yet

programmable architectures. For instance, the training of deep neural networks is a good match to Graphics Processing Units (GPU) [KSH12], which can perform a higher volume of dense linear algebra operations thanks to their many-core architecture compared to a CPU. As the popularity of deep learning increased, GPU vendors started to further specialize the architecture for these workloads [MDCL+18]. Another prominent example in this field is the development of Tensor Processing Units (TPU) by Google [JYP+17], that are designed to be very efficient at dense linear algebra.

Field-Programmable Gate Arrays (FPGA) have become another platform of focus for answering both flexibility and high performance demands of modern workloads. FPGAs are reconfigurable hardware devices that allow the implementation of custom digital logic without actually manufacturing integrated circuits. This way, they enable a cost-effective and convenient way to develop and deploy specialized hardware. Furthermore, thanks to their architectural flexibility and high I/O capability, FPGAs can be integrated in many places in a system: As a co-processor next to a CPU (either as a discrete PCIe device [vcu] or as a coherently attached processor [OSC+11]), as a network-attached standalone processor [WAHH15], or as a drop-in replacement for a NIC for network-faced processing [SAB+15].

These specific properties of FPGAs have led hyperscalers such as Microsoft and Amazon [ama] to deploy them in their datacenters. In Microsoft's case, the Catapult project [PCC+14] puts the FPGA as the backbone of the datacenter infrastructure: An FPGA is placed between each CPU-based server and the datacenter network, to act either as a co-processor for the CPU to offload workloads or as a bump-in-the-wire processor to process data as it is sent or received from the network. The Brainwave project [CFO+18] built on top of this infrastructure targets low latency deep learning inference workloads.

The way specialized hardware solutions are integrated into existing systems has a large effect on the potential gains from specialization. For instance, PCIe-attached devices such as GPUs require copying the data from CPUs memory to their own memory before processing can start. As a result, offload to a GPU only pays off for a relatively large amount of computation on limited sized datasets. Platforms with coherently attached accelerators such as the Xeon+FPGA from Intel [OSC+11] and IBM CAPI [SBJS15] aim to eliminate this disadvantage by allowing the accelerator to work on the main copy of the data, increasing the potential gains. Furthermore, these platforms open up the possibility for the accelerator and the CPU to work together in a fine granular work sharing scheme, leading to so-called hybrid processing capabilities.

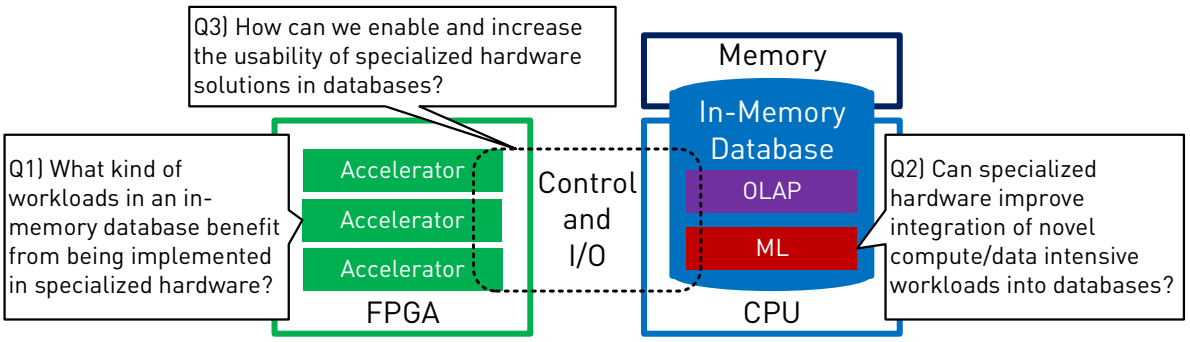



Figure 1.1: The overview of the system of focus and the research questions we ask.

## 1.2 Research Scope and Overview

The scope of this work is focused on improving the underlying hardware by developing specialized hardware solutions for in-database analytics and machine learning.

**System Overview**

We focus on using the FPGA as a co-processor next to a CPU running an in-memory database management system (DBMS), as depicted in Figure 1.1. In the co-processor setup, the main data to be processed resides in CPU's memory. The FPGA uses platform specific interfaces to access this memory, processes data and produces end or intermediate results for various algorithms. The advantage of the co-processor setup is the straightforward integration of FPGA-based accelerators into the system thanks to the unified memory model. Furthermore, any acceleration effort can be compared on a fair basis to a corresponding CPU implementation, leading to convenient design space and trade-off analyses. Nonetheless, the FPGA-based accelerator designs we present in this work are not fundamentally bound to the co-processor setup and can be utilized also in other integration scenarios.

Within the DBMS, we specifically target online analytical processing (OLAP) and machine learning (ML) workloads, and look to provide answers for the following questions:

**Q1) What kind of workloads in an in-memory database benefit from being implemented in specialized hardware?**

Generally, specialized hardware helps for workloads that are compute intensive. These are workloads that have a high computation per data access ratio. Specifically when using FPGAs, their architectural flexibility allows the design of very deep pipelines, and custom

vectorization at computation and memory access. However, FPGAs are at a disadvantage compared to CPUs or GPUs, because of an FPGA's lower operating clock frequency, due to its flexible routing capabilities. That is why, we find that the workloads that highly benefit from a combination of both deep pipelining and custom vectorization lead to a gain in terms of performance and efficiency, compared to being implemented on a CPU. In the following, we summarize the workloads we implemented and analyzed on the FPGA.

**Hashing for Database Operators.** OLAP workloads in a DBMS can be both compute and data intensive, especially since the queries associated with OLAP often include relational joins, aggregation, sorting, etc. Hashing is in the core of many of these operators and performing hashing in a robust way is important in ensuring high performance in their implementation [RAD15]. However, robust hash functions are also compute intensive, so they are a good candidate for being implemented in specialized hardware.

**Data Partitioning.** Relational joins are a very common operation in OLAP workloads, as the queries often require analysis of attributes from multiple tables, leading to denormalization of schemas. However, joins are both data and compute intensive. Recent work [BTAÖ13] has shown that high-fanout partitioned hash join to be particularly efficient on multi-core CPUs, however only if the hash function used is a simple modulo operation. Moreover, the high-fanout partitioning step of the algorithm creates random-access patterns when writing results to the memory. We show that using an FPGA-based partitioning engine can overcome both limitations; using a more robust hash function comes at no additional cost at performance, and a specialized on-chip cache can improve random-access behavior by performing targeted write-combining.

**Quantized Training.** Machine learning workloads are compute and data intensive. Generalized Linear Models (GLM), including linear regression, logistic regression, and support vector machines, are deployed in many domains for classification and regression tasks. Stochastic Gradient Descent (SGD) is one of the most prominent algorithms to train these models thanks its fast convergence properties. Recent work [ZLK$^+$17] has shown that high accuracy training is possible even if the input data is highly quantized, thanks to the error tolerant nature of machine learning. However, current general purpose architectures cannot take full advantage of heavily quantized values, because their arithmetic logic units (ALU) are designed for standard data types such as single-precision floating-point values. With an FPGA-based implementation, we take advantage of both pipelined processing and natively quantized arithmetic to design a compute engine with wider vectorization capability compared to available instructions on a CPU.

**Q2) Can specialized hardware improve the integration of novel compute/data intensive workloads into databases?**

Performing machine learning within the DBMS is a very attractive concept, driving many ongoing efforts both in academia [HRS+12, CVP+13, K+15] and industry [TBC+05, FML+12, AtCG+15]. The reason why this is interesting is, first, businesses have already massive amounts of data in DBMS, which are established and proven systems to manage large amounts of critical data. Ability to train models directly on tables managed in the DBMS is convenient, without the need to extract any data, worry about consistency and staleness. Second, if ML tasks are performed within the DBMS, many novel optimizations become available such as learning models over joins [KNP15], besides obvious advantages such as not having to maintain separate copies of the data for normal query processing and machine learning.

However, the integration of ML algorithms into DBMS comes with certain challenges. First, ML algorithms tend to be more compute intensive compared to OLAP, so the demands on the underlying hardware are different. Secondly, DBMSs are highly optimized systems with certain ways of managing and storing data, that can be unsuitable for ML algorithms. For instance, one of the optimizations for OLAP is the usage of columnar storage when storing tables [IGN+12]. Columnar storage is more efficient in terms of memory access in selective queries. Furthermore, columnar storage allows high compression capability, because same type of data points are stored next to each other; so DBMSs heavily utilize compression. Also, data in a DBMS can be transformed in other ways such as encryption, which is highly relevant in cloud deployments, making the data further unsuitable to use by ML algorithms.

We shot that specialized hardware helps eliminate the challenges associated with integrating ML functionality into a DBMS. First, specialized hardware helps handling the high computation demands of ML algorithms. Second, compute engines can be combined with data transformation modules (e.g., decompression, decryption), enabling training directly over compressed and encrypted data. When a CPU is used to train GLMs over compressed and encrypted data, its performance is reduced much, because the decompression and decryption times dominate. However on an FPGA, deep pipelined execution allows performing decompression, decryption and training all in a parallel dataflow fashion, without reducing the data processing rate.

**Q3) How can we enable and increase the usability of specialized hardware solutions in databases?**

Efficiently using specialized hardware solutions in large-scale multi-tenant software systems such as DBMSs remains an important challenge. Current specialized hardware solutions usually come with the following limitations: (1) limited programmability, (2) rigid execution flows, (3) high effort of development, and (4) platform boundedness. These limitations especially weaken the DBMS use case, because: (1) DBMS are multi-tenant systems that rely on shared execution capabilities to ensure fair progress or low response time guarantees, to name a few. (2) DBMS workloads are multi-faceted, so non-programmable and high-effort hardware solutions are only usable in limited settings. In the final part of this thesis, we try to overcome these limitations by focusing on the usability aspect of our previously developed FPGA-based data processing solutions in an in-memory database.

To further improve usability of specialized hardware solutions, we design a hardware-software architecture called PipeArch, that tackles the usability challenge from multiple perspectives: (1) A modular FPGA-based data processing unit enables programmability while keeping high performance characteristics such as deep pipelining and vectorization. This simplifies both hardware design and provides a generic architecture supporting multiple algorithms rather than just one. (2) The programmable architecture comes with the benefit of flexible execution flows. Thanks to this feature, we implement context switching capability, allowing for the first time fine-grained scheduling on an FPGA-based processor. (3) A software based runtime manager takes advantage of these capabilities to implement various scheduling algorithms such as shortest-job-first and round-robin, and the ability to migrate threads between FPGA-based processors for flexible load balancing. These capabilities provide better support for multi-tenancy in FPGA-based data processors. We implement a number of machine learning algorithms on PipeArch and evaluate the system from multiple aspects such as end-to-end performance, mixed workload execution capabilities, context switching overhead, and thread migration.

## 1.3 Thesis Statement

*Specialized hardware solutions based on co-processor FPGAs can provide performance improvements and reduce the computational burden on CPUs in heterogeneous systems, assuming an advanced data analytics and machine learning context within a database management system. Furthermore, the high effort and limited usability tied to the specialized hardware solutions in this context can be overcome by a modular and programmable hardware-software architecture, supporting shared execution scenarios.*

## 1.4 Contributions and Structure

Summarized per chapter, the contributions made by this thesis to the state-of-the-art is listed in the following.

In **Chapter 3**, we focus on specialized hardware efforts targeting OLAP workloads. We show that FPGA-based acceleration can help increase the robustness of hash functions used in databases and improve the performance/efficiency of the high-fanout data partitioning operation, as commonly used in optimized join algorithms:

- We present fully pipelined and vectorized FPGA-based implementations for *Murmur* and *Simple Tabulation* hash functions, achieving up to 6.6x speedup compared to a single threaded CPU implementation.
- We use the FPGA-based hashing in a hybrid hash table, where the hashing happens on the FPGA, and the maintenance and collision handling happens on the CPU.
- We present the design of an FPGA-based high fanout data partitioning engine, that takes advantage of flexible on-chip memory resources to do efficient and vectorized write-combining. The presented engine is able to match the partitioning throughput of a 10-core CPU, although having 3x less memory bandwidth.
- Based on the previous work in Section 3.1, we show how to use robust hash functions as part of an FPGA-based high fanout data partitioning engine.
- We use the FPGA-based partitioning as part of a hybrid radix join algorithm, where the partitioning happens on the FPGA and the build+probe happens on the CPU, showing hybrid execution capabilities of the target Xeon+FPGA device.

In **Chapter 4**, we show the performance/efficiency of a popular training algorithm in machine learning can be increased substantially by using quantized input data along with an FPGA-based specialized compute engine:

- We implement an FPGA-based pipelined and vectorized stochastic gradient descent (SGD) engine to train GLMs on single-precision floating-point input data, matching the performance of a 10-core Xeon CPU.

- We show how to modify the initial implementation to consume quantized data, while increasing the internal vectorization width such that data can be processed at the same rate while internal computation density increases.
- We perform an in-depth empirical study, analyzing the performance and SGD convergence trade-offs that depend on the quantization level and platform used.

In **Chapter 5**, we show how to perform efficient GLM training on columnar storage by algorithmic analysis and designing an FPGA-based accelerator that enables high performance training directly on compressed and encrypted data:

- We perform an algorithmic analysis and select cache-conscious partitioned stochastic coordinate descent (SCD) as the training algorithm to train GLMs on columnar storage, achieving both high processing rate and good convergence behavior.
- We present the design of an FPGA-based engine that can perform SCD in various configurations and with high performance.
- We show performing training directly on compressed and encrypted data slows down the CPU by an order of magnitude, so we combine the FPGA-based SCD engine with decompression and decryption modules to perform all operations in a pipelined manner, resulting in high performance even when training on transformed data.

In **Chapter 6**, we focus on the integration of specialized hardware solutions into a DBMS. With PipeArch, we consider the integration aspect from a shared execution and multi-tenancy perspective, designing a system that enables context switching and thread migration for threads running on a specialized FPGA-based computation engine, that delivers high performance while being programmable.

- We show how a programmable hardware architecture supporting a wide range of machine learning workloads can match the performance of specialized designs.
- We introduce context-switch capabilities on an FPGA-based data processing solution, enabling runtime scheduling, time-shared execution, and thread migration controlled by a software runtime manager.
- We provide a portable design easily deployable on both an Intel in-package shared-memory FPGA platform and a Xilinx discrete PCIe-attached FPGA platform.

- The overall performance shows up to 4x reduced median job runtime thanks to preemptive scheduling policies, and up to 3.2x speedup for generalized linear model training workloads compared to a 14-core Xeon CPU.

## 1.5 Related Publications

The publications this thesis is based on:

[KA16] Kaan Kara, Gustavo Alonso. **Fast and Robust Hashing for Database Operators**. *In Proceedings of 26th International Conference on Field Programmable Logic and Applications (FPL). August 2016.*

[KGA17] Kaan Kara, Jana Giceva, Gustavo Alonso. **FPGA-Based Data Partitioning**. *In Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD). May 2017.*

[KAA+17] Kaan Kara, Dan Alistarh, Gustavo Alonso, Onur Mutlu, Ce Zhang. **FPGA-accelerated Dense Linear Machine Learning: A Precision-Convergence Trade-off**. *In Proceedings of 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). May 2017.*

[KEZA18] Kaan Kara, Ken Eguro, Ce Zhang, Gustavo Alonso. **ColumnML: Column-Store Machine Learning with On-The-Fly Data Transformation**. *In Proceedings of the VLDB Endowment, 2018, Volume 12(4). August 2019.*

[KWZA] Kaan Kara, Zeke Wang, Ce Zhang, Gustavo Alonso. **doppioDB 2.0: Hardware Techniques for Improved Integration of Machine Learning into Databases**. *In Proceedings of the VLDB Endowment, 2018, Volume 12(12). August 2019.*

[KA] Kaan Kara, Gustavo Alonso. **PipeArch: Generic and Preemptively Scheduled FPGA-based Data Processing**. *Under submission.*

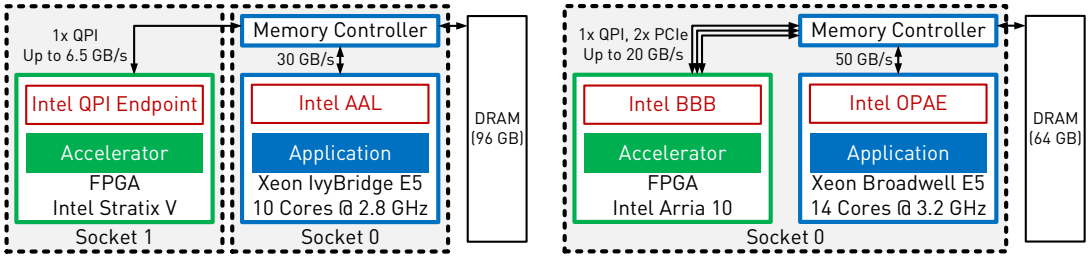
# 2 Background

## 2.1 FPGA as a Data Processing Accelerator

Field-Programmable Gate Arrays (FPGA) are hardware-programmable devices. They consist of a fabric of programmable and specialized resources (lookup tables, flip-flops, digital signal processors, and SRAM arrays) that can be connected in a flexible manner via programmable multiplexers to implement arbitrary logic, computation, and memory. FPGAs have been commonly used in embedded systems across many domains for signal processing and low-latency control tasks and as a prototyping platform for application specific integrated circuits (ASIC). FPGA as a target deployment platform for data processing acceleration is a relatively new concept. This thesis considers this use case of FPGAs, taking advantage of specialization capabilities offered by an FPGA's flexibility when it comes to hardware design. The target workloads are in-database analytics and machine learning, which are increasingly more in demand.

Accelerators based on an FPGA have two main advantages compared to fixed architectures such as a CPU or a GPU:

- Deep pipelining: Very deep and custom pipelines can be implemented to perform algorithms with higher efficiency compared to load/store-based instruction set architectures (ISA).
- Custom vectorization: Vectorization based on single-instruction-multiple-data (SIMD) parallelism can be utilized with high customization to perform either computation or to access on-chip memory resources.




(a) Xeon+FPGA v1. The FPGA and the CPU are on different sockets.

(b) Xeon+FPGA v2. Both the FPGA and the CPU are in the same package.

Figure 2.1: The Intel Xeon+FPGA platform.

Throughout this thesis, the specialized hardware solutions presented utilize a combination of these advantages to perform the data processing task with high efficiency. Depending on the task, one or the other advantage might be more pronounced. However in most cases, to observe a benefit from FPGA-based processing both need to be utilized to a high degree, because the FPGA is at a disadvantage compared to hardened circuits (as in CPU/GPU/ASIC) due to its lower clock frequency, practically in the range of 200-400 MHz. The lower clock frequency stems from the programmable routing capability, which is currently a fundamental limitation. This limitation is addressed with novel routing technologies such HyperFlex from Intel [Hut15] and might lead to higher frequencies being possible in the next generations.

## 2.2 Target Platforms

Throughout this thesis we use two FPGA-based platforms, where the FPGA is attached as a co-processor next to a CPU. In this section, we introduce these platforms in detail, discuss how various accelerators are implemented and deployed in each of them, and explain the software interface to interact with the FPGA-based accelerators focusing on control and memory access.

### 2.2.1 Intel Xeon+FPGA

The Intel Xeon+FPGA are a family of platforms developed thanks to ongoing efforts to integrate an FPGA as close as possible to a Xeon CPU's cache coherent fabric [OSC+11].

We used the first two generations of this platform for the projects in this thesis, as depicted in Figure 2.1.

**The first version Xeon+FPGA** consists of a dual socket with a 10-core CPU (Intel Xeon E5-2680 v2, 2.8 GHz) on one socket and an FPGA (Altera Stratix V 5SGXEA) on the other. The CPU and the FPGA are connected via QPI (QuickPath Interconnect). The FPGA has 64 B cache line and L3 cache-coherent access to the 96 GB of main memory located on the CPU socket. On the FPGA, an encrypted QPI end-point module provided by Intel handles the QPI protocol. This end-point implements a 128 KB two-way associative FPGA-local cache, using the Block-RAM (BRAM) resources. A so-called Accelerator Function Unit (AFU) implemented on the FPGA can access the shared memory pool by issuing read and write requests to the QPI end-point using physical addresses. Our measurements show the QPI bandwidth to be around 6.5 GB/s on this platform for combined read and write channels and with an equal amount of reads and writes.

Since the QPI end-point accepts only physical addresses, the address translation from virtual to physical has to take place on the FPGA using a page-table. Intel also provides an extended QPI end-point which handles the address translation but comes with 2 limitations: 1) The maximum amount of memory that is allocatable is 2 GB; 2) The bandwidth provided by the extended end-point is 20% less compared to the standard end-point. Therefore, we choose to use the standard end-point and implement our own fully pipelined virtual memory page-table using BRAMs. We can adjust the size of the page-table so that the entire main memory could be addressed by the FPGA.

The shared memory operation between the CPU and the FPGA works as follows: At start-up, the software application allocates the necessary amount of memory through the Intel provided API, consisting of 4 MB pages. It then transmits the 32-bit physical addresses of these pages to the FPGA, which uses them to populate its local page-table. During runtime, an accelerator on the FPGA can work on a fixed size virtual address space, where the size is determined by the number of 4 MB pages allocated. Any read or write access to the memory is then translated using the page-table. The translation takes 2 clock cycles, but since it is pipelined, the throughput remains one address per clock cycle. On the CPU side, the application gets 32-bit virtual addresses of the allocated 4 MB pages from the Intel API and keeps them in an array. Accesses to the shared memory are translated by a look-up into this array. The fact that a software application has to perform an additional address translation step is a current drawback of the framework. However, this can very often be circumvented if most of the memory accesses by the application are sequential

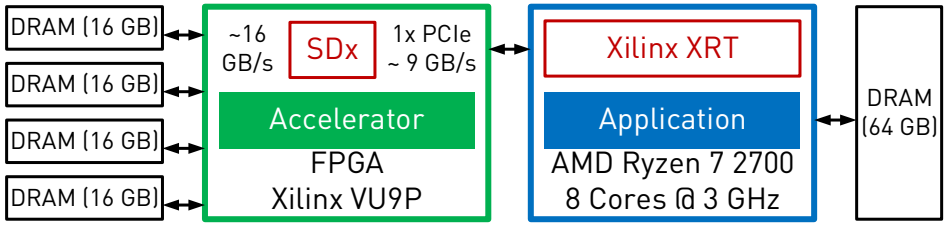



Figure 2.2: The Xilinx VCU1525 attached to an 8-core AMD CPU.

or if the working set fits into a 4 MB page. We observed in our experiments that if the application is written in a conscious way to bypass the additional translation step, no overhead is visible since the translation happens rarely.

**The second version Xeon+FPGA** provides significant improvements over the first version. On this platform a 14-core Broadwell E5 CPU is in the same package as an Arria 10 FPGA. The FPGA has coherent access to the main memory (64 GB) of the CPU via 1 QPI and 2 PCIe links. The read bandwidth when utilizing all 3 memory links is approximately 18 GB/s and the aggregate read-write bandwidth is 20 GB/s.

We implement accelerators for this platform using hardware description languages (VHDL or SystemVerilog), against a 64 B cacheline granular read/write-request based interface. This interface works with physical addressing, so address translation has to take place on the FPGA, similar to the first version of the platform. Since the address translation module provided by Intel eliminated the previously mentioned limitations in the first version platform, we use this readily available module. On the software side, Intel libraries provide a memory management API to allocate pinned memory that is accessible by both the CPU and the FPGA. Having the base pointer to this address spaces, the FPGA can read/write data to it arbitrarily.

### 2.2.2 Xilinx VCU1525 and SDx

Xilinx VCU1525 is a PCIe-attached card with a VU9P Ultrascale+ FPGA [vcu]. The FPGA has 4 DRAM banks, with each DDR providing around 16 GB/s read bandwidth, resulting in a total read bandwidth of 64 GB/s to 64 GB of memory, if each bank can be utilized fully. We use SDx framework and the Xilinx runtime (XRT) to interface with the FPGA-based accelerators. On the software side, Xilinx SDx is mainly used to allocate memory and move data between the CPU and the DRAM banks attached to

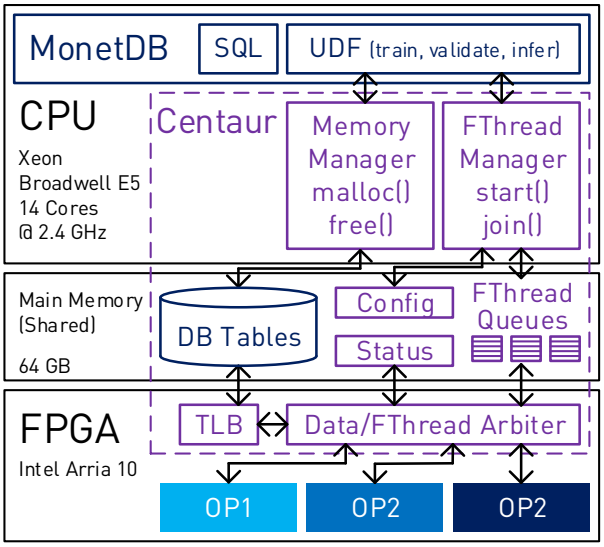



Figure 2.3: An overview of doppioDB: The CPU+FPGA platform and the integration of various operators into MonetDB via Centaur.

the FPGA. For certain operations this is a disadvantage compared to the Xeon+FPGA platform: The FPGA cannot work on the main copy of the data directly, but only on a copy. The advantage however is the much larger bandwidth on the FPGA compared to the Xeon+FPGA, thanks to 4 DRAM banks.

The FPGA-based accelerators can either be implemented using Vivado High-Level-Synthesis (HLS) or VHDL/SystemVerilog. For the projects utilizing this platform in this thesis we use SystemVerilog. Each accelerator can be attached to the DRAM banks via AXI4 memory mapped interfaces.

## 2.3 doppioDB: FPGA-based Data Processing in an Analytical Database

doppioDB is a branch of MonetDB, with FPGA-based data processing capabilities. It is built on the Xeon+FPGA platform by taking advantage of the coherently attached FPGA, that can work on the main copy of the tables stored in the DBMS. A hardware-software layer called Centaur [OSKA17] is used at integrating FPGA-based accelerators into the

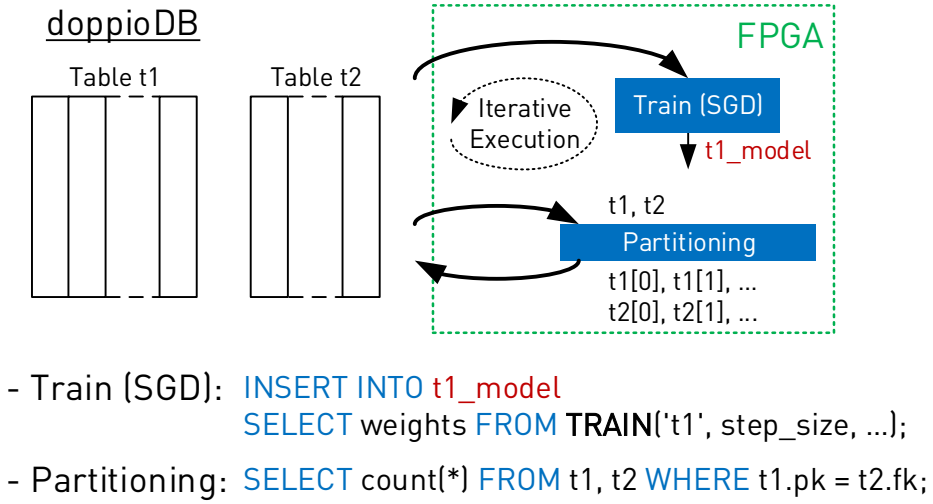



Figure 2.4: Overview of how FPGA-based operators can be used in doppioDB.

DBMS. In Figure 2.3, the components of the system running doppioDB are shown. In the following, we explain these components and their usage.

**MonetDB** is a main memory column-store database, highly optimized for analytical query processing. An important aspect of this database is that it allows the implementation of user-defined-functions (UDFs) in C. The usage of UDFs is highly flexible from SQL. In Figure 2.4, we show a couple of example SQL queries that use FPGA-based operators: Training with SGD and data partitioning. In the training case, a UDF is employed, which can accept entire tables as arguments by name. Data stored in columns can then be accessed efficiently via base pointers in C functions. In the partitioning case, the FPGA-based operator is used by the partitioned join implementation automatically without the need for user intervention.

**Centaur**, as shown in Figure 2.3, provides a set of libraries for memory and thread management to enable easy integration of multiple FPGA-based engines (so-called *FThreads*) into large-scale software systems. Centaur's memory manager dynamically allocates and frees chunks in the shared memory space (pinned by Intel libraries) and exposes them to MonetDB. On the FPGA, a translation lookaside buffer (TLB) is maintained with physical page addresses so that *FThreads* can access data in the shared memory using virtual addresses. Furthermore, Centaur's thread manager dynamically schedules software triggered *FThreads* onto available FPGA resources. These are queued until a corresponding engine

becomes available. For each *FThread* there is a separate queue in the shared memory along with regions containing configuration and status information. Centaur arbitrates memory access requests of *FThreads* on the FPGA and distributes bandwidth equally. How many *FThreads* can fit on an FPGA depends on available on-chip resources.

Thanks to the abstractions provided by doppioDB, we can use certain FPGA-based operators directly within the database, as we show in Chapters 4 and 5. Furthermore, we show how to improve certain aspects of doppioDB in Chapter 6: From a runtime perspective, doppioDB is limited to the first-come-first-serve scheduling policy for the FPGA-based operators and it does not allow context switching for these operators. From a development efficiency perspective, doppioDB does not provide support aside from a standardized interface for memory access. With PipeArch in Chapter 6, we present an alternative that is able to cover these requirements.

## 2.4 Related Work

Throughout the thesis, the detailed related work is presented in each chapter individually. In this section, we give an overview about the related work that studies using FPGA-based specialized hardware as an accelerator within the same domain as this thesis, with the goal to provide a general idea about its position in the state-of-the-art.

**Specialized hardware for online analytical processing (OLAP)** and its usefulness has been demonstrated for a variety of workloads such as joins [HSM+13, HANT15, WGLP13], aggregation [WIA14, DZT13, AHB+16], filtering/selection [ISA16, SIOA17], Skyline computation [WAT13], and extract-transform-load [FZEC17] workloads. In most of these works a complete relational operator is offloaded [HANT15, SIOA17, CO14] to process as much data as possible due to data copying overheads, or the data is processed on its way to the CPU from a storage device [IWA14, WIA14, SL11]. With the recently available CPU+FPGA platforms as we use in this thesis (Section 2.2.1), there are many novel ways to utilize FPGA-based processing thanks to the shared memory model and fine grained collaboration made possible between the CPU and the FPGA. Therefore, within the OLAP context, this thesis focuses on highlighting the collaborative processing opportunities between a coherently-attached FPGA and a multi-core CPU, with example workloads of non-cryptographic hashing in Section 3.1 and high-fanout data partitioning to support join processing in Section 3.2.

**Specialized hardware for machine learning** efforts mainly focused on deep learning (DL) [CFO+18, FOP+18, UFG+17]. For DL-training, FPGAs provide benefits only in a limited setting when a large part of computation can be performed in a quantized way [NVS+17]. Due to the highly compute intensive and regular nature of DL-training, GPUs or ASIC efforts such as the TPU [JYP+17] usually provide a better alternative. For DL-inference, FPGAs can be highly beneficial since high parallelism can be obtained even when processing a single sample, optimizing for the latency [FOP+18]. The projects presented in this thesis are related to the in-database machine learning setting. In-database machine learning can be highly practical, mainly because both commonplace SQL analytics and machine learning can be performed within the same ecosystem [HRS+12, TBC+05, FML+12], that is the DBMS. There are many specialized hardware efforts targeting the family of machine learning more associated with an in-database context such as dense linear models for regression and classification [MPA+16, MCC10], matrix factorization for recommender systems [MKS+18], decision tree ensembles [OAF+19], and clustering [HSIA18]. This thesis presents a first example of how an FPGA can be utilized to train dense linear models using quantized data (Chapter 4), that lead to an improved version [W+19] showing how a database index such as BitWeaving [LP13] can be consumed directly by an FPGA for the same training problem. This thesis also deals with how integrating machine learning into a DBMS creates issues such as assumptions about the data storage format. In this context, previous work [MKS+18] showed that FPGA-based solutions can be beneficial for extracting data from a database page-layout, designed for disk-access. Our work in Chapter 5 focuses on an in-memory column-store context, where native column-wise access and on-the-fly decompression are important.

**Increased usability of specialized hardware** is an extensively studied and active research field. Related efforts focus on high-level-synthesis (HLS) languages [Fei12, CCA+11] to increase the abstraction of designing FPGA-based accelerators, virtualizing FPGAs with partial reconfiguration [VPK18] to enable runtime flexibility, and providing accelerator soft-cores [Kap16, KJYH18] to increase programmability and runtime scheduling capability. This thesis provides a unique position with PipeArch (Chapter 6) in this domain, with a highly specialized yet programmable architecture. PipeArch is complementary to most related work in this field and its contributions can be combined with HLS, partial reconfiguration, and network-on-chip [KG15] solutions.

# 3 Online Analytical Processing: Hashing and Partitioning

## 3.1 Fast and Robust Hashing for Database Operators

Current trends in big data analytics with constantly increasing data sizes demand higher computing capacity. Non-cryptographic hashing is a common operation in data analytics, be it for traditional relational databases operators or more advanced analytics like machine learning. Since it is done so frequently, applications often use simple arithmetic hash functions, which can be computed in just a few CPU cycles [BTAÖ13], [BLP+14]. Intuitively, simple arithmetic hash functions, like modulo or multiply-shift, are not robust against the characteristics of the input data, namely the key-space [RAD15]. If the input data is skewed or has a specific distribution, the simple hash functions may produce high collision rates. Such a hash table will have values that are not distributed uniformly enough to achieve O(1) insertion or look-up cost.

In most cases, FPGAs are used as external accelerators connected to CPUs via PCI-express and a non-coherent communication protocol. This kind of architecture limits the acceleration capabilities of the FPGA for latency sensitive and random-access oriented applications, since the non-coherent interconnect forces an explicit copying of data in large chunks. Therefore, the established consensus is to utilize acceleration only for compute-intensive operations on large amounts of data; otherwise the overhead introduced by moving data to an external processor does not pay off [JRHK15]. Recently, there has been increased interest and development in heterogeneous architectures combining multi-core CPUs and

FPGAs via coherent interconnects. Platforms such as Intel Xeon+FPGA [OSC+11] and Xilinx Zynq [C+14], and coherent interconnects such as OpenCAPI [SSI+18] are available. Such heterogeneous architectures put the FPGA as a first-class citizen in the platform and give it cache-coherent access to the main memory. This type of architecture enables a shared memory programming model, paving the way for hybrid hardware-software applications. The applications can utilize acceleration justifiably even for small data sizes and moderately compute-intensive operations with an overhead for data access by the FPGA comparable to that experienced by a CPU.

**Contributions.** We explore the implementation of two robust hash functions (murmur hashing and simple tabulation hashing) on an FPGA, embedded into a heterogeneous multi-core CPU-FPGA platform. Our experiments show that our hardware hash functions are up to 6.6x faster than their software counterparts. We achieve this speed-up despite being limited by the memory bandwidth available to the FPGA. Our hardware hash functions are fully pipelined and saturate the QPI bandwidth, which means that in future platforms, if the bandwidth between the FPGA and main memory is higher, the speed-up would be proportionally higher. We also use hardware hashing in a hybrid hash table and show how existing software applications can benefit from using hardware functions without changes to their memory layout. The results are especially valuable for in-memory, column-store database operators, where hashing is performed very frequently on 64-bit keys [BTAÖ13], [BLP11] and robust hashing is required to be able to deal with skew.

### 3.1.1 Related Work

FPGAs have been used for accelerating cryptographic hashing extensively [DHV01], [SK05], [MCMM06], [PRR06], [SK10], which utilize the FPGA as a node in the network path to perform cryptographic hashing on streaming data. There are some hash table implementations on FPGAs [IABV13], [ISAV16] focusing on implementing an entire hash table as part of a larger application, for example a key-value store.

To our knowledge, non-cryptographic hashing, which would only produce hash values to be used by a software application, has not yet been studied as a candidate for FPGA acceleration. The reason is that hashing is only a moderately compute-intensive operation and the overhead introduced by a round-trip to an external accelerator would seem excessive. Yet, a recent analysis [RAD15] showed the importance of selecting the right hash function and the hashing scheme in the context of databases. In the provided analysis, the

trade-off between performance and robustness among hash functions is often mentioned. Our contribution lies in showing that an FPGA implementation can break this trade-off by providing both robustness and high performance.

### 3.1.2 Hash Functions

In our evaluation we use the following 6 non-cryptographic hash functions, which are widely used in database engines and are also included in the extensive analysis in [RAD15]:

**Modulo:** A hash value of $n$ bits is produced by taking the $n$ least-significant bits of a $w$-bit key.

**Multiply-Shift:** A hash value of $n$ bits is calculated as follows out of a $w$-bit key, where Z is an odd $w$-bit integer:

$$hash = (key \cdot Z \, mod \, 2^w)/2^{w-n}$$

**Murmur:** It is a frequently used hash function in practice because of its relatively simple computation and robustness. We use the 64-bit finalizer both in software and hardware implementations [App]:

```
1 key = key ⊕ (key >> 33)
2 key = key * 0xff51afd7ed558ccd
3 key = key ⊕ (key >> 33)
4 key = key * 0xc4ceb9fe1a85ec53
5 hash = key ⊕ (key >> 33)
```

**Simple Tabulation:** It is a very robust hash function, based on random value look-ups. It can be proven that $O(1)$ hash table performance (insertion, deletion, look-up) is achieved when simple tabulation and linear probing are used together [PT12]. During its calculation a $w$-bit key is split into $w/8$ characters $c_1, ..., c_{w/8}$. Look-ups are performed at pre-populated (with true random values) tables $T_i$ with $c_i$ as the look-up address. The resulting hash value is obtained by XORing the output values from the tables:

$$hash = \oplus_{i=1}^{w/8} T_i[c_i]$$

**LookUp3 and City Hash:** These are well established hash functions, often used in practice. We use their source code without any changes [Jen], [Goo]. They are included in the analysis mainly for comparison purposes.

Table 3.1: Resource usage of the *Murmur* and *Simple Tabulation* hash function implementations on the target FPGA.

| Unit | Logic | BRAM | DSP |
|---|---|---|---|
| Total | 34% | 5% | 50% |
| QPI Endpoint | 30.8% | 3.3% | 0% |
| Murmur Hasher | 0.9% | 0% | 50% |
| SimpleTab Hasher | 0.74% | 1% | 0% |

### 3.1.3 Implementation

The target platform for this project is the first generation Intel Xeon+FPGA[OSC+11], introduced in Section 2.2.1. In one read request the accelerator receives a 64 B cache line containing 8 64-bit keys. As a result, parallel hashing with 8 hash function units is possible as depicted in Figure 3.1. The data is received out-of-order in terms of addressing. To maintain the ordering in writes, we save the addresses of the received cache lines in a FIFO queue and use them to write back the hash values to the same address plus an offset. This results in a memory layout where keys and hashes are known to belong to each other because of their ordering in memory, a common strategy in column store databases. Thus, the accelerator does not need to write the keys back, only the hashes.

Since murmur hashing is basically a series of bitshifts, XORs and multiplications, it is very suitable for a pipeline implementation of few stages. Two 64-bit multiplications are implemented using 16 cascaded DSPs. 8 murmur hashers need a total 128 DSPs, which is 50% of the available DSPs on the target FPGA. Simple tabulation in hardware is implemented using BRAMs as look-up tables populated with true random values. The one time population of tables happens prior to operation. One BRAM holds 256 32-bit values, resulting in 1 KB tables. 8 tables per hashing unit and 8 hashing units in total results in 64 KB of BRAM usage, which is 1% of available BRAM capacity.

All hardware implementations are done in VHDL. The logic for both hash functions is able to consume and produce a 64 B cache line per clock cycle, utilizing 100% of the available QPI bandwidth. In Table 3.1 the total resource usage is depicted when both hash functions are synthesized and loaded onto the FPGA at the same time. Apart from the QPI endpoint and the hash functions, some glue logic is needed for example to save addresses or to select which hash function to use at runtime.

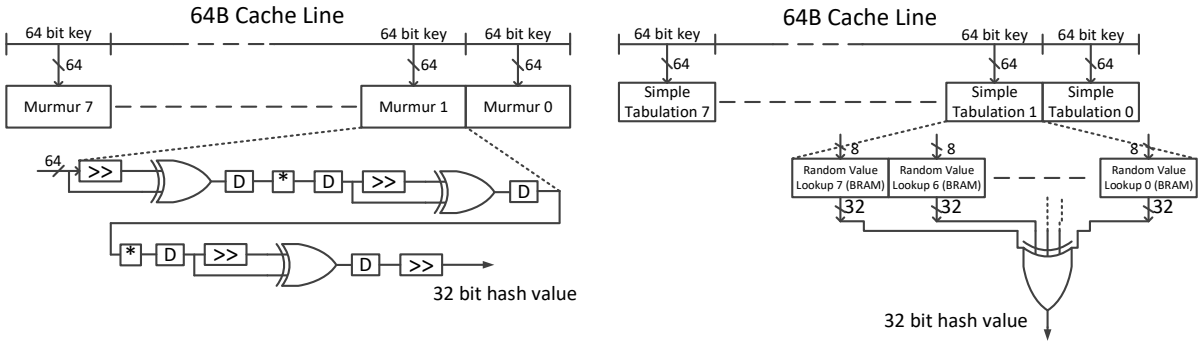



Figure 3.1: *Murmur* and *Simple Tabulation* hashing in hardware.

Table 3.2: Data distributions for the evaluation.

| Linear | Random | Grid | Reverse Grid |
|---|---|---|---|
| 0x0000_0001 | 0x2E4F_5929 | 0x1111_1111 | 0x1111_1111 |
| 0x0000_0002 | 0x82FA_C7B1 | 0x1111_1112 | 0x2111_1111 |
| 0x0000_0003 | 0x186C_BA1F | 0x1111_1113 | 0x3111_1111 |
| ... | ... | ... | ... |
| 0x0001_1AF0 | ... | 0x111E_14E1 | 0x1E41_E111 |
| ... | ... | ... | ... |

## 3.1.4 Experimental Evaluation

### 3.1.4.1 Data Distribution

In the experiments we use N 64-bit keys and 4 different key distributions similar to [RAD15], as shown in Table 3.2. In the linear distribution the keys are in the range $[1 : N]$. The keys in the random distribution are generated by the C standard library pseudo-random generator in the range $[1 : 2^{64} - 1]$. In the grid distribution every byte of a 64 bit key takes a value between 1 and 14. They are generated by incrementing the least significant byte until it reaches 14 and then it is reset to 1 and the next least significant byte is incremented. The reverse grid distribution follows the same pattern as the grid distribution; however bytes are incremented starting from the most significant byte. Both grid distributions represent a different type of dense key-space and hashing these kinds of keys in real applications might be necessary at times (such as certain address patterns or strings). Every generated key distribution is shuffled randomly before experiments, so that no ordered data artifacts are produced.

#### 3.1.4.2 Setup and Methodology

In our experiments we use 6 software hash functions (modulo, multiply-shift, murmur, simple tabulation, lookup3 and city hash) and 2 hardware hash functions (murmur and simple tabulation). Regarding hardware hashing performance, the only difference between the two hardware hash functions is the number of clock cycles they take to produce the hash value. This does not affect performance, since all hardware implementations are fully pipelined. Therefore, we observe the same measurements for both hardware hash functions. All performance measurements are performed on the first version of Intel Xeon+FPGA. Software is written in C++ and compiled with gcc-4.8.2 optimized at -O3. Each measurement is performed with warm caches and with maximum CPU frequency.

#### 3.1.4.3 Experiment 1: Robustness of Hash Functions

In our first experiment we aim to demonstrate the importance of robustness in hashing. To achieve this, we generate 1,468,000 keys using the 4 data distributions described above. The keys are hashed using 6 different hash functions in software. With the resulting hash values, 2 hash tables are built using linear probing and bucket chaining schemes, respectively. Out of the produced 32-bit hash values, 21 most-significant bits are used during hash table insertions. Thus, each 64-bit key hashes to a 21-bit hash (2,097,152 unique values), which results in a 70% fill rate for the hash tables. Both hash tables experience a serious performance reduction if the utilized hash function produces values with high collision rates.

In Table 3.3 and 3.4 we observe that simple arithmetic hash functions (modulo, multiply-shift) have a tendency to fail depending on the input data distribution. Although they behave perfectly for linearly distributed keys, modulo produces many colliding hash values for both grid distributions and multiply-shift is also inadequate for reverse grid distribution. On the other hand murmur, simple tabulation, lookup3 and city hash behave independently of the data distribution.

#### 3.1.4.4 Experiment 2: Performance of Hash Functions

This experiment considers only hashing performance, i.e., data processing rate. We perform either software or hardware hashing on keys and measure the total execution time. The same measurement was performed on a varying the number of keys from $2^{20}$ up to

Table 3.3: Number of average probes in linear probing.

| Key-Space/Hash Func. | Linear | Random | Grid | Reverse Grid |
|---|---|---|---|---|
| modulo | 1 | 2.1686 | 306357.65 | 674958.81 |
| multiply-shift | 1 | 2.1616 | 1.6417 | 17.0139 |
| murmur | 2.1735 | 2.1742 | 2.1677 | 2.1683 |
| simpletab | 2.1584 | 2.1602 | 2.2045 | 2.1369 |
| lookup3 | 2.1678 | 2.1599 | 2.1713 | 2.1667 |
| city | 2.1634 | 2.1694 | 2.1661 | 2.1659 |

Table 3.4: Average and maximum chain length in bucket chaining.

| Key-Space/Hash Func. | | Linear | Random | Grid | Rev. Grid |
|---|---|---|---|---|---|
| modulo | Avg. | 1 | 1.70108 | 534.98 | 498943.73 |
| | Max. | 1 | 7 | 535 | 537824 |
| multiply-shift | Avg. | 1 | 1.69997 | 1.51494 | 14.02882 |
| | Max. | 1 | 7 | 5 | 28 |
| murmur | Avg. | 1.70116 | 1.69966 | 1.7002 | 1.70055 |
| | Max. | 8 | 8 | 8 | 8 |
| simpletab | Avg. | 1.70088 | 1.70017 | 1.69647 | 1.70408 |
| | Max. | 8 | 8 | 8 | 8 |
| lookup3 | Avg. | 1.70081 | 1.70166 | 1.70177 | 1.70073 |
| | Max. | 8 | 8 | 8 | 7 |
| city | Avg. | 1.70119 | 1.69951 | 1.70149 | 1.69940 |
| | Max. | 8 | 8 | 8 | 8 |

$2^{26}$ to observe scalability, which resulted in linear increases in execution times. In Figure 3.2 we observe that simple tabulation hashing is 6.6x and murmur hashing is 1.7x faster in hardware compared to their software counterparts.

#### 3.1.4.5 Experiment 3: Hybrid Hash Table Build

In this experiment a hash table is built using linear probing. Here, two factors affect overall performance: Hash calculation speed and the number of necessary probes to insert the key into the hash table. The number of necessary probes is less, if the produced hash values collide with low probability. Since the build time consists of both hashing and insertion, it is a good metric to evaluate overall hashing quality. A hash function that is both fast to calculate and robust will result in the shortest build times. Similar to

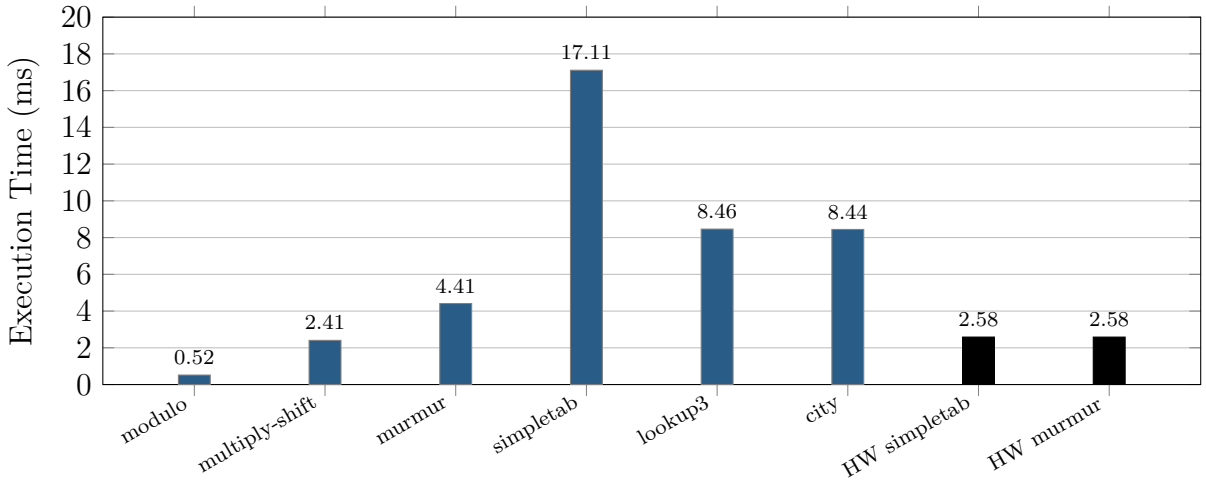



Figure 3.2: Hashing time for $2^{20}$ keys for various hash functions.

Table 3.5: Hash table build times.

| Key-Space/Hash Func. | Linear | Random | Grid | Reverse Grid |
|---|---|---|---|---|
| modulo | 7.72 ms | 22.59 ms | 510.08 s | 1123.16 s |
| multiply-shift | 11.88 ms | 28.42 ms | **23.40 ms** | 60.03 ms |
| murmur | 31.25 ms | 32.24 ms | 32.29 ms | 31.38 ms |
| simpletab | 60.25 ms | 61.47 ms | 60.69 ms | 60.90 ms |
| lookup3 | 39.87 ms | 40.73 ms | 41.37 ms | 40.22 ms |
| city | 39.52 ms | 41.47 ms | 41.08 ms | 40.13 ms |
| HW simpletab | 24.26 ms | 24.38 ms | **24.23 ms** | 24.28 ms |
| HW murmur | 24.21 ms | 24.22 ms | **24.26 ms** | 24.25 ms |

Experiment 1, 1,468,000 keys are generated in 4 distributions and inserted into a $2^{21}$ keys capacity hash table, resulting in a 70% fill rate. The results in Table 3.5 again show the lack of robustness in modulo and multiply-shift, leading to increased build times due to high collision rates, despite being fast in pure hash calculation.

In case of hardware hashing, the CPU and the FPGA collaborate and utilize the shared memory as follows: As soon as the FPGA writes a cache line containing the hashes to memory, the CPU iterates over the hashes in that cache line and inserts the keys into the hash table. Thus, the FPGA can be regarded as the data producer and the CPU as the data consumer, both operating in a pipelined fashion, one cache-line at a time instead of in large batches. This hybrid execution enables hardware acceleration without any visible overhead and hides data transfer latencies over QPI to the FPGA.

We observe the absence of visible overhead in hardware acceleration best when we compare the build times using hardware hashing and multiply-shift hashing for grid distribution in Table 3.5 (in bold). We conclude this after the following analysis: From the pure hashing performance measurements in Figure 3.2 we know that multiply-shift in software and hashing in hardware have almost the same performance. Furthermore, we observed in the robustness measurements that multiply-shift behaves even slightly better than both simple tabulation and murmur for grid distribution. Keeping these two results in mind and observing the same build times in Experiment 3 leads us to conclude that using an FPGA to implement a more expensive hash function leads to the same results as using simple hash functions in software. Yet, the resulting system is far more robust and capable of supporting different data distributions than existing solutions, thereby showing that the FPGA can be used not only to attain better performance but also better quality of results.

## 3.2 FPGA-based Data Partitioning

Modern in-memory analytical database engines achieve high throughput by carefully tuning their implementation to the underlying hardware [MBK02, BTAÖ13, BLP+14]. This often involves a fine-grained partitioning to cluster and split data into smaller cache size blocks to improve locality and facilitate parallel processing. Recent work [SCD16] has confirmed the importance of data partitioning for joins in an analytical query context. Similar results exist for other relational operators [PR14]. Unfortunately, even though it improves the performance of the subsequent processing stages (e.g., in-cache build and probe for the hash join operator), partitioning comes at a cost of additional passes over the data and is very heavy on random data access. In many cases, it can account for a significant portion of the execution time, nearly 90% [SCD16]. Since it is an important, yet expensive sub-operator, partitioning has been extensively studied to make it faster on modern multi-core processors [BATÖ13, PR14, SKD15] and to explore possible hardware acceleration [WHZ15, WBKR13].

In this project, we explore the design and implementation of an FPGA-based data partitioning accelerator. We leverage the flexibility of a hybrid architecture, a 2-socket machine combining an FPGA and a CPU introduced in Section 2.2.1, to show that data partitioning can be effectively accelerated by an FPGA-based design when compared to CPU-based implementations [PR14].

The context for the work is, however, not only hybrid architectures but the increasing heterogeneity of multi-core architectures. In this heterogeneity, FPGAs are playing an increasingly relevant role both as an accelerator as well as a platform for testing hardware designs that are eventually embedded in the CPU. One prominent example of this is Microsoft's Catapult which uses a network of FPGA nodes in the datacenter to accelerate, e.g., page rank algorithms [PCC+14]. Other such hybrid platforms include IBM's POWER9 via OpenCAPI [SSI+18] and Intel's research oriented experimental architecture Xeon+FPGA [OSC+11]. As demonstrated in Section 3.1 with non-cryptographic hashing, Intel Xeon+FPGA, which we also use in this project, brings an FPGA closer to the CPU enabling true hybrid applications where part of the program executes on the CPU and part of it on the FPGA. Especially because of its hybrid nature, this type of architecture enables the exploration of which CPU intensive parts of an application can be offloaded to an FPGA as it has been done for GPUs [PMK14].

**Contributions.** The hash function used for partitioning plays a crucial role but robust hash functions are expensive [RAD15]. Reusing the FPGA-based hash function implementations presented in Section 3.1, we show how to perform robust hashing during partitioning with no performance loss. Furthermore, we show that the partitioning operation can be implemented as a fully pipelined hardware circuit, with no internal stalls or locks, capable of accepting an input and producing an output at every clock cycle. To our knowledge this is the first FPGA partitioner to achieve this and improves throughput by 1.7x over the best reported FPGA partitioner [WHZ15]. Finally, we compare our FPGA based partitioning with a state-of-the-art CPU implementation in isolation and as part of a partitioned hash join. The experiments show that the FPGA partitioner can achieve the same performance as a state-of-the-art parallel CPU implementation, despite the FPGA having 3x less memory bandwidth and an order of magnitude slower clock frequency. The cost model we developed shows that, in future architectures without such structural barriers, FPGA-based partitioning will be the most efficient way to partition data.

### 3.2.1 Target Platform Specific Observations

In this project we use the first version of the Xeon+FPGA (Section 2.2.1). Before discussing performance in later sections, we micro-benchmark this platform regarding the characteristics of the partitioning workload. In Figure 3.3, a comparison of the memory bandwidth available to the CPU and the FPGA is shown, measured for varying sequential

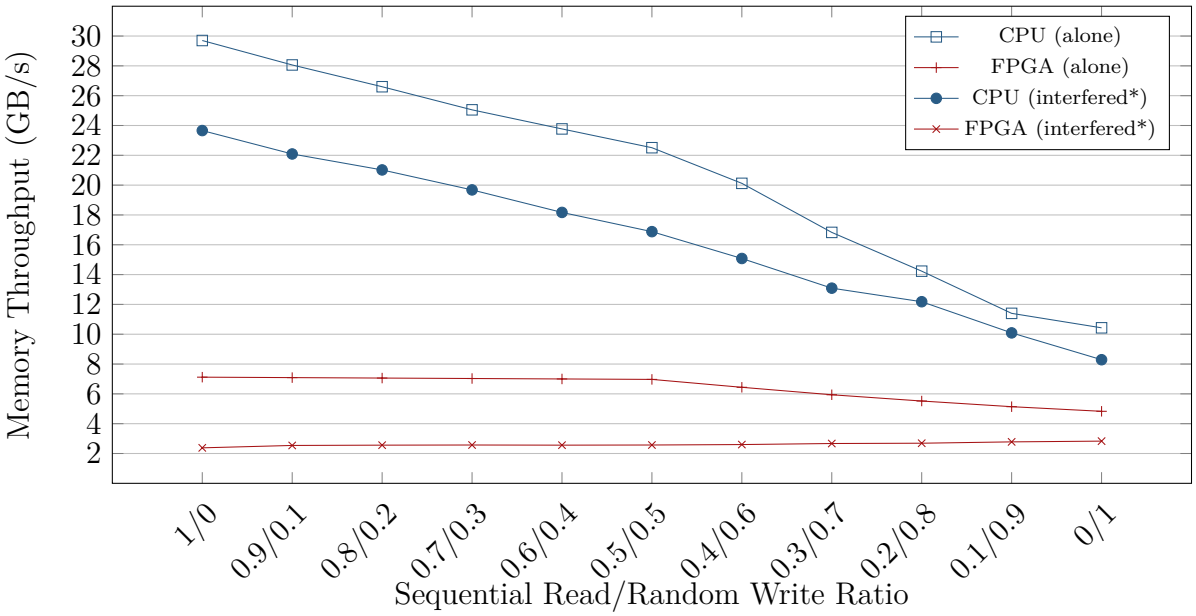



Figure 3.3: Memory bandwidth available to the CPU and QPI bandwidth available to the FPGA depending on the sequential read to random write ratio. *The FPGA sends a balanced ratio of read/write requests (0.5/0.5) at the highest possible rate, while the CPU read/write ratio changes as shown on the x-axis.

Table 3.6: Memory access behavior depending on which socket has lastly written to the memory.

| | CPU reads sequentially | CPU reads randomly |
|---|---|---|
| CPU writes | 0.1381 s | 1.1537 s |
| FPGA writes | 0.1533 s | 2.4876 s |

read to random write ratios, since this is the memory access characteristic that is relevant for the partitioning operation. We observe that when the CPU accesses a memory region which was lastly written by the FPGA, the memory access takes significantly longer in comparison to accessing a memory region lastly written by the CPU. In the micro-benchmark, we first allocate 512 MB of memory and either the CPU or the FPGA fills that region with data. Then, the CPU reads the data from that region (1) sequentially, (2) randomly. The results of the single-threaded experiment is presented in Table 3.6. After the FPGA writes to the memory region, no matter how many times the CPU reads it, it does not get faster. Only after the CPU writes that same region, do the reads become just as fast.

We suspect that this is a side-effect of the cache coherence protocol between two sockets connected by QPI. When the FPGA writes some cache-lines to the memory, the snooping filter on the CPU socket marks those addresses as belonging to the FPGA socket. When the CPU accesses those addresses, they are snooped on the FPGA socket, which causes a delay. Furthermore, the snooping filter gets only updated through writes and not reads. In a homogeneous 2-socket machine with 2 CPUs, this is not an issue because both sockets would have the same amount of L3 cache (in this case 25 MB). The probability of a cache-line residing in the L3 of the other socket would be very high, if that cache-line was last written by that socket. However in the Xeon+FPGA architecture, the FPGA has a cache of only 128 KB and any cache-line that is snooped on the FPGA socket is most likely not found. In short, the snooping mechanism designed for a homogeneous multi-socket architecture causes problems in a heterogeneous multi-socket architecture.

This behavior particularly affects the hybrid join because during the build+probe phase the CPU reads regions of memory last written by the FPGA, when it created the partitions. During the build phase the effect is not as high, because the partitions of the build relation are read sequentially. However, during the probe phase, the build relation needs to be accessed randomly, following the bucket chaining method from [MBK02] and the CPU cannot prefetch data to hide the effects of the needless snooping.

In our experiments, the build+probe phase after the FPGA partitioning is always disadvantaged by this behavior. However, the Xeon+FPGA platform is an experimental prototype, and we expect future version not to have this issue.

### 3.2.2 CPU-based Partitioning

#### 3.2.2.1 Background

In a database, a data partitioner reads a relation and puts tuples in their respective partitions depending on some attribute of the input key, as shown in Algorithm 1. This attribute can be determined by either some simple calculation, e.g., taking a certain number of least significant bits of the key in the case of radix partitioning, or something more complex, e.g., computing a hash value of the key in the case of hash partitioning. Thus, the means of determining which partition a tuple belongs to is a factor affecting the partitioning throughput.

```
1 foreach tuple t in relation R do
2 |   i = hash(t.key)
3 |   partitions[i][counts[i]] = t
4 |   counts[i]++
5 end
```

**Algorithm 1:** Partitioning algorithm.

```
1 foreach tuple t in relation R do
2 |   i = hash(t.key)
3 |   buffers[i][counts[i] mod N] = t
4 |   counts[i]++
5 |   if counts[i] mod N == 0 then
6 |   |   copy buffers[i] to partitions[i]
7 |   end
8 end
```

**Algorithm 2:** Partitioning with software-managed buffers.

Another important aspect is how the algorithm is designed to access the memory when putting the tuples into their respective partitions, called the shuffling phase. Since the shuffling is very heavy on random-access, the performance is limited by TLB misses. Earlier work by Manegol et al. [MBK02] has focused on dividing the partitioning into multiple stages with the goal of limiting the shuffling fan-out of each stage, so that TLB misses can be minimized. Surprisingly, the multiple passes over the data required by this approach pay off in terms of performance. Later on, a more sophisticated solution proposed by Balkesen et al. [BATÖ13] used software-managed cache-resident buffers, an idea first introduced for radix sort by Satish et al. [SKC+10], to improve radix partitioning. The cache-resident buffers, each usually having the size of a cache line, are used to accumulate a certain number of tuples, depending on the tuple size. If a buffer for a certain partition gets full, it is written to the memory, as shown in Algorithm 2.

The size of each buffer (N) should be set so that all the buffers fit into L1 to achieve maximum performance. The advantage of this technique is that it prevents frequent TLB misses without the need of reducing the partitioning fan-out. Thus, a single pass partitioning can be performed very fast. An additional improvement to this was proposed by Wassenberg et al. [WS11] to use non-temporal SIMD instructions for directly writing

the buffers to their destinations in the memory, bypassing the caches. That way the corresponding cache-lines do not need to be fetched and the pollution of caches is avoided. If non-temporal writes are used, N depends also on the SIMD width.

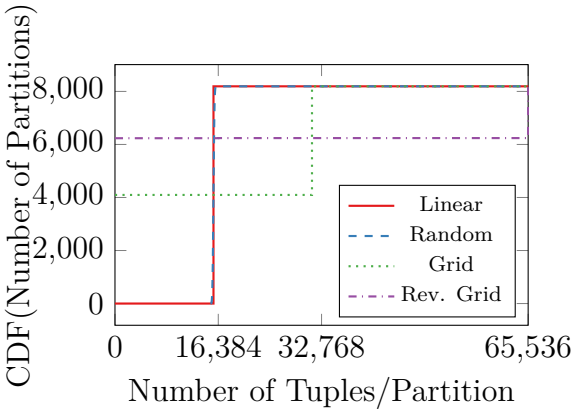
Polychroniou et al. [PR14] and Schuhknecht et al. [SKD15] have both performed extensive experimental analysis on data partitioning and confirmed that best throughput can be achieved with the above mentioned optimizations. Accordingly, in the rest of this work we use the open-sourced implementation from Balkesen et al. [BATÖ13] as the software baseline and use a *single-pass partitioning* with *software-managed buffers* and *non-temporal writes* enabled.

#### 3.2.2.2 Radix vs Hash Partitioning

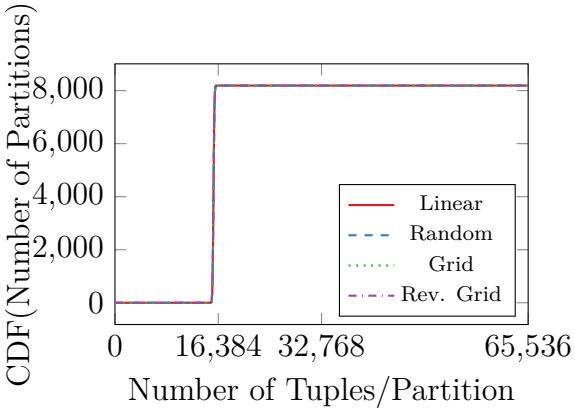
Richter et al. [RAD15] describe the importance of implementing robust hash functions for analytical workloads. In particular, they have shown that for certain key distributions simple and inexpensive radix-bit based hashing can be very ineffective in achieving a well distributed hash value space. On the other hand, robust hash functions can produce infrequently colliding and well distributed hash values for every key distribution but they are computationally costly. Consequently, there is a trade-off between hashing robustness and performance, as also discussed extensively in Section 3.1. To observe this trade-off in the context of data partitioning, we perform both radix and hash partitioning on 4 different key distributions (following [RAD15]), as introduced earlier in Section 3.1.4.1: Linear, Random, Grid, and Reverse Grid.

When radix partitioning is used, the resulting partitions may have unbalanced distributions as depicted in Figure 3.4a. On the other hand, if hash partitioning is used with a robust hash function, such as murmur hashing [App], the created partitions have approximately the same number of tuples as shown in Figure 3.4b.

Although hash partitioning is robust against various key distributions, using a powerful hash function lowers the partitioning throughput as depicted in Figure 3.5. However, as the number of threads is increased, the partitioning process becomes memory bound. Consequently, there are free clock cycles available as the CPU waits on memory requests. These free clock cycles can be used for calculating a more powerful hash function. Thus, the throughput slowdown observed with few threads disappears.




(a) Radix partitioning is applied.




(b) Hash partitioning is applied.

Figure 3.4: The distribution of tuples across 8192 partitions represented as a cumulative distribution function.

#### 3.2.2.3 Partitioned Hash Join

The relational equi-join is a primary component of almost all analytical workloads and constitutes a significant portion of the execution time of a query. Therefore, it has received a lot of attention in terms of how to improve its performance on modern architectures. A recent paper by Schuh et al. [SCD16] provides a detailed analysis of implementations and optimizations for both hash- and sort-based joins published in the last few years [KKL+09, BATÖ13, BTAÖ13, BLP11, LLA+13]. The conclusion is that partitioned, hardware-conscious, radix hash-joins have a clear performance advantage over non-partitioned and sort-based joins on modern multi-core architectures for large and non-skewed relations. Hence, in this work we evaluate how offloading the partitioning

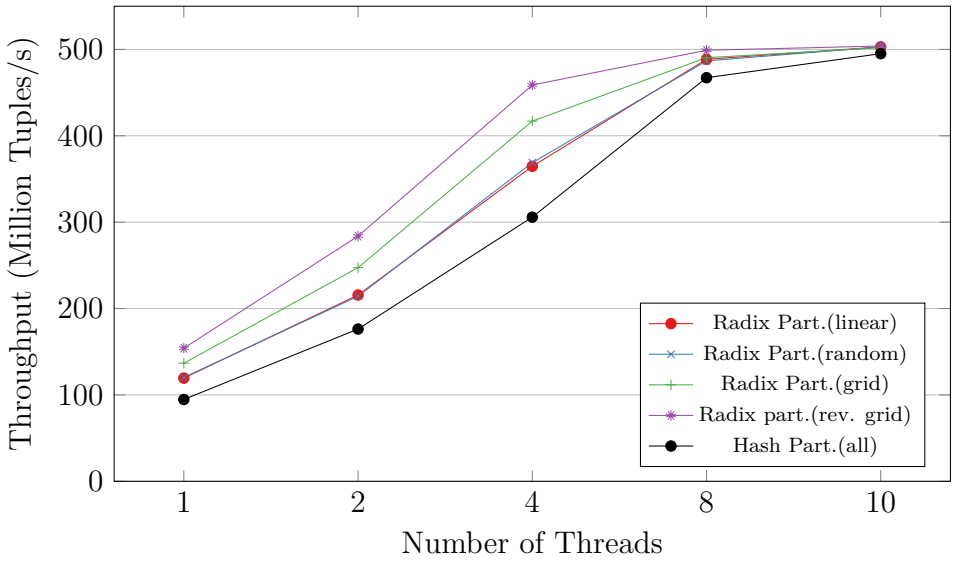



Figure 3.5: CPU Partitioning throughput with 8B tuples, for varying key distributions and partitioning methods. Hash partitioning delivers for every key distribution the same throughput.

phase to a hardware accelerator affects the performance of radix hash join algorithm.

The partitioned hash join algorithm (also known as *radix join*) operates in several stages:

1. Both relations ($R$ and $S$) are partitioned using radix partitioning, so that each partition fits into cache. More details about the implementation and various optimizations are discussed in Section 3.2.2.1.
2. For each partition, a build and probe phase follows:
   (a) During the build phase, a cache resident hash table is built from a partition of $R$.
   (b) During the probe phase, the tuples of the corresponding partition of $S$ are scanned and for each one, the hash table is probed to find a match.

### 3.2.3 FPGA-based Partitioning

In this section we give a detailed description of the VHDL implementation of the FPGA partitioner. Because the accelerators on the Xeon+FPGA architecture access the memory in 64B cache-line granularity, the partitioner circuit is designed to work on that data width. The design is fully pipelined requiring no internal stalls or locking mechanisms

```
1  Input: 64-bit tuple <key,payload>
2  Output: N-bit hash and 64-bit tuple <key,payload>
3  if do_hash == 1 then
4  |   key = key ⊕ (key >> 16)
5  |   key = key * 0x85ebca6b
6  |   key = key ⊕ (key >> 13)
7  |   key = key * 0xc2b2ae35
8  |   key = key ⊕ (key >> 16)
9  |   hash = N LSBs of key
10 else
11 |   hash = N LSBs of key
12 end
```

**Algorithm 3:** Hash Function Module for 4B keys. Pseudo-Code for VHDL, where each line is always active.

regardless of input type. Therefore, the partitioner circuit is able to consume and produce a cache-line at every clock cycle.

When it comes to tuple width, the design supports various tuple sizes (e.g., 8B, 16B, 32B and 64B). Throughout the first parts of this section, we explain the partitioner architecture in the 8B tuple configuration, since this is the smallest granularity we are able to achieve and poses the largest challenge for write combining. Besides, 8B tuples <4B key, 4B payload> are a common scheme in most of the previous work [SCD16, BTAÖ13] and that allows for direct comparisons afterwards in the evaluation. We later show how to change the design to work on larger tuples and how this considerably reduces resource consumption on the FPGA. Additionally, the partitioner has multiple modes of operation for accommodating both column and row store memory layouts, and for handling data with large amounts of skew, as explained in Section 3.2.3.5.

#### 3.2.3.1 Hash Function Module

Figure 3.6 shows the top level design of the circuit. Every tuple in a received cache-line first passes through a hash function module, which can be configured to do either murmur hashing [App] or a radix-bit operation taking $N$ least significant bits of the key. The important thing to keep in mind is that in the synthesized hardware for the pseudo-

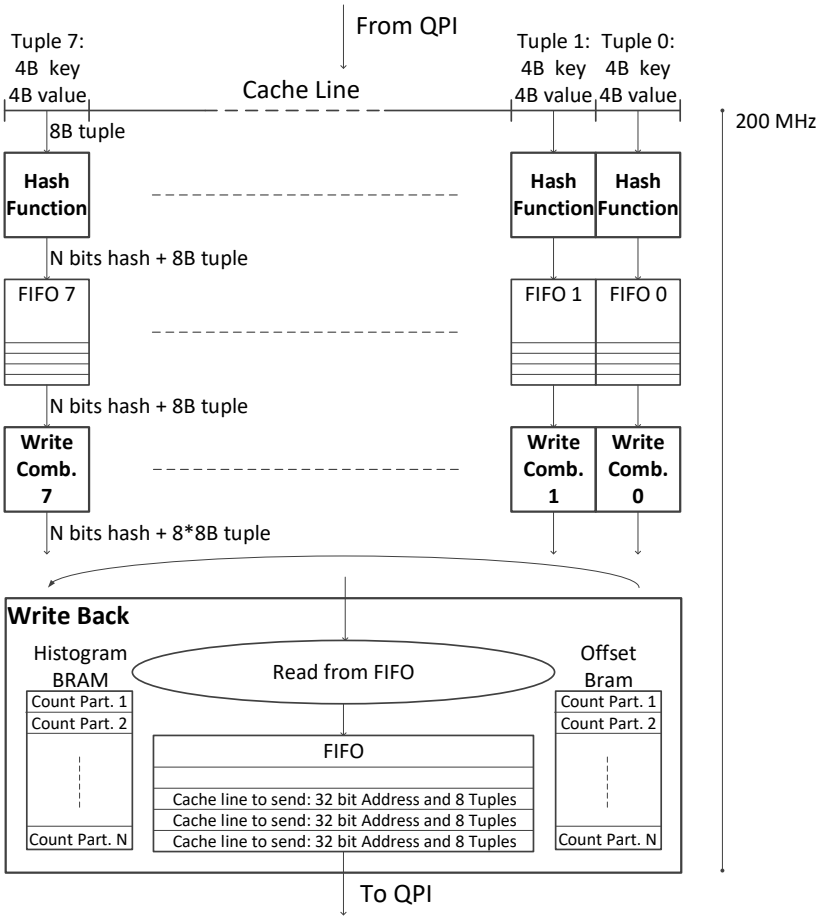



Figure 3.6: Top level design of the hardware partitioner for 8B tuples.

code in Algorithm 3, every calculation (Lines 4-9) is a stage of a pipeline, which is very different from a sequential execution point-of-view of software. For example, when Line 9 executes the multiplication on the first received key, Line 8 can execute the bit-shift on the second received key and so on. Therefore, the hash function module can produce an output at every clock cycle, regardless of how many intermediate stages are inserted into the calculation pipeline. The only thing that increases with additional pipeline stages is the latency. For murmur hashing the latency is 5 clock cycles: $(1/200MHz) \cdot 5 = 25ns$.

#### 3.2.3.2 Write Combiner Module

The output of every hash function module is placed into a first-in, first-out buffer (FIFO), waiting to be read by a write combiner module. The job of the write combiner is to put 8

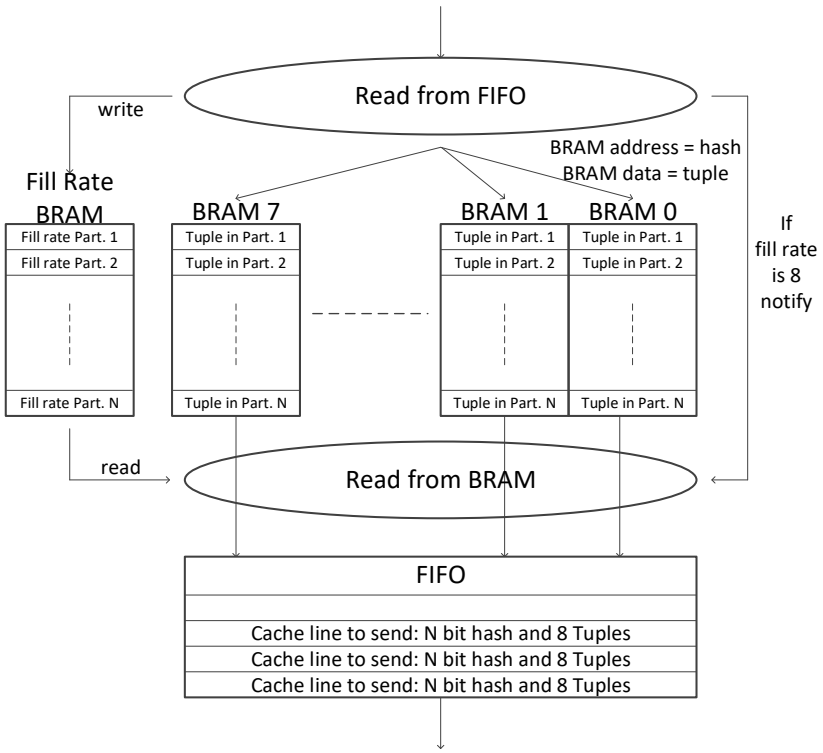



Figure 3.7: Design of the write combiner module for 8B tuples.

tuples belonging to the same partition together in a cache-line before they are written back to the memory. To demonstrate the benefits of write combining, consider the case when no write combiner is used: For every tuple going into a partition first its corresponding cache-line from the memory has to be fetched (64B read). Then, the tuple is inserted into the cache-line at a certain offset and written back to memory (64B write). This operation takes place for every tuple entering the partitioner, say T tuples. In total, the amount of memory transfers *excluding* the initial sequential read of tuples is $(64 + 64) \cdot T\,Bytes$. With write combining enabled we have to write approximately as much data as we have read: $64 \cdot T/8\,Bytes$, a 16x improvement over the no write combining option. Since the hardware partitioner in the current architecture is already bound by memory bandwidth, we choose to implement a fully-pipelined write combiner to achieve the best performance on the current platform.

Here we describe in more detail how the circuit avoids internal locking in the write combiner module. The module reads a tuple from its input FIFO as soon as it sees that the FIFO is not empty. It then reads from an internal BRAM the fill rate of the partition, to which

the tuple belongs. The fill rate holds a value between 0 and 7 (the number of previously received tuples belonging to the same partition), specifying into which BRAM the tuple should be put (see Figure 3.7). Reading the fill rate from the BRAM takes 2 clock cycles. However, during this time the pipeline does not need to be stalled since the BRAM can output a value at every clock cycle. The design challenge here is inserting a forwarding register outside the BRAMs, to handle the cases where a read and a write occur to the same address at the same clock cycle. Basically, a comparison logic recognizes if the current tuple goes into the same partition as one of the previous 2 tuples (Lines 3 and 5 in Algorithm 4). If that is the case, the issued read of the fill rate 2 cycles ago is ignored and the in-flight fill-rate of either one of the matching hashes is forwarded (Lines 4 or 6 in Code 4). Otherwise, the requested fill rate is read to determine which BRAM the tuple belongs to. If the fill rate for any partition reaches 7, it is first reset to 0, the tuple is written to the last BRAM, and then a read from the corresponding addresses of all 8 BRAMs is requested. The actual read from the BRAMs happens 1 clock cycle later, since the BRAMs operate with 1 cycle latency. During the read cycle, if a write occurs to the same address as the read, there is no problem because of this 1 clock cycle latency. No data gets lost and no pipeline stalls are incurred regardless of any input pattern.

At the end of a partitioning run, some partitions in the write combiner BRAMs may remain empty. In fact, this happens almost every time since the scattering of tuples to 8 write combiners are irregular and some cache-lines remain partially filled in the end. To write back every tuple a flush is performed, where every address of the BRAMs is read sequentially and full cache-lines are put into the output FIFO. To obtain a full cache-line in this case, the empty slots are filled with dummy keys, which later on won't be regarded by the software application. Because of this non-perfect gathering after the scattering, the partitioner circuit writes some more data than it receives. In the worst case, every partition in a write combiner module would be filled with one single tuple, making the other 7 tuples at every partition a dummy key. This overhead is in principle no different than the one incurred through aligning and padding to enable the use of SIMD or optimize cache accesses on a CPU.

#### 3.2.3.3 Write Back Module

This module reads the output FIFO of the write combiners in a round-robin fashion and puts the cache-lines in a last stage FIFO to be sent to the main memory via QPI (See

```
1  Input: N-bit hash and 64-bit tuple
2  Output: N-bit hash and 512-bit combined_tuple
3  if hash == hash_1d then
4  |   which_BRAM = which_BRAM_1d + 1
5  else if hash == hash_2d then
6  |   which_BRAM = which_BRAM_2d + 1
7  else
8  |   which_BRAM = fill_rate[hash]
9  end
10 if which_BRAM == 7 then
11 |   fill_rate[hash] = 0
12 |   read_from_BRAM = True
13 else
14 |   fill_rate[hash]++
15 end
16 BRAM[which_BRAM][hash] = tuple
17 if read_from_BRAM_2d == True then
18 |   combined_tuple = BRAM[7][hash_1d],BRAM[6][hash_1d], .., BRAM[0][hash_1d]
19 end
```

**Algorithm 4:** Write Combiner Module. 1d and 2d represent the same signals after 1 and 2 clock cycles, respectively. Pseudo-Code for VHDL, where each line is always active.

Figure 3.6). There are 2 BRAMs which are used to calculate the end destinations of tuples. The first BRAM holds the prefix sum for the histogram that can optionally be built in an initial run over the data (see Section 3.2.3.5). If the histogram is populated, the prefix sum is used to obtain the partition's base address in memory. If the histogram is not populated, a calculated base address via the fixed size partition is used. A second BRAM holds the counts of how many cache-lines have already been written to a certain partition. These counts are used as an offset to the base address to obtain the end address of a current cache-line, which can then be sent. For maintaining the integrity of the offset BRAM, the forwarding logic described in Section 3.2.3.2 is used.

The circuit is able to produce a cache-line at every clock cycle when the entire pipeline is filled, but the QPI bandwidth cannot handle this and puts back-pressure on the write

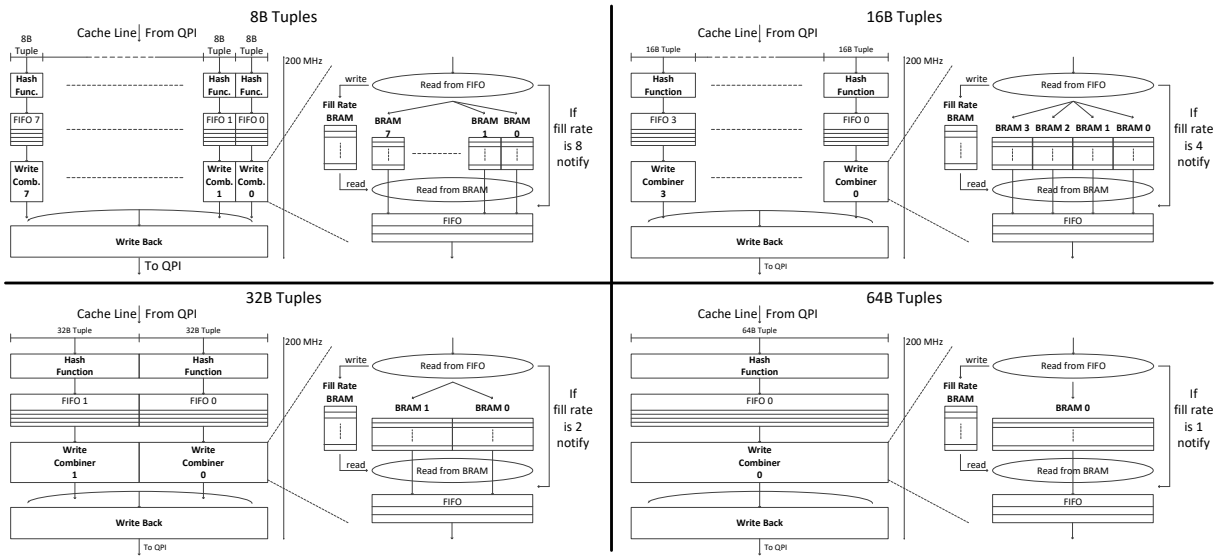



Figure 3.8: Changes to the design to support wider tuples.

Table 3.7: Resource usage depending on tuple width configuration.

| Tuple width | Logic units | BRAM | DSP blocks |
|---|---|---|---|
| 8B | 37% | 76% | 14% |
| 16B | 28% | 42% | 21% |
| 32B | 27% | 24% | 11% |
| 64B | 27% | 15% | 6% |

back module. This back-pressure has to be propagated along the pipeline to ensure that no FIFO experiences an overflow, which would cause data to be lost. We handle this by issuing only so many read requests as there are free slots in the first stage FIFOs just after the hash function modules (see Figure 3.6). If the QPI *write* bandwidth would be more than 12.8 GB/s, which is the output throughput of the circuit, no back-pressure would be put and there would always be free slots in the first stage FIFOs so that new cache-lines would be requested at every clock cycle.

#### 3.2.3.4 Configuring for Wider Tuples

The most complex and resource consuming part of the partitioner is the write combiner module, partly since it has to assemble small tuples to build a full cache-line. Therefore, as we increase the tuple width, the complexity of the write combiner decreases considerably

and the overall design becomes much simpler as depicted in Figure 3.8. However, there is also one part of the design that may become more complex with wider tuples: The hash function. With increasing size of the key to be hashed, more arithmetic units of the FPGA or more pipeline stages may be needed. Nevertheless, the throughput remains a cache-line per clock cycle across all configurations. In Table 3.7 we can observe how the resource usage drops with wider tuples. The only increase observed is for DSP blocks (responsible for arithmetic operations on the FPGA) when going up from 8B to 16B. This is due to the hash function requiring more multiplications per clock cycle to be able to hash 8B keys instead of 4B keys. However, for 32B and 64B tuples, the DSP block usage drops since the write combiner becomes much simpler (less addresses need to be computed).

Figure 3.9 shows the throughput in tuples per second, which understandably decreases with wider tuples, since the partitioner is bandwidth bound. However, the total amount of data processed remains nearly the same, indicating that the partitioner consumes and produces cache-lines at the same rate regardless of the tuple width (as predicted by the analytical model in Section 3.2.3.6).

#### 3.2.3.5 Different Modes of Operation

The partitioner has 2 binary configuration parameters, resulting in 4 modes of operation in total.

1. How to format the output: HIST or PAD mode.

   (a) Histogram Building Mode (HIST). In this mode the partitioner does a first pass over the relation to build a histogram. During the first pass, no data is written back, and the histogram is built using an internal BRAM. During a second pass, the tuples are written out to their partitions using the prefix sum obtained from the saved histogram. In this mode, intermediate memory for holding the partitions is minimized. This mode is also robust against skew, because the number of tuples in each partition is known before writing out begins.

   (b) Padding Mode (PAD). In this mode the partitioner preassigns a fixed size to every partition, which is calculated by: $\#Tuples/\#Partitions + Padding$. As the padding gets larger, the partitioner becomes more robust against skew. In this mode only one pass over the data is done and the tuples are written directly to their partitions by using the fixed sized prefix sum. If one partition gets filled, the operation aborts and

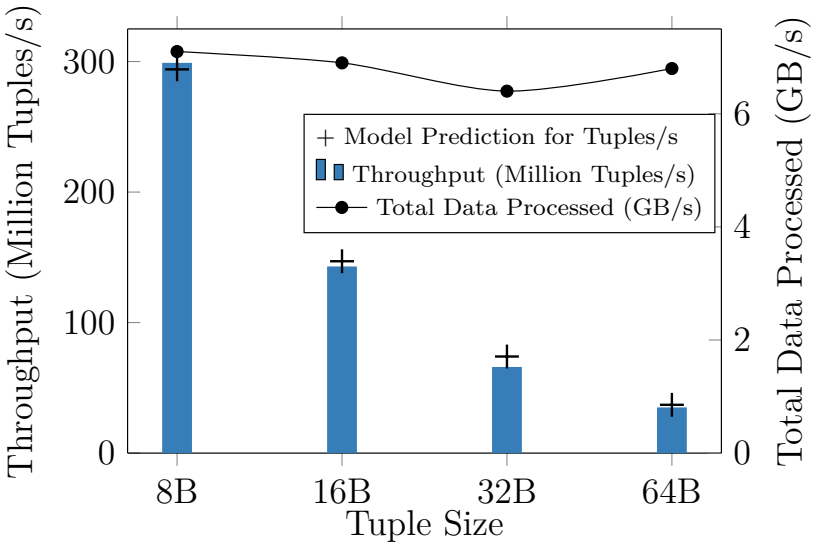



Figure 3.9: Throughput in tuples per second and total amount of data processed with changing tuple width (Mode: HIST/RID).

falls back to a CPU based partitioner. This should happen very rarely and only under large skews with a Zipf factor of more than 0.25.

2. Whether the partitioner works in a column store mode or not: RID or VRID.

   (a) Record ID Mode (RID). In this mode the tuples reside in the main memory as the partitioner expects them: ¡xB key, yB payload¿.

   (b) Virtual Record ID Mode (VRID) is used by column store databases. In this mode the keys and the payloads are assumed to be stored in separate arrays in the memory, only associated by their ordering in the arrays. However, partitioning does not maintain the ordering of tuples in the intermediate result, that is the created partitions. Therefore, in this mode the FPGA only reads the key array as ¡xB key¿ and a virtual record ID is appended to that key on the FPGA, creating a tuple ¡xB key, 4B VRID¿. After the partitioning takes place, the real tuple can be materialized using the VRIDs to associate keys with their payloads.

#### 3.2.3.6 Analytical Model of the FPGA Circuit

Table 3.8 shows the notation of all parameters that we use in this section when developing the model for the partitioning implementation.

Table 3.8: Summary of notation used in cost model.

| Parameter | Description | Values or Units |
|---|---|---|
| $f_{FPGA}$ | Clock frequency of the FPGA | 200 MHz |
| $T_{FPGA}$ | Clock period of the FPGA | 5 ns |
| $CL$ | Width of a cache line | 64 Bytes |
| $W$ | Width of a tuple in bytes | 8, 16, 32 or 64 |
| $r$ | Seq. read/rand. write ratio | 2, 1, 0.5 |
| $T_{mem}$ | Time for memory access | in seconds |
| $T_{process}$ | Time to process | in seconds |
| $B_{FPGA}$ | Partitioner throughput | in tuples/second |
| $L_{FPGA}$ | Partitioner latency | in seconds |
| $f_{mode}$ | Mode factor | 2(HIST), 1(PAD) |
| $B(r)$ | QPI Bandwidth for r | in Bytes/seconds |
| $c_{hashing}$ | Cycles for hashing | 5 |
| $c_{writecomb}$ | Cycles for write combining | 65540 |
| $c_{fifos}$ | Cycles for fifo accesses | 4 |

Let's consider the total processing rate of the FPGA partitioner for $N$ tuples and in units of $tuples/s$:

$$P_{total} = min\{\frac{N}{T_{process}}, \frac{N}{T_{mem}}\} \tag{3.1}$$

The FPGA partitioner completes its execution in:

$$T_{process} = f_{mode}(\frac{N}{B_{FPGA}} + L_{FPGA}) \tag{3.2}$$

where,

$$B_{FPGA} = \frac{CL}{W} f_{FPGA} \tag{3.3}$$

$$L_{FPGA} = (c_{hashing} + c_{writecomb} + c_{fifos}) \cdot T_{FPGA} \tag{3.4}$$

Thus, the processing rate of the FPGA in $tuples/s$ is:

$$P_{FPGA} = \frac{N}{T_{process}} = \frac{1}{f_{mode}(\frac{1}{B_{FPGA}} + \frac{L_{FPGA}}{N})} \tag{3.5}$$

For a sufficiently high $N$, the term $L_{FPGA}/N$ becomes orders of magnitudes smaller than $1/B_{FPGA}$ and the latency is hidden.

The memory access rate (in $tuples/s$) is, when $r \cdot N$ tuples are read and $N$ tuples are written, where a tuple has W Bytes:

$$P_{mem} = \frac{N}{T_{mem}} = \frac{N}{W(Nr+N)/B(r)} = \frac{B(r)}{W(r+1)} \tag{3.6}$$

Thus, total processing rate of the FPGA partitioner becomes:

$$P_{total} = min\{\frac{1}{f_{mode}(\frac{1}{B_{FPGA}} + \frac{L_{FPGA}}{N})}, \frac{B(r)}{W(r+1)}\} \tag{3.7}$$

In the current architecture, the second term in equation 3.7 is always smaller than the first term. Therefore, it defines the rate at which the FPGA partitioner processes the tuples.

#### 3.2.3.7 Performance Analysis

In this section we determine the throughput of the FPGA partitioner using the different modes of operation. Whether radix or hash partitioning is used does not affect performance since computing a hash comes virtually at no additional cost. Figure 3.10 shows the performance of 4 modes of operation of the FPGA partitioner with respect to prior work, the CPU based partitioning and the raw FPGA partitioning throughput. All experiments are performed on the Xeon+FPGA platform and the numbers represent end-to-end partitioning throughput, with the exception of the raw FPGA numbers. The raw FPGA numbers are obtained with the following method to show the throughput capabilities of the partitioner when it is not limited by the bandwidth: An FPGA internal wrapper around the partitioner is implemented to emulate QPI memory access behavior, however with a combined read and write bandwidth of 25.6 GB/s. The wrapper generates tuples internally, feeds the cache-lines to the partitioner when requested, and gets the processed cache-lines from the partitioner to disregard them.

We observe that using both the PAD and VRID modes increases the throughput. Using PAD instead of HIST is clearly faster, because only one scan over the data has to be done instead of two, although the same amount of data has to be written. Using VRID instead of RID also leads to an increase in throughput. This actually is an experimental proof that the hardware partitioner on the FPGA is memory bound. More specifically, in VRID mode, for each cache-line the FPGA receives, two cache-lines are generated internally by appending the virtual record IDs as explained in Section 3.2.3.5. In total the partitioning circuit receives the same number of cache-lines, but over the QPI only half the number of

cache lines are read compared to RID mode. Removing some of the reads from the QPI bandwidth lowers the back-pressure on the writes and the overall throughput increases.

The best direct comparison to CPU partitioning is the HIST/ RID mode, because the CPU algorithm also builds a histogram and uses tuples in ¡4B key, 4B payload¿ layout. However, the partitioning algorithm for the CPU builds the histogram out of necessity, in order to remove synchronization between multiple threads, so that each thread accesses a specific part of memory while writing out the partitions. The FPGA implementation can be seen as single-threaded from a software point-of-view because the memory is accessed from only one agent. Therefore, there is no need to build a histogram for synchronization, an advantage for the FPGA partitioner. We see that the FPGA partitioner reaches the same throughput as the 10-threaded CPU partitioner on the Xeon+FPGA platform. When compared to related work, we observe that we improve the FPGA partitioning throughput reported by Wang et al. [WHZ15]. In comparison to the results reported by Polychroniou et al. [PR14], the raw throughput of our FPGA partitioner in PAD mode is 45% higher when compared with 32 cores.

#### 3.2.3.8 Model Validation

In this section we validate the model. For this, we select a sufficiently large number of tuples $N = 128 \cdot 10^6$ and assume a tuple width of $W = 8B$:

$$P_{total} = min\{\frac{1 \cdot tuples/s}{f_{mode}(6.25\,\text{e}{-10} + 2.58\,\text{e}{-12})}, \frac{B(r)}{W(r+1)}\} \tag{3.8}$$

Obviously, the latency term has become 2 orders of magnitude smaller than the output rate for this $N$, consequently its effect is minimal:

$$P_{total} = min\{f_{mode} \cdot 1.593\,\text{e}9\,tuples/s, \frac{B(r)}{W(r+1)}\} \tag{3.9}$$

For different modes of operation of the FPGA partitioner, the read to write ratio $r$ changes. From Figure 3.3 we can look-up the matching bandwidth for a particular ratio $B(r)$ and it should give us the processing rate, which then can be matched to the experimental results in Figure 3.10.

- In HIST/RID mode the FPGA partitioner reads twice as much data, since the first scan is just for the internal build of the histogram $r = 2$:

$$P_{total} = \frac{7.05GB/s}{8B/tuple \cdot 3} = 294\,Mtuples/s$$

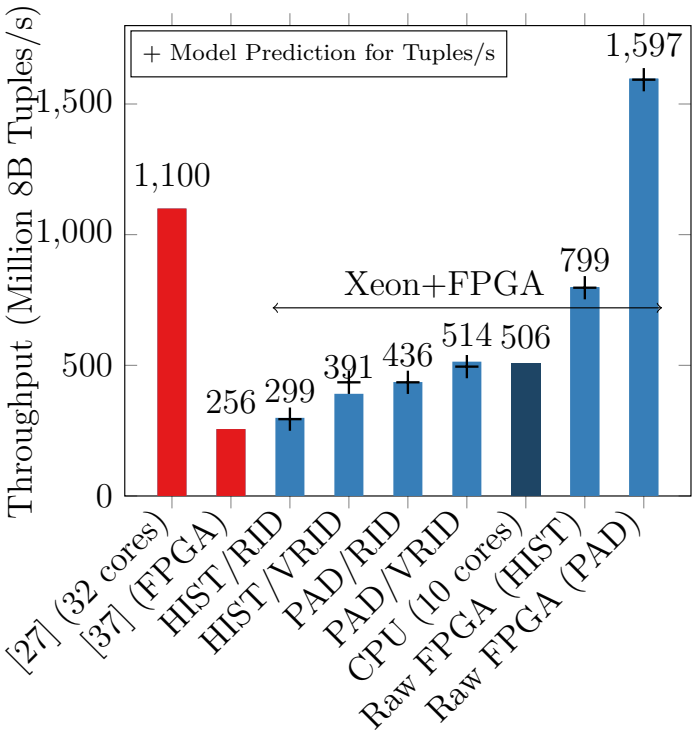



Figure 3.10: Throughput of hardware partitioner for its 4 different configurations. For all the results the number of partitions is 8192, the tuple size is 8B.

- In HIST/VRID and PAD/RID mode read ratio is equal to write ratio $r = 1$:

$$P_{total} = \frac{6.97GB/s}{8B/tuple \cdot 2} = 435\,Mtuples/s$$

- In PAD/VRID mode read ratio is half the write ratio $r = 0.5$:

$$P_{total} = \frac{5.94GB/s}{8B/tuple \cdot 1.5} = 495\,Mtuples/s$$

Comparing the derived values with the measured ones, we can see that the model matches the experiments within 10%. The model does not capture every detail of the implementation for the sake of simplicity, such as triggering the start of the FPGA partitioning by passing the pointers, the flushing of the pipeline or writing a histogram back. For example, the reason why PAD/RID mode is faster than the HIST/VRID mode in experiments is that, in HIST mode, the FPGA partitioner completely flushes the pipeline while building the histogram during the first phase and then the pipeline has to be filled again during

Table 3.9: Workloads used in experiments.

| Name | #Tuples R | #Tuples S | Key Distribution |
|---|---|---|---|
| Workload A | $128 \cdot 10^6$ | $128 \cdot 10^6$ | Linear |
| Workload B | $16 \cdot 2^{20}$ | $256 \cdot 2^{20}$ | Linear |
| Workload C | $128 \cdot 10^6$ | $128 \cdot 10^6$ | Random |
| Workload D | $128 \cdot 10^6$ | $128 \cdot 10^6$ | Grid |
| Workload E | $128 \cdot 10^6$ | $128 \cdot 10^6$ | Reverse Grid |

the second phase which adds to the overall latency. We choose not to further detail the model, as the FPGA partitioner remains bound on the memory access bandwidth.

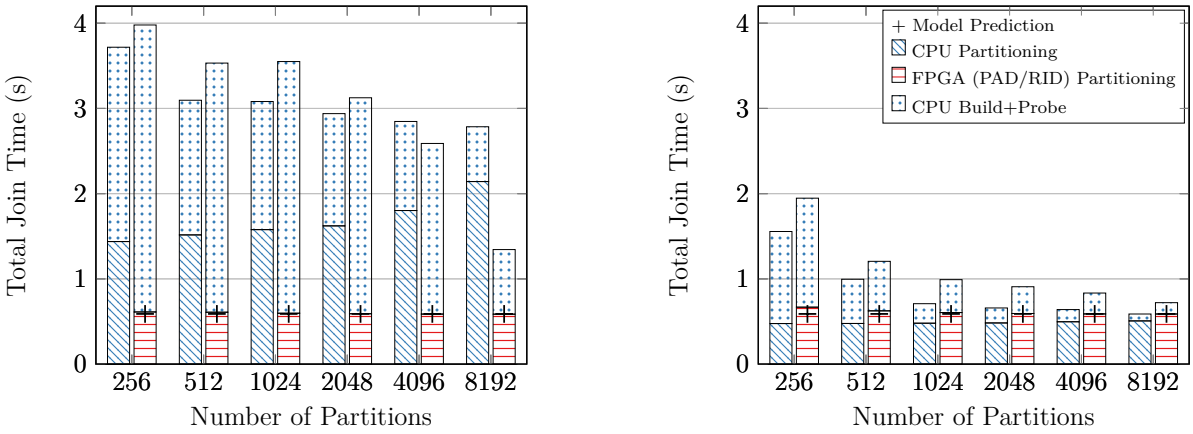
The validated model shows that, if the second term in equation 3.7 ever becomes larger, by providing a high enough bandwidth around 25.6 GB/s to the FPGA, the first term would define the throughput, which will become 1.6 Billion tuples/s. This is 45% faster than the highest absolute partitioning throughput reported by a 64-threaded CPU solution on a 4-socket 32-core machine[PR14]. Now, this improvement is achievable on an FPGA with 200 MHz frequency. If the provided design is hardened as a macro on the CPU die, which can then be clocked in the GHz range, one could expect an even higher throughput performance. Even a better utilization of the design would be to have it integrated in a DRAM chip, which could do near memory processing, similar to what Mirzadeh et al. [MKFG15] discusses for whole joins.

### 3.2.4 Evaluation

We evaluate the proposed partitioner when executed as part of a radix hash join. We use the workloads in Table 3.9 in our experiments, with the key distributions introduced in Section 3.2.2.2. All experiments performed here use 8B tuples. Since the join is bandwidth bound both on the CPU and the FPGA, performing it on wider tuples only results in linear changes for tuples per second, whereas total data processed per second remains the same [BTAÖ13].

#### 3.2.4.1 Different Number of Partitions

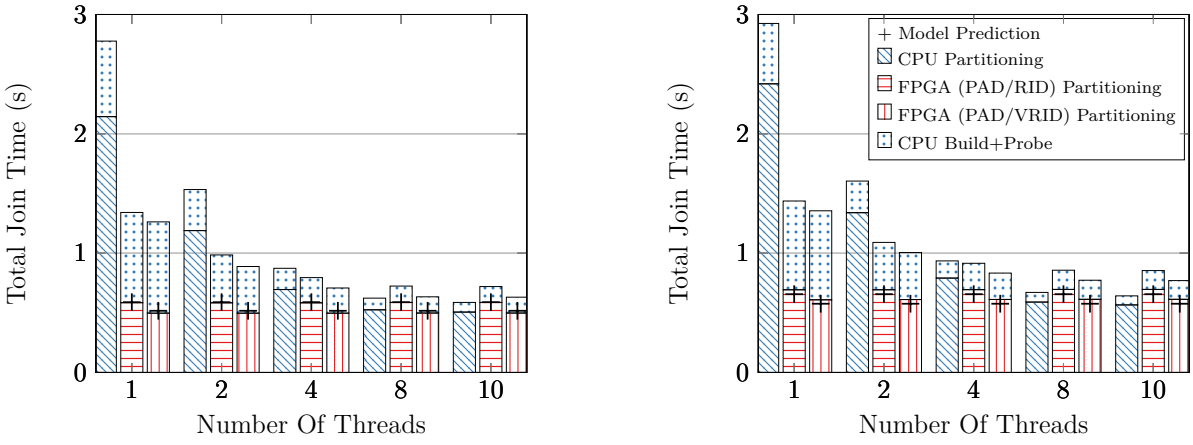
In this experiment we perform the join on workload A with an increasing number of partitions to see how the CPU and FPGA partitioning behave, and how the CPU build+probe




(a) Execution is single threaded.

(b) Execution is 10 threaded.

Figure 3.11: Join performance with increasing number of partitions. Join is performed on workload A.



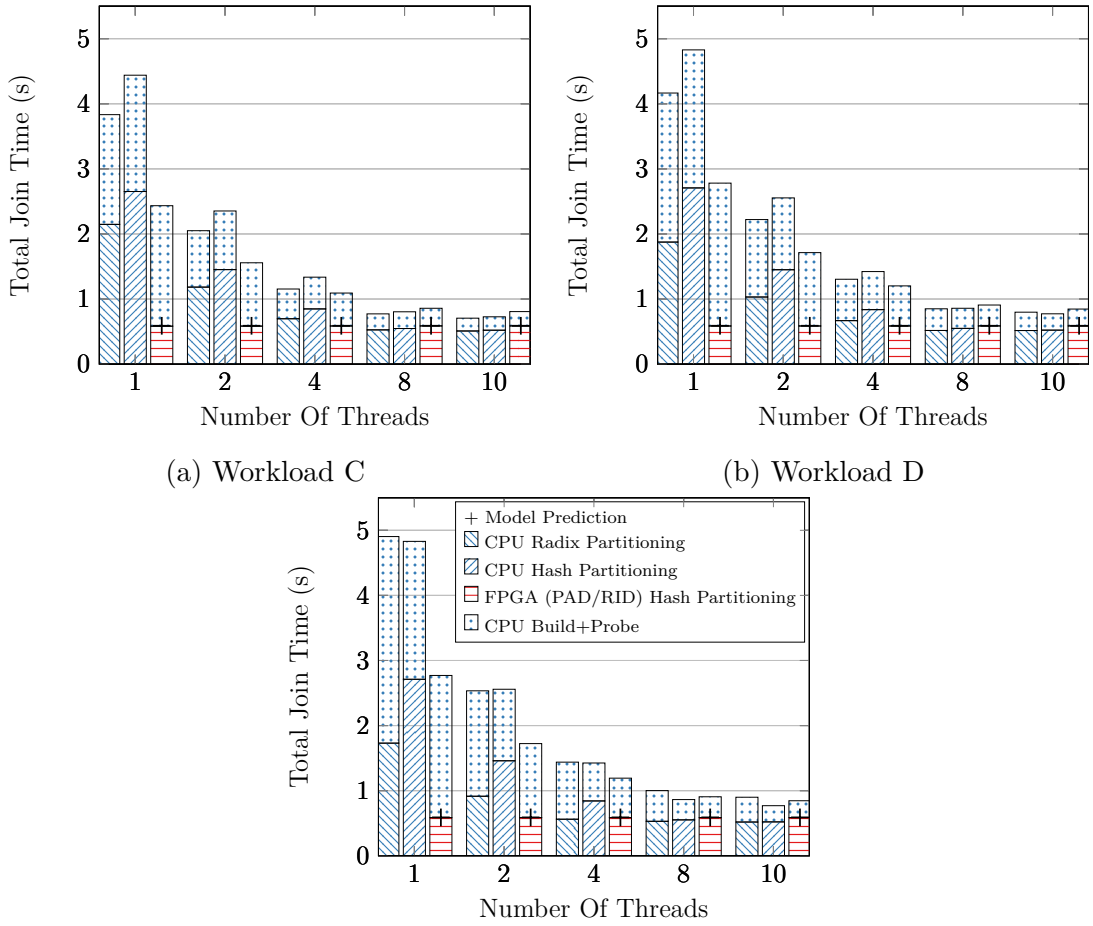

(a) Workload A

(b) Workload B

Figure 3.12: Join performance for increasing number of software threads. Number of partitions is set to 8192. Join is performed on workloads A and B.

phase is affected. Figure 3.11a shows the results for the single threaded join and Figure 3.11b for the 10-threaded. When we say 10-threaded join in the context of hybrid joins, we mean that after the FPGA partitioning the CPU build+probe phase is 10-threaded. For the pure CPU join, both partitioning and build+probe phases are 10-threaded. In this experiment, the partitioner can work in PAD mode, since the workload does not have any skew. Also, we choose RID mode so that it is a direct comparison to the CPU.

The results indicate that as the number of partitions increases, a single threaded CPU




(a) Workload C

(b) Workload D

(c) Workload E

Figure 3.13: Join performance with for increasing number of software threads. Number of partitions is set to 8192. Join is performed on workloads C,D and E after having either radix or hash partitioning.

join spends more time on partitioning. On the other hand, FPGA partitioning delivers the same performance regardless of the number of partitions. Similar to the behavior of the CPU join, the build+probe performance increases for the hybrid join as well with increasing number of partitions. For lower number of partitions, the build+probe takes longer time than the partitioning. This is due to the fact that for large relations as in workload A, a low number of partitions is not enough to split the data into small enough, cache-fitting blocks.

The build+probe performance after FPGA partitioning is always slower compared to being performed after CPU partitioning, although both of them execute the same algorithm.

This happens due to the cache-coherency protocol as explained in Section 3.2.1. For the 10-threaded execution depicted in Figure 3.11b the CPU partitioning becomes slightly faster than the FPGA one. It also appears to be memory bandwidth bound during the partitioning phase, since the performance remains the same across all the number of partitions.

#### 3.2.4.2 Different Relation Sizes and Ratios

In this experiment we fix the number of partitions to 8192, which delivers the best performance for build+probe. We perform the pure CPU join and the hybrid join on workloads A and B, representing joins between similar sized relations and joins between a smaller build relation and a larger probe relation, respectively. We observe the join performance with increasing number of CPU threads in Figures 3.12a and 3.12b. Again, in the context of the hybrid join the number of threads only refers to the build+probe phase coming after the FPGA partitioning. The partitioner can work in PAD mode, since both workloads are without skew. Otherwise, we choose to evaluate both RID and VRID mode to see how they compare against each other and the CPU.

The FPGA partitioner reaches its best performance in VRID mode, since it has to read half the amount of data from the main memory, saving bandwidth. The hybrid join throughput for workload A in this mode is 406 Million tuples/s and the CPU join throughput is 436 Million tuples/s, both reported for 10 threaded execution. In this mode, the FPGA *partitioning* seems to be slightly faster than the 10-threaded CPU one, but the assumption is that the database is a column-store. If the tuples need to be materialized with the payloads later for the query, this will be an additional cost that does not occur in RID mode. However, this is no different than an additional materialization cost that also occurs in column-store database engines.

Both CPU and FPGA partitioning for workload B are slightly slower, since the probe relation is larger than the one in workload A and during writing the tuples to their respective partitions random-access over a wider range of memory is required. The build+probe performance after the FPGA partitioning again is throttled by the cache coherence protocol as explained in Section 3.2.1.

#### 3.2.4.3 Different Key Distributions

Executing the join for relations of different key distributions provides us a way to see in Figure 3.13, whether the difference in robustness of hash or radix partitioning has any effect on the performance of the build+probe phase. We perform radix and hash partitioning on the CPU, and just hash partitioning on the FPGA, since hash calculation on the FPGA comes at no additional cost. For workload C, no benefit for the build+probe phase comes from using hash partitioning, since the input keys are distributed randomly. In that case, radix partitioning delivers a good enough distribution. For workloads D and E however, we can observe a visible improvement of build+probe time, when hash partitioning is used: 11% for workload D and 35% for workload E, both observed from the 10-threaded execution. In contrast, the results also show up to 50% increase in the CPU partitioning time when hash partitioning is used, confirming the performance reduction through the added cost of hash computation. For the 10-threaded execution, the CPU does not seem to suffer from doing hash partitioning, because it is memory bound and has free cycles to compute the complex hash function. However, this is only the case for 10-threaded execution, when the whole CPU is used for this operation, whereas with the FPGA we can get the same robustness for free. This is another example where the advantage of the FPGA implementation shows itself, because even a complex hash function calculation does not slow it down and it deterministically delivers the same performance.

#### 3.2.4.4 Effect of Skew

If one of the relations is skewed following the Zipf distribution law with a factor larger than 0.25, the PAD mode of the FPGA partitioner fails for realistic padding sizes, leading to overflowed partitions. When this happens, the HIST mode must be used to ensure no overflow occurring. The detection time for the failure of the PAD mode is random and depends on the arrival order of the tuples, how they are hashed and the size of the relation. The failure is detected when one of the counters for a partition exceeds the preassigned fixed size. In the worst case, this might happen at the very end of a partitioning run. Then, the procedure has to start from the beginning in HIST mode, which is able handle any Zipf skew factor.

In Figure 3.14 the join performance for 10-threaded execution is given, comparing CPU partitioning and FPGA partitioning. We see that the FPGA partitioner is slower compared to the CPU one, when HIST/RID mode is used. Note that, this is again a limitation of

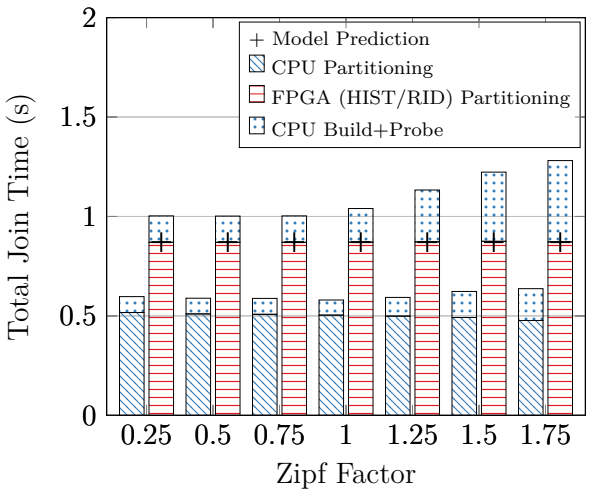



Figure 3.14: Join performance on workload A, when relation S is skewed. Execution is 10-threaded.

the bandwidth that is available to the FPGA. As we have shown in our analytical model, the partitioner throughput can reach 800 Million tuples/s in HIST/RID mode, when not limited by the bandwidth (see Section 3.2.3.7). This means that the FPGA partitioning time in Figure 3.14 would be approximately 0.32 seconds, 1.56x faster than the 10-core Xeon.

### 3.2.5 Related Work

Relational database joins have been the subject of hardware acceleration in many previous works. In the following, all reported throughput numbers are for 8B tuples and similar relation sizes and ratios to provide as fair a comparison as possible. Kaldeway et al. [KLMV12] port hash join algorithms to a GPU exploiting massive parallelism and achieve 480 Million tuples/s throughput. Sukhwani et al. [SMT+12] use an FPGA to accelerate predicate evaluation and decompression to improve the performance of queries utilizing those sub-operators. Werner el al. [WGLP13] provide one of the early works implementing simple join algorithms on an FPGA at very small scales (evaluation is up to 5000 tuples per relation, reaching a throughput of 2.5 Million tuples/s) and the data is assumed to be on the FPGA prior to the join operation. Halstead et al. [HSM+13] implement a non-partitioned hash join on an FPGA and report the simulated throughput for the probe phase, where the FPGA probing outperforms the CPU by an order of magnitude.

As the results reported are based on a simulation, data transfer overheads and integration challenges are not addressed. As a follow-up of this work, Halstead et al. [HANT15] implement a non-partitioned hash join on a Convey-MX architecture with multiple FPGAs and shared global memory. The logic on the FPGAs is designed to hide the memory-latency by sustaining a high utilization of the available memory bandwidth (76 GB/s) through deep-pipelining and having thousands of concurrent hardware threads. The design relies on in-order responses to memory requests and direct support of atomic operations for the FPGA, which is currently only available in the Convey-MX architecture. With 4 FPGAs, the system achieves around 620 Million tuples/s join throughput. Casper et al. [CO14] present a sort-merge based equi-join implemented completely on an FPGA. Based on this work, Chen et al.[CP16] perform a hybrid sort-merge join and accelerate the sorting phase with a bitonic sorter on an FPGA, which is part of a hybrid CPU-FPGA platform designed mainly for embedded low-end applications. Because of the memory bandwidth limitations of the target platform, the resulting design does not outperform previous work in terms of absolute throughput (with 86 Million tuples/s). Ueda et al. [UIO15] focus more on partial reconfiguration, reconfiguring the FPGA at runtime either with a sort-merge join or a hash join pre-compiled circuit depending on relation sizes to provide optimal performance. Jha et al. [JHL+15] and Polychroniou et al. [PRR15] both implement non-partitioned and partitioned hash joins on a many-core (Xeon Phi) architecture. The join throughput reported by Jha et al. [JHL+15] is 450 Million tuples/s with the non-partitioned hash join and by Polychroniou et al. [PRR15] 740 Million tuples/s with the partitioned hash join on the Xeon Phi. Another interesting work for accelerating join processing by Mirzadeh et al. [MKFG15] suggests doing near memory processing, to execute the join without the CPU ever touching the actual data. Although absolute throughput numbers are not reported, it is stated that near memory join processing outperforms CPU-based ones for both hash and sort-based joins, through a highly parallelized design and very high internal bandwidth. There has been also many advancements for CPU-based join implementations in recent years [BLP11, BTAÖ13, BLP+14, LLA+13]. Schuh et al. [SCD16] performed a detailed experimental analysis of previous work and suggested their own improvements for NUMA-aware joins. On a server with 4 CPUs each with 15 cores, they report a throughput of 1800 Million tuples/s.

Data partitioning as an important sub-operator in database engines has been studied in previous work. Polychroniou et al. [PR14] provide an extensive analysis on data partitioning across several dimensions, such as the partitioning type (radix, hash or range)

and the shuffling strategy. It has been shown that for more than 16 partitions, write-combining partitioning with non-temporal writes directly to the memory bypassing the cache performs the best. The reported partitioning throughput is 1.1 Billion tuples/s for 8192 partitions with 64-threaded parallel execution on a 32-core server. Schuhknecht et al. [SKD15] present a set of experiments executing radix partitioning. Known optimizations (write-combining, non-temporal stores etc.) are enabled and novel optimizations (prefetching for writes, micro row layouts) are added step-by-step to observe their effects on the total execution time. The experiments in this study are all single-treaded and the reported throughput is 77 Million tuples/s. Wu et al.[WBKR13] present a range partitioner accelerator designed as an ASIC and the simulated throughput is reported to be 312 Million tuples/s for 511 partitions. The fastest to date FPGA data partitioning implementation is presented by Wang et al.[WHZ15] with 256 Million tuples/s for 8192 partitions. They improved an existing OpenCL implementation of a partitioner and deployed it on an FPGA. The data to be partitioned is assumed to reside in a DRAM connected directly to the FPGA. The resulting partitions are written back to the same DRAM, making a transfer over PCIe to the host memory necessary if the partitioned data is to be used by the CPU for subsequent operations.

For reference, the design we propose in this work achieves for 8192 partitions a raw partitioning throughput of 1597 Million tuples/s. On the Xeon+FPGA platform, where we prototype our design and perform experiments, we measure a maximum end-to-end partitioning throughput of 514 Million tuples/s. Our hybrid join achieves a maximum throughput of 406 Million tuples/s.

### 3.2.6 Discussion

This work's primary focus is on the partitioning sub-operator and its integration within a CPU+FPGA hybrid platform for hash joins, but the proposed techniques and ideas can be applied in a broader setting.

FPGAs excel at processing deep pipelines, vectorized instructions, and computationally intensive workloads. However, their main constraint until now has been the limited integration with the CPU. Therein lies the potential of hybrid architectures. With this work, we show that even a tightly integrated and CPU-optimized algorithm as the radix join can benefit from FPGA co-processing. This opens up the possibility of using FPGA co-processing on a wider range of algorithms, from which compute intensive parts can

be extracted and accelerated. For example, the partitioning we have described can also be used for a hardware conscious group by aggregation [AHB+16] and in other operators involving partitioning [PR14]. Recent literature provides multiple examples for operations which can be offloaded to an FPGA: Sub-operators such as bitonic sorting networks [CP16], histograms [IWA14], hash functions [KA16]; full operators such as pattern matching and regular expressions [ISA16, MTA10]; and eventually new functions typically not supported such as skyline operators [WAT13].

FPGAs are bandwidth bound on most workloads because of their very high internal processing rate and parallelism. Therefore, we currently try to optimize our designs for the limited bandwidth, hence the write combiner in our partitioner. Sequential access (e.g., table scans) and stream processing are something FPGAs are very good at. In the current state where the bandwidth is so scarce, doing random access from the FPGA is not a good option (e.g., using database indexes), unless it is part of the algorithm as it was for the partitioner writes. In this context, the data layout is also important to consider. Column stores can benefit greatly from deep pipelines implementing complex analytics queries. Similarly, when processing compressed columns (a de facto standard for analytical workloads), decompression and compression can be done for free on the FPGA, as we show later in Chapter 5 as the first and the last steps of a processing pipeline. For row stores, the flexible vectorized instructions (e.g., a customized SIMD) can be very beneficial, enabling multiple predicate evaluations per only one row access.

Regarding the integration of FPGA co-processing into a DBMS, depending on the nature of the algorithm there are several possibilities. Some can be integrated as part of the DBMS query processing by invoking the FPGA sub-operators for a more complex relational operator. This is the method used in this work. A similar approach was also explored by He et al. in the context of CPU+GPU co-processing [HZH14]. An alternative method for DBMS integration is to implement relational operators or complex analytics as user defined functions (UDF) executed on the FPGA, similar to doppioDB (Section 2.3) and what Sidler et al. [SIOA17] propose.

As part of the future work, we see two main use cases for the partitioner presented in this work. The first one is a tight integration into a DBMS, following the ideas presented in [SIOA17], for accelerating end-to-end execution time of joins. The second one is to have the FPGA partitioner directly connected to the network to distribute the data across machines using RDMA for the highly scalable distributed joins presented by Barthels et al. in [BLAK15, BMS+17].

# 4 Quantized Dense Linear Machine Learning

FPGAs have become increasingly popular hardware accelerators for machine learning due to the inherent parallelism and the deeply-pipelined computation they offer. Recently, FPGAs have been used to accelerate both training and inference for a range of machine learning models (e.g., generalized linear models [RB16, MCB+12, KBTP15] and deep learning [GAGN15, FLK+11]), using various optimization algorithms (e.g., conjugate gradient [RC10, KDK11] and stochastic gradient descent [MPA+16, cad09]).

One prominent feature of machine learning algorithms, especially those trained with stochastic gradient descent, is that they can tolerate certain types of noise and errors incurred during execution and still return statistically the same answer. This observation has enabled a range of system optimizations such as lock-free [RRWN11] and asynchronous execution [YLL+16]. Recently, one emerging line of research has focused on developing machine learning algorithms that can tolerate a different type of noise—low-precision data representation and/or computation [GAGN15, KZF11, PGTXFN+14, LMG11]. Using low-precision data for training on FPGAs has been considered before, using nearest-value (naive) rounding to reduce data precision [RB16, MLG10, LMG11].

In this project, we focus on the question: *"What is impact of using low precision data for FPGA-based stochastic gradient descent on (1) performance and (2) result quality?"* To address this question, we explore a novel way of reducing the precision of data—stochastic quantization, which has been shown by recent machine learning studies [ZLK+17, DSZOR15, AGL+17] to produce high quality and unbiased results for training dense linear models.

**Machine Learning Scope.** We focus on training one of the simplest class of machine learning models—dense linear models. Despite their simplicity, dense linear models are fundamental for applications such as regression and classification, compressive sensing, and image reconstruction. For applications such as human-in-the-loop analytics and feature selection, one often needs to train hundreds or thousands of models. Thus, the training speed is important.

**Performance Objective.** SGD is an iterative algorithm that performs multiple passes over the data (so called *epochs*). There are two decoupled metrics to assess the performance of SGD: (1) *statistical efficiency*, the number of epochs ($N_{epochs}$) the algorithm needs to converge, and (2) *hardware efficiency*, the time the algorithm requires to execute each epoch ($T_{epoch}$). Our objective is to increase the hardware efficiency, by lowering $T_{epoch}$, and maintain statistical efficiency, by keeping $N_{epochs}$ the same. We show that this is possible on an FPGA when using stochastically quantized data.

**Design Space.** The reason quantized data leads to better hardware efficiency is simple: The FPGA needs to read less volume of data per epoch. However, the parallelism of the design has to be increased to be able to process quantized data at the same rate as full-precision data. Furthermore, stochastic quantization has been shown to maintain statistical efficiency [ZLK+17]. However, the quantization level (precision) needs to be an adjustable parameter, since its effects on statistical efficiency highly depend on the data set characteristics, among other SGD related configurations. Thus, the design space required for complete control over both hardware and statistical efficiency of SGD contains the following parameters: 1) design decisions for the implemented circuit, 2) data precision, and 3) SGD algorithm configuration parameters (learning rate, mini-batch size).

**Contributions.** To explore different points in the design space, we design the following prototypes: We first present an FPGA-based SGD implementation working on 32-bit floating-point data (`floatFSGD`). Apart from being scalable (handling high dimensionality) and resource-efficient, `floatFSGD`'s performance is on par with a 10-core CPU, despite being bound on the available memory bandwidth on our current platform. Then, we increase `floatFSGD`'s internal parallelism, so that it can process quantized data—`qFSGD` is up to 11× faster than `floatFSGD` and up to 10.6× faster than the fastest 10-threaded CPU version of SGD we have access to. Furthermore, we reveal a new trade-off space that helps us to determine the most efficient way to do SGD on an FPGA:

(I) We explore different circuit designs for `qFSGD` to achieve better scalability in order to process lower precision data. We show that with 1-bit quantization linear scaling does not

work, leading to a compute bound design.

(II) As we vary the precision and the quantization strategies (stochastic quantization vs. naive rounding), we find that:

1. The lower the precision, the better the hardware efficiency, because of higher bandwidth utilization (less volume of data being read).
2. The lower the precision, the worse the statistical efficiency, because of increased variance in the data.
3. The optimal precision regarding statistical and hardware efficiency depends on the data set, especially its number of features.
4. Stochastic quantization leads to an unbiased convergence of SGD, compared to biased naive rounding, while the latter does not require any preprocessing of input data.

(III) We experiment with various datasets and algorithmic configurations to explore the training quality vs. performance trade-off. Among other aspects, we look into the number of quantized samples needed, before `qFSGD` can be used. Our key conclusion is that the FPGA-based SGD on stochastically quantized data is a very efficient and well performing way of training dense linear machine learning models.

## 4.1 Background

### 4.1.1 Stochastic Gradient Descent (SGD)

We consider the following problem: Given a dataset $(a_i)_{i=1,N}$ of $D$-dimensional data points, each with its own label $(b_i)_{i=1,N}$, we wish to identify the dense linear model $\mathbf{x}$ which minimizes the classification loss over this dataset:

$$\arg\min_x Q(x) = \frac{1}{N}\sum_{i=1}^{N} \text{loss}_i(\mathbf{x}) = \frac{1}{N}\sum_{i=1}^{N} \frac{(\mathbf{a_i}\mathbf{x} - b_i)^2}{2} \tag{4.1}$$

- $\mathbf{a_i} \in \mathbb{R}^{1\times D}$, a single sample of data set $\mathbf{a} \in \mathbb{R}^{N\times D}$
- $b_i \in \mathbb{R}$, the corresponding true inference value to $\mathbf{a_i}$
- $\mathbf{x} \in \mathbb{R}^{D\times 1}$, the model to be trained and used for inference

- $N \in \mathbb{N}$, the number of samples
- $D \in \mathbb{N}$, the number of features per sample

A standard tool for solving this problem is stochastic gradient descent (SGD), consisting of the iterative process in Algorithm 9. For a small enough step size $\gamma$, SGD will converge to the *optimal* solution [Bot10].

**Data:** dataset $a_1, a_2, \ldots, a_N$ of $D$-dimensional data
**Result:** optimal value of the model $\mathbf{x}$
Initially, $\mathbf{x}$ is zero;
**while** *not converged* **do**
  **for** *i from* 1 *to* $N$ **do**
    $\mathbf{g_i} = \frac{\partial \text{loss}_i(\mathbf{x})}{\partial \mathbf{x}} = (\mathbf{a_i} \cdot \mathbf{x} - b_i)\mathbf{a_i}$ ;
    $\mathbf{x} \leftarrow \mathbf{x} - \gamma \mathbf{g_i}$ ;
  **end**
**end**

**Algorithm 5:** Stochastic Gradient Descent

### 4.1.2 Stochastic Rounding (Quantization)

Recent work by Zhang et al. [ZLK+17] shows that SGD convergence can be guaranteed even if the data undergoes a compression process called *stochastic quantization* before it is used in the gradient update. We now provide a brief explanation of this procedure. Assume that the data consist of floating-point values contained in an interval $[L, U]$. We *quantize* each data point $a_{i,j}$ to one of $s$ levels, as follows. First, we split the interval $[L, U]$ into $s-1$ intervals of equal length $\Delta = (U-L)/(s-1)$. Then, each data point is rounded stochastically to one of the endpoints of its interval:

$$Q_s^{L,U}(a_{i,j}) = \begin{cases} (\left\lfloor \frac{a_{i,j}}{\Delta} \right\rfloor + 1)\Delta & \text{with prob. } a_{i,j} - \left\lfloor \frac{a_{i,j}}{\Delta} \right\rfloor \Delta \\ \left\lfloor \frac{a_{i,j}}{\Delta} \right\rfloor \Delta & \text{otherwise.} \end{cases} \tag{4.2}$$

Example: Quantize value 0.7 between [0,1] with 2 levels. $\Delta = 0.\bar{3}$

$$Q_2^{0,1}(0.7) = \begin{cases} 1 & \text{with prob. } 0.7 \\ 0 & \text{with prob. } 0.3 \end{cases}$$

This quantization procedure is chosen so that the *expected* quantized value returned equals the value itself, that is:

$$\mathbf{E}[Q(a_{i,j})] = a_{i,j}. \tag{4.3}$$

In other words, if we iterate the quantization procedure on a sample, the average of the returned values would converge to the value of the sample. The key observation by Zhang et al. [ZLK+17] is that SGD still converges even if samples are quantized in this way. However, to preserve correctness, we must take two independent quantizations $Q'$ and $Q''$ for each sample $a_i$, and update the gradient value to:

$$\hat{\mathbf{g}}_i = (Q'(\mathbf{a_i})^T\mathbf{x} - b_i)Q''(\mathbf{a_i}) \tag{4.4}$$

This choice of update ensures that $\mathbf{E}[\hat{\mathbf{g}}_i] = \mathbf{g}_i.$, i.e., the update is an *unbiased estimator* of the true gradient, which in turn ensures convergence of SGD.

## 4.2 Implementation

The target platform for this project is the first version of Intel Xeon+FPGA (Section 2.2.1). The designs we present in this work are not dependent on the cache-coherency features of the Xeon+FPGA: the processing is done in a streaming fashion, where memory bandwidth, not memory latency, determines performance So, the designs can be integrated into other FPGA platforms (e.g., attached to the network or to storage) with ease.

### 4.2.1 FPGA-SGD on float data (`floatFSGD`)

We first present an SGD implementation that works on 32-bit floating-point data (Figure 4.1), a common data representation in machine learning. As the data is accessed in a 64B cache-lines, the circuit is designed to work on this data width. It is able to consume a cache-line at every clock cycle (200 MHz), resulting in the processing rate of 12.8 GB/s.

**Scale to # of features:** The challenging part of the design is to make it capable of handling a number of features $D$ that is larger than 16, which is the default width of the pipeline. This is possible since all vector algebra in Algorithm 9 can be performed iteratively, where each portion contains 16 values. To stay cache-line aligned, we use zero-padding if $D \bmod 16 \neq 0$. Thus, we can calculate how many cache-lines it takes for $\mathbf{a_i}$ (one row in the set) to be completely received:

$$\#\mathbf{a_i} \text{ cache-lines} = \begin{cases} D/16 & \text{if } D \bmod 16 = 0 \\ \frac{D+(16-D \bmod 16)}{16} & \text{if } D \bmod 16 \neq 0 \end{cases}$$

The only parameter determining the scalability of `floatFSGD` is the maximum dimensionality $D_{max}$, because it determines the amount of BRAM needed for storing the model $\mathbf{x}$. We choose $D_{max}$ to be 8192, which is more than enough for most existing linear dense model training examples. The design can handle any number of samples, $N$, since training is done in a streaming fashion.

**Walk-through of computation pipeline:** In the following, we explain each stage of the computation pipeline. The first stage is the dot product $\mathbf{a_i} \cdot \mathbf{x} = \sum_{j=1}^{D} a_{i,j} x_j$. When a cache-line containing a part of vector $\mathbf{a_i}$ arrives, it is first multiplied (❶) with the corresponding part of vector $\mathbf{x}$, in floating-point with multiplier IPs,[1] which have 5-cycle latency and throughput of one result per cycle. Then, the values are converted to a 32-bit integer (❷) by multiplication with a large constant that is configurable during runtime, depending on the value range desired. This *float2fixed* conversion takes 1 cycle. After that, an adder tree (❸) accumulates 16 values, each layer taking 1 cycle. At the output of the adder tree is an accumulator (❹). It accumulates the results coming out of the adder tree for the pre-calculated number $\#\mathbf{a_i}$ cache-lines, building the final value for the dot product. The dot product result is converted back to floating-point (❺), since the next stages of the calculation will be performed with floating-point data. In the next part, the rest of the gradient calculation takes place. First, the scalar value $b$ (the true inference value in the data set), is subtracted from the dot product (❻) using a floating-point adder IP, which has 7-cycle latency and throughput of one result per cycle. The $b$ value is received some cycles before the subtraction takes place and is placed into a FIFO (Ⓑ), waiting there until the dot product result is ready. After the subtraction, a floating-point multiplication takes place with step size $\gamma$ (❼), which can be configured to any *float* value. At the end of this step, we have a scalar value $\gamma(\mathbf{a_i} \cdot \mathbf{x} - b_i)$, which needs to be multiplied with vector $\mathbf{a_i}$.

[1] All floating point IPs are created via Altera Quartus II 13.1.

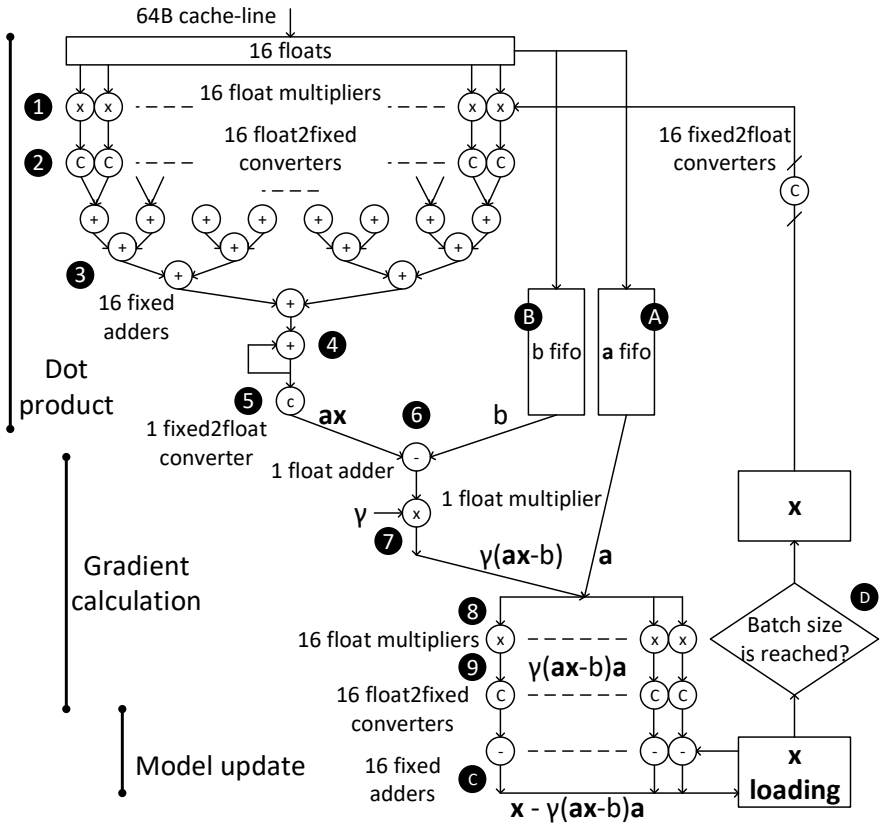



Figure 4.1: Computation pipeline for `floatFSGD`, latency: 36 cycles, data width: 64B, processing rate: 64B/cycle.

At this stage, a FIFO (Ⓐ) already contains all parts of vector $\mathbf{a_i}$, because the incoming cache-lines are written to this FIFO simultaneously as they were sent to the dot product calculation. The scalar-vector multiplication (❽) takes place in floating-point, where all parts of $\mathbf{a_i}$ are multiplied with the same scalar value. This gives the gradient $\mathbf{g_i}$ one part at a time, which undergoes *float2fixed* conversion (❾), so that a cycle-by-cycle update of the model $\mathbf{x}$ (Ⓒ) can take place. This would not be possible with a floating-point adder having 7-cycle latency, since the result of the current calculation is needed in the next cycle. The gradient is applied to the corresponding part of the model as it becomes available. After the last part of the gradient is subtracted from the model, the update for $\mathbf{a_i}$ is completed. After all rows go through the same calculation, one epoch is completed (Algorithm 9).

**Staleness vs. batch size (as a result of pipelined execution):** The model updated and the model read for the dot product are separate (Figure 4.1). Only when a certain batch size (the number of already processed $\mathbf{a_i}$) is reached, the updated model is carried

on to the actual model (❶). The reason is the latency introduced by the computation pipeline: in theory, the whole gradient calculation and the update to the model as in Algorithm 9 should be an atomic operation. However, to exploit deep-pipelining, we don't perform this operation atomically. Instead, we keep the actual model and the updated model separate and carry out the accumulated update only when a certain batch size is reached (called a mini-batch SGD). The batch size is a configurable parameter, which should be set to the latency of the pipeline (36 cycles) to avoid any so-called stale updates.

**End-to-end float vs. hybrid computation:** We choose a hybrid (*float+fixed*) over end-to-end *float* computation, because a 7-cycle floating-point addition latency leads to: (1) A high latency adder tree (❸) that imposes a larger batch-size to avoid staleness, slowing down the convergence rate, (2) not being able to do a cycle-by-cycle accumulation (❼), since the result of an ongoing addition is required in the next cycle. Thus, to keep the processing rate at 64B/cycle, we choose a hybrid design that eliminates both these disadvantages.

### 4.2.2 FPGA-SGD on quantized data (`qFSGD`)

We explain how we change the `floatFSGD` design to work on quantized data. The main purpose is simple: Instead of reading *float* data (only 16 values in a cache-line), we quantize the data beforehand, so that more than 16 values fit into a cache-line, thus reading less volume of data in total. There is one main challenge in making the FPGA-SGD work on quantized data: scaling out the `floatFSGD` pipeline so that it can work on more than 16 values in parallel. Before we explain how this is achieved, we first review the quantization options we consider and how the data layout looks like.

**Quantization for `qFSGD`:** Equation (4.2) shows that, given a non-integer value, the quantization still might produce a non-integer value. However, floating-point arithmetic induced by non-integer values are hard to implement on the FPGA and scaling out such a design would be difficult. We take advantage of the fact that we can select the quantization variables $[L, U]$ and $s$ aptly, so that only integer values are produced. Table 4.1 shows our choices for these values.

After selecting a quantization precision (one of *Q1*, *Q2*, *Q4* or *Q8*; powers of two to stay cache-line aligned), the data set (the values in matrix $\mathbf{a}$) must be normalized to the selected quantization's corresponding $[L, U]$. At this stage, the sign of the data set is considered for the normalization: we do not normalize a negative data set into a positive interval

Table 4.1: Choice of quantization levels, lower and upper bounds, so that only integer values are produced.

| Levels | Data set positive | Data set negative | Needed bits |
|---|---|---|---|
| s=2 | $[L, U] = [0, 1]$ | N/A | 1, *Q1* |
| s=3 | $[L, U] = [0, 2]$ | $[L, U] = [-1, 1]$ | 2, *Q2* |
| s=9 | $[L, U] = [0, 8]$ | $[L, U] = [-4, 4]$ | 4, *Q4* |
| s=129 | $[L, U] = [0, 128]$ | $[L, U] = [-64, 64]$ | 8, *Q8* |

in order to keep existing zeros (maintain sparsity). Thus, we do not use *Q1* for negative datasets.

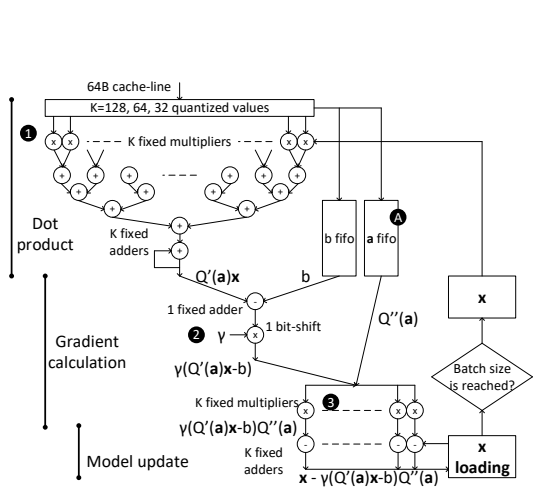
**The layout of quantized data:** As we showed in Section 4.1.2, to calculate the correct gradient, we need 2 quantization samples of the same data point. That is, if we, for example, select *Q8*, a quantized sample has 8 bits and we need 2 of them to calculate the gradient; the actual amount of bits we use is 16. That's why, when we perform quantization on a data set, we always create 2 samples and store them in memory next to each other. Thus, we can calculate how many quantized values can fit into one cache-line, a value we call $K$, in Table 4.2. The value $K$ dictates the amount of zero-padding we need to perform, in order to be cache-line aligned, similar to as it did for `floatFSGD`. Thus, the number of cache-lines required to receive one quantized row $Q(\mathbf{a_i})$ can be calculated:

$$\#\mathbf{a_i} \text{ cache-lines}(K) = \begin{cases} D/K & \text{if } D \bmod K = 0 \\ \frac{D+(K-D \bmod K)}{K} & \text{if } D \bmod K \neq 0 \end{cases} \tag{4.5}$$

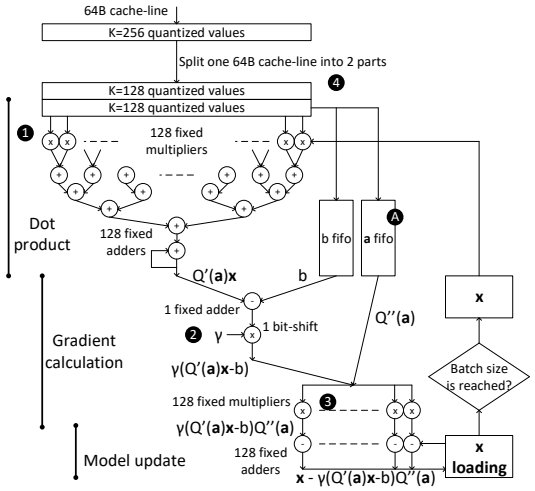
Table 4.2: Number of received values in a single cache-line.

| **Data type** | *Q1* | *Q2* | *Q4* | *Q8* | *float* |
|---|---|---|---|---|---|
| **# of values in a cache-line,** $K$ | 256 | 128 | 64 | 32 | 16 |
| **Processing rate (GB/S),** $PR$ | 6.4 | 12.8 | 12.8 | 12.8 | 12.8 |

**When and where does quantization happen?:** Unfortunately, there is no way to perform stochastic quantization on the fly and gain the same performance benefits on the target platform. Some naive ideas prove to be useless in this regard: (1) the 10-core CPU can't create samples at the rate of FPGA's memory bandwidth, (2) naively rounding the data once, and then creating stochastically quantized samples on the FPGA is not possible,




(a) *Q2*, *Q4* and *Q8* `qFSGD`,latency: $log(K)$+5 cycles, data width: 64B, processing rate: 64B/cycle.

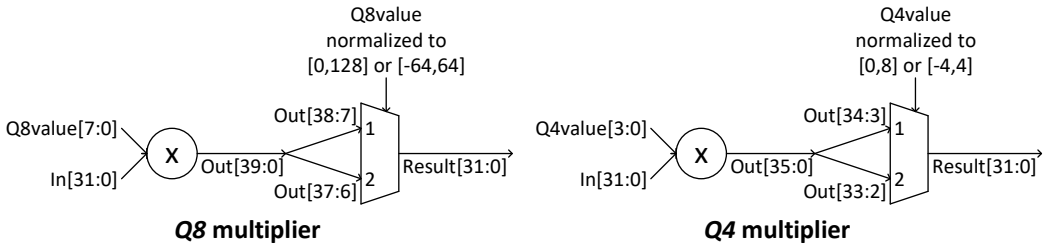



(b) *Q1* `qFSGD`, latency: 12 cycles, data width: 32B, processing rate: 32B/cycle.

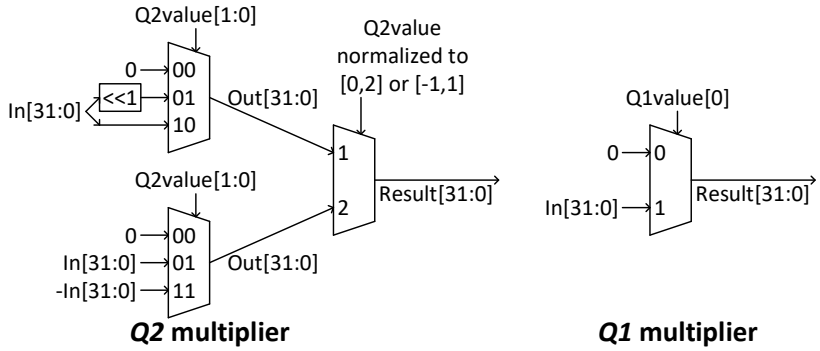
Figure 4.2: Computation pipelines for all quantizations. Although for *Q2*, *Q4* and *Q8*, the pipeline width scales out and maintains 64B width, for *Q1* it does not scale out and the pipeline width needs to be halved, making *Q1* `qFSGD` compute bound.

since quantization depends on the full-precision value itself. Thus, in the current system, the quantization happens as a pre-calculation step on the CPU, before running `qFSGD`. To achieve perfect statistical soundness, we have to create as many quantized samples (so called indexes) of the same data set as the number of epochs. However, creating a separate index for each epoch might be undesirable, because of space or time overheads (exact memory space needed: # of indexes $\times$ $N_{CL}$, from Equation 4.6). We consider the possibility of reusing indexes and its effect on statistical efficiency in Section 4.3.

**Computation pipeline for quantized data (Figure 4.2):** The selection of $Qx$, which determines the width of the pipeline is a generic parameter that can be set before synthesis. Thus, each $Qx$ results in a different bitstream. The explanation here only focuses on the differences from `floatFSGD` and how the pipeline is scaled out. The first thing to note is that $Qx$ pipelines work only on integer data, so there is no need for converters. This is because the arriving quantized data $Q(\mathbf{a_i})$ is already in integer form, as explained previously, and the inference values $b$ are also converted to integer by multiplication with a large constant. Another difference here is that for a given value in vector $\mathbf{a_i}$, 2 quantized samples arrive due to the double sampling method. The first sample is given to the dot




(a) Signed integer multiplication for *Q8* and *Q4* data, implemented with DSPs.




(b) Signed integer multiplication for *Q2* and *Q1* data, implemented with combinational logic.

Figure 4.3: Multiplication implementations depending on the quantization type.

product calculation (❶) and the second sample is put into a FIFO (Ⓐ), where it is kept until the dot product result is ready, as depicted in Figure 4.2. The last difference is applying the step size $\gamma$ (❷), which is actually a division. Since we are dealing with integer data here, we choose to apply $\gamma$ as a bit-shift operation. By how many bits the value is shifted to the right is a runtime configurable parameter, allowing adjustments according to the data set characteristics.

**Scaling out for quantized data and trade-offs:** Scaling out the pipeline for *Q8* and *Q4* is straightforward using conventional signed multipliers (❸) implemented by DSP resources, followed by a bit-shift, to keep the data width at 32-bit (Figure 4.3a). However, for *Q2* and *Q1*, we can do a more efficient multiplication using multiplexers (Figure 4.3b), since one of the multiplicands is only 2-bit and 1-bit, respectively. Doing this efficient multiplication allows the *Q2* pipeline to scale to 128-value parallelism, which would have otherwise required a 100% usage of the available DSP resources on the target FPGA (see Table 4.3). However, the pipeline shown in Figure 4.2a does not scale to 256-value parallelism (we can't meet timing with the target frequency of 200 MHz), even though the

Table 4.3: Resource consumption for computation pipelines.

| Data type | Logic (ALMs) | DSP | BRAM (bits) |
|---|---|---|---|
| *float* | 38% (89194) | 12% (33) | 7% (3.471K) |
| *Q8* | 35% (82152) | 25% (64) | 6% (3.145K) |
| *Q4* | 36% (84500) | 50% (128) | 6% (3.145K) |
| *Q2*, *Q1* | 43% (100930) | 1% (2) | 6% (3.145K) |

*Q1* multiplier is just one multiplexer. The main issue here is (1) the bus for propagating the model from the BRAM to compute units becomes too wide (8192 signals), (2) the adder tree becomes too wide and deep. Note that, we still have to perform addition in full-precision, because we can't simplify the addition as we have done with the multiplication. Using compressor trees [KZ14] instead of standard adder trees also did not help in meeting timing. For this reason, we decided to halve the `qFSGD` pipeline to process *Q1* data (Figure 4.2b). To do so, we split an arriving cache-line into 2 parts (❹), which are processed sequentially by the pipeline shown in Figure 4.2b. The processing rate of this pipeline is 32B/cycle, or 6.4 GB/s, which is slightly less than the memory bandwidth available in our platform.

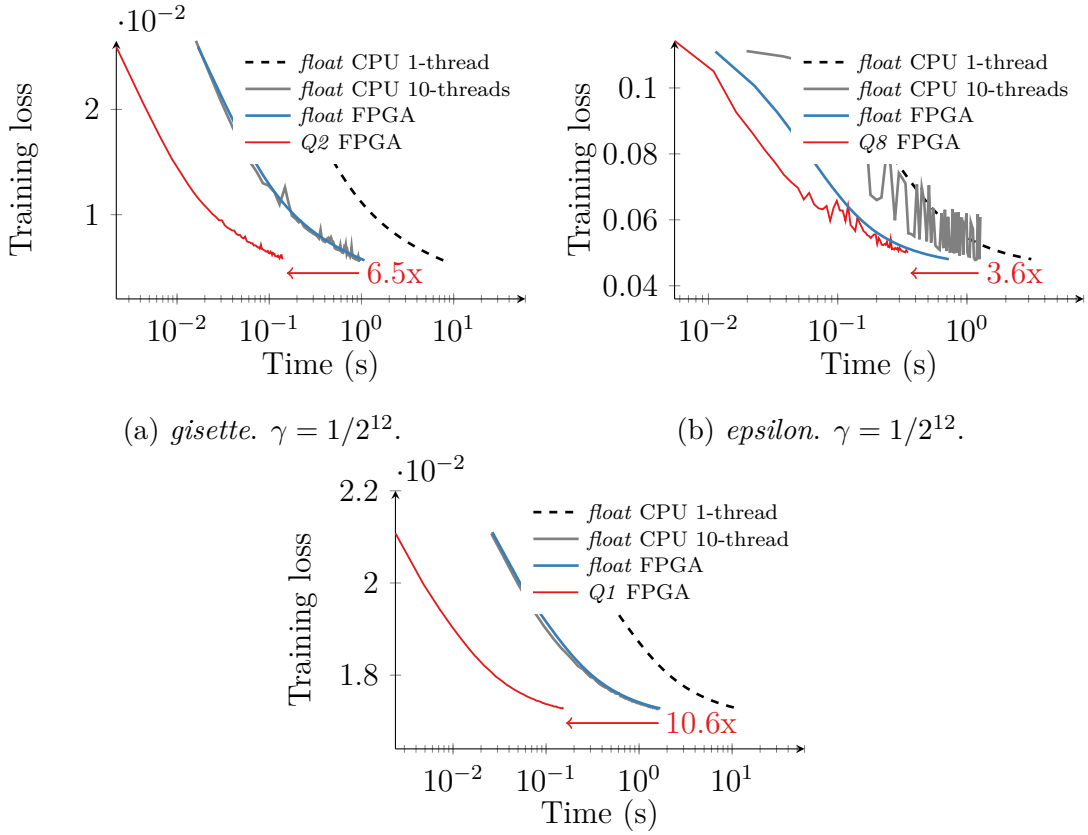
**Model for predicting the speedup with quantized data (hardware efficiency):** We can now create a simple model to predict if using quantization will provide any speedup, depending on the dimensionality of the data set, the selected precision, and memory bandwidth. The total number of cache-lines `floatFSGD` or `qFSGD` needs to read is:

$$\#\text{ of cache-lines } N_{CL}(K) = N \cdot \#\mathbf{a_i} \text{ cache-lines}(K) \tag{4.6}$$

The time for each SGD epoch with the processing rate of the circuit ($PR$) and available platform bandwidth $B$ is thus:

$$T_{epoch} = \begin{cases} N_{CL}(K)/B & \text{if } PR > B \\ N_{CL}(K)/PR & \text{else} \end{cases} \tag{4.7}$$

The model shows that as long as quantization leads to a reduction of the data size, $T_{epoch}$ can be reduced. An exception to this occurs between *Q1* and *Q2*, when the bandwidth available is larger than 12.8 GB/s. There, *Q1* is not faster than *Q2* (both are compute bound): even when the data size is halved with *Q1*, *Q2* can process twice the data twice as fast. However, *Q1* can still be interesting in platforms with higher memory




(a) *gisette*. $\gamma = 1/2^{12}$.

(b) *epsilon*. $\gamma = 1/2^{12}$.

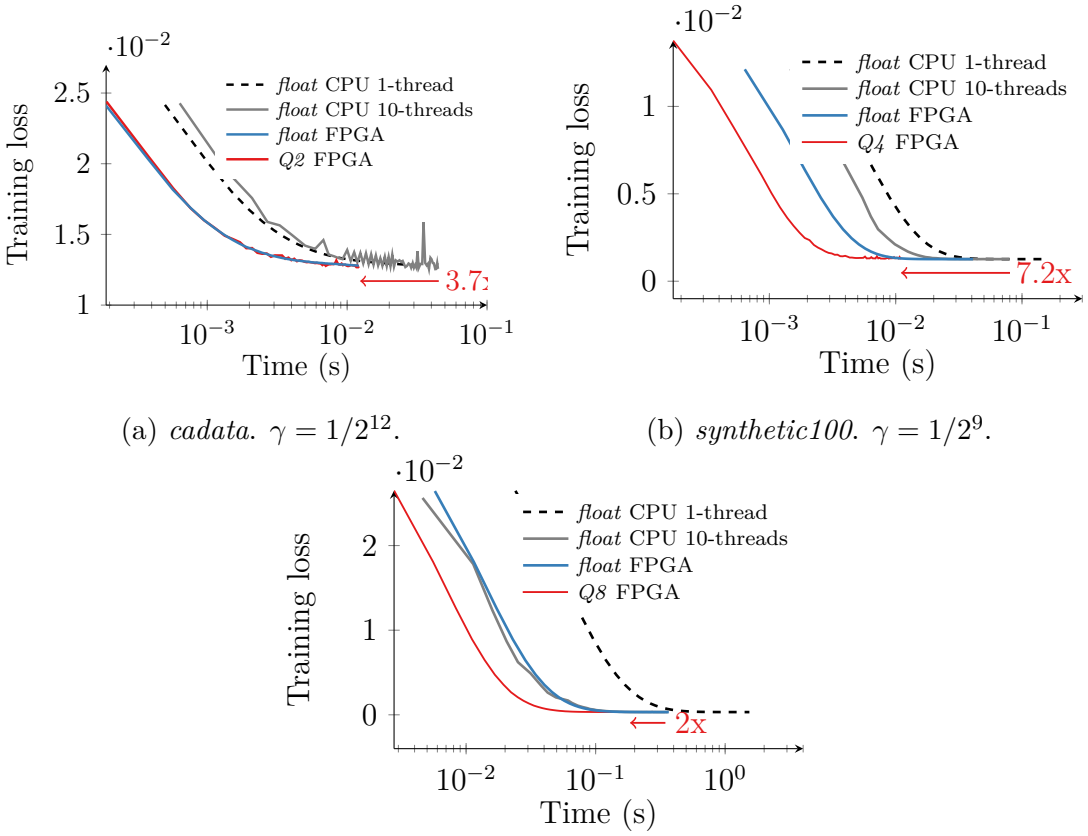
(c) *mnist*, trained for digit 7. $\gamma = 1/2^{15}$.

Figure 4.4: SGD on classification data. All curves represent 64 SGD epochs. Speedup shown for *Qx* FPGA vs. *float* CPU 10-threads.

bandwidth, because one can put multiple `qFSGD` instances, multiplying the bandwidth requirements, making *Q1* still more attractive compared to *Q2*. As for the decision between *Q2*, *Q4*, *Q8*, and *float*, their processing rates are the same, so lower-precision quantization always leads to speedup regardless of bandwidth, unless cache-aligned zero padding causes $\#\mathbf{a_i}$ cache-lines($K$) to be the same.

## 4.3 Experimental Evaluation

The main hypothesis that we would like to experimentally validate is: (1) low-precision data representation created via stochastic quantization can be used for training dense linear models while maintaining quality, and (2) since the processor doing the training needs to read less data per epoch, using quantized data provides speedup. To validate this, we run




(a) *cadata*. $\gamma = 1/2^{12}$.

(b) *synthetic100*. $\gamma = 1/2^{9}$.

(c) *synthetic1000*. $\gamma = 1/2^{9}$.

Figure 4.5: SGD on regression data. All curves represent 64 SGD epochs. Speedup shown for *Qx* FPGA vs. *float* CPU 10-threads.

our FPGA-SGD on various datasets having different characteristics (see Table 4.4). As the CPU baseline, we use both a single-threaded SGD doing exactly the same calculation as `floatFSGD` and a high performance parallel library called "Hogwild!"[RRWN11] working on *float* data, with a mini-batch size of 36 (equivalent to `floatFSGD`). The multi-thread parallelism in Hogwild! is achieved through asynchronous updates: each thread works on a separate portion of the data and applies gradient updates to a common model without any synchronization. The asynchrony might reduce the statistical efficiency, especially if the data set is dense. Both CPU baselines make use of vectorized instructions and are compiled with GCC 4.8.4, with -O3 enabled.

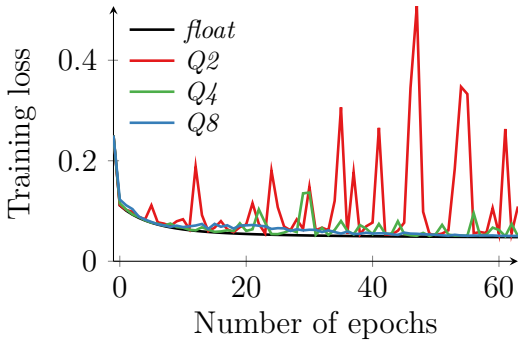
**Methodology:** Since we apply the step size as a bit-shift operation, we choose one of the following step sizes, which results in the smallest loss for the full-precision data after 64 epochs: $(1/2^{6}, 1/2^{9}, 1/2^{12}, 1/2^{15})$. With a given constant step size, we run FPGA-SGD on all the precision variations that we have implemented (*Q1*-only for classification data-, *Q2*,

Table 4.4: Datasets used in experimental evaluation.

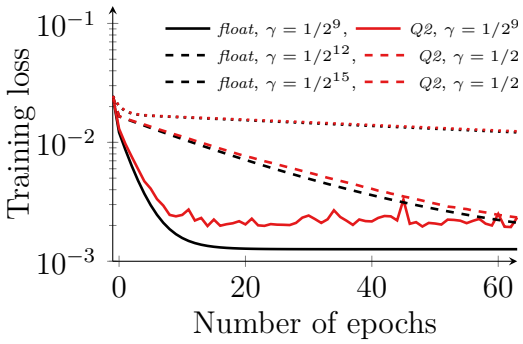
| Name | Training size | Testing size | # Features | # Classes |
|---|---|---|---|---|
| cadata | 20,640 | | 8 | regression |
| music | 463,715 | | 90 | regression |
| synthetic100 | 10,000 | | 100 | regression |
| synthetic1000 | 10,000 | | 1000 | regression |
| mnist | 60,000 | 10,000 | 780 | 10 |
| gisette | 6000 | 1000 | 5000 | 2 |
| epsilon | 10,000 | 10,000 | 2000 | 2 |

*Q4*, *Q8*, *float*). For each data set, we present the loss function over time in Figures 4.4 and 4.5, showing 4 curves: single-threaded and a 10-threaded CPU-SGD for *float*, `floatFSGD`, and `qFSGD` for the **smallest** precision data that **has converged within 1% of the original loss**. We would like to show the difference in time for all implementations to converge to the same loss, emphasizing the speedup we can achieve with `qFSGD` compared to full-precision variants.

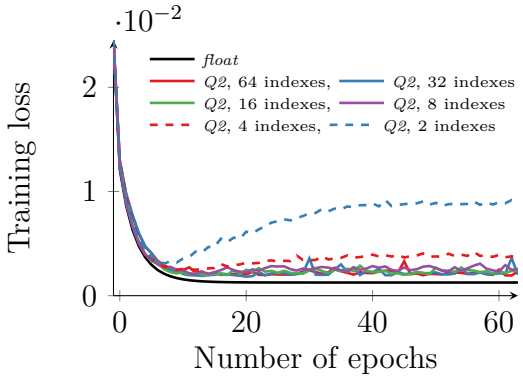
**Main results:** In Figure 4.4 we observe that for all classification datasets, `qFSGD` achieves a speedup while maintaining convergence quality. For *gisette* in Figure 4.4a, *Q2* reaches the same loss 6.9x faster than Hogwild!. Due to the high variance data in *epsilon*, both Hogwild! and *Qx* curves seem to be unstable (Figure 4.4b). We can see that `floatFSGD` in this case behaves well, providing both 1.8x speedup over Hogwild! and better convergence quality. The results for the *music* data set are very similar to *epsilon*, so we omit them for space reasons. On the *mnist* data set (Figure 4.4c), we can even use *Q1* without losing any convergence quality, showing that the characteristics of the data set heavily effect the choice of quantization precision, which justifies having multiple *Qx* implementations. For *mnist*, *Q1* `qFSGD` provides 10.6x speedup over Hogwild! and 11x speedup over `floatFSGD`. In Figure 4.5a, `floatFSGD` converges as fast as *Q2* for *cadata*, because they read the same number of cache-lines. The reason is the cache aligned zero padding (set $D = 8$ and respective $K$s for *Q2* and *float* into Equation (4.5)). For *synthetic100* and *synthetic1000*, we need to use *Q4* and *Q8*, respectively, to achieve the same convergence quality as *float* within the same number of epochs. In Figure 4.5c, Hogwild! convergence is slightly faster than that of `floatFSGD`, and *Q8* provides 2x speedup over that. Hogwild! becomes faster with higher dimensionality (compare Figures 4.5b and 4.5c), because with lower



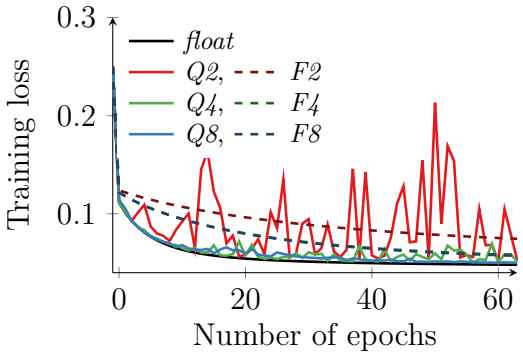

(a) Effect of different precision quantization on *epsilon*. $\gamma = 1/2^{12}$.




(b) Effect of decreasing the step size on *Q2* with *synthetic100*.




(c) Effect of reusing indexes with *synthetic100*.




(d) Naive rounding vs. stochastic quantization

Figure 4.6: SGD on various data sets, showing the effects of data precision, step size $\gamma$, index reuse, and naive rounding.

dimensionality cache pollution occurs more frequently [RRWN11].

**The outcome of the main results:** We can converge to the same loss using quantized data within the same number of epochs as with full-precision data; thereby achieving better hardware efficiency while maintaining statistical efficiency. To achieve this, the precision has to be selected carefully (which we did empirically). Predicting the ideal precision for a given data set is out of the scope of this work.

### 4.3.1 Effects of quantized SGD parameters:

We now study the effects of various parameters on the convergence quality. We have performed the same experiments on all our datasets and observed similar results. Here we present a subset (due to space constraints) of our experiments.

**Precision** Previously, we observed that, for some datasets, at least *Q8* precision is needed to be within 1% of the original loss given the same number of epochs. Figure 4.6a shows how the convergence curves look like, if we insist on using lower precision data on *epsilon*. The inherently higher variance of *epsilon* is amplified by having quantized data, and so statistical efficiency drops. This can be fixed by applying a smaller step size, as we discuss next.

**Step Size** A higher step size causes higher variance during quantized SGD, since the error introduced by quantization has a greater effect during each gradient update. In the first part of the experimental section, we chose step sizes optimized for *float* precision data. This is why very low precision data does not converge to the original loss for some datasets: the variance caused by *float*-optimized step size is too high. In Figure 4.6b, SGD for all datasets can converge even with *Q2* data if the step size is chosen to be small enough. The downside of a smaller step size is the slower convergence rate.

**Dimensionality** Higher dimensional data causes higher variance in `qFSGD`, since the summed-up quantization error for each data sample is greater. We see this effect when comparing Figures 4.5b and 4.5c: while for *synthetic100*, *Q4* provides high-quality convergence, for *synthetic1000*, we need at least *Q8*.

**Reusing indexes** In all previous experiments, the number of indexes for an SGD run is chosen to be equivalent to the number of epochs for keeping statistical soundness, as explained in Section 4.2.2. Here, we discuss the possibility of reusing indexes, and how it affects SGD convergence quality. Theoretically, reusing indexes of quantized data introduces bias to the gradient, since not every quantized sample is statistically independent. Figure 4.6c shows the effects of reusing indexes, thereby causing samples not to be statistically independent. We empirically conclude that using more than 16 indexes is enough to get the same convergence quality, but using fewer than 16 causes the convergence curve to be biased.

**Naive Rounding vs. Quantization** Figure 4.6d shows the difference between having stochastically quantized data ($Qx$) vs. naively rounded (to the nearest integer) data ($Fx$). If the data is naively rounded, the original optimization problem changes; the global minimum the algorithm converges to is a different one than the original one, showing itself as a bias in the convergence curve.

Table 4.5: Multi-class classification on *mnist*. Run times are for training 10 models with 100 iterations and $\gamma = 1/2^{15}$.

| Precision | Accuracy for 10 digits | Training time (s) |
|---|---|---|
| *float* CPU-SGD | 85.82% | 19.0354s |
| *Q1* `qFSGD` | 85.87% | 2.4083s |

### 4.3.2 Classification accuracy:

We discuss the accuracy for *mnist*, for which 10 separate models (for each digit) are trained. This can be parallelized with the CPU very well since each model can be trained completely independently. The CPU reaches a processing rate of 10 GB/s for this test, the highest we observe for the CPU. On the FPGA, we need to train 10 models one after the other, since there is only one SGD instance. In Table 4.5, *Q1* achieves 8x speedup against the highest performing CPU implementation we have, while maintaining the same multi-classification accuracy.

## 4.4 Related Work

FPGAs have been extensively used for accelerating and prototyping machine learning algorithms. Most of the existing work focuses on classification with a pre-trained model [LMG11, KBTP15, MSK+11] since classification is thought to be the more frequent operation, often with real-time constraints. As the demand for machine learning applications and data sizes keep growing, accelerating training also becomes highly important. Contrary to common belief, training is an operation that has to be run many times, often as a human-in-the-loop process, until an optimal model is produced.

There is a large body of work considering FPGA as the target processor for linear model and neural network training. Mahajan et al. [MPA+16] present Tabla, a framework for automatically generating SGD solvers with different loss functions. The architecture of the generated circuit resembles a MIMD processor with static scheduling, making the approach general purpose. The highest dimensional data set tested is *mnist*. No absolute performance numbers are presented. Kesler et al. [KDK11] focus on accelerating linear algebra operations and present simulation results of a CPU-integrated accelerator architecture. A maximum matrix size of 150 is presented. Roldao et al. [RC10] consider accelerating a

conjugate gradient algorithm, which takes symmetric matrices of a maximum size 58 as input. Cadambi et al. [cad09] consider using low precision (naive rounding) arithmetic, achieving a maximum of 256-value parallelism with 4-bit precision, for support vector machine (SVM, also a linear classifier) training. Their FPGA accelerator is implemented as a coprocessor for accelerating the multiply-accumulate part of the algorithm. Bin Rabieah et al. [RB16] implement SVM training with custom precision data (4-bit and 8-bit, via naive rounding). The parallelism on the FPGA is exploited by partitioning the data, and then training multiple models with each partition and, at the end, aggregating the models, similarly to the asynchronous update method used for achieving parallelism for SGD on a CPU. Majumdar et al. [MCB+12] present a framework—MAPLE, not necessarily designed for FPGAs, but using an FPGA as a prototyping platform. Its architecture is based on a many core with large on-chip memory, enabling highly parallel computation. The FPGA-SGD we presented achieves the highest dimensional scalability among existing work, considering reported test datasets and workloads.

To our knowledge, the only efforts considering stochastic quantization as a viable data representation for deep neural network training and combining this with an FPGA accelerator are presented by Gupta et al. [GAGN15] and Courbariaux et al. [CBD14]. Gupta et al. [GAGN15] use the FPGA as a matrix multiplication accelerator working on quantized data. Courbariaux et al. [CBD14] focus on reducing multiplication precision on FPGAs, not necessarily on stochastically quantized data. In contrast to these, our study focuses on the detailed analysis of statistical vs. hardware efficiency trade-offs when using stochastically quantized data for dense linear model training on FPGAs.

## 4.5 Discussion

In this project, we presented highly scalable and configurable FPGA-based stochastic gradient descent implementations for generalized linear model training using full-precision and quantized data. Our key takeaways are: (1) The FPGA-based consumption of low-precision data via stochastic quantization leads to significant performance improvements during training, while the convergence quality can be maintained. (2) Despite the presented advantages, there is a multi-variate trade-off space consisting of precision vs. end-to-end runtime, convergence quality, design complexity of the implementation etc. We performed an extensive empirical analysis showing how the trade-off space can be navigated. (3) Nevertheless, the trade-offs necessitate an automated decision making process

to select the optimal quantization depending on various factors such as the properties of the dataset and the target hardware, motivating further research in the area.

One limitation of our presented FPGA-based implementations is that each design can work on a particular data precision. Therefore, with the current designs it is not possible to change the precision of the consumed data at runtime. A follow-up work to this project by Wang et al. [W+19] presents a design using bit-serial multiplication can allow changing the precision used at runtime without any overhead, leading to a more flexible FPGA-based quantized SGD solution.

# 5 Column-Store Suitable Machine Learning

Integrating advanced analytics such as machine learning into a database management system and taking advantage of hardware accelerators for data processing are two important directions in which databases are evolving. Online analytical processing engines (OLAP) are the natural place to implement these advanced capabilities, being the type of data processing system with rich functionality and the ability to deal with large amounts of data. Yet, the problem of efficiently integrating ML with OLAP is not trivial, especially if the functionality of the latter needs to be preserved. In this project, we take a step forward in understanding the question: *How can we integrate ML into a column-store DBMS without disrupting either DBMS efficiency or ML quality and performance?*

This work is built upon previous research in the following three areas.

**In-DBMS Machine Learning.** Integrating machine learning into DBMS is an ongoing effort in both academia and industry. Current prominent systems include MADlib [HRS+12], SimSQL [CVP+13], SAP HANA PAL [FML+12] and various products from Oracle [TBC+05], Impala [K+15], and LogicBlox [AtCG+15]. The combination of ML and DBMS is attractive because businesses have massive amounts of data residing in their existing DBMS. Furthermore, relational operators can be used to preprocess and denormalize a complex schema conveniently before executing ML tasks [KNP15].

**Column-Store for Analytical Workloads.** Column-stores are the standard solution for OLAP workloads [IGN+12, LBH+15, APM16]. When combined with main memory processing and techniques like compression [ABH+13], they become highly efficient in

processing large amounts of data. However, it is not clear whether the column-store format is suitable for most ML algorithms, notably stochastic gradient descent (SGD), which access all attributes of a tuple at the same time; processing tuples by row. In fact, existing in-DBMS ML systems tend to work on row stores [HRS+12, CVP+13, AtCG+15].

**New Hardware for Data Processing.** Emerging new hardware—FPGA, GPU—has the potential to improve processing efficiency in a DBMS through specialization of commonly used sub-operators such as hashing [KA16] (Section 3.1), partitioning [KGA17] (Section 3.2), sorting [SJ17] or advanced analytics [MKS+18]. Specialized hardware is also becoming available in data centers: Microsoft uses FPGAs [PCC+14] to accelerate Bing queries [Put14] and neural network inference [CFO+18]; Baidu uses FPGA instances in the cloud to process SQL; Intel develops hybrid CPU+FPGA platforms enabling tight integration of specialized hardware next to the CPU [OSC+11]. In most of these systems, efficient I/O is required to move data to the accelerator; otherwise the performance advantages disappear behind the cost of data movement. Column-stores are a better fit to achieve this purpose as column oriented processing makes it easier to move data from memory while still being able to exploit the available parallelism in the hardware—especially when the columns are compressed, thereby increasing I/O efficiency.

In this work, we focus on a specific problem: *Given a column-store database, how can we efficiently support training generalized linear models (GLMs)?* We provide a solution by first choosing an established ML training algorithm that natively accesses data column-wise: stochastic coordinate descent (SCD) [SST11]. This algorithm, however, is not efficient due to lack of cache locality and the complexity of model management during the learning process. Therefore, we apply a partitioned SCD algorithm, inspired by recent work of Jaggi et al. [JST+14] to improve cache-locality and performance. Both the standard and the partitioned SCD work on raw data values. Yet, data in column-stores is usually compressed (often twice, through dictionary compression and run-length or delta encoding) and also encrypted [ABE+13]. When adding the necessary steps to deal with compression and encryption, we observe that the overall performance on a CPU suffers. To overcome this, we use an FPGA to do on-the-fly data transformation. We choose an FPGA as accelerator because its architectural flexibility enables us to put vastly different processing tasks, such as data transformation (decompression, decryption) and ML training (SCD) into a dataflow pipeline, resulting in high throughput for all of them.

**Contributions.** We employ a partitioned stochastic coordinate descent algorithm (pSCD) to achieve cache-efficient training natively on column-stores, leading to higher performance

both on a multicore CPU and an FPGA. Next, we analyze the performance of pSCD against standard SCD and SGD on both row and column-stores by performing an in-depth experimental evaluation on a wide range of datasets, showing the effectiveness of pSCD for real-world ML tasks. To handle on-the-fly data transformation, as a first step, we provide a detailed implementation of an FPGA-based training engine able to perform SCD and pSCD. We analyze memory access and computational complexity for both algorithms. We validate our design on a CPU+FPGA platform, showing that it saturates the memory bandwidth available to the FPGA. Finally, we extend the FPGA-based training with data transformation steps. We utilize those steps to show that performing delta-encoding decompression and/or AES-256 decryption on-the-fly comes at practically no performance reduction for the SCD/pSCD engine on the FPGA, whereas on the CPU it leads to significant decrease in the training throughput.

## 5.1 Background

In this section, we explain the advantages of stochastic coordinate descent (SCD) against the popularly implemented stochastic gradient descent (SGD) algorithm when the underlying data is organized as a column-store and provide the background on SCD.

**Problem Definition.** We aim at solving optimization problems of the form:

$$\min_{\mathbf{x}\in\mathbb{R}^n} \left( \frac{1}{m} \sum_{i=1}^{m} J(\langle \mathbf{x}, \mathbf{a}_i \rangle, b_i) \right) + \lambda \|\mathbf{x}\|_1 \tag{5.1}$$

$$J = \begin{cases} \frac{1}{2}(\langle \mathbf{x}, \mathbf{a}_i \rangle - b_i)^2 & \text{for Lasso} \\ -b_i \log(h_{\mathbf{x}}(\mathbf{a}_i)) - (1 - b_i) \log(1 - h_{\mathbf{x}}(\mathbf{a}_i)) & \text{for Logreg} \end{cases} \tag{5.2}$$

$\bullet\ h_{\mathbf{x}}(\mathbf{a}_i) = 1/(1 + \exp(-\langle \mathbf{x}, \mathbf{a}_i \rangle))$ is the sigmoid function.

where $(\mathbf{a}_1, b_1), ..., (\mathbf{a}_m, b_m) \in ([-1, 1]^n \times [0, 1])$ is a set of samples and $J : \mathbb{R}^n \times \mathbb{R} \to [0, \infty)$ is a non-negative convex loss function. $\lambda$ is the regularization parameter adjusting the strength of regularization, that tries to prevent overfitting to the training data by limiting how much the norm of the model $\mathbf{x}$ can grow. We use $\ell_1$ regularization, turning the optimization problem to Lasso [Tib96] for quadratic loss and logistic regression (Logreg) for cross-entropy loss, as shown in Equation (6.2). For denoting a column, we use the notation $\mathbf{a}_{:,j} \in [-1, 1]^m$, indicating the $j$th column/feature of the sample matrix $[\mathbf{a}_1; \mathbf{a}_2; ...; \mathbf{a}_m]$, where each $\mathbf{a}_i$ is a row.

### 5.1.1 SGD on Column-Stores

For all gradient-based optimization methods, the knowledge of the inner product $\langle \mathbf{x}, \mathbf{a}_i \rangle$ is required to compute a gradient estimate. Any gradient-descent algorithm, e.g., most notably SGD, computes this inner product at each iteration of the algorithm. This implies that a sample $\mathbf{a}_i$ has to be accessed *completely* at each iteration requiring all features in $\mathbf{a}_i \in \mathbb{R}^n$ to be read in a row-wise fashion.

It is still possible to perform SGD on a column-store, but to make reading from distant addresses in the memory efficient, one has to read *in batches* due to row-buffer locality of DRAMs [KZTK15]: Reading multiple cachelines from one column, storing them in the cache, reading from the next column, and so on. This is practically an on-the-fly column-store to row-store conversion that is disadvantageous: It requires cache space proportional to the number of features in the data set and a large batch size in order to read as sequentially as possible [ZR14]. We show the disadvantage of working on a column-store for SGD empirically in Section 4.

Coordinate-descent based algorithms eliminate this problem by enabling a way of accessing the samples *one feature at a time*, which natively corresponds to column-wise access.

### 5.1.2 Stochastic Coordinate Descent

Shalev-Shwartz et al. [SST11] introduce SCD and provide a bound on the runtime for convergence. The core idea in SCD is to maintain a vector $\mathbf{z} \in \mathbb{R}^m, z_i = \langle \mathbf{x}, \mathbf{a}_i \rangle$ containing the results of the inner-products between the model and the samples. It is then possible to always apply the change that was done to one coordinate of the model ($x_j$) also to the inner-product vector $\mathbf{z}$, as shown in Algorithm 10. This maintains up-to-date inner products between all samples and the current model in vector $\mathbf{z}$. As a result, an update to the model requires accessing only one coordinate—one feature—for all samples, which equals to a column-wise access pattern.

We perform SCD by randomly (without replacement) selecting a feature at each iteration. As we show in Algorithm 10, the model $\mathbf{x}$ is initialized to 0 and therefore, the inner-product vector $\mathbf{z}$ also starts at 0. An epoch corresponds to processing the entire data set: Each column is accessed completely to compute the partial gradient either for Lasso or Logreg. Then, the corresponding coordinate of the model is updated with the partial gradient. Finally, an inner-product vector update takes place to keep the inner products

**Initialize:**
- $\mathbf{x} = 0$, $\mathbf{z} = 0$, step size $\alpha$
- $S(\mathbf{z}) = \begin{cases} \mathbf{z} & \text{for Lasso} \\ 1/(1+\exp(-\mathbf{z})) & \text{for Logreg} \end{cases}$
- $T(x_j, g_j) = \begin{cases} \alpha g_j + \alpha\lambda & x_j - \alpha g_j > \alpha\lambda \\ \alpha g_j - \alpha\lambda & x_j - \alpha g_j < -\alpha\lambda \\ x_j & \text{else (to set } x_j = 0) \end{cases}$

**for** *epoch = 1, 2, ...* **do**
  — randomly without replacement
  **for** $j$ = *shuffle(1, ..., n)* **do**
    $g_j = \frac{1}{m}(S(\mathbf{z}) - \mathbf{b}) \cdot \mathbf{a}_{:,j}$ — partial gradient computation
    $\mu = T(x_j, g_j)$ — thresholding due to regularization
    $x_j = x_j - \mu$ — coordinate update
    $\mathbf{z} = \mathbf{z} - \mu\mathbf{a}_{:,j}$ — inner-product vector update
  **end**
**end**

**Algorithm 6:** Stochastic Coordinate Descent

in $\mathbf{z}$ up-to-date, the last step in Algorithm 10. All steps of the algorithm access samples column-wise. We perform multiple epochs until convergence for the optimization problem is observed, by evaluating the loss function.

### 5.1.3 System Overview

Our implementation is based on doppioDB [SIO+17], as introduced previously in Section 2.3. The high level diagram in Figure 5.1 shows how we perform training and inference using user defined functions (UDF). A table name and hyperparameters are given as arguments to a UDF, in which we implement our CPU and FPGA based algorithms. The model produced by the training is stored as a database internal data structure in doppioDB—similar to a database index. Inference can be performed also with UDFs if a model to a corresponding table has been trained beforehand. MonetDB uses compression by default only on strings, so we implement numeric compression and also encryption as part of the UDFs for test purposes.

**Evaluation Setup.** Our target platform is the second generation Intel Xeon+FPGA, as in Section 2.2.1, which we use for all experiments. We use *gcc 5.4* and compile the binaries

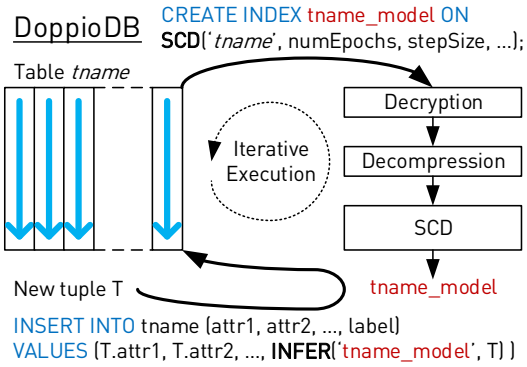



Figure 5.1: System overview showing the training and inference procedures in doppioDB.

with "*-O3 -march=native*". We set the frequency governor of the CPU to *performance*, enabling it to run at 3.2 GHz peak frequency during program execution.

## 5.2 Cache-Conscious SCD

Although column-wise access of SCD suits column-store DBMS well, it has one drawback compared to SGD: The intermediate state that needs to be kept is the inner-product vector $\mathbf{z} \in \mathbb{R}^m$ which may be much larger than the model $\mathbf{x} \in \mathbb{R}^n$ itself, since the number of samples is usually much larger than the number of features ($m \gg n$). We use a partitioned version of SCD (pSCD), inspired by the CoCoA algorithm introduced by Jaggi et al. [JST+14]. CoCoA aims at reducing the communication frequency when performing distributed dual coordinate ascent. Our main goal with pSCD is to reduce the amount of intermediate state that is kept. As a result, first, the memory access complexity can be reduced because of cache-locality, and second, the algorithm becomes trivially parallelizable, with only infrequent need for synchronization.

**Method.** The difference in pSCD is mainly that it processes all features in a partition, before moving onto the next one, thus requiring to keep only a partition-sized portion of the inner-product vector $\mathbf{z}$ and label vector $\mathbf{b}$, as depicted in Figure 5.2. However, when we split the gradient updates this way, the partial gradient that we compute is valid only for the current partition. Therefore, we can't apply the coordinate update step to a global model $\mathbf{x}$; we have to keep a separate model $\mathbf{x}[k]$ for each partition and apply updates locally, as in Algorithm 7. This is equivalent to training $K$ separate models when a dataset

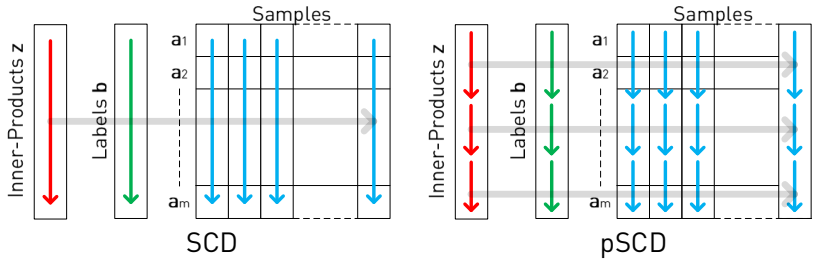



Figure 5.2: A simplified representation of the data access patterns of SCD and pSCD. The crucial advantage of pSCD is that while processing the samples, only partition-sized portions of the inner-product and label vectors need to be accessed.

is divided into $K$ subsets. To achieve a common model, we perform model averaging and update the entire inner-product vector every $P$ epochs, the so-called global inner-product update (last part in Algorithm 7). Notice that all steps in the algorithm still access data column-wise, even though a column is not scanned in its entirety as in SCD, but in large partitions/subsets.

**Convergence Rate.** Jaggi et al. [JST+14] perform a theoretical analysis on the convergence rate of partitioned coordinate methods. They show that the convergence rate of partitioned methods is equal to non-partitioned block-coordinate descent methods [RT16], if the individual optimization problem for each partition is solved to optimality. Another way of analyzing the convergence properties of pSCD is from an update staleness perspective, which suits having the frequency of model aggregation as a tuning parameter: The convergence rate is affected mainly because we are introducing staleness to the algorithm by updating $K$ models independently, then aggregating those every $P$ epochs. The update scheme is similar to *stale synchronous parallel* [HCC+13, ZC17]. Using a similar approach, we assume $K$ (given by the number of partitions) independent workers and $S = nP$ as our deterministic staleness value, where $n$ is the number of features and $P$ is the global inner-product update period. Ho et al. [HCC+13] show that the noisy model state due to staleness is at most $KS = KnP$ updates away from the reference state and this factor is proportional to the convergence rate. In conclusion, the convergence rate of pSCD is closer to standard SCD for smaller $K$, $n$, or $P$. In practice, we can adjust the staleness by selecting a smaller global inner-product update period $P$, thus resulting in a better convergence rate, as we show in the next section.

**Runtime overhead.** Performing a global inner-product update ((2) in Algorithm 7) has

**Initialize:**
· $\mathbf{x}[K] = 0$, $\mathbf{z} = 0$, step size $\alpha$
· $S(\mathbf{z})$ and $T(x_j, g_j)$ as in Algorithm 10
· partition size $M$, number of partitions $K = m/M$
· inner-product update period $P$
**for** *epoch = 1, 2, ...* **do**
  **for** *k = 0, ..., K-1 (each partition)* **do**
    — randomly without replacement
    **for** *j = shuffle(1, ..., n)* **do**
      $subset = kM + 1, ..., kM + M$
      — partial gradient computation
      $g_j = (S(\mathbf{z}_{subset}) - \mathbf{b}_{subset}) \cdot \mathbf{a}_{subset,j}$
      — thresholding due to regularization
      $\mu = T(x[k]_j, g_j)$
      — coordinate update
      $x[k]_j = x[k]_j - \mu$
      — inner-product vector update
      $\mathbf{z}_{subset} = \mathbf{z}_{subset} - \mu \mathbf{a}_{subset,j}$
    **end**
  **end**
  — global inner-product update with the averaged model
  **if** *epoch* mod $P$ **then**
    $\bar{\mathbf{x}} = (\mathbf{x}[0] + ... + \mathbf{x}[K-1])/K$
    $\mathbf{z} = 0$
    **for** *k = 0, ..., K-1 (each partition)* **do**
      $subset = kM + 1, ..., kM + M$
      **for** *j= 1, ..., n* **do**
        $\mathbf{z}_{subset} = \mathbf{z}_{subset} + \bar{x}_j \mathbf{a}_{subset,j}$
      **end**
    **end**
  **end**

**end**

(1)

(2)

**Algorithm 7:** Partitioned SCD

the same memory access cost as the main part ((1) in Algorithm 7). If the main part (1) has a runtime of $T_{main}$, the runtime for one pSCD epoch can be approximated by $T_{epoch} = T_{main} \times (1 + 1/P)$. If a relatively high $P$ value delivers good convergence, the overhead by part (2) becomes minimal, because (2) needs to be executed infrequently.

Table 5.1: Datasets used in the evaluation.

| Name | # Samples | # Features | Size | Type |
|---|---|---|---|---|
| IM | 332,800 | 2,048 | 2,726 MB | classification |
| AEA | 32,769 | 126 | 16,5 MB | classification |
| KDD1 | 391,088 | 2,399 | 2,188 MB | classification |
| KDD2 | 131,329 | 2,330 | 1,224 MB | classification |
| SYN1 | 33,554,432 | 16 | 2,147 MB | regression |
| SYN2 | 2,097,152 | 256 | 2,147 MB | regression |

**Limitation.** One limitation of pSCD in a DBMS is that it assumes the input data is *shuffled*—when the data is ordered, the convergence of the pSCD approach could be slower. In our CPU and FPGA based implementations, we support both pSCD and standard SCD, the latter is scan order resilient. The user may use both depending on the assumptions on the underlying data. The study of scan order for stochastic first order methods is a challenging problem open for decades. Recent theoretical studies on this topic [GOP15, Sha16, RR12] do not completely eliminate data shuffling.

**Evaluation.** To evaluate the overall efficiency of pSCD, we need to compare both its convergence rate and data processing rate to those of standard SCD. Compared to standard SCD, the convergence rate, termed *statistical efficiency*, is expected to be less for pSCD due to its staleness. However, the data processing rate, termed *hardware efficiency*, is expected to be higher for pSCD due to its cache-locality. In the following sections, our goal is to show that, for pSCD, the advantage offered by hardware efficiency is larger than the disadvantage in its statistical efficiency. Thus, pSCD can lead to an overall shorter training time.

### 5.2.1 Statistical Efficiency

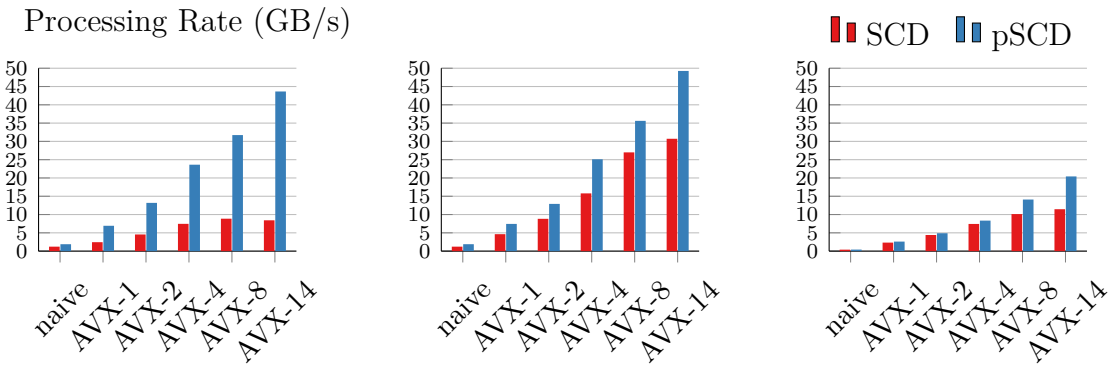
In this section, we evaluate the statistical efficiency—convergence behavior—of pSCD against standard SCD, using datasets from real-world use cases (Table 6.4). We are interested in comparing the convergence rate and the resulting model quality when using either SCD or pSCD. Due to the large staleness introduced by pSCD, our expectation is to observe a deviation in the final loss achieved after $N$ epochs when comparing the two algorithms. Our goal here is to show that, in practice, the deviation is very limited and can further be adjusted by tuning the global inner-product update period $P$.

Table 5.2: Training quality results comparing SCD and pSCD, with varying inner-product update period $P$. For pSCD results, we use red for worse and green for better results compared to SCD.

| Configuration | ValidationMetric | SCD | pSCD $P$=$\infty$ | pSCD $P$=100 | pSCD $P$=10 |
|---|---|---|---|---|---|
| **Data set: IM**<br>Epoch = 200<br>Train Size = 266k<br>Test Size = 66k<br>Partition Size = 16k | **Log Loss** | 0.10154 | 0.10575 +4.15% | 0.10380 +2.23% | 0.10191 +0.36% |
| | **Test Accuracy** | 96.17% | 96.071% -0.10% | 96.109% -0.06% | 96.196% +0.03% |
| **Data set: AEA**<br>Epochs = 5k<br>Train Size = 32k<br>Test Size = 59k<br>Partition Size = 16k | **Log Loss** | 0.13531 | 0.25927 +91.61% | 0.14972 +10.65% | 0.13947 +3.07% |
| | **Test AUC** | 0.91029 | 0.86880 -4.56% | 0.90891 -0.15% | 0.91013 -0.02% |
| **Data set: KDD1**<br>Epochs = 1k<br>Train Size = 391k<br>Test Size = 45k<br>Partition Size = 16k | **Log Loss** | 0.24672 | 0.24712 +0.16% | 0.24701 +0.12% | 0.24698 +0.11% |
| | **Test AUC** | 0.62430 | 0.62369 -0.10% | 0.62274 -0.25% | 0.62226 -0.33% |
| **Data set: KDD2**<br>Epochs = 1k<br>Train Size = 131k<br>Test Size = 45k<br>Partition Size = 16k | **Log Loss** | 0.32285 | 0.32294 +0.03% | 0.32286 +0.003% | 0.32285 0% |
| | **Test AUC** | 0.61145 | 0.61144 -0.002% | 0.61145 0% | 0.61139 0.01% |

**Datasets.** We select a variety of datasets which have relational properties from *Kaggle* competitions, using MLBench from Liu et al. [LZZ+18] as a guideline. We also create a data set of our own to have a large-scale real-world classification task: We run *InceptionV3* [SVI+16] neural network on ImageNet (ILSVRC 2012 contest) to extract 2048 features from images, resulting in the IM data set.

- **IM**: Half the samples contain features extracted from cat images (classes 281, 282, 283, 284) and the other half from dog images (classes 153, 230, 235, 238), for classifying cats and dogs.
- **AEA**: Winning features [aea] for Amazon employee resource access/denial prediction competition.
- **KDD1 and KDD2**: Winning features [kdd] for the KDD Cup 2014 competition, predicting excitement about projects.



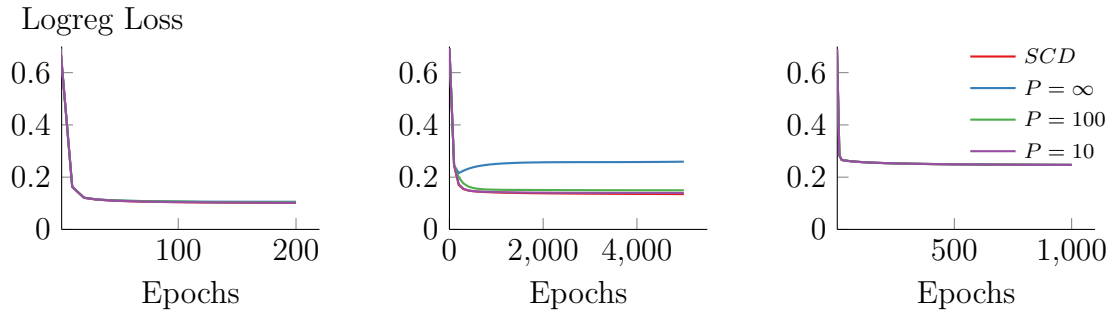

(a) Lasso, SYN1 (b) Lasso, SYN2 (c) Logreg, IM

Figure 5.3: SCD and pSCD, throughput for SYN1, SYN2 and IM. AVX-N denotes using an N-threaded CPU implementation with AVX intrinsics. Partition size: 16384. For pSCD $P = 10$.

- **SYN1 and SYN2**: Synthetically generated datasets having varying number of samples and features with uniform random noise, for throughput measurement purposes.

**Methodology.** For each data set, we select a certain configuration and perform first SCD, then pSCD with varying inner-product update periods, $P$. Step size is held constant at 4. We evaluate the convergence both by the optimization objective (loss function) and a test score: for IM we split the data set into 80/20 training/test sets; for AEA, KDD1 and KDD2 we perform inference on the test sets given in their respective *Kaggle* competitions and submit the predictions to *Kaggle* to obtain an area under curve (AUC) score. The scores we obtain are ranked in the top 50 in *Kaggle* for the respective datasets.

**Analysis.** The results are in Table 5.2: We present the deviation from the SCD results as percentages. As expected, the negative impact by the staleness of pSCD shows itself in both the loss function evaluation and the test scores. However, for all datasets, the negative effect is very limited. Furthermore, it can be decreased by using a lower global inner-product update period $P$: We observe loss and test score values closer to the ones obtained with SCD when using lower $P$ ($= 10$), which means a global inner-product update is performed more frequently. Performing the global update frequently turns out to be especially important for AEA data set. The reason for this is that it has a low number of features (32k), leading to just two partitions (each sized 16k). When those two partitions have substantially different distributions, frequent global updates are needed to share information regarding optimization steps the two partitions take separately during




(a) IM. (b) AEA. (c) KDD1.

Figure 5.4: Convergence of the Logreg loss for three data sets trained with either SCD or pSCD with varying $P$.

pSCD. The fact that even relatively infrequent global updates lead to very satisfactory results in Table 5.2 shows the usefulness of pSCD to train generalized linear models, for real-world problems. In Figure 5.4, we plot the Logreg loss over the number of epochs performed for 3 datasets. Apart from AEA, all curves are overlapped, making a visual distinction not possible. For AEA, we observe that for $P = 10$, a total visual overlap happens. From the results in this section, we see that not only the final loss with pSCD is very close to SCD, but the *empirical convergence rate* is also very similar. Thus, we conclude the statistical efficiency of pSCD is very close to that of SCD.

### 5.2.2 Hardware Efficiency

Previously we showed that epochs of SCD and pSCD are *statistically* very similar. Now our goal is to show that an epoch of pSCD can be performed faster on a multi-core CPU, compared to SCD; thus leading to an overall shorter training time. We are interested in the sample processing rate reported in GB/s, calculated by dividing the total size of all samples in a data set (number of samples $\times$ number of features $\times$ 4 Bytes) by the time required for one epoch.

**CPU performance.** On the CPU we have multi-threaded implementations of both SCD and pSCD, using Intel intrinsics to take full advantage of the AVX (256-bits) and fused multiply-add (FMA) instructions. We parallelize standard SCD by distributing the partial gradient computation to multiple threads and we synchronize before the coordinate update step for each feature. Parallelizing pSCD is much simpler, since each thread can independently work on its own partitions without any need for synchronization, which is

only needed when performing a global inner-product update. In Figure 5.3, we report numbers using synthetic and IM datasets, covering a range of dimensionality and sample count properties. The advantage of using intrinsics and multi-core parallelism (AVX-N, denoting N-threaded CPU implementation) is clearly visible, compared to a naive single-threaded implementation. For Lasso with AVX-14 doing pSCD, the processing rate reaches the memory bandwidth of the CPU (Figure 5.3b). This is thanks to the well parallelizable and cache-local processing nature of pSCD. The performance difference between SCD and pSCD is most noticeable for SYN1, in Figure 5.3a: The large number of samples in that data set leads to a large inner-product vector (∼135 MB) that cannot be kept in the CPU's cache. SCD needs to read the inner-product vector from main memory during gradient computation and inner-product update, leading to lower compute efficiency. For datasets where the inner-product vector fits the last level cache, for example with SYN2 (∼ 8 MB), this effect is less detrimental, however still visible (Figure 5.3b). Doing Logreg on the CPU has larger overhead due to having to compute the sigmoid function during gradient computation, therefore the throughput is substantially reduced for IM (Figure 5.3c).

**Conclusion.** We have shown that pSCD has better hardware efficiency than SCD on a multi-core CPU. Although this is already useful when the target platform is a multi-core CPU, the advantage of pSCD over SCD will be more pronounced for an FPGA implementation, as it will be discussed in Section 5.5.

## 5.3 Empirical Comparison to SGD

In this section we empirically compare the coordinate descent based methods (SCD and pSCD) with stochastic gradient descent (SGD) regarding convergence speed. Our goal is to understand under which circumstances SCD/pSCD is preferable over SGD, focusing on the assumption of columnar storage.

**Algorithms.** The algorithms used in this analysis are given in Table 5.3. For SGD, we use the term *minibatch* to refer to how many samples are used to compute a partial gradient. As a baseline (SGD-tf), we use the standard SGD optimizer in Tensorflow v1.11 to perform logistic regression (Logreg). The performance of SGD-tf for a minibatch size of 1 is quite low (we assume due to the overhead of calling its C++ back-end frequently), so we use a minibatch size of 512 for SGD-tf. The other algorithms are our own AVX-optimized CPU implementations. SGD-row performs standard SGD with a minibatch

Table 5.3: Algorithms used in comparison analysis. AVX-N denotes using an N-threaded CPU implementation with AVX intrinsics.

| Name | Minibatch Size | Step Size | Impl. | Storage |
|---|---|---|---|---|
| SGD-tf | 512 | 0.1 | default | row-store |
| SGD-row | 1 | 0.01 | AVX-1 | row-store |
| SGD-col-1 | 1 | 0.01 | AVX-1 | column-store |
| SGD-col-8 | 8 | 0.1 | AVX-1 | column-store |
| SGD-col-64 | 64 | 0.5 | AVX-1 | column-store |
| SGD-col-512 | 512 | 0.9 | AVX-1 | column-store |
| **Name** | **Partition Size** | **Step Size** | **Impl.** | **Storage** |
| SCD | - | 4 | AVX-1 | column-store |
| pSCD | 16384 | 4 | AVX-1 | column-store |

size of 1 on row-store data. Although its minibatch size is much lower than what we used in Tensorflow, it achieves a higher throughput due to being a native implementation and therefore provides a good baseline for SGD performance on row-stores. SGD-col-1 to SGD-col-512 perform *SGD on column-store data*, with varying minibatch sizes. SCD and pSCD are coordinate-descent based methods as described previously, working on column-store data.

**Methodology.** We use single-threaded CPU implementations for each algorithm to obtain a fair comparison. We explain the reason for doing a single threaded analysis in the following. SGD when training linear models is not straightforward to parallelize because of its iterative nature: In each iteration the model under training is updated and the next iteration requires the up-to-date model, creating a tight dependency. There are many studies about modifying SGD by relaxing certain constraints, such as allowing asynchronous updates to enable better thread parallelism [RRWN11, DSZOR15, NO14]. However, since the efficiency of the algorithm depending on the storage layout is an orthogonal issue to how it is modified for better thread parallelism, using standard SGD with one thread serves the purposes of our analysis. Although SCD/pSCD is straightforward to parallelize without any modifications to the algorithm, to remain fair regarding how much compute resources are used, we use also one threaded implementations for SCD/pSCD in this analysis. All algorithms are implemented using AVX (256-bits) instructions to take full advantage of one core. For all algorithms, we tune the step size by sweeping over a

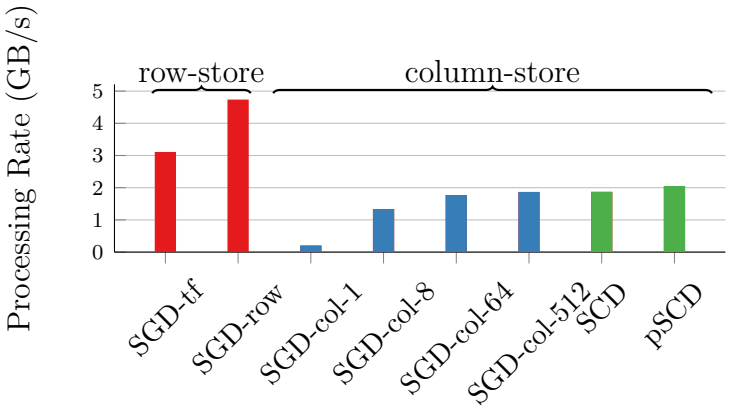


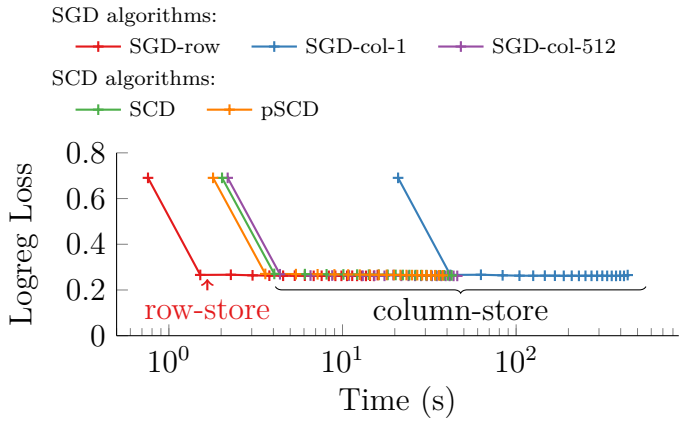

Figure 5.5: Throughput of algorithms while running Logreg on IM. For pSCD $P = 10$.

range and then use the one that delivers the best final loss. For SGD algorithms, the step size is diminished at each epoch (step size/epoch number).

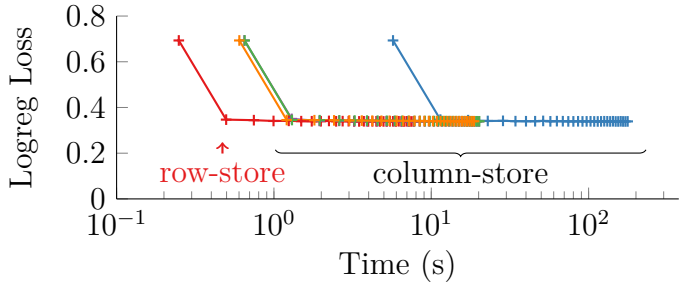
**Analysis of SGD performance.** Figure 5.5 shows the processing rate of all algorithms when running Logreg on IM data set. Higher bars indicate that the algorithm can execute epochs faster, corresponding to better *hardware efficiency*. When the underlying storage layout is a row-store, SGD is the clear winner in hardware efficiency. However, when the same algorithm is run on a column-store (SGD-col-1), the throughput drops more than 20x. This is because, regular SGD—having a minibatch size of 1—is very inefficient when gathering individual values of a row from different memory locations. Moreover, with a minibatch size of 1, any means of achieving instruction level parallelism is also blocked due to a tight data dependency, where each iteration depends on the previous iteration's result. This observation is consistent with previous work, analyzing SGD performance on row vs. column stores [ZR14].

Still, there is a way to improve the hardware efficiency of SGD on column-stores: Increasing the minibatch size. This eliminates the tight data dependency, allowing unrolling the innermost loop of the algorithm and the usage of SIMD parallelism on minibatch sized columns. Furthermore, since we read minibatch size chunks from the memory contiguously, we also increase memory access efficiency. As we observe in Figure 5.5, larger minibatch variants of SGD (SGD-col-8 ... SGD-col-512) have a much higher processing rate, nearly reaching SCD/pSCD on columns-stores.

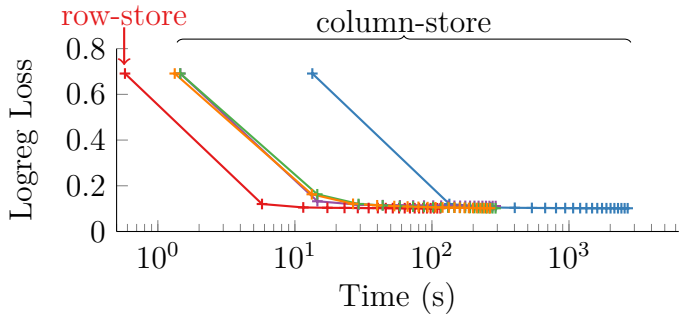
**Hardware and statistical efficiency combined.** To get a complete picture of algorithm efficiency, we evaluate hardware and statistical efficiency combined. While the hardware




(a) Convergence over time for KDD1.




(b) Convergence over time for KDD2.




(c) Convergence over time for IM.

Figure 5.6: Convergence of the Logreg loss using different training algorithms on three data sets, plotted over time to observe the combined effect of hardware and statistical efficiencies. Partition size: 16384. For pSCD $P = 10$.

efficiency tells us how fast an algorithm can complete a training epoch (corresponding to the results in Figure 5.5), the statistical efficiency tells us how much optimization progress the algorithm makes in that epoch.

In Figure 5.6 we show the convergence curves obtained by running different algorithms on

different datasets, plotted over time. Thus we can observe which algorithm reaches a lower optimization objective faster. While the statistical efficiency for SGD-row and SGD-col-1 are—by definition—exactly the same, the difference in hardware efficiency discussed earlier leads to SGD-row's much faster convergence, as observed for all datasets. The statistical efficiency of SGD-col-512 can remain practically unaffected by increasing the initial step size (Table 5.3) and it offers a good alternative to SCD in terms of hardware efficiency, if the underlying data format is column-store. The statistical efficiency of coordinate descent methods is slightly lower than SGD variants for IM data set (Figure 5.6c), having a large number of samples to number of features ratio; nevertheless SCD/pSCD reaches as good a solution after the same number of epochs. The difference in convergence rates is explained well in theory [BCN18] that favors SGD for datasets with large number of samples to number of features ratio.

**FPGA-related considerations.** Large-minibatch-SGD gives a comparable hardware efficiency to pSCD even when working with a column-store, on a CPU. We discuss here if large-minibatch-SGD would be also a viable alternative to pSCD for FPGA acceleration. We consider FPGA as a target platform, because our final goal is to efficiently combine on-the-fly data transformation with a training algorithm. When implementing large-minibatch-SGD on FPGA, following difficulties regarding resource consumption may arise:

(1) For efficient usage of the limited memory bandwidth, the model needs to be kept in FPGA on-chip memory (BRAM). Otherwise, half the memory bandwidth will be used for reading the model. This problem does not exist in pSCD, as only one coordinate of the model needs to be updated per each partition, thus leading to less BRAM usage.

(2) For SGD, a large minibatch size is required to saturate the memory bandwidth when reading individual columns from a column-store. Empirical studies on our platform show that around 32 cachelines (4KB) need to be read from subsequent addresses to saturate the bandwidth [SIO+17], leading to a minibatch size of 512. Besides potentially lowering the statistical efficiency [BCN18], larger minibatch sizes would also increase the BRAM usage as discussed next.

(3) In SGD, a row is required twice during an update, once in the initial dot product and once in the partial gradient computation. To avoid wasting memory bandwidth, an entire minibatch of all rows need to be buffered during an update, leading to large BRAM usage. For instance, to run KDD1 with 2399 features and a batch size of 512, we need 4.9 MB of BRAM which is around 70% of our target FPGA. Such high resource usage makes meeting

timing constraints for an FPGA design more difficult, potentially requiring reducing the clock frequency. This problem does not exist in pSCD, since only one column is needed per update and therefore resource usage does not scale with dimensionality.

With these arguments, large-minibatch-SGD seems to be less suitable than pSCD for an FPGA implementation when working on column-stores. However, when working on row-stores, efficient linear model training designs have been proposed on FPGAs [KAA+17] as well, potentially reaching the same hardware efficiency as pSCD.

**Conclusion.** When the underlying storage format is row-store, the clear choice of algorithm is SGD, providing both high hardware and statistical efficiency. This conclusion is also inline with existing in-DBMS machine learning solutions [HRS+12, CVP+13, AtCG+15], which work on row-store format and use SGD for training. When we are dealing with column-stores, both large-minibatch-SGD and SCD/pSCD emerge as good candidate algorithms. The main advantage of pSCD over large-minibatch-SGD is its suitability for an FPGA implementation when processing column-store data. This leads to an efficient and scalable FPGA implementation, which in turn can be combined with other necessary data processing modules such as decompression and decryption, as we discuss in Section 5.5.

## 5.4 Non-Disruptive Integration

A primary limitation of integrating ML algorithms into a DBMS is that the two systems usually have different assumptions about the underlying data. In previous sections, we described the inconsistency between the nature of SGD and the storage layout in column-stores. We offered pSCD as a solution, enabling both statistical and hardware efficient generalized linear model training on a column-store, with a CPU. However, DBMS usually store and manage data in ways that are inherently unfriendly to machine learning algorithms. For instance, column-stores are very suitable to being compressed efficiently thanks to the fact that items of the same type are stored next to each other. DBMS take advantage of this and store tables compressed. Sensitive applications also might choose to keep the data encrypted at all times. However, ML algorithms by default cannot work directly on compressed or encrypted data.

The naive solution of creating a separate copy of data properly formatted for ML purposes is usually less than ideal. First, it creates maintenance problems and requires additional

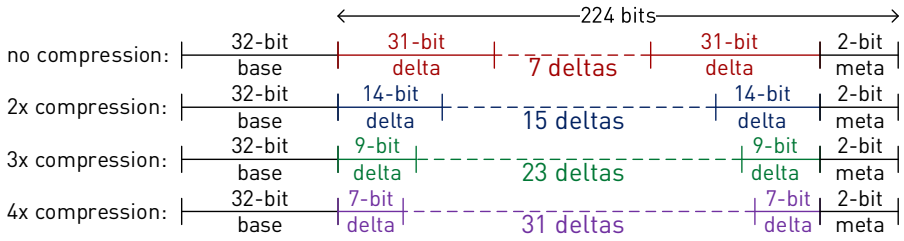



Figure 5.7: An illustration of the block-based delta encoding scheme.

storage by creating two copies. Second, it defeats the benefits of certain transformations: Keeping a decompressed copy defeats the purpose of efficient storage and keeping a decrypted copy defeats the security objective. Therefore, non-disruptive integration requires on-the-fly data transformation from its in-storage state to that required by ML only when it is needed. We consider two such data transformations: delta-encoding decompression and AES decryption.

**Block-based Delta-Encoding.** The encoding tries to find small delta series in a column and packs the base and the deltas in 256-bit wide blocks, ergo block-based. During decompression, the blocks can be processed independently, thus allowing a high degree of parallelism. Choosing 256-bit wide blocks provides a nice trade-off between compression capability vs. ease of decompression: A larger block would increase compression capability, while decreasing the ease of decompression due to less independence.

Depending on the largest delta found in a series, a compressed block can have the following number of values (Figure 5.7): 8, 16, 24 or 32. The first value in a 256-bit block is always the base, which takes 32 bits. The rest of the block is used to store deltas: 7, 15, 23 or 31, taking 31, 14, 9 or 7 bits, respectively. At the end of each block, 2 bits are reserved for meta data storing the information about the compression rate applied to the current block.

**Decompression Performance.** We perform a micro-benchmark on our target CPU to analyze decompression rate and how it scales with the number of threads used. In Figure 5.8, we show the decompression rate in comparison to both read bandwidth and simultaneous read/write bandwidth on the target CPU. We observe that decompression is compute bound; as a result, the decompression rate scales linearly with the number of threads until the core count on the target CPU—14 cores—is reached. Therefore, we conclude that on-the-fly decompression on the CPU is expected to lead to substantial

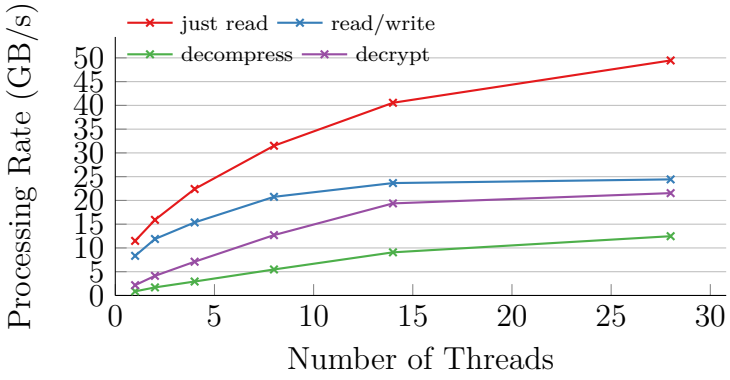



Figure 5.8: CPU scaling of read bandwidth, read/write bandwidth, decryption and decompression rates.

```
1  AESDEC (in[127:0], out[127:0], last):
2      temp = InvShiftRows(in)
3      temp = InvSubBytes(temp)
4      if last then
5          out = temp
6      else
7          out = InvMixColumns(temp)
8      end
9  AESDEC-256 (in[127:0], out[127:0])
10     AESDEC (in, temp1, 0)
11     ... — multiple executions of AESDEC
12     AESDEC (temp12, temp13, 0)
13     AESDEC (temp13, out, 1)
```

**Algorithm 8:** Steps in AES-256 decryption

throughput reduction if performed before SCD.

**AES-256 CBC Decryption.** We use the Advanced Encryption Standard (AES) with 256-bit key and "Cipher Block Chaining" (CBC) as the block cipher mode—this determines the block-size for data that has to be encrypted and decrypted together. We select the block size of CBC to be the partition size of our ML algorithms. On the target Xeon CPU, AES encryption/decryption is supported by specialized intrinsic instructions which we use to implement AES-256 in CBC mode.

**Decryption Performance.** The decryption steps are shown in Algorithm 8, of which

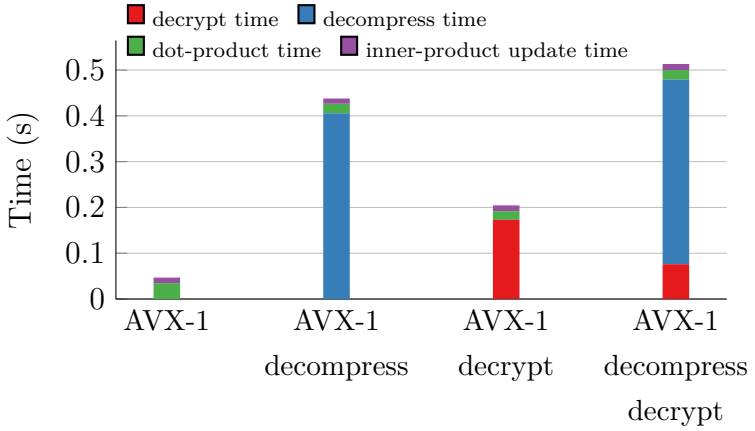



Figure 5.9: CPU breakdown analysis for pSCD with on-the-fly data transformation. Data: 1 Million samples, 90 features. Partition size: 8192. For pSCD $P = 10$.

`AESDEC` is an intrinsic instruction providing 1.78 cycles/byte performance according to the Intel AES intrinsics manual [aes]. This is consistent with our micro-benchmark in Figure 5.8, considering a peak clock frequency of 3.2 GHz. Although scaling better than decompression, decryption is still compute bound, even when using all cores available.

**On-The-Fly Data Transformation on CPU.** In Figure 5.9 we show a timing breakdown of individual parts when performing pSCD with on-the-fly data transformation. As expected, data transformation times dominate on the CPU. Decompression on-the-fly is more costly than decryption on-the-fly; when data is both compressed and encrypted, the decompression time dominates. When we perform both encryption and compression, we first compress the data in order to benefit from inherent numerical properties for a better compression rate and then encrypt the compressed data. This has the side effect of decryption being shorter when performed on compressed data, simply because less data has to be decrypted, as we see in the rightmost column in Figure 5.9. The breakdown experiment confirms that on-the-fly data transformation when present indeed dominates the runtime of ML training on a CPU.

## 5.5 Specialized Hardware

Since data transformation is so costly on the CPU, but is also required for a seamless integration of ML into a DBMS, we offer a specialized hardware solution. Our goal is to develop a data processing pipeline performing both data transformation tasks and ma-

chine learning. FPGAs excel at pipeline parallelism and due to their micro-architectural flexibility they also offer acceleration for vastly different compute tasks. Therefore, they are a suitable target platform for providing ML with on-the-fly data reconstruction. In this section we provide an in-depth description of an FPGA-based *SCD Engine* and the implementation of two data transformation tasks on the FPGA: decompression and decryption.

### 5.5.1 FPGA-based SCD Engine

The *SCD Engine* is designed to be runtime configurable regarding many parameters: Data properties (number of samples and features), partition size, global inner-product update period, and whether to perform SCD or pSCD. We can also dynamically choose which data transformation slots (e.g., decompression and decryption) to activate. Figure 5.10 shows a high-level diagram with the essential blocks of the *SCD Engine*.

#### 5.5.1.1 Fetch Engine

A *Fetch Engine* is responsible for generating read requests with correct addresses to the memory; to read the inner-products, the labels, and the samples—in partitions. The read responses (64 B cachelines) arrive out-of-order from the memory links, so the *Fetch Engine* performs reordering of cachelines according to their transaction IDs, using a large enough (256 lines) internal reorder buffer.

**Address Calculation.** Since we perform address translation on the FPGA, the *SCD Engine* can work on a virtual address space. At the start of each epoch, the offsets for the inner-products, the labels, and the feature columns are read by the *Fetch Engine* and stored internally for calculating final addresses.

The essential part of final address calculation is partition offset determination, for which there are two modes of operation: (1) If the data is stored regularly—if the offsets for partitions can be computed—, then the partition offsets are calculated on the engine and the final address can be determined with those. (2) If the data is stored irregularly—for instance in the compressed case each partition might occupy a different size in memory—, then the partition offsets are stored in the memory and they first have to be read by the *Fetch Engine* before the final addresses can be determined.

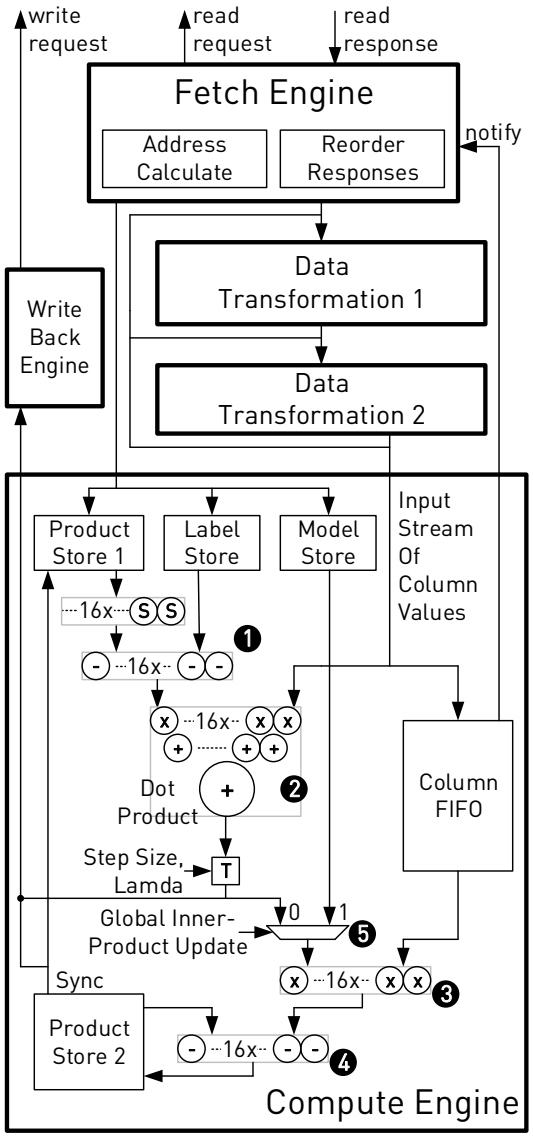



Figure 5.10: A high-level diagram showing the parts of the *SCD engine* on the FPGA.

In the latter mode, where the partition offsets have to be read from the memory, the *Fetch Engine* has to wait as long as the memory latency before it can start requesting the partition. This causes a substantial reduction in read throughput, if only one *SCD Engine* is employed. However, when multiple *SCD Engines* are employed, as discussed later in Section 5.5.1.4, there are enough requests generated to hide this effect, as we show later in our evaluation. The effect is similar to that observed in join algorithms where a large number of threads can help hide memory latencies [BLP11].

**Fetch Frequency.** The *Fetch Engine* is allowed to request one complete partition without any control by the *Compute Engine*, which is further down in the pipeline. However, in order to start requesting the next partition, a notification has to come from the *Compute Engine*, as shown in Figure 5.10. This notification is generated once an entire partition is received by the *Compute Engine*. The longer the latency of the data transformation, the more the *Fetch Engine* has to wait until it can start requesting a new partition. The throughput reduction caused by this usually remains small, even for high latency operations such AES-256 decryption, since the read time for the partition sizes we are using (e.g., 64 KB) is much larger than internal data transformation latencies.

**SCD vs. pSCD.** The read pattern of the *Fetch Engine* depends on whether the FPGA is supposed to perform SCD or pSCD. In the pSCD case, a partition from the inner-products and labels are fetched and stored in the *Compute Engine*, then the *Fetch Engine* proceeds to reading the corresponding partitions from all feature columns. Since the inner-products and the labels are read only once per partition in the case of pSCD, the memory access complexity of one epoch is proportional to $nm$, where $n$ is the number of features and $m$ is the number of samples. In the SCD case, the access complexity is larger, because the intermediate state that needs to be kept is much larger than in pSCD: For *each column*, the labels are read once, the inner-products and all feature columns are read twice (once for the gradient computation, once for the inner-product update), and the inner-products are written once, resulting in a memory access complexity proportional to $6nm$.

**Global Inner-Product Update.** In pSCD, we perform a global inner-product update at every $P$ epochs, with an aggregated model. The *Fetch Engine* enables this operation by reading the aggregated model from the memory and then scanning all columns in the partition pattern, just as in normal mode of operation. Labels and current inner-products do not have to be read, as they are not required in this operation. The memory access complexity of this operation is equal to performing one epoch of pSCD, $nm$, where $n$ is the number of features and $m$ is the number of samples.

#### 5.5.1.2 Compute Engine

The *Compute Engine* is responsible for calculating the gradient of the current partition and for updating the local inner-product. It can also perform a global inner-product update with an averaged model kept in *Model Store*, a step required in pSCD. It is designed to consume one 64 B cacheline, with 16 single-precision floating-point values at every clock

cycle, resulting in a processing rate of 12.8 GB/s when clocked at 200 MHz.

**Compute Pipeline.** In the following, we give a high-level description of the compute pipeline shown in Figure 5.10. Here, we first explain the mode of operation, when *Global Inner-Product Update* (Figure 5.10) is not performed. Partition-sized inner-products and labels are kept in *Product Store 1* and *Label Store*. When a feature column is fed into the *Compute Engine*, first the differences between inner-products (for Lasso without and for Logreg with *sigmoid S*) and labels are calculated (❶), then the dot product between the differences and the column values is calculated (❷). In the meantime, the column is written into the partition-sized *Column FIFO*, waiting there to be used during the inner-product update. Once the dot product is ready, it is multiplied with the step size and regularization term is applied as in the thresholding function in Algorithm 10. The resulting value is the model update for the current coordinate, or feature. This value is written back to the memory and also used internally in the *Compute Engine* to update the local inner-product. The inner-product update takes place by multiplying the column values from *Column FIFO* with the model update (❸) and subtracting the values from the local inner-product (❹) residing in *Product Store 2*. Notice that we keep two inner-product stores in the *Compute Engine*. The reason is that we use single-port read/write Block-RAM (BRAM) resources on the FPGA to implement local storage. As a result, at every clock cycle data from only a single address can be read from a BRAM. Since we need to read the inner-product both at the input for the dot product and during the inner-product update, we keep two replicas of the inner-product and update both of them at the same time.

**Global Inner-Product Update.** In the second mode of operation, when *Global Inner-Product Update* (Figure 5.10) is active, the aggregated model is kept in the *Model Store*. The columns are read just as in normal mode of operation but instead of performing a dot product, only a scalar-vector multiplication (❹) is performed between the values in the model store (❺) and values from the *Column FIFO*. Mathematically, the goal of this computation is to calculate the dot products between the aggregated model and all samples in the data set, to obtain an up-to-date inner-product vector.

#### 5.5.1.3 Write Back Engine

The *Write Back Engine* is responsible for writing the model updates and updated inner-products back to the memory. Since the memory interface is granular as a 64 B cacheline,

writes also have to be issued in this granularity. For inner-product writes, this happens naturally, because the interface to *Product Stores* are 64 B wide. For model updates, however, the *Write Back Engine* concatenates 16 of them to have a complete cacheline before requesting a write to the memory. That means model updates are written for every 16 columns processed. If the number of columns modulo 16 is larger than 0, the end of processing for the final column triggers the write back with a padded cacheline.

**SCD vs. pSCD.** For the pSCD case, the updated inner-products are written back after *all* columns are processed, resulting in a write complexity proportional to $m$, the number of samples. For the SCD case, the updated inner-products have to be written back after each processed column, since local storage is not large enough, resulting in a write complexity proportional to $mn$.

#### 5.5.1.4 Employing Multiple SCD Engines

One *SCD Engine* can potentially process data close to its maximum processing rate (64 B $\times$ 200 MHz = 12.8 GB/s), if a very large partition size is selected. A large partition size means that the *Fetch Engine* can request more data at a time without waiting. However, in our evaluation, we observe that even using a relatively large partition (64 KB) leads to around 9 GB/s sample processing rate for the *SCD Engine*, due to the high memory access latency ($\sim$250 ns). Furthermore, when operating in dynamic partition offset reading mode, the high memory latency hurts performance even more, as explained in Section 5.5.1.1, around 5 GB/s sample processing rate. To overcome this limitation and to saturate the available memory bandwidth for the FPGA ($\sim$17 GB/s), we put multiple *SCD Engines* to work. The engines can process samples completely independently due to the well parallelizable nature of pSCD.

To work with multiple *SCD Engines*, we need a load balancer, as in Figure 5.11. The load balancer talks to the Intel Endpoint at 400 MHz, enabling a peak data processing rate of 25.6 GB/s. There are asynchronous FIFOs in the load balancer, responsible for the necessary clock domain crossing: 4 FIFOs for requests, 4 FIFOs for responses. The reason for keeping *SCD Engines* clocked at 200 MHz is to make meeting the timing requirements for the FPGA much easier; also, 4 *SCD Engines* at 200 MHz provide an aggregate processing rate already enough to saturate the memory bandwidth. During runtime, we can select how many engines we want to use. We use this to benchmark how overall throughput can be improved when multiple engines are employed, as discussed during the evaluation.

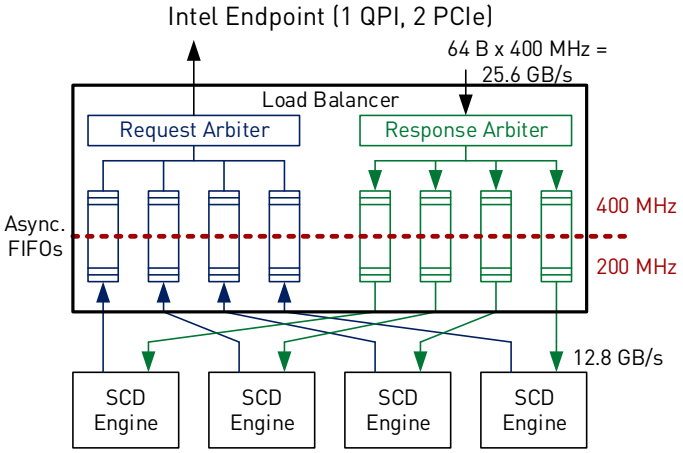



Figure 5.11: Load balancing to employ multiple *SCD Engines* on the FPGA.

With 4 *SCD Engines*, the memory latency can be hidden well and enough requests are generated to saturate the memory bandwidth.

Regarding load balancing, we assign partitions as equally as possible among *SCD Engines*. Each *Engine* reads and writes data from its own partitions in the memory. Thus, the load balancer is passive and merely arbitrates access to the memory port: read/write requests from each Engine are tracked and replies are forwarded accordingly. The load balancer has to keep track of as many cacheline requests as *Engines* can produce in-flight, that is 256 cachelines.

### 5.5.2 On-The-Fly Data Transformation

The advantage of the architectural flexibility of an FPGA shows itself most when data processing can be done as a pipeline. Deep pipelines on FPGAs turn the problem of achieving high throughput into a question of *whether there are enough resources*, instead of counting cycles as done for a CPU. As much as the available resources on the FPGA allow, data processing modules can be put in a pipeline, resulting in an overall throughput determined by the slowest module. If all modules process data at the same rate, then the number of modules in the pipeline does not affect the throughput, it only affects resource consumption. Thanks to pipeline parallelism, when we put data transformation slots in front of the *Compute Engine*, we enable on-the-fly data transformation at no throughput reduction—if the data transformation happens at the same rate as the *Compute Engine*,

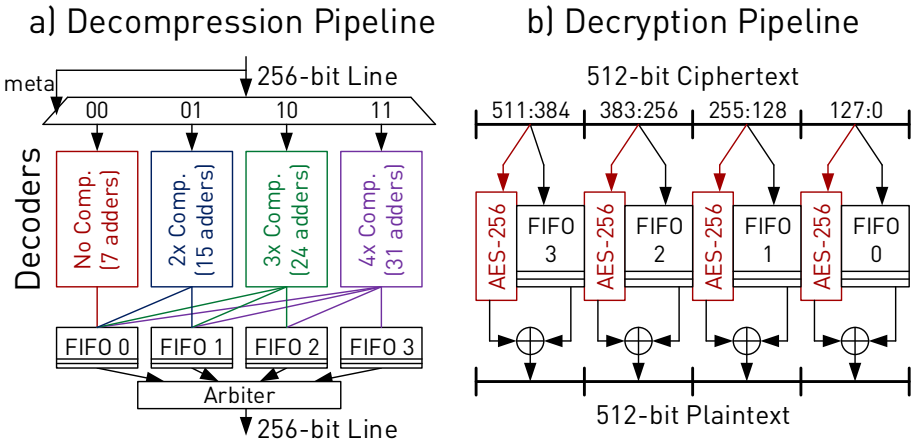
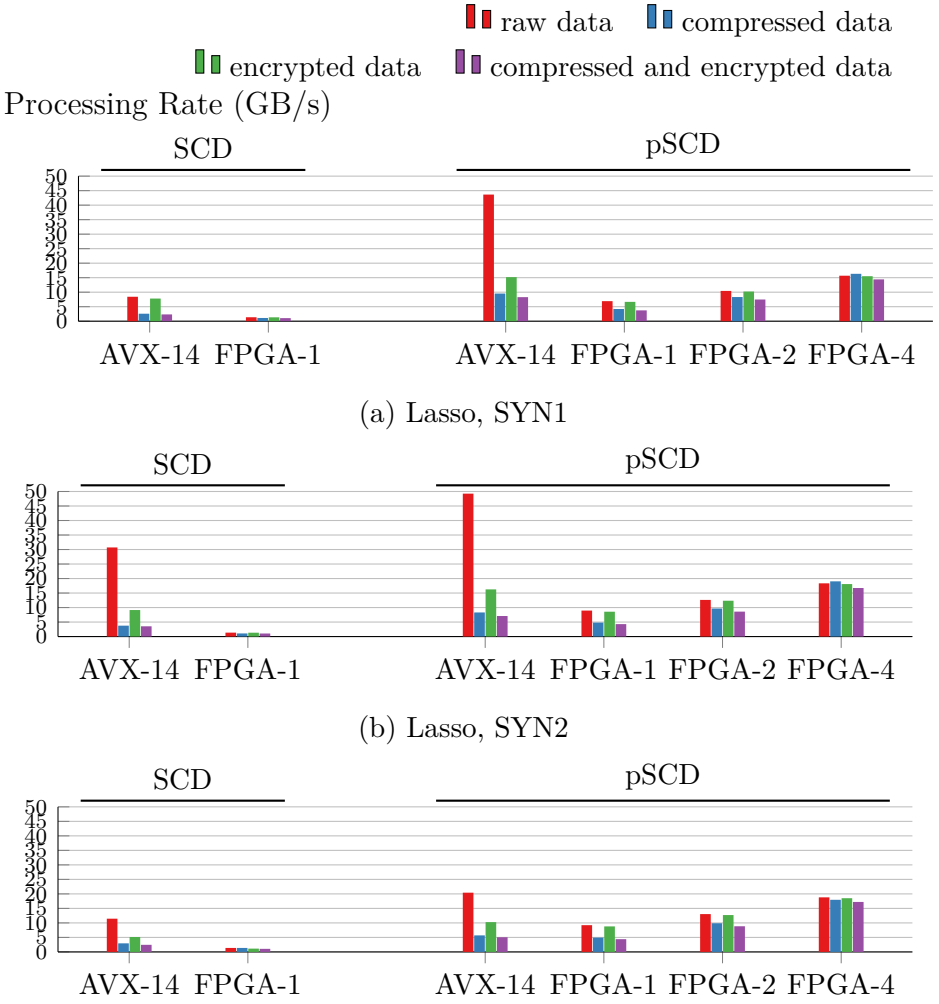



Figure 5.12: Decompression and decryption pipelines on the FPGA.

that is 64 B/cycle.

**Delta-Encoding Decompression** FPGAs excel at compression and decompression tasks [RBK07, SMT+12, FKBH15], mainly because they inherently can access and manipulate data in a bitwise manner. We implement block-based delta decompression on the FPGA using spatial parallelism to handle multiple values in the same cycle, using multiple adders. Determining the FPGA-based decompression performance is straightforward by analyzing the design, shown in Figure 5.12. A 256-bit input line is fed to its corresponding decoder, depending on its meta bits. In the case of 4x compression, the pipeline has to produce 4 times as many lines as it consumes. Therefore, its consumption rate in the worst case is 1 line per 4 cycles. We put two such decompression pipelines when doing decompression in front of the *Compute Engine* in order to handle 512-bit cachelines. In the worst case, the theoretical rate of decompression is 4x less than the rate of the *Compute Engine*. In practice we do not observe any performance reduction related to this, mainly because one *SCD Engine* does not read data as frequently to saturate memory bandwidth in dynamic partition offset reading mode (Section 5.5.1.1).

**AES-256 CBC Decryption** When performing decryption on the FPGA, we eliminate the performance disadvantages of the CPU. Since FPGA-based designs are specialized and use a restricted memory interface, having decrypted values only on the FPGA might also provide a security advantage, although recent work [RPD+18] has shown side-channel attacks are possible in multi-tenant FPGA settings. In Figure 5.12 we show the pipelined design for performing AES-256 decryption in CBC mode. Each *AES-256* block is im-




(a) Lasso, SYN1

(b) Lasso, SYN2

(c) Logreg, IM

Figure 5.13: SCD and pSCD, throughput for SYN1, SYN2 and IM. AVX-N denotes using an N-threaded CPU implementation with AVX intrinsics. FPGA-N denotes using N *SCD Engines* simultaneously. Partition size: 16384. For pSCD $P = 10$.

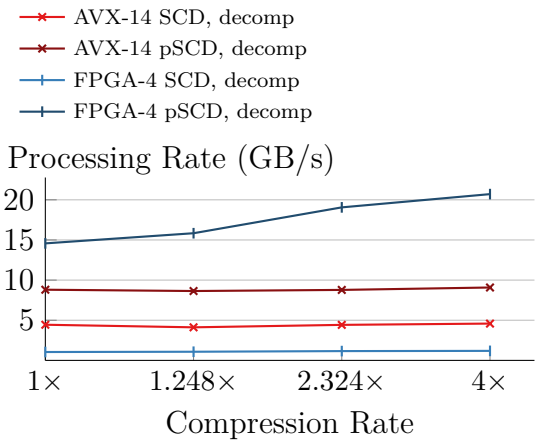
plemented as a deep pipeline on the FPGA, performing the transformations shown in Algorithm 8. We implement the arithmetic functions in the Galois field using look-up tables on the FPGA. The pipelined design in Figure 5.12 is able to decrypt data at 12.8 GB/s. When put in front of the *SCD Engine*, it causes no reduction in throughput.

### 5.5.3 Evaluation with FPGA

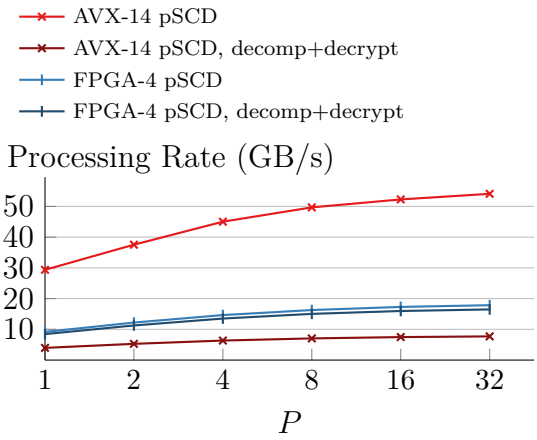
In this section we aim to show the processing rate of the FPGA for pSCD, focusing on the capability to perform on-the-fly data transformation (decryption/decompression) without performance reduction, thus offering better overall hardware efficiency for pSCD than a multi-core CPU. In Figure 5.13, we present the performance for the CPU and FPGA designs performing SCD and pSCD with on-the-fly delta-encoding decompression and AES256-CBC decryption. We report the processing rate in GB/s, calculated by dividing the total size of all samples in a data set (number of samples × number of features × 4 Bytes) by the time required for one epoch.

**FPGA performance.** The reduced memory access complexity of pSCD helps the FPGA more than it helps the CPU , simply because the FPGA has a lower memory bandwidth (∼17 GB/s) compared to the CPU (∼50 GB/s). As a result, we observe a relatively low performance for the FPGA performing standard SCD compared to the CPU (Figure 5.13). When we perform pSCD, the FPGA processes samples at a much higher rate, due to the better intermediate state locality provided by partition-based processing. With 4 *SCD Engines* operating in parallel (FPGA-4), the FPGA processes samples at the maximum available memory bandwidth of ∼17 GB/s, for relatively high dimensional datasets, SYN2 and IM. For SYN1, the processing rate for samples is around 15 GB/s, lower compared to the other ones, because SYN1 has less columns; therefore more inner-product updates happen per column processed.

**Data transformation on the FPGA.** The advantage of the FPGA shows itself most when performing on-the-fly data transformation. When doing decompression, the FPGA-1 performance is lower because partition offsets have to be fetched from the memory dynamically, as explained in Section 5.5.1.1. However, the latency caused by this dynamic offset fetching can be hidden when more *SCD Engines* are employed: For FPGA-4, we see that processing compressed data might even increase performance on the FPGA (Figures 5.13a and 5.13b) because in total less data has to be read from the memory. Performing just decryption on-the-fly is shown to come at no performance reduction. Performing both decompression and decryption causes a slight decrease in the processing rate, around 8%, because of the circuit latency introduced by these operations: The *Fetch Engine* has to wait slightly longer until it can request a new partition. The effect of this latency could be hidden by using a larger batch size than we used for these experiments (64 KB), which comes at a higher on-chip storage consumption cost.




(a) Compression rate analysis. For pSCD $P = 10$.




(b) Global inner-product update period $P$ analysis.

Figure 5.14: Sample processing rate shown at different compression rates and increasing global inner-product update periods $P$ on the multi-core CPU and FPGA. Lasso, SYN2. Partition size: 16384.

**Effect of the compression rate.** We show the effect of the cost of decompression on the sample processing rate on the CPU and the FPGA in Figure 5.14a. Higher compression rate means higher cost of decompression, since more data is produced through delta addition (Section 5.5.2). Against intuition, sample processing rate increases with higher compression rate (=higher cost) for pSCD on the FPGA. This shows that reading less data—thanks to higher compression rate—is more beneficial than the increasing decompression cost on the FPGA. Furthermore, it shows that the *Compute Engines* are able to keep up with higher processing rates than the memory bandwidth, which is the bottleneck of FPGA-based pSCD when compression rate is 1×. The CPU also reads less data with a higher compression rate, but higher decompression cost amortizes this leading to a constant processing rate.

**Effect of global inner-product update period $P$.** In Figure 5.14b, we show the effect of the global inner-product update period for pSCD. Performing a global inner-product update takes a similar amount of time as performing one pSCD epoch without the global update for Lasso, as analyzed in Section 5.2. We see this behavior in the experiment, a higher $P$ resulting in a higher processing rate both on the CPU and the FPGA. As shown in Section 5.2.1, high $P$ values such as 10 still lead to high quality training, so that the overhead by the global inner-product update can be kept minimal.

**About inference performance.** Since the FPGA is already memory bandwidth bound performing training with pSCD, inference on the FPGA would not be any faster than the rates we observe in Figure 5.13. For the CPU, significant throughput increase is also not expected when doing only inference, because just the inner-product update time is eliminated and that is only a small portion of the total runtime, as we showed previously in Figure 5.9.

**Conclusion.** We have showed the advantage of using an FPGA design, if the data needs to be decompressed or decrypted before a machine learning task. However, if the data exists in a non-transformed state, a multi-core CPU offers a faster solution. It is important to mention that the FPGA design becomes memory bandwidth bound and future FPGA-based architectures offering more memory bandwidth could turn this trade-off more in favor of FPGA-based solutions, even if the data does not need to be transformed.

## 5.6 Related Work

**ML in DBMS.** As ML-based data analytics becomes commonplace, integrating ML functionality into a DBMS is ever more important and therefore a very active research field [TBC+05, FML+12, FKRR12, HRS+12, KTD+13, KNP15, KBY17, LAT18]. The SAP HANA [FML+12] Predictive Analysis Library enables a wide range of ML algorithms in the form of SQLScript procedures. Both stochastic gradient descent and cyclical coordinate descent (similar to standard SCD in this work) are given as possible training algorithms, however details about the implementations are not disclosed. Zhang et al. [ZR14] provide an in-depth study on which ML algorithms to use depending on the storage layout, focusing on a NUMA-based CPU system, rather than the cache-conscious or specialized hardware approaches we take in this work. Most of the remaining work in this area focuses on row-store databases: Feng el al. [FKRR12] consider *gradient descent* based methods, noting its similar data access pattern with SQL aggregation and integrate it into a row-store DBMS. Hellerstein et al. [HRS+12] propose MADlib enabling SQL-based ML with user-defined-functions (UDF) primarily on a row-store database (PostgreSQL); similar to how we use MonetDB [IGN+12] UDFs, its main difference being columnar storage. To our knowledge, our focus on performing ML natively on column-store DBMS via coordinate-descent methods and our approach in using specialized hardware to work on transformed data with high performance is unique among existing work.

**Stochastic Coordinate Descent.** Shalev-Shwartz et al. [SST11] introduce stochastic coordinate update at each iteration and provide a theoretical convergence analysis with tight convergence bounds. Zhao et al. [ZYW+14] use a minibatch-based coordinate descent approach, where they, for each epoch, first compute an exact gradient and then use that during minibatch-wise access to adjust partial gradients. Our approach with pSCD is similar to this, as we perform global inner-product updates every $P$ epochs, similar to computing the exact gradient; however, our approach is different in that $K$ models can be updated independently, thus allowing staled updates and enabling parallel processing with infrequent synchronization. Jaggi et al. [JST+14] introduce CoCoA, a communication efficient distributed dual coordinate ascent algorithm. Our approach is similar in that it also partitions a dual variable (the inner-product vector) into disjunct pieces and performs local optimization in a distributed way followed by model averaging. While they analyze the local optimization and subsequent aggregation independently, we analyze the effect of the frequency of model averaging from a staleness point of view, similar to Ho et al. [HCC+13] and keep the number of coordinate descent steps the same, regardless of how many aggregation steps are performed. Liu et al. [LW15] perform SCD asynchronously to enable better parallelism and show near linear speed-ups in distributed settings, however not eliminating cache inefficiencies. Recently, coordinate-based methods have also been considered for deep neural network (DNN) training [ZB17].

**Specialized Hardware in DBMS.** Database acceleration with specialized hardware, in the form of FPGAs, GPUs and ASICs, has been a very relevant topic in recent literature [KA16, KGA17, SIOA17, WZY+14, SJ17, LSK+15]. Oracle's Sparc M7 processor [LSK+15] contains so-called database analytic accelerator (DAX) engines on silicon, performing predicate evaluation and decompression on-the-fly, the latter similar to our approaches in this work. Fang et al. [FZEC17] also target on-the-fly data transformation with an ASIC design, focusing on offloading extract-transform-load workloads from the CPU, albeit not combining it with other computation directly on specialized hardware, as we do in this work with decompression/decryption and SCD/pSCD. Istvan et al. [ISA17] implement a persistent key-value store, similar to a noSQL database, on an FPGA-based system, using its on-the-fly data transformation capabilities. However, since we are interested in maintaining OLAP capability while providing efficient ML in this work, noSQL databases are not a direct option.

**Specialized Hardware for ML.** Due to the data and compute intensive nature of ML, specialized hardware designed for these algorithms has a large potential to accelerate

both training [JYP+17, KGA17, MKS+18] and inference [UFG+17, OZZA17]. Similar to this work, Kara et al. [KGA17] accelerate generalized linear model training on an FPGA; however they use SGD on row-stores and focus on optimizing the architecture for quantized input data, whereas we focus on coordinate-descent based methods and performing on-the-fly data transformation.

## 5.7 Discussion

In this project we have focused on generalized model training on column-stores using coordinate descent based methods. We use partition based stochastic coordinate descent (pSCD), that improves the memory access complexity of SCD, leading to better training performance both on the CPU and FPGA. We show the staleness of pSCD for model updates can be fine tuned, leading to high quality convergence. On the systems side, we presented an FPGA-based system capable of performing various compute intensive data transformation tasks—decompression and decryption—in a pipeline before an SCD engine, performing either SCD or pSCD on the FPGA. We compared our FPGA-based system to an AVX-optimized multi-core CPU implementation, showing that the multi-core CPU is faster on raw data. However, once it has to perform on-the-fly data transformation, its performance is reduced significantly; whereas the FPGA sustains high throughput even when it performs decompression/decryption, due to pipeline parallelism. We also compared pSCD to more popular SGD and discussed under which circumstances the choice of pSCD over SGD makes sense.

The resulting system shows the advantages of using specialized hardware for dataflow processing and machine learning, in a column-store database setting. One limitation is the limited applicability of the proposed design. For instance, if the target database uses a different compression scheme than delta encoding or if the storage format is row-oriented making SGD more preferable, the hardware design needs to change to adapt to these properties of the system. In the next chapter, we tackle this limitation by introducing a more programmable solution for specialized hardware development.

# 6 Generic and Context-Switch Capable Data Processing on FPGAs

In previous chapters, we showed the potential advantages of FPGA-based processing for OLAP and machine learning workloads within an in-memory database management system. Despite the advantages in certain settings, existing FPGA-based accelerators are often not as widely used as they could be. The lower operating clock frequency of FPGAs requires highly specialized designs to achieve the necessary performance and efficiency but often at the cost of following limitations:

1. The accelerator's functionality is usually limited to a single operation.
2. The designs enforce a rigid execution flow, i.e., the start-done way of interacting with the accelerator due to the lack of context-switch capability.
3. The level of difficulty in developing new accelerators and extending/modifying existing ones are high.
4. Integration flexibility and reuse remains limited. For instance, an accelerator designed for a Xilinx FPGA cannot be deployed on an Intel FPGA with ease.

These limitations especially impact the database use-case, because: (1) DBMS are multi-tenant systems relying on shared execution capabilities to ensure fair progress for users and response time guarantees. (2) DBMS workloads are multi-faceted. As we discussed in previous chapters FPGA-based processing can be valuable regarding many aspects in this

context, ranging from OLAP (Chapter 3) to machine learning (Chapter 4), and integration (Chapter 5). Therefore, if we could support more use-cases with one accelerator or if we could modify/extend it with ease, the usability of it will increase substantially.

To overcome these limitations and as part of the larger effort within this thesis to use FPGA accelerators to extend and improve data processing engines, in this project we explore how to implement acceleration functionality within an FPGA that is general (i.e., supports a family of different operators rather than just one) and includes functionality for workload management in the form of threading, context-switching, and thread migration. The goal is to reach a design that is an intermediate point between a fully specialized circuit and a general purpose processor running on the FPGA while, at the same time, augmenting the system level functionality to the point that is somewhat comparable to that of a conventional operating system in terms of thread management. The resulting prototype, PipeArch is, to our knowledge, the first design reaching the high performance typical of an FPGA while also providing the system support needed in real settings.

**Contributions.** With PipeArch, we show how a programmable hardware architecture can support a wide range of machine learning workloads while matching the performance of fully specialized designs. PipeArch introduces runtime scheduling capabilities to an FPGA, enabled by the context-switch capability, leading to features such as time-shared execution and thread migration controlled by a software runtime manager for load balancing. Furthermore, it provides a portable design easily deployable on both an Intel in-package shared-memory FPGA platform and a Xilinx discrete PCIe-attached FPGA platform. The evaluation highlights significant improvements such as up to 4x reduced median job runtime thanks to runtime scheduling policies, and up to 3.2x speedup for generalized linear model training workloads compared to a 14-core Xeon CPU.

## 6.1 Design Goals

**In terms of hardware**, PipeArch's goal is to improve *programmability* and *reduce development effort* while maintaining the advantages of FPGA designs such as vectorization and deep pipelining. To this end, we propose a programmable processing unit, PipeArch-PU, with *custom vectorized subroutines* able to perform a wide range of machine learning tasks as fast as highly specialized previous work [KAA+17, KEZA18] and significantly faster than previous more generic solutions [MKS+18].

In order to match the performance of fully specialized accelerators, a key advantage of FPGA-based designs needs to be maintained: Deep pipelining. We implement it through the introduction of hardware subroutines and the capability of asynchronously issuing these subroutines, such that a set of subroutines are active at the same time in a producer-consumer relationship. Furthermore, we enable runtime scheduling via context-switching of the hardware threads.

PipeArch also aims at reducing the development effort. Programming is software-driven by composition of hardware subroutines with well defined interfaces. We show this with examples of vectorized subroutines (e.g., DotSigmoid, MultiplySubtract) and use them to construct a variety of machine learning algorithms. The development effort is reduced over standard FPGA programming techniques because new subroutines can easily be added based on the interfaces with key connecting functionality (e.g., programmability, data access, subroutine invocation, on-chip memory usage) made available by the platform.

**In terms of software**, the runtime manager, PipeArch-RT, monitors and dynamically schedules threads to available PipeArch-PU instances on the FPGA, providing thread management capabilities on an FPFA. PipeArch-RT can at runtime schedule threads and migrate them between instances. These features are becoming critical. First, FPGA devices keep getting larger, increasing the possibility of sharing a single device among multiple applications [SIO+17, OSKA17]. Second, FPGAs are increasingly being deployed in datacenters and the cloud [PCC+14, ama], where dynamically sharing hardware resources is of utmost importance. Currently, we are not aware of any virtualization being done at the resource level in existing systems. For instance, FPGAs in AWS are assigned as a whole to a user. By combining context-switching capability in PipeArch-PU with PipeArch-RT implementing scheduling, we take a first step towards enabling virtualization and fine-grained time-sharing on FPGAs.

**Target workloads.** The workloads implemented in the current version of PipeArch include generalized linear model (GLM) training for classification and regression tasks as well as low-rank matrix factorization (LRMF) for recommender systems. Regarding GLMs, we can train linear regression and multi-class logistic regression models, with either stochastic gradient descent (SGD) [Zha04] or stochastic coordinate descent (SCD) [SST11] as the optimization algorithm. Each optimization algorithm provides distinct benefits regarding the data access efficiency and convergence characteristics during training, as discussed previously in Chapter 5. LRMF [CR09] is widely used to create recommender systems, factorizing a large sparse matrix (e.g., users rating movies) into two smaller matrices with

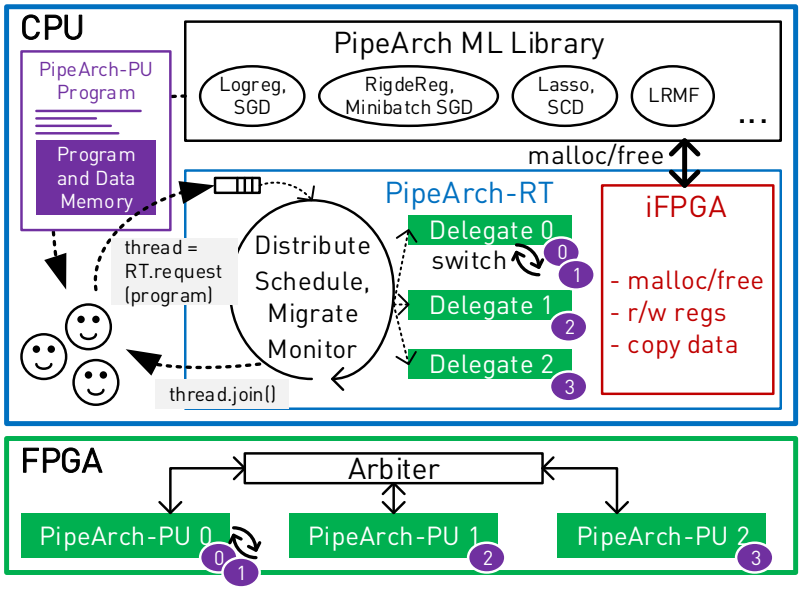



Figure 6.1: PipeArch System Overview: Users submit programs to PipeArch-RT which creates threads to execute them on available PipeArch-PU instances on the FPGA.

latent variables. We solve LRMF problems with SGD via alternating updates [RR13]. The goal behind these workloads is to showcase PipeArch's programmability, extensibility, and overall efficiency.

## 6.2 System Overview

Figure 6.1 shows an overview of PipeArch. Users interact with PipeArch by requesting the execution of a PipeArch-PU program, provided by a library containing template machine learning programs adjusted at runtime with user specific arguments (e.g., hyperparameters, number of epochs). PipeArch-PU programs are a set of subroutine calls and control instructions with specific operands. The programs contain pointers to corresponding data in the memory. PipeArch-RT maintains a set of active threads with the help of delegate objects, assigned to available PipeArch-PU instances. An FPGA interface class abstracts platform specific interaction such as memory allocation, register access and copying data. Users wait on thread completion by calling a join, similar to the pthreads API.

Thanks to context-switching capabilities, multiple threads can progress in a time-multiplexed manner on a single PipeArch-PU instance, as illustrated in Figure 6.1 on instance 0. Furthermore, PipeArch-RT supports dynamic load balancing: As soon as an instance becomes free, a thread is migrated to the free instance. Various scheduling schemes can be implemented in PipeArch-RT, as we demonstrate in the evaluation section with round-robin, first-come-first-served, and shortest-job-first.

## 6.3 Background and Related Work

PipeArch combines ideas from multiple fields. Following, we summarize related work to provide the necessary background to understand the contributions and design of PipeArch.

**1. Frameworks for FPGA-based computing** focus on the integration of FPGA-based accelerators into large-scale systems, mainly via abstraction of interfaces. Exposing FPGA-based processing units as POSIX-like threads to software has been considered as a way of abstracting invocation, synchronization and shared memory usage [LP09, AA09, IS11, IBH+13, OSKA17]. Earlier studies in this line of work, ReconOS [LP09] and hthreads [AA09] expose hardware accelerators as threads, however they assume system-on-chip platforms (FPGA is next to an embedded CPU). Centaur [OSKA17] focuses on integrating accelerated operators into databases in a shared-memory CPU-FPGA system [OSC+11]. The main difference in PipeArch compared to this line of work is that we take a holistic approach and provide support also for accelerator development and dynamic scheduling. Furthermore, we make less assumptions about the underlying hardware than systems such as ReconOS [LP09], which consider only embedded CPUs or FUSE [IS11] that integrates accelerators with respect to a soft CPU.

Another focus area of frameworks to support FPGA-based computing is the virtualization of FPGA resources for cloud computing [CSZ+14, BSB+14, AGV+17, ZXX+17, ZLZC18]. Vaishnav et al. [VPK18] provide an extensive survey of this field. These studies focus on partial reconfiguration (PR), which involves reprogramming a pre-defined region of the FPGA at runtime. The PR regions are treated as dynamically scheduled, time-shared computing units, as a way of virtualizing FPGA-based processing. The main drawback is the high cost of PR, which can be in the order of seconds depending on the size of the region, making fine-grained time-sharing impractical. Also, these solutions do not address the problem of capturing the state of a hardware thread. PipeArch is complementary to

this line of work: Each PipeArch-PU instance can occupy a PR region and it can help with virtualization tasks thanks to the software-driven execution and monitoring capabilities. PipeArch also provides novel ways to capture the state of hardware threads, enabling functionality not available in current virtualization solutions.

**2. Context-switching in FPGA-based computing** has been proposed [HTK15, KGS17, MVKGR16] via PR to capture the logic and on-chip memory contents as a way of context saving and then restoring. These methods are designed around the lack of support for capturing/restoring fabric state in FPGA devices. Although these systems do provide a non-disruptive way of context-switching, the proposed methods have several limitations: First, the methods are specific to a given FPGA device and vendor as they are based on low-level bitstream manipulation; second, the cost of PR is increased since runtime scheduling requires more frequent (bidirectional) reconfiguration; and third, the states residing on specialized computation resources (DSPs) cannot be captured, limiting the designs in fundamental ways. PipeArch removes all these limitations as context-switches are performed at the logical level and do not rely on the FPGA fabric. However, PipeArch's method requires to use PipeArch-PU and allows context-switches only at certain points during program execution.

A further method of supporting context-switching on FPGA-based accelerators is via including scan-chains to extract data from flip-flops and memory elements [KHT07, BMR16]. Bourge et al. [BMR16] propose a method showing how to do this automatically as part of a high-level-synthesis toolchain. The main drawback of this method is the hardware overhead caused (reported up to 85% increase in resource consumption for certain algorithms). This is despite the fact that only 32-bit wide scan-chains are used. With PipeArch, we use 512-bit vectorization and target acceleration compared to a high-end CPUs, so employing this type of mechanism would be detrimental to the performance goals. The capability of context-switch in PipeArch arises as a side-effect of the programmable architecture and by itself does not incur any increase in resource consumption. Furthermore, the workloads we are targeting, namely linear machine learning, are fundamentally different to the previous work in this domain.

**3. FPGA-based soft vector processors** have been proposed [YSR08, CSB+11, CFM12, SL12, SEOL14, Kap16, KJYH18] as a way of making FPGA-based processing more programmable while keeping the benefits of spatial parallelism. Chou et al. [CSB+11] propose the VEGAS architecture as an improvement over earlier work [YSR08], thanks to features such as scratchpad memory and address registers. The main difference to PipeArch is our

use of subroutines that can be pipelined compared to the use of vectorized instructions in VEGAS. Severance et al. [SEOL14] propose a way to pipeline custom vector instructions as part of a soft processor. While this is good for algorithms with simple dataflow patterns, it is limiting for algorithms with tight read-write dependencies such as those used in machine learning. Thus, the main difference in PipeArch is in its higher flexibility in consuming data by subroutines (from multiple memory regions thanks to the spatial layout) allowing pipelined execution.

While having an ISA increases programmability, it also puts limitations on the degree of specialization. The trade-off is key to achieve the full potential of FPGA-based processing. Soft processor solutions even when vectorized are limited in performance compared to modern CPUs/GPUs due to the FPGA's lower clock frequency. Therefore, proposals on soft cores typically use the performance of other soft cores on FPGAs as a baseline rather than real applications on multi-core CPUs or GPUs.

**4. Dataflow-optimized architectures** have been studied extensively [Cor97, GSM+99, BKM+04] as a way to increase performance and efficiency compared to general purpose processors. PipeArch draws inspiration from these works and transfers ideas into a modern data processing acceleration setting. Transport-Triggered Architectures (TTA) [HC95, Cor97, HHHH05, JTV+18] aim to increase instruction-level-parallelism (ILP) exploiting deep pipelining between instructions thanks to the dataflow nature of target workloads. In TTAs, computation occurs as a side-effect of data movement. So-called *Function Units* and *Register Files* (*Subroutines* and *Regions* as counterparts in PipeArch) are connected via transport busses, enabling software bypassing: results are delivered directly to the next instruction. PipeRench [GSM+99] is another early effort in prioritizing dataflow-based computing, proposing a multiple-instruction-multiple-data (MIMD) architecture with reconfigurable dataflow between simple ALUs. Explicit Data Graph Execution (EDGE) [NSBK01, BKM+04, GMC+09] is another family of dataflow-oriented architectures, that exposes the spatial microarchitecture for compiler optimization.

A common theme in these architectures is that they push complexity from hardware to software: Either the programmer or the compiler has to make additional effort in utilizing the available mechanisms to take advantage of dataflow-based computing. Furthermore, both TTA and EDGE efforts still aim to cater for general purpose processing. The main difference in PipeArch is that it starts from a fully specialized accelerator point-of-view. Therefore, data processing *has to* happen in a dataflow fashion regardless of software optimization: Since PipeArch is still highly specialized and aims to support a family of

workloads (rather than being a general purpose processor), the hardware is already optimized and the difficulty of software development is low, as we show later in Section 6.5.3. Furthermore, also due to its proximity to fully specialized solutions, in PipeArch data is processed in bulk and there is no concept of register files, but rather only scratchpad memory (*Regions*).

**5. Overlays and networks-on-chip (NoC)** solutions are additional layers of abstraction on top of the FPGA fabric to increase the runtime flexibility [GHN+12, PDM12, CS13, CHM+14, LNS15] and to make the communication of spatially laid-out processing units more streamlined [KG15, FYAE14]. Govindaraju et al. [GHN+12] propose a switching network of functional units to dynamically create pipelines for a specific task, where the overhead of the network limits the prototype to 4 functional units. NoC solutions such as Hoplite [KG15] are complementary to PipeArch: One of our limitations is the static network between hardware subroutines and on-chip memory regions. Although providing high bandwidth and low latency, this limits functionality; so connecting them using an NoC might be an interesting approach in making PipeArch more generic, without sacrificing much performance. This is an aspect left for future work.

**6. Specialized hardware for machine learning** is being widely used both for training [JYP+17, KAA+17, MKS+18, W+19] and inference [OZZA17, UFG+17, CHZ+19] mainly because general purpose processing has difficulties satisfying the compute/data intensive nature of machine learning. Mahajan et al. [MKS+18] propose a framework (DAnA) for generating FPGA-based accelerators to perform in-database machine learning. Their ISA-based design provides high programmability and enables tight integration with a database thanks to efficient page accesses. PipeArch favors more specialization in the hardware design, so we are able to achieve higher performance across all workloads (detailed comparison to related work in Section 6.8.1). Our runtime scheduling capabilities are also interesting from a database integration perspective and combined with page access techniques from DAnA, it can be a more complete in-database machine learning platform. A recent interesting use case of FPGAs has been shown by Chung et al. [CFO+18] focusing on low-latency neural network inference rather than training as in PipeArch. They propose a distributed architecture utilizing the large-scale FPGA deployment of Microsoft in their datacenters [PCC+14], and showcase the architectural flexibility of FPGAs in customizing the numerical format for the core computation.

## 6.4 Target Platforms and Setup

PipeArch is designed to be portable across FPGA platforms with different characteristics. We deploy it on two platforms, shown in Figure 6.2: Intel Xeon+FPGA and PCIe-attached Xilinx VCU1525, as introduced in Section 2.2. The portability is achieved thanks to the encapsulation of low-level device specific memory management and control in a template class we call *iFPGA* (Figure 6.1), which uses *Open Programmable Acceleration Engine (OPAE)* [opa] for the Xeon+FPGA, and SDAccel [Wir14] for VCU1525.

**Intel Xeon+FPGA v2** [OSC+11] combines an FPGA (Arria 10) with a 14-core Broadwell Xeon CPU in the same package. Thanks to the shared memory architecture, the FPGA can directly work on the main copy of the data without the need for additional copying, as in discrete PCIe-attached FPGA cards. Its disadvantage is the limited memory bandwidth (19 GB/s), usually the bottleneck for data intensive applications.

**Xilinx VCU1525** is a PCIe-attached FPGA card with a VU9P Ultrascale+ FPGA [vcu]. The FPGA has 4 DRAM banks, with each DDR providing around 16 GB/s read bandwidth, resulting in a total read bandwidth of 64 GB/s to 64 GB of memory, if each bank can be utilized fully. The disadvantage is the need to copy data to the FPGA-attached memory first. This problem is less pronounced for the training phase in machine learning, because the input data is accessed iteratively; so the data is copied once, but used multiple times.

**Differences.** The setup on the two target platforms is different, because of the non-uniform (bank-wise) memory access on VCU1525. A PipeArch-RT instance manages threads running on PipeArch-PU instances, which access memory *uniformly*. This is needed because a migrated thread must be able to access its memory region from the new instance it has been migrated to. In Figure 6.2 the resulting difference in setups is highlighted: One PipeArch-RT instance is enough on the Xeon+FPGA, because all PipeArch-PU instances access memory uniformly via an arbiter. On VCU1525 however, each PipeArch-PU instance is connected to its own DRAM bank. Since there are 4 DRAM banks, there are also 4 PipeArch-RT instances on the CPU to monitor threads. Due to non-uniformity, thread migration is not performed on VCU1525.

The number of PipeArch-PU instances on each FPGA depends on the resource consumption. The limiting resource in both cases is the amount of on-chip memory (BRAM), allowing us to put 2 instances on Xeon+FPGA and 4 instances on VCU1525. Because of the non-uniform memory access on VCU1525, different combinations of how we connect

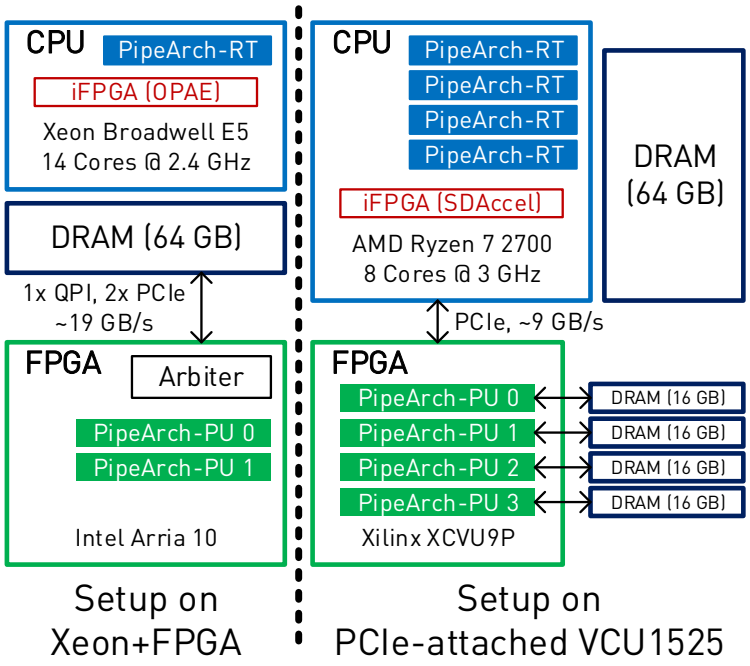



Figure 6.2: PipeArch setup on both target platforms: Intel Xeon+FPGA and PCIe-attached VCU1525.

4 available instances to 4 memory banks are available: For instance, as opposed to connecting each PipeArch-PU instance to its own bank (Figure 6.2), we could connect all 4 instances to just 1 bank to achieve uniformity, but resulting in a decrease in data access bandwidth, now 4 times less. Since machine learning workloads tend to be data intensive, we choose to connect each instance to its own bank, optimizing for bandwidth.

## 6.5 PipeArch Processing Unit (PipeArch-PU)

In this section we explain the design principles behind PipeArch-PU in detail, that help us reach programmability combined with high performance and reduced development effort for a variety of FPGA-based acceleration efforts.

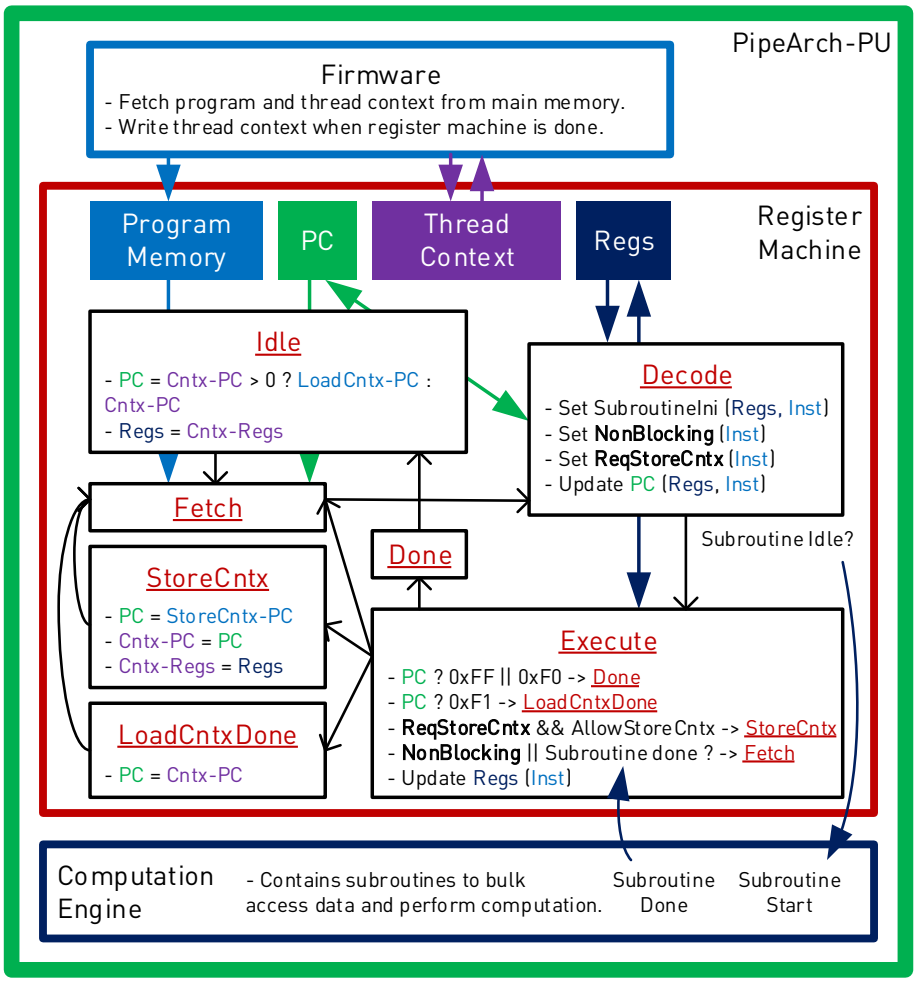



Figure 6.3: A PipeArch-PU instance with a focus on the Register Machine, its states, transitions, and data structures.

### 6.5.1 PipeArch-PU Register Machine

At the heart of a PipeArch-PU instance is a Register Machine (RM) executing programs. Figure 6.3 shows a PipeArch-PU with a focus on the RM's states, transitions and data structures. We design the RM focusing on 2 capabilities while keeping other aspects simple: (1) asynchronous execution for the subroutines and (2) runtime scheduling of threads.

**Execution Overview.** At the start, the RM is in the Idle state. The RM transitions to the Fetch when a program is fully written to the program memory and the thread

context is read. Both the program and the thread context reside in the address space in main memory that a PipeArch-PU instance is assigned to, and their transfer is performed by the Firmware as in Figure 6.3. Fetch reads instructions from the program memory and transitions to Decode. Decode constructs the subroutine initiation (*SubroutineIni*) from the current instruction and registers. This is where we can for instance calculate the starting address for a bulk memory access, combining a register (used as an index) and part of the instruction (used as an offset). At Decode the program counter (PC) is updated based on current registers and instruction, allowing us to implement branches and loops via jumps. Decode blocks until the requested subroutine becomes idle, then transitions to Execute. Execute transition either to Done, StoreCntx, or LoadCntxDone, depending on the current PC and whether a context-switch has been requested. At Execute the registers are updated according to the current instruction: Integer addition and subtraction can be performed on the registers, which are mainly used as indexes (e.g, sample, minibatch, epoch). Eventually the program reaches the terminating PC (0xFF) and RM transitions to Done, at which point the Firmware writes the thread context back to main memory.

**Asynchronous Execution.** Unless a subroutine is initiated with the *NonBlocking* flag, Execute blocks until the subroutine is done. Otherwise the RM can immediately transition to Fetch to continue program execution. This is what allows multiple subroutines to be active at the same time and is an enabler for pipeline parallelism between multiple subroutines. Notably, we need to maintain correctness while allowing asynchronous execution. This is ensured by the program, allowing asynchronous execution when the produced-consumer relationships between subroutines are suited, and by blocking at the right instruction to ensure synchronization. We discuss this in more detail by the examples of implementing SGD, SCD and LRMF in Section 6.7.

**Context-Switch Capability.** A thread starting execution on the RM will restore its context that has been previously stored. This starts at Idle, checking whether the PC in the thread context (*Cntx-PC*) is larger than 0: If true, the program jumps to a group of subroutines (at *LoadCntx-PC*) that restores the state of the Computation Engine from the main memory. At the end of this group, the RM will land on LoadCntxDone and continue normal execution of the thread. When context-switch is requested, the RM will transition to StoreCntx after Execute whenever possible. Here, the program will jump to a group of subroutines (at *StoreCntx-PC*) that stores the state of the Computation Engine to the main memory. At the end of this group, the RM will land on Done, allowing the start of another thread, completing the context-switch.

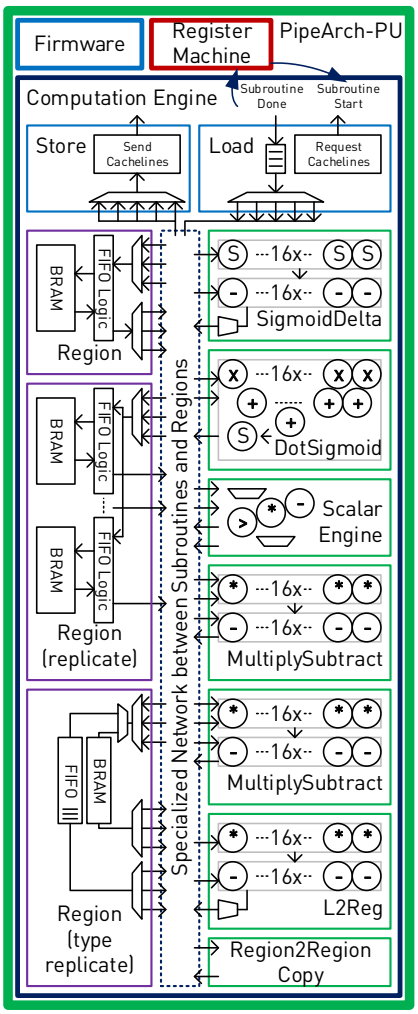



Figure 6.4: A PipeArch-PU instance with a focus on the Computation Engine, its subroutines and regions. With the shown engine a wide variety of machine learning tasks can be completed with high performance.

### 6.5.2 PipeArch-PU Computation Engine

**Overview.** The computation engine is a collection of *subroutines*, on-chip memory *regions*, and a specialized network between these subroutines and regions to cater for producer-consumer relationships to implement the machine learning algorithms of interest, as shown in Figure 6.4.

There are 2 subroutines to access main memory: Load reads data from the main memory

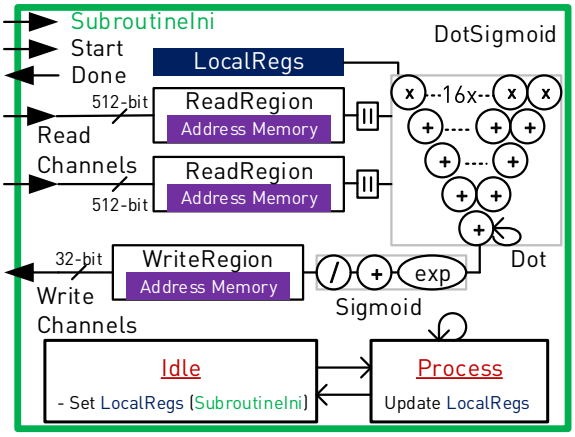



Figure 6.5: DotSigmoid subroutine design, with its interfaces, computation pipeline and internal state.

and writes to the regions, and Store does the opposite. In the context of machine learning, they are mainly used for reading training data and for writing trained models back. These subroutines are connected to all regions; Load can write data coming from the main memory to any of the regions in a broadcast fashion. Store can read data from only one region at a time, since there is one physical channel to the main memory for writing.

There are 6 computation subroutines; each designed to perform a compute intensive part of an algorithm with high efficiency thanks to 512-bit vectorization and internal pipelining. The 512-bit wide inputs and outputs are directly connected to one of the regions via the specialized network, creating data communication channels between subroutines.

**Subroutines** are functional modules on the FPGA, that perform either memory access or vectorized computation. They are invoked by the register machine in a blocking or non-blocking manner as explained in Section 6.5.1. The DotSigmoid subroutine is shown in Figure 6.5 as an example. The control interface consists of Start, Done, and *SubroutineIni* signals. *SubroutineIni* may contain different information for each subroutine: For instance, Load requires memory address and length, whereas DotSigmoid requires vector dimensionality and whether sigmoid after dot-product is performed. Upon initiation, the internal state transitions from Idle to Process after setting the *LocalRegs* according to *SubroutineIni*.

*SubroutineIni* also contains the information about the number of iterations a subroutine will be repeated and how *LocalRegs* will be updated for each iteration. The capability of initiation *with iterations* is particularly useful to eliminate looping overheads for dense

computation blocks, which are what subroutines mainly perform. For instance, we initiate DotSigmoid with iterations for minibatch SGD, since in that algorithm the same operation needs to be performed on a minibatch of samples, creating high computation density.

Each subroutine accesses data from regions via read/write channels that are up to 512-bit wide. A subroutine can have as many read/write channels as necessary. For each read/write channel there is a corresponding *ReadRegion* and *WriteRegion* module with its own address memory (Figure 6.5). These modules are designed to access data from regions either assuming a RAM or a FIFO interface. Thanks to the dedicated address memory, they can perform complex access patterns, which is needed for sparse computation tasks, for instance at LRFM. The address memories of *ReadRegion*/ *WriteRegion* modules are populated by Load, so arbitrary complex access patterns can be realized. The computation a subroutine performs can be complex, combining vectorization and deep pipelining: In the example of DotSigmoid (Figure 6.5) a dot-product is performed in a vectorized manner on 16 single-precision floating-point values followed by a sigmoid function, in a fully pipelined fashion. Clocked at 200 MHz, the processing rate for DotSigmoid is 25.6 GB/s, consuming data from 2 channels.

**Regions** are on-chip memory modules using block RAM (BRAM) resources on the FPGA. They can have multiple inputs and outputs, each up to 512-bits wide. They provide a specialized on-chip cache, tuned to the producer-consumer relationships of the subroutines. The regions can be configured depending on the needs of subroutines as (1) regular, (2) replicated, and (3) type replicated, as depicted in Figure 6.4.

In the regular mode, physically only one BRAM is used that can be interfaced either in a FIFO mode or just as a regular 64 B addressed memory. The advantage is the conservative usage of BRAM resources, however only one input and one output channel can be active at a clock cycle, so the region is accessed in a time-shared manner. In the replicated mode, as many BRAMs are used as read channels. All writes are replicated to all BRAMs. The advantage is that all read channels can be used simultaneously, which is beneficial in case multiple subroutines need to consume the same data. Finally, in the type replicated mode, we have separate BRAM and FIFO instantiations, allowing read and write channels to use the region either in FIFO or BRAM modes simultaneously.

Table 6.1: Region access behavior of *ReadRegion*/*WriteRegion*.

| regionaccess[31:0] | | | | | |
|---|---|---|---|---|---|
| BRAM | FIFO | Length in 64 B | Keep Count at Iterations | Use Address Memory | Offset in 64 B |
| [31] | [30] | [29:16] | [15] | [14] | [13:0] |

Table 6.2: Examples of subroutine instructions

| **Subroutine** | **Instruction (regionaccess)** | **Instruction (specific)** |
|---|---|---|
| Load | out[6] | offset; length; localoffset |
| Store | in[3] | offset; length; localoffset |
| SigDelta, DotSig | in[2],out[1] | dimension; sigmoid; iterations |
| ScalarEngine | in[1],out[1] | type; algo; stepsize; iterations |
| MultiplySubtract | in[2],out[2] | dimension; iterations |
| L2Reg | in[2],out[2] | dimension; regularization |

### 6.5.3 Programming PipeArch-PU

PipeArch programs consist of a set of subroutine and control instructions. Subroutine instructions consist of subroutine-specific information and directives for data access from regions: *regionaccess* as shown in Table 6.1. Examples of subroutine instructions are in Table 6.2. Each instruction can be made either blocking or non-blocking. Correctness needs to be ensured by the programmer; hazards that might happen due to asynchronous execution has to be avoided while writing programs via blocking calls when necessary.

## 6.6 PipeArch Runtime Manager (PipeArch-RT)

PipeArch-RT is a two-threaded application on the host CPU. The first (helper) thread listens to incoming PipeArch-PU programs submitted by users, and puts them into a queue. The second (main) thread dynamically schedules requested programs on available PipeArch-PU instances on the FPGA. It continuously monitors which threads are running, idle, waiting for execution, and finished.

**Scheduling.** 3 scheduling mechanisms are implemented by PipeArch-RT: first-come-first-served (FCFS), round-robin (RR), and shortest-job-first (SJF); of which RR and SJF take advantage of context-switching. FCFS completes programs without context-switches in

Table 6.3: Machine learning algorithms in PipeArch.

| ID | Model | Regularizer | Optimizer |
|---|---|---|---|
| Ridgereg | Ridge Regression | L2 | SGD |
| LogregL2 | Logistic Regression | L2 | SGD |
| Lasso | Lasso | L1 | SCD |
| LogregL1 | Logistic Regression | L1 | SCD |
| LRMF | Low-Rank Matrix Factor. | L2 | Alt. SGD |

the order they are submitted. RR preempts the active thread as soon as possible and schedules threads in a round-robin fashion, without any priorities. SJF gives the program with the shortest runtime the highest priority. Whenever a new program is submitted with a shorter total runtime than the remaining runtime of the active program, the latter will be preempted. The runtime of programs is approximated by their content and how much data they consume, since this is statically available knowledge.

**Context-Switch.** Runtime scheduling requires the capability to store and restore the context of the threads running on PipeArch-PU. In Section 6.5.1 we explained the mechanisms enabling this in hardware. With those in place, implementing context-switches by PipeArch-RT is simple: It sets a status register in the PipeArch-PU, requesting a context switch. When the active thread completes the procedure of storing its entire context (status and intermediate data), it notifies PipeArch-RT. Then, a new thread can be started. The overhead caused by the context-switch depends mainly on the size of the intermediate data a program keeps. For the machine learning algorithms we implement, this is usually the model being trained.

**Thread Migration.** In the case where multiple PipeArch-PU instances are maintained by PipeArch-RT, the latter also can perform thread migration to balance load. Once the context of a thread is stored, it is straightforward to continue its execution on another PipeArch-PU instance the next time it will be scheduled for execution.

## 6.7 Machine Learning on PipeArch

We implement generalized linear model training and low-rank matrix factorization (Table 6.3) using the capabilities of the PipeArch-PU presented in Section 6.5.

**Generalized Linear Models (GLM)** are widely used in regression and classification

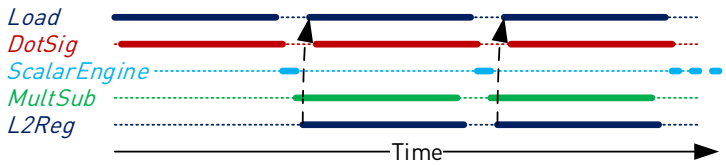



Figure 6.6: SGD inner-loop execution

tasks, also applied often in a transfer learning setting [PY09], where the last layer of a neural network is retrained on a new task. In this category we aim at solving the optimization problems of the following form:

$$\min_{\mathbf{x}\in\mathbb{R}^n} \left( \frac{1}{m} \sum_{i=1}^{m} J(\langle \mathbf{x}, \mathbf{a}_i \rangle, b_i) \right) + \underbrace{\lambda_1 \|\mathbf{x}\|_1}_{\text{For L1-reg}} + \underbrace{\lambda_2 \|\mathbf{x}\|_2^2}_{\text{For L2-reg}} \tag{6.1}$$

$$J = \begin{cases} \frac{1}{2}(\langle \mathbf{x}, \mathbf{a}_i \rangle - b_i)^2 & \text{for Rigdereg and Lasso} \\ -b_i \log(h_{\mathbf{x}}(\mathbf{a}_i)) - (1 - b_i) \log(1 - h_{\mathbf{x}}(\mathbf{a}_i)) & \text{for Logreg} \end{cases} \tag{6.2}$$

$h_{\mathbf{x}}(\mathbf{a}_i) = 1/(1 + \exp(-\langle \mathbf{x}, \mathbf{a}_i \rangle))$ is the sigmoid function.

where $(\mathbf{a}_1, b_1), ..., (\mathbf{a}_m, b_m) \in ([-1, 1]^n \times \mathbb{R})$ is a set of samples and $J : \mathbb{R}^n \times \mathbb{R} \to [0, \infty)$ is a non-negative convex loss function. Lasso and Ridge Regression are for regression tasks, and Logistic Regression is for classification tasks. Lasso and LogregL1 are solved using stochastic coordinate descent (SCD) [SST11], since these models are regularized with the L1-norm and favor sparsity, which SCD is efficient at optimizing. RigdeReg and LogregL2 are solved using stochastic gradient descent (SGD) [Zha04] with the ability to vary minibatch sizes.

**SGD** implementation for Rigdereg and LogregL2 is shown in Algorithm 9. The **black** text is pseudocode showing how the algorithm is implemented logically and the blue text is showing simplified PipeArch code to present how we implement the algorithm on PipeArch. At the beginning, we load the model to one of the on-chip memory regions with Load. We then perform so-called epochs (a complete scan of the entire dataset) to train the model. We process data in so-called partitions of size $P$. This is required because the size of the on-chip memory regions are limited. For instance, from the $b_{1...m}$ vector (labels) we only load $P$ values at a time. The partition size also dictates the frequency with which context-switching is allowed by Yield().

In the innermost loop is the core computation of SGD. Here, the tight producer-consumer relationships and deep pipelining capabilities in PipeArch-PU become important: All

1 **Initialize:**
2 · $\mathbf{x} = 0$, step size $\alpha$, partition size $P$
3 · $S(\mathbf{z}) = \begin{cases} \mathbf{z} & \text{for Ridgereg} \\ 1/(1+\exp(-\mathbf{z})) & \text{for LogregL2} \end{cases}$
4 $Load(xAddr, \mathbf{x})$
5 **for** *epoch = 1, 2, ...* **do**
6  **for** $p$ = *1, ..., m/P* **do**
7  $offset = p \cdot m/P$
8  $Load(\mathbf{a}_{offset+1...P})$, $Load(bAddr, b_{offset+1...P})$
9  **for** $i$ = *offset + shuffle(1, ..., P)* **do**
10  $dot = S(\langle \mathbf{x}^t, \mathbf{a}_i \rangle)$ { $Load(aAddr, \mathbf{a}_i)$; $DotSig(dot, xAddr, aAddr, ...)$; $Block()$
11  $dot = \alpha(dot - b_i)$ { $ScalarEngine(dot, \alpha, bAddr, ...)$
12  $\mathbf{x}^{t+1} = \mathbf{x}^t - dot \cdot \mathbf{a}_i^T$ { $MultiplySubtract(xAddr, dot, aAddr, ...)$
13  $\mathbf{x}^{t+1} = \mathbf{x}^{t+1} - 2\alpha\lambda_2\mathbf{x}^t$ { $L2Reg(xAddr, \alpha, \lambda_2, ...)$
14  **end**
15  $Yield()$
16  **end**
17  $Store(\mathbf{x}, xAddr)$
18 **end**
19 $StoreContext:$ $Store(\mathbf{x}, xAddr)$
20 $LoadContext:$ $Load(xAddr, \mathbf{x})$

**Algorithm 9:** SGD for Rigdereg and LogregL2

subroutines in the innermost loop are initiated in a non-blocking way except for DotSig, whose result needs to be ready before moving on. This leads to high compute density as illustrated in the timing diagram in Figure 6.6: In the core inner loop, all important subroutines are active at the same time; Load consuming the newly produced model by L2Reg in a pipelined manner. This is one example about what allows PipeArch to perform dense computation tasks with high performance.

**SCD** updates the model one feature—coordinate—at a time as shown in Algorithm 10, as opposed to updating the entire model for each sample as in SGD. This leads to a column-wise access of the dataset as opposed to row-wise access in SGD. The column-wise access can be beneficial in systems that store data in columnar format such as in-memory databases [IGN+12]; being able to run both SGD and SCD on the same PipeArch-PU instance showcases the programmability and flexibility in accessing the data. In Algorithm 10, the data is accessed again in partitions of size $P$ to fit on-chip memory regions and in the innermost loop we again take advantage of non-blocking initiation of subroutines. In SCD, we do not need StoreContext or LoadContext groups, because at the

**Initialize:**
· $\mathbf{x} = 0$, $\mathbf{z} = 0$, step size $\alpha$, partition size $P$
· $S(\mathbf{z})$ as in Algorithm 9
· $T(x_j, g_j) = \begin{cases} \alpha g_j + \alpha\lambda_1 & x_j - \alpha g_j > \alpha\lambda_1 \\ \alpha g_j - \alpha\lambda_1 & x_j - \alpha g_j < -\alpha\lambda_1 \\ x_j & \text{else (to set } x_j = 0) \end{cases}$
**for** *epoch = 1, 2, ...* **do**
  **for** *p = 1, ..., m/P* **do**
    $Load(xAddr, \mathbf{x})$, $Load(zAddr, \mathbf{z}_p)$, $Load(bAddr, \mathbf{b}_p)$
    **for** *j = shuffle(1, ..., n)* **do**
      $g = (S(\mathbf{z}_p) - \mathbf{b}_p) \cdot \mathbf{a}_{p,j}$ { $Load(aAddr, \mathbf{a}_{p,j})$ / $SigDelta(zAddr, bAddr, ...)$ / $DotSig(g, aAddr, ...)$ / $Block()$ }
      $\mu = T(x_j, g)$
      $x_j = x_j - \mu$ { $ScalarEngine(\mu, \alpha, g, xAddr, ...)$
      $\mathbf{z}_p = \mathbf{z}_p - \mu\mathbf{a}_{p,j}$ { $MultiplySubtract(zAddr, \mu, aAddr, ...)$
    **end**
    $Store(\mathbf{x}, xAddr)$
    $Store(\mathbf{z}_p, zAddr)$
    $Yield()$
  **end**
**end**

**Algorithm 10:** SCD for Lasso and LogregL1

point where we allow context-switching (end of a partition) all intermediate data has been already stored to the main memory.

**Low-Rank Matrix Factorization (LRMF)** is used to factorize a large sparse matrix into two smaller matrices, to extract latent variables and complete the empty entries in the original sparse matrix (e.g., users' ratings of movies). This method is used in recommender systems and was part of the top solution in the Netflix challenge [BL+07]. The optimization problem is:

$$\min_{\mathbf{M}\in\mathbb{R}^{m\times d}, \mathbf{U}\in\mathbb{R}^{u\times d}} \|\mathbf{M}\mathbf{U}^T - \mathbf{X}\|_F^2 + \lambda\|\mathbf{M}\|_F^2 + \lambda\|\mathbf{U}\|_F^2 \tag{6.3}$$

where $\mathbf{X} \in \mathbb{R}^{m\times u}$ is a sparse input matrix with $x$ non-zero entries that needs to be approximated. $\mathbf{M} \in \mathbb{R}^{m\times d}$ and $\mathbf{U} \in \mathbb{R}^{u\times d}$ are the low-dimensional matrices ($d \ll min(m, u)$), that the optimization algorithm aims to find. The Algorithm 11 shows the logical and the simplified PipeArch-PU implementations of LRMF. We process matrices in tiles of size $T$ to fit on-chip memory regions. The main challenge in implementing LRMF is the sparse access to both $\mathbf{M}$ and $\mathbf{U}$ tiles. This is possible in PipeArch-PU thanks to address memories in modules where we read from on-chip memory regions. Accordingly, we need to

**Initialize:**
· $\mathbf{M} = rand^{m \times d}$, $\mathbf{U} = 0$, step size $\alpha$, tile size $T$
**for** *epoch = 1, 2, ...* **do**
  **for** $t_m$ *= 1, ..., m/T* **do**
    $Load(AddressRegs, Sparse\ Indexes\ at\ t_m(\mathbf{M}, \mathbf{U}, \mathbf{X}))$, $Load(MAddr, \mathbf{M}_{t_m})$
    **for** $t_u$ *= 1, ..., u/T* **do**
      $Load(UAddr, \mathbf{U}_{t_u})$, $Load(XAddr, \mathbf{X}_{t_m,t_u})$
      $Block()$
      **for** $x_{row}, x_{col}, x_{value}$*: non-zero entries in* $\mathbf{X}_{t_m,t_u}$ **do**
        $error = \alpha(\langle \mathbf{M}_{x_{row}}, \mathbf{U}_{x_{col}} \rangle - x_{value})$ { $DotSig(MAddr, UAddr, ...)$ / $ScalarEngine(error, XAddr, ...)$
        $\mathbf{M}_{x_{row}} = \mathbf{M}_{x_{row}} - (error \cdot \mathbf{U}_{x_{col}} + \lambda \mathbf{M}_{x_{row}})$
        { $MultiplySubtract(MAddr, error, UAddr, ...)$
        $\mathbf{U}_{x_{col}} = \mathbf{U}_{x_{col}} - (error \cdot \mathbf{M}_{x_{row}} + \lambda \mathbf{U}_{x_{col}})$
        { $MultiplySubtract(UAddr, error, MAddr, ...)$
      **end**
      $Block()$
      $Store(\mathbf{U}_{t_u}, UAddr)$
    **end**
    $Store(\mathbf{M}_{t_m}, MAddr)$
    $Yield()$
  **end**
**end**

**Algorithm 11:** Tiled Alternating SGD for LRMF

populate these address memories via Load with the corresponding addresses given by the location of entries in the target matrix $\mathbf{X}$. The innermost loop again takes advantage of pipelining subroutines, where no blocking is necessary, because updates to matrix entries are independent. Thus, we only block before and after the innermost loop.

## 6.8 Evaluation

We evaluate our system on both target platforms with datasets presented in Table 6.4, covering a wide range of workloads including real-world use cases.

- AEA [aea] (Amazon employee access/denial prediction) and KDD [kdd] (KDD Cup 2014) are datasets from *Kaggle* competitions. We use the features Liu et al. [LZZ+18] extracted from these and train the linear models directly on them.
- IM is a large-scale multi-class classification task we created with transfer learning [PY09] in mind: We run *InceptionV3* [SVI+16] neural network on ImageNet to

Table 6.4: Datasets used in the evaluation.

| Name | # Samples | # Features | # Classes | Size | Model |
|---|---|---|---|---|---|
| AEA | 32,769 | 126 | binary | 17 MB | LogregL1 |
| KDD | 131,329 | 2,330 | binary | 1,224 MB | LogregL1 |
| IM | 166,400 | 2,048 | 16 | 1,363 MB | LogregL2 |
| MNIST | 60,000 | 784 | 10 | 188 MB | LogregL2 |
| SYN_Logreg | 524,288 | 1,024 | binary | 2,147 MB | LogregL2 |

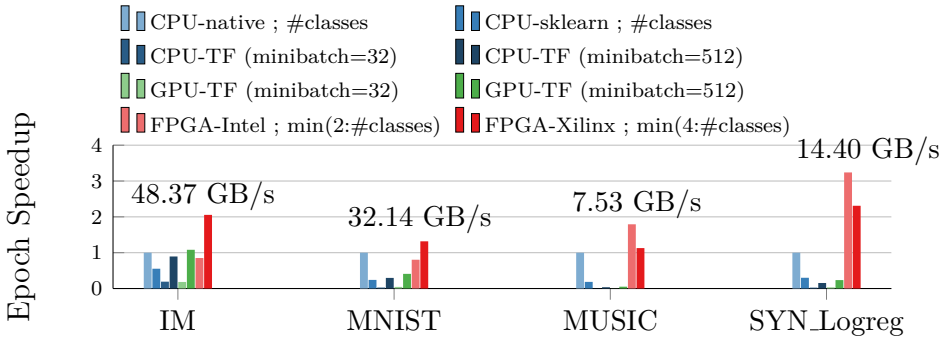
| Name | # Samples | # Features | Size | Model |
|---|---|---|---|---|
| MUSIC | 448,000 | 90 | 161 MB | Ridgereg |
| SYN_Lasso1 | 33,554,432 | 16 | 2,147 MB | Lasso |
| SYN_Lasso2 | 2,097,152 | 256 | 2,147 MB | Lasso |

| Name | $m$ | $u$ | $d$ | $x$ | Size | Model |
|---|---|---|---|---|---|---|
| NETFLIX | 4,499 | 470,758 | 32 | 24M | 350 MB | LRMF |
| SYN_LRMF1 | 4,096 | 524,288 | 32 | 50M | 715 MB | LRMF |
| SYN_LRMF2 | 4,096 | 524,288 | 32 | 100M | 1,366 MB | LRMF |

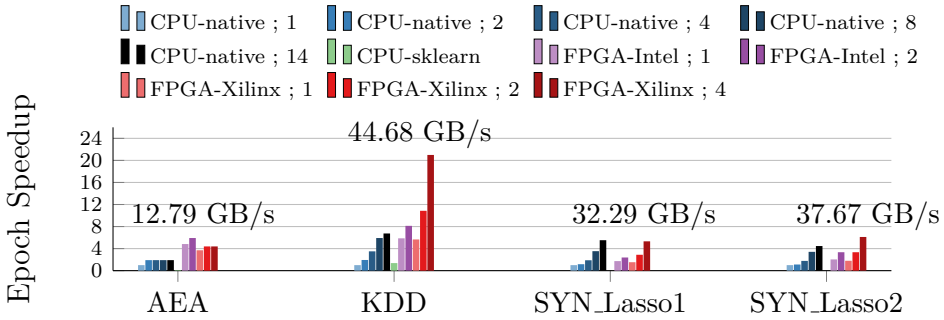
extract 2048 features from images of different classes. The task is to train the final layer responsible for the classification.

- MUSIC [mus] is a regression task to predict the year a song was released in, given certain extracted features from them.
- NETFLIX [BL+07] is a dataset of anonymous users' ratings of movies, released for a competition. It was won by a team utilizing LRMF as one of their primary methods.
- We also use various synthetic datasets to show different characteristics of our system.

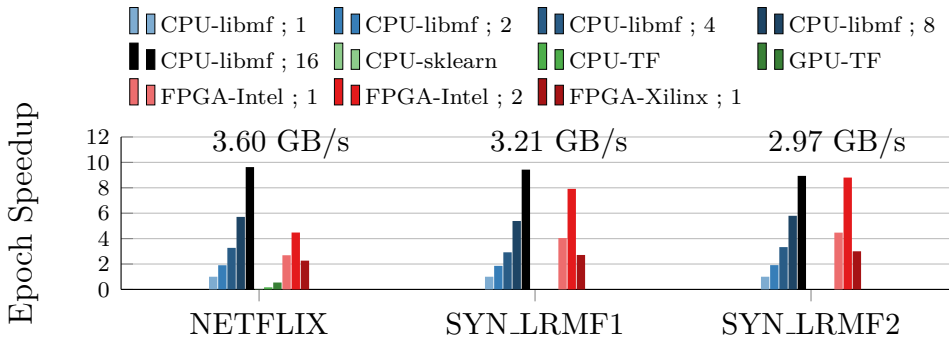
**Evaluation Setup.** For the baseline experiments we use (1) the 14-core Xeon Broadwell (in the Xeon+FPGA platform) and (2) the NVDIA QUADRO M6000 with 24 GB of memory attached to an 8-core Intel i7-5960X. On the CPU, we use *gcc 5.4* and compile with "*-O3 -march=native*". The frequency governor is set to *performance*, running at 3.2 GHz during program execution. For the GPU baselines, we use Tensorflow (v1.11) to implement the target algorithms. Tensorflow uses CUDA libraries to utilize the GPU.




(a) SGD workloads. Notation for legend labels: *platform (properties) ; number of threads*. Except for 2 Tensorflow numbers (CPU-TF and GPU-TF), SGD minibatch equals to 32.




(b) SCD workloads. Notation for legend labels: *platform ; number of threads*. Partition size equals to 16384 for all experiments.




(c) LRMF workloads. Notation for legend labels: *platform ; number of threads*. Tile size equals to 256 for all experiments.

Figure 6.7: Individual workload performance showing the speedup for epoch runtime, plotted relative to the workload and the dataset. The processing rate ($dataset\ size / epoch\ runtime$) is given for the largest bar as an absolute reference.

### 6.8.1 Individual Workload Performance

First, we analyze the individual workload performance categorizing them according to the optimization algorithm: SGD, SCD, and LRMF. For each algorithm we perform the same number of epochs (a complete scan of the dataset) and compare runtimes normalized over the CPU run. Since the optimization algorithms used are the same, the convergence behavior (e.g., statistical efficiency) is similar across platforms, so that the same number of epochs progress the optimization problem similarly. Results are presented in Figure 6.7.

**SGD.** We solve LogregL2 and Ridgereg models with SGD. For multiclass datasets (IM, MNIST) we use one-vs-all method and parallelize by creating a separate SGD thread for each class. As a baseline, we use our own AVX-optimized implementation, the scikit-learn function *SGDClassifier*, and Tensorflow *GradientDescentOptimizer* running either on the CPU or the GPU. In Figure 6.7a we observe for parallelizable problems (IM, MNIST), FPGA-Xilinx reaches peak performance thanks to 4 PipeArch instances connected to their own DRAM banks giving high aggregate bandwidth. The GPU runtimes do not provide much speedup over the CPU, since the GPU is underutilized (on average 27%, reported by nvidia-smi) due to the models being relatively small leading to limited many-core parallelism. Also, the overhead from Tensorflow is detrimental especially for a minibatch size of 32, which we normally use for other CPU and FPGA runs, so that we had to increase the minibatch to 512 for the GPU runs. For binary classification and regression problems (MUSIC, SYN_Logreg), FPGA-Intel reaches a higher throughput, because only one PipeArch instance can be utilized and the instances on FPGA-Intel are clocked faster (300 MHz vs 200 MHz).

**SCD.** We solve LogregL1 and Lasso models with partitioned SCD, leading to good parallelism if the dataset is large. As the baseline we use the AVX-optimized multi-core implementation from related work [KEZA18] and *LogisticRegression* function from scikit-learn with liblinear solver (Figure 6.7b). For KDD, FPGA-Xilinx reaches almost peak performance again thanks to utilizing 4 PipeArch instances, 2x faster than FPGA-Intel, which also reaches its memory bandwidth at around 17 GB/s with 2 PipeArch instances. For Lasso models, the CPU is also fast since there is no sigmoid function in the gradient computation, making the algorithm less compute intensive. The reason why FPGA is slower at these models is the smaller dimensionality compared to KDD, leading to more frequent memory transfers during an epoch.

**LRMF.** For the baseline (Figure 6.7c), we use the optimized multi-core implementation

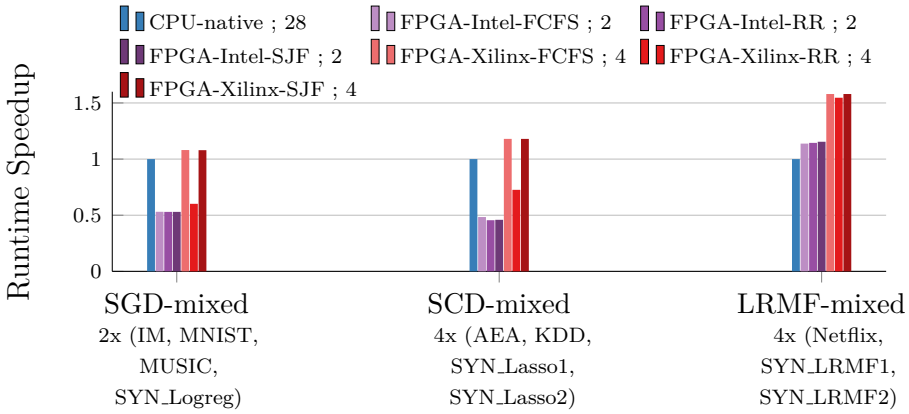



Figure 6.8: Mixed workload runtime comparison. All jobs are submitted at the same time and total runtime is measured. For FPGA runs, different scheduling policies are used. Notation for legend labels: *platform ; number of cores/instances*

from LIBMF [CZJL15]. The scikit-learn and Tensorflow implementations are performed only as a sanity check for convergence, however their performance is quite low mainly because these frameworks are not optimized to work on sparse problems. The alternating SGD method for solving LRMF can be parallelized [RR13], however intermediate state exchange is necessary among workers during optimization. With FPGA-Intel, we can do this because both PipeArch instances access the same shared memory, so no-copy data exchange is possible. However, with FPGA-Xilinx this is not possible because each PipeArch instance has access to its own DRAM bank. Therefore, FPGA-Intel reaches a higher performance at solving a single LRMF problem (FPGA-Xilinx can solve 4 independent problems in parallel as we show later). The CPU is faster than both FPGA solutions at this problem when all cores are utilized, mainly because it has a larger cache (25 MB) compared to FPGA-Intel (4 MB): The CPU can use larger tile sizes and perform efficient sparse access on more data. For this reason, the denser the problem, the better the FPGA comparison becomes (SYN_LRMF1, SYN_LRMF2), because the FPGA performs more computation with each tile.

**Comparison to Related Work.** To the best of our knowledge, PipeArch is the first specialized design to support the presented combinations of optimization algorithms/models. We compare our performance to recent FPGA-based designs targeting these algorithms. Wang et al. [W+19] propose a highly specialized design for SGD on the Xeon+FPGA, focusing on quantized input data. We surpass their performance (˜15 GB/s) for up to 3.2x with 32-bit input data even with a more generic design, mainly thanks to our deployment

on VCU1525. Mahajan et al. [MKS+18] implement SGD and LRMF on the same Xilinx FPGA (VU9P) as this work, however the processing rate is not reported. Thanks to the authors' response to our inquiry, we calculate the maximum processing rate they reach for SGD and LRMF to present as fair a comparison as possible: ˜3.76 GB/s for SGD and ˜0.79 GB/s for LRMF; we can surpass these up to 12.9x and 3.8x respectively. Kara et al. [KEZA18] present an FPGA-based SCD design, reaching up to ˜18 GB/s on the Xeon+FPGA, which we match on the same platform and surpass by 2.5x using VCU1525.

### 6.8.2 Mixed Workload Performance

For the mixed workload experiments multiple jobs are submitted to the PipeArch-RT at the same time and measure total and per-job completion times. This allows us to evaluate the system from a throughput-oriented perspective and also analyze the effect of different scheduling policies.

**Throughput.** Figure 6.8 shows the total runtime speedup of mixed workloads (grouped by optimization algorithm). Both FPGA solutions saturate their respective memory bandwidths with first-come-first-served (FCFS) scheduling policy. With that, FPGA-Intel is around 2x slower compared to using the entire 14-core Xeon CPU (2 threads per core) for SGD and SCD workloads. FPGA-Xilinx is slightly faster thanks to its higher aggregate bandwidth with 4 DRAM banks. A key observation is the slowdown at FPGA-Xilinx when round-robin (RR) scheduling is used. The reason for this is the higher cost of context-switching by PipeArch instances on this platform: The context is first stored at the FPGA-local DRAM, then it is read from it and transferred to the CPU memory via PCIe. This latency (˜810 $\mu$s) caused mainly due to the memory model of OpenCL and PCIe-based communication is around 7x higher compared to the context-switch latency on the Intel platform (˜117 $\mu$s), making the round-robin policy inefficient on FPGA-Xilinx. However, shortest-job-first (SJF) policy behaves well on both platforms leading to high throughput. For LRMF, now that 4 independent jobs can be executed in parallel on FPGA-Xilinx, we see a total runtime advantage of around 1.5x compared to the CPU. This could not be achieved for a single LRMF job due to the nonuniform access to 4 DRAM banks as explained in Section 6.8.1

**Individual Completion Times.** Figure 6.9 plots histograms showing the number jobs from a mixed workload (48 jobs of 3 categories with varying lengths) completed in a certain time interval. The distribution of completion times depends on the scheduling

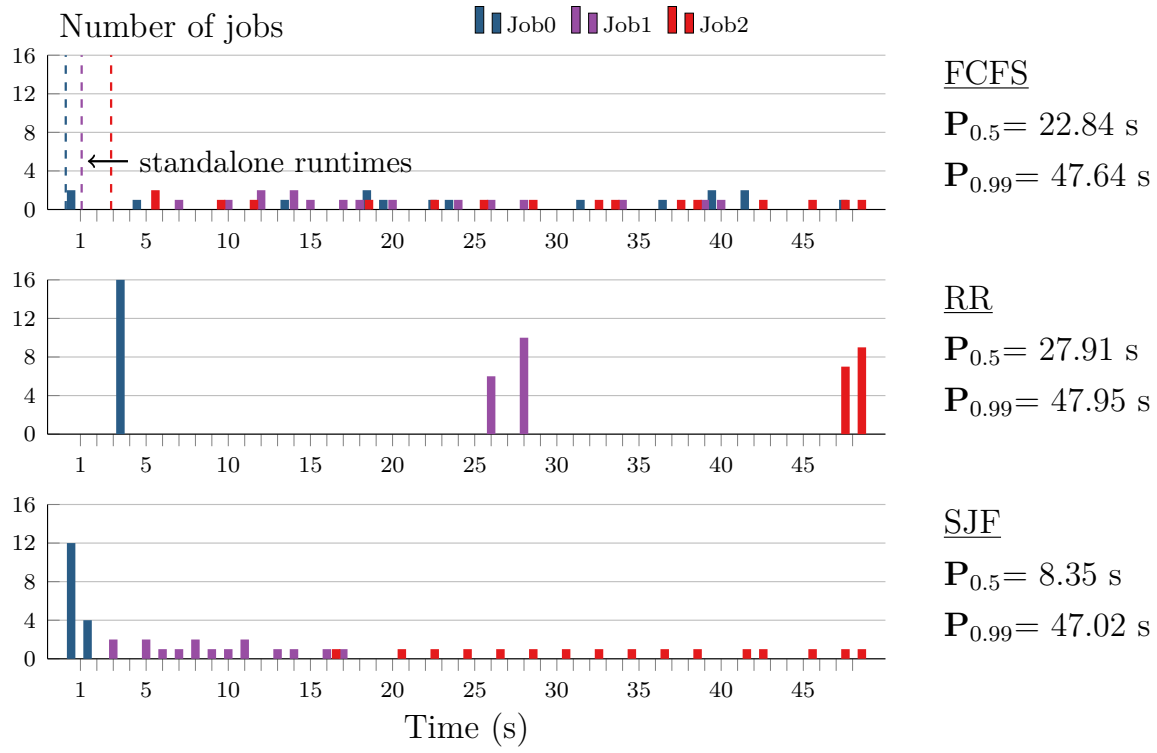



Figure 6.9: Histograms of individual job runtimes, performed on FPGA-Intel with 2 PipeArch instances: 48 jobs of 3 kinds are submitted to PipeArch at the same time and individual runtimes are gathered with different scheduling policies. Median ($\mathbf{P}_{0.5}$) and 99th percentile ($\mathbf{P}_{0.99}$) numbers are shown.

policy: FCFS leads to an even distribution, because a started job is executed until it is finished, with no preemption. RR progresses all submitted jobs equally, leading to distinct points at which jobs are finished corresponding to their standalone runtimes. SJF prioritizes shorter jobs leading to much lower median completion time ($\mathbf{P}_{0.5}$) compared to FCFS and RR, showing the advantage of the capability to perform different scheduling policies.

**Context-Switch Granularity.** Figure 6.10 shows how the total completion time is affected by the context-switch granularity. The lower the partition size, the more frequently will a context-switch occur, especially for RR policy. This is evident for FPGA-Xilinx because of higher context-switch cost as previously explained, leading to inefficient runs with RR policy and low partition sizes. For FPGA-Intel, all scheduling policies and partition sizes behave well, thanks to the low cost of context-switching with to the coherent shared memory architecture. However, FPGA-Xilinx reaches lower total runtimes because of its

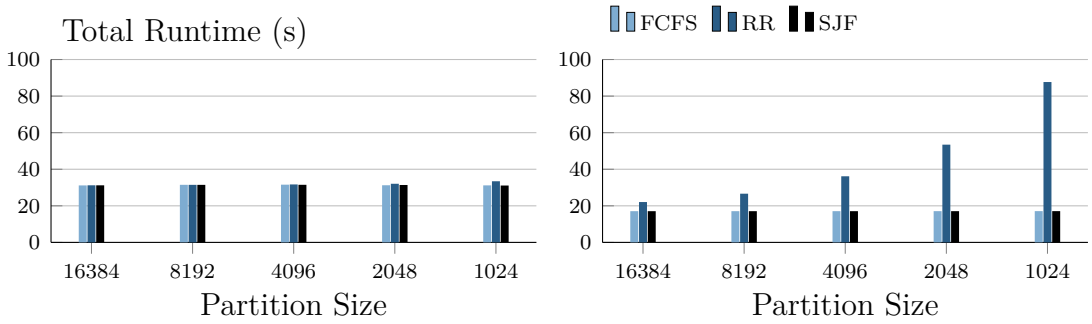



Figure 6.10: Effect of context switch granularity on total completion time when 32 jobs with varying lengths are submitted to PipeArch-RT at the same time.

higher aggregate memory bandwidth.

**Thread Migration Capability.** Table 6.5 shows how the total runtime for a mixed workload is affected when thread migration between PipeArch instances is enabled. This experiment can currently only be performed on FPGA-Intel, because both PipeArch instances access a shared memory as opposed to non-uniform access on FPGA-Xilinx. The benefit of enabling thread migration is evident for all scheduling policies thanks to better load balancing between PipeArch instances, especially for the total runtime rather than the median.

**High Priority Job Runtime.** Figure 6.11 shows what happens if a high priority job is submitted to a busy system depending on whether context-switching is enabled or not. For both platforms, if context-switching is enabled, the high priority job is completed approximately at the same time as if it was running alone in the system. When context-

Table 6.5: Effect of enabling thread migration on workload completion times, shown for different scheduling policies. The percentiles (50%, 75%, 99%) for individual runtimes of 48 jobs and the total runtime is shown in seconds. Experiments are performed on FPGA-Intel with 2 PipeArch instances.

| | **Thread Mig. disabled** | | | **Thread Mig. enabled** | | |
|---|---|---|---|---|---|---|
| **Perc.** | **FCFS** | **RR** | **SJF** | **FCFS** | **RR** | **SJF** |
| 50% | 3.53 | 15.51 | 3.54 | 3.53 | 12.32 | 3.54 |
| 75% | 14.37 | 27.23 | 14.38 | 11.91 | 20.85 | 11.92 |
| 99% | 26.52 | 27.23 | 26.51 | 20.75 | 20.85 | 20.76 |
| Total | 27.03 | 27.23 | 27.02 | 20.95 | 20.85 | 20.96 |

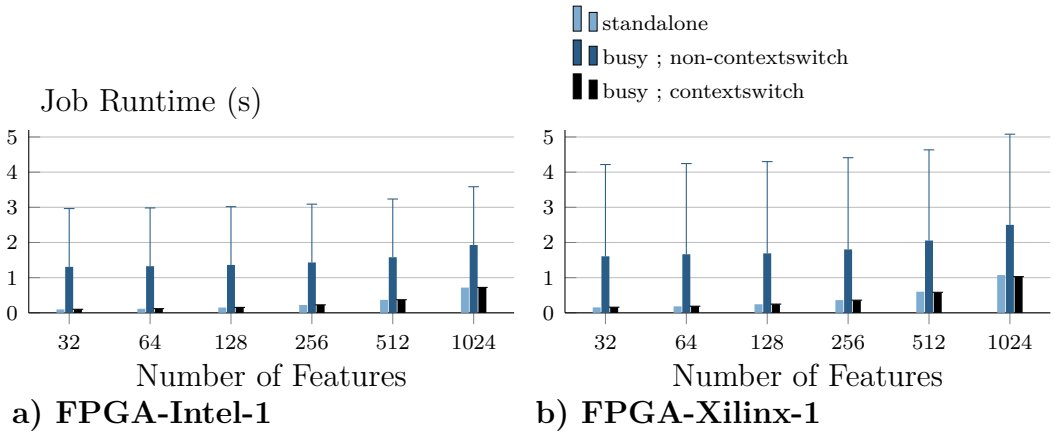



Figure 6.11: High priority job runtime on a busy system. The bars show the average runtime of 5 independent runs and the error bars show the maximum among these runs.

switching is disabled, both the average and maximum runtimes measured are significantly longer because the currently active job has to be executed until completion before the high priority job can be started. Thus, we observe the advantage of having the ability to preempt active jobs in case there are high priority jobs with low latency requirements in the system.

## 6.9 Discussion

In this chapter we presented PipeArch, an architecture spanning hardware and software, enabling novel ways for generic and preemptively scheduled FPGA-based data processing. We implemented a variety of machine learning algorithms, reaching performance similar to fully specialized solutions thanks to maintaining pipeline and SIMD parallelism within the hardware design, while providing dynamic scheduling and control capabilities from software for better system support. These capabilities shall help a more flexible and widely applicable deployment of FPGA-based accelerators than currently possible.

An outcome of this project and also previous chapters in the thesis show that an FPGA-based solution does not always lead to acceleration in comparison to using an entire high-end CPU, for instance in the case of the mixed workload experiment in Figure 6.8. This is not surprising when one considers the power consumption of these processors: With the presented designs loaded, the FPGA on Intel Xeon+FPGA consumes ~29 W (DRAM excluded, uses CPU's memory), the FPGA on VCU1525 uses ~47 W (DRAM included),

whereas the 14-core Xeon can draw more than 300 W. Thus, the FPGA solutions are much more efficient in terms of energy consumption. Therefore, when one considers the problem of optimizing datacenters, using an FPGA provides substantial advantages: Instead of dedicating an entire CPU to satisfy compute intensive algorithms, an FPGA can replace it and perform these operations with the same or better performance, consuming an order of magnitude less energy thanks to specialization. Reducing the compute burden of CPUs with a more efficient alternative is one of the main reasons why major cloud providers are including FPGAs in their datacenters; for instance Microsoft's Catapult project [PCC+14] puts a network-facing FPGA in front of each server blade to offload computation [CFO+18].

Regarding the algorithms we target with PipeArch, they represent a base set for in-database machine learning to evaluate our ideas about providing a generic architecture with better system support than the state-of-the-art. However, they do not represent all angles of improvement that FPGAs might offer over general purpose processors. More compute intensive workloads such as deep learning inference has been shown [UFG+17, FOP+18] to benefit much from FPGA-based processing, especially for providing low-latency. Other workloads that are massively parallel but with irregular access patterns such as inference in decision tree ensembles have been shown to work especially well on FPGAs [OAF+19]. Furthermore, thanks to their architectural flexibility and deep pipelining, computation on FPGAs can be extended: As we showed in Chapter 5, on-the-fly data transformation such as decompression/decryption can be added to existing modules to enable processing directly on compressed/encrypted data without reduced overall throughput. The principles behind PipeArch are orthogonal improvements to these already known advantages of FPGA-based data processing. The presented PipeArch-PU can be modified/extended to be suitable for these other use cases, while benefiting from the presented programming model, dynamic scheduling, and code reuse.

# 7 Conclusions

## 7.1 Summary

This thesis tackles the challenge of using specialized hardware solutions to perform in-database analytics and machine learning from multiple perspectives: (1) individual accelerator efforts resulting in insights into suitability of workloads for specialized hardware, (2) challenges and opportunities arising from integration of novel workloads into databases using specialized hardware, and (3) increasing the usability and flexibility of deployment of the proposed solutions.

In Chapter 1, we presented a motivation and summarized our contributions to the state-of-the-art. In Chapter 2, we discuss using the FPGA as a data processing accelerator and introduce the target platforms we use for the projects in this thesis. We also introduced doppioDB, a branch of MonetDB, which we used for multiple projects in this thesis to show an end-to-end integration of FPGA-based data processing efforts in a database management system.

In Chapter 3, we focused on robust hashing and high-fanout data partitioning workloads, both important sub-operators in query engines optimized for online analytical processing. In Section 3.1, we implemented robust hash functions in a heterogeneous multi-core CPU-FPGA platform with shared memory. We showed that these hash functions achieve significant speed-up while providing increased robustness to skewed data distributions. We also utilized hardware hashing in a main memory hash table and achieved the fastest average build times among all hash functions and data distributions presented, as a result of fast and robust hashing. In Section 3.2, we presented an FPGA-based data parti-

tioner for radix hash joins. It is a novel hardware design avoiding internal pipeline stalls and locks, achieving state-of-the-art throughput for high-fanout data partitioning using an FPGA. We tested the implementation on a research-oriented heterogeneous platform, Intel Xeon+FPGA. In an empirical analysis incorporating a wide range of workloads, we showed that the hybrid join is on par regarding performance with the state-of-the-art 10-threaded CPU solution. We developed an analytical model of the FPGA partitioner showing that it is bound on the memory bandwidth.

In Chapter 4, we presented various highly scalable and parameterizable FPGA-based stochastic gradient descent implementations for performing linear model training. We showed that the efficiency of training linear models can be increased substantially via the usage of low-precision data obtained with stochastic quantization. Processing heavily quantized data with highest efficiency requires specialized instructions and custom vectorization that are unavailable in current fixed architectures, so we took advantage of the architectural flexibility of an FPGA to develop custom engines that can natively consume quantized data for SGD. We empirically analyzed the trade-off space resulting due to the usage of quantization, showing how precision effects end-to-end training time. We showed that FPGA-based computation can outperform state-of-the-art parallel CPU solutions, while being bound on the memory bandwidth available.

In Chapter 5, we focused on generalized model training on column-stores using coordinate descent based methods. We used partition based stochastic coordinate descent (pSCD), that improves the memory access complexity of SCD, leading to better training performance both on the CPU and FPGA. We showed the staleness of pSCD for model updates can be fine tuned, leading to high quality convergence. On the systems side, we presented an FPGA-based system capable of performing various compute intensive data transformation tasks—decompression and decryption—in a pipeline before an SCD engine, performing either SCD or pSCD on the FPGA. We compared our FPGA-based system to an AVX-optimized multi-core CPU implementation, showing that the multi-core CPU is faster on raw data. However, once it has to perform on-the-fly data transformation, its performance is reduced significantly; whereas the FPGA sustains high throughput even when it performs decompression/decryption, thanks to pipeline parallelism. Finally, we compared pSCD to more popular SGD and discussed under which circumstances the choice of pSCD over SGD makes sense.

In Chapter 6, we presented PipeArch, an architecture spanning hardware and software, enabling novel ways for generic and context-switch capable FPGA-based data processing.

With PipeArch, we aimed at programmability while maintaining advantages of highly specialized architectures being often deployed on FPGAs. To achieve this, we developed a register machine that is able to invoke multiple so-called *Subroutines* in parallel, working in a pipelined producer-consumer fashion. The pipelined execution is enabled by a specialized network between the *Subroutines* and so-called memory *Regions*. Thanks to the programmability in PipeArch, we implemented a variety of machine learning algorithms for generalized linear model training and matrix factorization, reaching performance similar to fully specialized solutions, while providing dynamic scheduling and control capabilities from software.

## 7.2 Research Outlook

### 7.2.1 Data Access Challenge

In most of the solutions we presented in this work, the accelerator is bound on the bandwidth for its data access. This is not a fundamental limitation in FPGA-based systems, but is a characteristic of the platforms currently available. As research has shown the importance of having high bandwidth access to data for FPGAs, vendors are developing novel platforms with better capabilities. For instance, FPGAs with high bandwidth memory (HBM) are made available by Xilinx [u28]. On these platforms, the FPGA has access to 8 GBs of memory uniformly via 32 ports, each 256-bits wide. If the logic is clocked at 400 MHz, the theoretical bandwidth can be as high as 410 GB/s.

With these recent developments, the data access challenge seems to be less critical. However, novel systems raise many questions and the capabilities might not be easy to utilize to their full extent. For instance, reaching the theoretical bandwidth of HBM on an FPGA alongside practical applications seems to be quite difficult due to the following reasons:

1. Reaching a clock frequency of 400 MHz with an application utilizing most of the resources on a modern FPGA is difficult. The maximum bandwidth available by the HBM is linearly proportional to the clock frequency.

2. The design methodology for high HBM bandwidth utilization requires a scale-out computation architecture, leading to high resource usage and very wide busses to be routed. This, in turn, limits the highest reachable clock frequency.

3. The HBM on Xilinx FPGAs is connected to one so called Super Logic Region (SLR). There are multiple SLRs on modern Xilinx FPGAs and accessing logic between SLRs adds high net delay, enforcing lower clock frequency. When reaching high utilization, the usage of multiple SLRs cannot be avoided, forcing logic from a distant SLR to access the SLR with the HBM. This, in turn, causes high net delays and the clock frequency needs to be lowered.

4. To provide high bandwidth, the HBM consists of multiple physically separate DRAM banks. There is a hardened crossbar logic in front of these, to allow each of the 32 ports to access any of the banks. If the data to be accessed happens to be placed on one bank, it creates congestion in the crossbar for that bank, so that each port receives 1/32th of the peak bandwidth. Consequently, smart placement and partitioning of data is very crucial in achieving as high a bandwidth as HBM promises.

Besides the above mentioned limitations, the inclusion of HBM creates a non-uniform memory system. This is similar to the VCU1525 platform with its 4 DRAM banks as opposed to the Xeon+FPGA, both introduced in Chapter 2. The non-uniform memory makes system design more difficult, forcing copies of data. Collaborative processing as presented in Chapter 3, where parts of the data is consumed by the FPGA and parts by the CPU does not benefit from the non-uniformity, as the interconnect between the CPU and the FPGA becomes the bottleneck. The family of applications that might benefit much from HBM contain algorithms accessing data iteratively such as training machine learning models as presented in Chapters 4, 5 and 6. In Chapter 6, we have already showed the advantage of having higher bandwidth on VCU1525 against the Xeon+FPGA for this family of algorithms, so the HBM can potentially give much better results.

There is much to be explored in this area for future work. Since collaborative processing seems to be more difficult in non-uniform memory systems, abstraction layers are needed to hide this non-uniformity for the software that is responsible for the data placement. Also, novel FPGAs [spe] with vector-routing or hardened network-on-chip [SGA+19] architectures might help overcome some of the mentioned limitations in reaching the peak potential of HBM, since they eliminate the need for bit-wise routing, making on-chip data movement much more efficient. The accelerator efforts we presented in this work can thus be revisited in those novel platforms with potentially much higher acceleration as a result.

A further field of research that has potential to address the data access challenge is processing-in-memory (PIM). PIM is becoming a reality thanks to 3D-stacking tech-

nologies that allow combining logic layers with DRAM layers in the same chip [JK12, AYMC15]. If the logic layer is an FPGA, one could apply solutions similar to the ones presented in this thesis to perform computation directly in the memory, benefiting from low latency, high bandwidth, and energy efficiency. One difficulty in using this technology is that the logic layer to be implemented could be limited in area and power consumption due to cooling restrictions, compared to dedicated dies as normally used. This calls for a different and more simplistic family of workloads [BGK+18] that might benefit from PIM, with the main goal of *reducing data movement between the memory and the processor*.

### 7.2.2 Lower Productivity Challenge

One of the primary challenges in developing FPGA-based data processing systems remains the lower productivity compared to developing software-only solutions. There are multiple factors causing this:

1. A fundamental difficulty arises from the architectural flexibility offered by FPGAs; the very same reason why they offer acceleration capability for many workloads make them also harder to use: With flexibility in hardware, the design space becomes larger, increasing the time to develop an optimized solution.

2. The primary programming model using hardware description languages (HDL) is low level and does not offer abstractions that are commonplace in software development. However, this is rapidly improving as interest in FPGA-based solutions increases, with so-called high-level-synthesis (HLS) tools such as Vivado HLS from Xilinx [Fei12], HLS Compiler from Intel [intb], and start-up efforts spun off from academia such as LegUp [CCA+11]. These tools have their own shortcomings such as producing suboptimal designs compared to HDL-based solutions or requiring still a deep understanding of hardware design principles. Nevertheless, they offer many abstractions and even nowadays can increase the productivity in specialized hardware development substantially.

3. The interface between an FPGA-based accelerator and any other component in the system with regards to both control and data movement is usually manually designed. This increases the effort in using the accelerators in established large-scale software systems. However, this area is also rapidly improving with frameworks such as SDAccel from Xilinx [Wir14], OpenCL from Intel [inta], OC-Accel from IBM [oca], and solutions from academia like Centaur [OSKA17] and PipeArch as presented in Chapter 6.

These frameworks offer abstractions for control and data movement between an FPGA accelerator and a CPU. Further efforts for exposing the FPGA's memory as an extension to the CPU's own memory address space are also underway [SDS+18]. This will streamline data movement between these two platforms and make writing CPU-side code that interacts with the FPGA much easier.

4. The tools for FPGA development are not as mature as software development. One reason for this is the much smaller developer community compared to software. However, there are also some fundamental challenges in designing tools for FPGA development. Placement and routing of custom logic on FPGA fabric is an NP-complete problem [WTMS94]. Compilation can take many hours and this problem becomes more pronounced as FPGA devices keep getting larger, resulting in longer compilation times.

5. Even when an FPGA-based accelerator has been tested for correct behavior via simulation, getting a working solution in hardware can still take long, due to the necessity of meeting the timing requirements. An accelerator, after being placed and routed, must adhere to setup and hold times enforced by the target clock frequency. Apart from the obvious solution of reducing the clock frequency at the expense of losing performance, designers might iterate over their design adding more registers, choosing different resources for certain blocks (e.g., LUTRAM vs. Block-RAM or Ultra-RAM), or doing manual floorplanning. This iterative process might increase time to a working solution substantially.

Although some of these challenges are fundamental and arise naturally from working with flexible hardware, there is still a large room for improvement. We tried to address some of the deployment and programming model challenges with PipeArch in Chapter 6. PipeArch can be used as a starting point for further research in this area:

1. We designed the network between the *Subroutines* and *Regions* manually considering the algorithms of interest. A valuable contribution would be to automate this process. Based on an intermediate representation such as MLIR [LP19], one could come up with a method of generating this specialized network between *Subroutines* and *Regions* while adhering to the further constraints determined by the presented version of PipeArch.

2. After a specialized version of PipeArch is obtained with the above method, programming this architecture is done with a low-level language using *Subroutines* directly via

their instruction interface. Although not as complex as optimizing code for a RISC processor (thanks to a much course-granular instructions), certain optimizations are still required such as decisions on non-blocking vs. blocking instructions. A compiler can be developed to generate PipeArch code automatically from a higher level domain specific language.

3. The network between the *Subroutines* and *Regions* is static and point-to-point. A switched network in the form of an overlay [KG15] could be interesting to evaluate, to make the architecture more programmable at the expense of losing performance.

# List of Tables

# List of Figures

# Bibliography


[AA09] J. Agron and D. Andrews. "Building Heterogeneous Reconfigurable Systems with a Hardware Microkernel." In *Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pp. 393–402. ACM, 2009.

[ABE+13] A. Arasu, S. Blanas, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan. "Orthogonal Security with Cipherbase." In *CIDR*. Citeseer, 2013.

[ABH+13] D. Abadi, P. Boncz, S. Harizopoulos, S. Idreos, S. Madden, et al. "The Design and Implementation of Modern Column-Oriented Database Systems." *Foundations and Trends® in Databases*, vol. 5, no. 3, 197–280, 2013.

[aea] "Amazon Employee Access Dataset." `https://github.com/owenzhang/Kaggle-AmazonChallenge2013`.

[aes] "Intel AES Instructions Manual." `https://www.intel.com/content/dam/doc/white-paper/advanced-encryption\-standard-new-instructions-set-paper.pdf`.

[AGL+17] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic. "QSGD: Communication-efficient SGD via Gradient Quantization and Encoding." In *Advances in Neural Information Processing Systems*, pp. 1709–1720. 2017.

[AGV+17] M. Asiatici, N. George, K. Vipin, S. A. Fahmy, and P. Ienne. "Virtualized Execution Runtime for FPGA Accelerators in the Cloud." *Ieee Access*, vol. 5, 1900–1910, 2017.

[AHB+16] I. Absalyamov, R. Halstead, P. Budhkar, W. Najjar, et al. "FPGA-Accelerated Group-by Aggregation Using Synchronizing Caches." In *ACM DaMoN*. 2016.

[ama] "Amazon F1 Instances." `aws.amazon.com/ec2/instance-types/f1/`.

[APM16] J. Arulraj, A. Pavlo, and P. Menon. "Bridging the Archipelago Between Row-Stores and Column-Stores for Hybrid Workloads." In *Proceedings of the 2016 International Conference on Management of Data*, pp. 583–598. ACM, 2016.

[App] A. Appleby. "Murmur Hash Function." `https://github.com/aappleby/smhasher`.

[AtCG+15] M. Aref, B. ten Cate, T. J. Green, B. Kimelfeld, D. Olteanu, E. Pasalic, T. L. Veldhuizen, and G. Washburn. "Design and Implementation of the LogicBlox System." In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pp. 1371–1382. ACM, 2015.

[AYMC15] J. Ahn, S. Yoo, O. Mutlu, and K. Choi. "PIM-enabled Instructions: A Low-Overhead, Locality-Aware Processing-In-Memory Architecture." In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pp. 336–348. IEEE, 2015.

[BATÖ13] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu. "Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited." *Proceedings of the VLDB Endowment*, vol. 7, no. 1, 85–96, 2013.

[BCN18] L. Bottou, F. E. Curtis, and J. Nocedal. "Optimization Methods for Large-Scale Machine Learning." *SIAM Review*, vol. 60, no. 2, 223–311, 2018.

[BGK+18] A. Boroumand, S. Ghose, Y. Kim, R. Ausavarungnirun, E. Shiu, R. Thakur, D. Kim, A. Kuusela, A. Knies, P. Ranganathan, et al. "Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks." In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 316–331. 2018.

[BKG+18] C. Balkesen, N. Kunal, G. Giannikis, P. Fender, S. Sundara, F. Schmidt, J. Wen, S. Agrawal, A. Raghavan, V. Varadarajan, et al. "RAPID: In-Memory Analytical Query Processing Engine with Extreme Performance per Watt." In *Proceedings of the 2018 International Conference on Management of Data*, pp. 1407–1419. ACM, 2018.

[BKM+04] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, and W. Yoder. "Scaling to the End of Silicon with EDGE Architectures." *Computer*, vol. 37, no. 7, 44–55, 2004.

[BL+07] J. Bennett, S. Lanning, et al. "The Netflix Prize." In *Proceedings of KDD cup and workshop*, vol. 2007, p. 35. New York, NY, USA., 2007.

[BLAK15] C. Barthels, S. Loesing, G. Alonso, and D. Kossmann. "Rack-Scale In-Memory Join Processing Using RDMA." In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pp. 1463–1475. ACM, 2015.

[BLP11] S. Blanas, Y. Li, and J. M. Patel. "Design and Evaluation of Main Memory Hash Join Algorithms for Multi-Core CPUs." In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pp. 37–48. ACM, 2011.

[BLP+14] R. Barber, G. Lohman, I. Pandis, et al. "Memory-Efficient Hash Joins." *Proceedings of the VLDB Endowment*, vol. 8, no. 4, 353–364, 2014.

[BMR16] A. Bourge, O. Muller, and F. Rousseau. "Generating Efficient Context-Switch Capable Circuits Through Autonomous Design Flow." *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 10, no. 1, 1–23, 2016.

[BMS+17] C. Barthels, I. Müller, T. Schneider, G. Alonso, and T. Hoefler. "Distributed Join Algorithms on Thousands of Cores." *Proceedings of the VLDB Endowment*, vol. 10, no. 5, 517–528, 2017.

[Bot10] L. Bottou. "Large-Scale Machine Learning with Stochastic Gradient Descent." In *Proceedings of COMPSTAT'2010*, pp. 177–186. Springer, 2010.

[BP18] P. A. Beerel and M. Pedram. "Opportunities for Machine Learning in Electronic Design Automation." In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–5. IEEE, 2018.

[BSB+14] S. Byma, J. G. Steffan, H. Bannazadeh, A. L. Garcia, and P. Chow. "FPGAs in the Cloud: Booting Virtualized Hardware Accelerators with OpenStack." In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, pp. 109–116. IEEE, 2014.

[BTAÖ13] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. "Main-Memory Hash Joins on Multi-Core CPUs: Tuning to the Underlying Hardware." In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pp. 362–373. IEEE, 2013.

[C+14] L. H. Crockett et al. *The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc.* 2014.

[cad09] "A Massively Parallel FPGA-Based Coprocessor for Support Vector Machines, author=Cadambi, Srihari and Durdanovic, Igor and Jakkula, Venkata and Sankaradass, Murugan and Cosatto, Eric and Chakradhar, Srimat and Graf, Hans Peter." In *2009 17th IEEE Symposium on Field Programmable Custom Computing Machines*, pp. 115–122. IEEE, 2009.

[CBD14] M. Courbariaux, Y. Bengio, and J.-P. David. "Training Deep Neural Networks with Low Precision Multiplications." *arXiv preprint arXiv:1412.7024*, 2014.

[CCA+11] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski. "LegUp: High-Level Synthesis for FPGA-based Processor/Accelerator Systems." In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pp. 33–36. ACM, 2011.

[CEW+18] H. Chen, O. Engkvist, Y. Wang, M. Olivecrona, and T. Blaschke. "The Rise of Deep Learning in Drug Discovery." *Drug discovery today*, vol. 23, no. 6, 1241–1250, 2018.

[CFM12] H. Y. Cheah, S. A. Fahmy, and D. L. Maskell. "iDEA: A DSP Block Based FPGA Soft Processor." In *2012 International Conference on Field-Programmable Technology*, pp. 151–158. IEEE, 2012.

[CFO+18] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, et al. "Serving DNNs in Real Time at Datacenter Scale with Project Brainwave." *IEEE Micro*, vol. 38, no. 2, 8–20, 2018.

[CHM+14] J. Cong, H. Huang, C. Ma, B. Xiao, and P. Zhou. "A Fully Pipelined and Dynamically Composable Architecture of CGRA." In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, pp. 9–16. IEEE, 2014.

[CHZ+19] Y. Chen, J. He, X. Zhang, C. Hao, and D. Chen. "Cloud-DNN: An Open Framework for Mapping DNN Models to Cloud FPGAs." In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 73–82. ACM, 2019.

[CO14] J. Casper and K. Olukotun. "Hardware Acceleration of Database Operations." In *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*, pp. 151–160. ACM, 2014.

[Cor97] H. Corporaal. *Microprocessor Architectures: from VLIW to TTA*. John Wiley & Sons, Inc., 1997.

[CP16] R. Chen and V. K. Prasanna. "Accelerating Equi-Join on a CPU-FPGA Heterogeneous Platform." In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 212–219. IEEE, 2016.

[CR09] E. J. Candès and B. Recht. "Exact Matrix Completion via Convex Optimization." *Foundations of Computational mathematics*, vol. 9, no. 6, 717, 2009.

[CS13] J. Coole and G. Stitt. "Fast, Flexible High-Level Synthesis from OpenCL Using Reconfiguration Contexts." *IEEE Micro*, vol. 34, no. 1, 42–53, 2013.

[CSB+11] C. H. Chou, A. Severance, A. D. Brant, Z. Liu, S. Sant, and G. G. Lemieux. "VEGAS: Soft Vector Processor with Scratchpad Memory." In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pp. 15–24. ACM, 2011.

[CSZ+14] F. Chen, Y. Shan, Y. Zhang, Y. Wang, H. Franke, X. Chang, and K. Wang. "Enabling FPGAs in the Cloud." In *Proceedings of the 11th ACM Conference on Computing Frontiers*, p. 3. ACM, 2014.

[CVP+13] Z. Cai, Z. Vagena, L. Perez, S. Arumugam, P. J. Haas, and C. Jermaine. "Simulation of Database-Valued Markov Chains Using SimSQL." In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pp. 637–648. ACM, 2013.

[CZJL15] W.-S. Chin, Y. Zhuang, Y.-C. Juan, and C.-J. Lin. "A Fast Parallel Stochastic Gradient Method for Matrix Factorization in Shared Memory Systems." *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 6, no. 1, 2, 2015.

[DHV01] J. Deepakumara, H. M. Heys, and R. Venkatesan. "FPGA Implementation of MD5 Hash Algorithm." In *Canadian Conference on Electrical and Computer Engineering 2001. Conference Proceedings*, vol. 2, pp. 919–924. IEEE, 2001.

[DSZOR15] C. M. De Sa, C. Zhang, K. Olukotun, and C. Ré. "Taming the Wild: A Unified Analysis of Hogwild-Style Algorithms." In *Advances in neural information processing systems*, pp. 2674–2682. 2015.

[DZT13] C. Dennl, D. Ziener, and J. Teich. "Acceleration of SQL Restrictions and Aggregations through FPGA-based Dynamic Partial Reconfiguration." In *2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*, pp. 25–28. IEEE, 2013.

[EBA+11] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. "Dark Silicon and the End of Multicore Scaling." In *2011 38th Annual international symposium on computer architecture (ISCA)*, pp. 365–376. IEEE, 2011.

[Eec17] L. Eeckhout. "Is Moore's Law Slowing Down? What's Next?" *IEEE Micro*, , no. 4, 4–5, 2017.

[Fei12] T. Feist. "Vivado Design Suite." *White Paper*, vol. 5, 30, 2012.

[FKBH15] J. Fowers, J.-Y. Kim, D. Burger, and S. Hauck. "A Scalable High-Bandwidth Architecture for Lossless Compression on FPGAs." In *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*, pp. 52–59. IEEE, 2015.

[FKRR12] X. Feng, A. Kumar, B. Recht, and C. Ré. "Towards a Unified Architecture for In-RDBMS Analytics." In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pp. 325–336. ACM, 2012.

[FLK+11] C. Farabet, Y. LeCun, K. Kavukcuoglu, E. Culurciello, B. Martini, P. Akselrod, and S. Talay. "Large-Scale FPGA-Based Convolutional Networks." *Scaling up Machine Learning: Parallel and Distributed Approaches*, pp. 399–419, 2011.

[FLP+18] F. Franchetti, T. M. Low, D. T. Popovici, R. M. Veras, D. G. Spampinato, J. R. Johnson, M. Püschel, J. C. Hoe, and J. M. Moura. "SPIRAL: Extreme Performance Portability." *Proceedings of the IEEE*, vol. 106, no. 11, 1935–1968, 2018.

[FML+12] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. "The SAP HANA Database–An Architecture Overview." *IEEE Data Eng. Bull.*, vol. 35, no. 1, 28–33, 2012.

[FOP+18] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, et al. "A Configurable Cloud-Scale DNN Processor for Real-Time AI." In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 1–14. IEEE, 2018.

[FYAE14] K. Fleming, H.-J. Yang, M. Adler, and J. Emer. "The LEAP FPGA Operating System." In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–8. IEEE, 2014.

[FZEC17] Y. Fang, C. Zou, A. J. Elmore, and A. A. Chien. "UDP: A Programmable Accelerator for Extract-Transform-Load Workloads and More." In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 55–68. ACM, 2017.

[GAGN15] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan. "Deep Learning with Limited Numerical Precision." In *International Conference on Machine Learning*, pp. 1737–1746. 2015.

[GHN+12] V. Govindaraju, C.-H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim. "Dyser: Unifying Functionality and Parallelism Specialization for Energy-Efficient Computing." *IEEE Micro*, vol. 32, no. 5, 38–51, 2012.

[GMC+09] M. Gebhart, B. A. Maher, K. E. Coons, J. Diamond, P. Gratz, M. Marino, N. Ranganathan, B. Robatmili, A. Smith, J. Burrill, et al. "An Evaluation of the TRIPS Computer System." *ACM SIGARCH Computer Architecture News*, vol. 37, no. 1, 1–12, 2009.

[Goo] Google. "City Hash Function." `https://github.com/google/cityhash`.

[GOP15] M. Gürbüzbalaban, A. Ozdaglar, and P. Parrilo. "Why Random Reshuffling Beats Stochastic Gradient Descent." *arXiv preprint arXiv:1510.08560*, 2015.

[GSM+99] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, and R. Laufer. "PipeRench: A Coprocessor for Streaming Multimedia Acceleration." In *Proceedings of the 26th International Symposium on Computer Architecture (Cat. No. 99CB36367)*, pp. 28–39. IEEE, 1999.

[HANT15] R. J. Halstead, I. Absalyamov, W. A. Najjar, and V. J. Tsotras. "FPGA-based Multithreading for In-Memory Hash Joins." In *CIDR*. 2015.

[HC95] J. Hoogerbrugge and H. Corporaal. "Automatic Synthesis of Transport Triggered Processors." In *Proc. First Ann. Conf. Advanced School for Computing and Imaging, Heijen, The Netherlands*. 1995.

[HCC+13] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing. "More Effective Distributed ML via

a Stale Synchronous Parallel Parameter Server." In *Advances in neural information processing systems*, pp. 1223–1231. 2013.

[HHHH05] P. Hamalainen, J. Heikkinen, M. Hannikainen, and T. D. Hamalainen. "Design of Transport Triggered Architecture Processors for Wireless Encryption." In *8th Euromicro Conference on Digital System Design (DSD'05)*, pp. 144–152. IEEE, 2005.

[HRS+12] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, et al. "The MADlib Analytics Library: or MAD skills, the SQL." *Proceedings of the VLDB Endowment*, vol. 5, no. 12, 1700–1711, 2012.

[HSIA18] Z. He, D. Sidler, Z. István, and G. Alonso. "A Flexible K-means Operator for Hybrid Databases." In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 368–3683. IEEE, 2018.

[HSM+13] R. J. Halstead, B. Sukhwani, H. Min, M. Thoennes, P. Dube, S. Asaad, and B. Iyer. "Accelerating Join Operation for Relational Databases with FPGAs." In *2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*, pp. 17–20. IEEE, 2013.

[HTK15] M. Happe, A. Traber, and A. Keller. "Preemptive Hardware Multitasking in ReconOS." In *International Symposium on Applied Reconfigurable Computing*, pp. 79–90. Springer, 2015.

[Hut15] M. Hutton. "Understanding How the New Hyperflex Architecture Enables Next-Generation High-Performance Systems." *Altera White Paper. Retrieved April*, 2015.

[HZH14] J. He, S. Zhang, and B. He. "In-Cache Query Co-Processing on Coupled CPU-GPU Architectures." *Proceedings of the VLDB Endowment*, vol. 8, no. 4, 329–340, 2014.

[HZK+19] J. T. Halloran, H. Zhang, K. Kara, C. Renggli, M. The, C. Zhang, D. M. Rocke, L. KaÌĹll, and W. S. Noble. "Speeding Up Percolator." *Journal of proteome research*, vol. 18, no. 9, 3353–3359, 2019.

[IABV13] Z. István, G. Alonso, M. Blott, and K. Vissers. "A Flexible Hash Table Design for 10Gbps Key-Value Stores on FPGAs." In *2013 23rd International Conference on Field programmable Logic and Applications*, pp. 1–8. IEEE, 2013.

[IBH+13] X. Iturbe, K. Benkrid, C. Hong, A. Ebrahim, R. Torrego, I. Martinez, T. Arslan, and J. Perez. "R3TOS: A Novel Reliable Reconfigurable Real-Time Operating System for Highly Adaptive, Efficient, and Dependable Computing on FPGAs." *IEEE Transactions on computers*, vol. 62, no. 8, 1542–1556, 2013.

[IGN+12] S. Idreos, F. Groffen, N. Nes, S. Manegold, S. Mullender, and M. Kersten. "MonetDB: Two Decades of Research in Column-Oriented Database Architectures." *Data Engineering*, vol. 40, 2012.

[inta] "Intel FPGA SDK for OpenCL." `https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html`.

[intb] "Intel HLS Compiler." `https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html`.

[IS11] A. Ismail and L. Shannon. "FUSE: Front-End User Framework for O/S Abstraction of Hardware Accelerators." In *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*, pp. 170–177. IEEE, 2011.

[ISA16] Z. István, D. Sidler, and G. Alonso. "Runtime Parameterizable Regular Expression Operators for Databases." In *Field-Programmable Custom Computing Machines (FCCM), 2016 IEEE 24th Annual International Symposium on*, pp. 204–211. IEEE, 2016.

[ISA17] Z. István, D. Sidler, and G. Alonso. "Caribou: Intelligent Distributed Storage." *Proceedings of the VLDB Endowment*, vol. 10, no. 11, 1202–1213, 2017.

[ISAV16] Z. István, D. Sidler, G. Alonso, and M. Vukolic. "Consensus in a Box: Inexpensive Coordination in Hardware." In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*, pp. 425–438. 2016.

[IWA14] Z. Istvan, L. Woods, and G. Alonso. "Histograms as a Side Effect of Data Movement for Big Data." In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pp. 1567–1578. ACM, 2014.

[Jen] B. Jenkins. "LookUp3 Hash Function." `http://burtleburtle.net/bob/c/`.

[JHL+15] S. Jha, B. He, M. Lu, X. Cheng, and H. P. Huynh. "Improving Main Memory Hash Joins on Intel Xeon Phi Processors: An Experimental Approach." *Proceedings of the VLDB Endowment*, vol. 8, no. 6, 642–653, 2015.

[JK12] J. Jeddeloh and B. Keeth. "Hybrid Memory Cube New DRAM Architecture Increases Density and Performance." In *2012 symposium on VLSI technology (VLSIT)*, pp. 87–88. IEEE, 2012.

[JRHK15] M. Jacobsen, D. Richmond, M. Hogains, and R. Kastner. "RIFFA 2.1: A Reusable Integration Framework for FPGA Accelerators." *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 8, no. 4, 22, 2015.

[JST+14] M. Jaggi, V. Smith, M. Takác, J. Terhorst, S. Krishnan, T. Hofmann, and M. I. Jordan. "Communication-Efficient Distributed Dual Coordinate Ascent." In *Advances in Neural Information Processing Systems*, pp. 3068–3076. 2014.

[JTV+18] P. Jääskeläinen, A. Tervo, G. P. Vayá, T. Viitanen, N. Behmann, J. Takala, and H. Blume. "Transport-Triggered Soft Cores." In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 83–90. IEEE, 2018.

[JYP+17] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, et al. "In-datacenter Performance Analysis of a Tensor Processing Unit." In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 1–12. ACM, 2017.

[K+15] M. Kornacker et al. "Impala: A Modern, Open-Source SQL Engine for Hadoop." In *Proceedings of the 7th Biennial Conference on Innovative Data Systems Research*. 2015.

[KA] K. Kara and G. Alonso. "PipeArch: Generic and Preemptively Scheduled FPGA-based Data Processing."

[KA16] K. Kara and G. Alonso. "Fast and Robust Hashing for Database Operators." In *Field Programmable Logic and Applications (FPL), 2016 26th International Conference on*, pp. 1–4. IEEE, 2016.

[KAA+17] K. Kara, D. Alistarh, G. Alonso, O. Mutlu, and C. Zhang. "FPGA-Accelerated Dense Linear Machine Learning: A Precision-Convergence Trade-off." In *Field-Programmable Custom Computing Machines (FCCM), 2017 IEEE 25th Annual International Symposium on*, pp. 160–167. IEEE, 2017.

[Kap16] N. Kapre. "Optimizing Soft Vector Processing in FPGA-Based Embedded Systems." *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 9, no. 3, 17, 2016.

[KBTP15] C. Kyrkou, C.-S. Bouganis, T. Theocharides, and M. M. Polycarpou. "Embedded Hardware-Efficient Real-Time Classification with Cascade Support Vector Machines." *IEEE transactions on neural networks and learning systems*, vol. 27, no. 1, 99–112, 2015.

[KBY17] A. Kumar, M. Boehm, and J. Yang. "Data Management in Machine Learning: Challenges, Techniques, and Systems." In *Proceedings of the 2017 ACM International Conference on Management of Data*, pp. 1717–1722. ACM, 2017.

[kdd] "KDD Dataset." `https://www.datarobot.com/blog/datarobot-the-2014-kdd-cup`.

[KDK11] D. Kesler, B. Deka, and R. Kumar. "A Hardware Acceleration Technique for Gradient Descent and Conjugate Gradient." In *2011 IEEE 9th Symposium on Application Specific Processors (SASP)*, pp. 94–101. IEEE, 2011.

[KEZA18] K. Kara, K. Eguro, C. Zhang, and G. Alonso. "ColumnML: Column-Store Machine Learning with On-the-Fly Data Transformation." *Proceedings of the VLDB Endowment*, vol. 12, no. 4, 348–361, 2018.

[KG15] N. Kapre and J. Gray. "Hoplite: Building Austere Overlay NOCs for FPGAs." In *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–8. IEEE, 2015.

[KGA17] K. Kara, J. Giceva, and G. Alonso. "FPGA-Based Data Partitioning." In *Proceedings of the 2017 ACM International Conference on Management of Data*, pp. 433–445. ACM, 2017.

[KGS17] O. Knodel, P. R. Genssler, and R. G. Spallek. "Migration of Long-Running Tasks between Reconfigurable Resources using Virtualization." *ACM SIGARCH Computer Architecture News*, vol. 44, no. 4, 56–61, 2017.

[KHT07] D. Koch, C. Haubelt, and J. Teich. "Efficient Hardware Checkpointing: Concepts, Overhead Analysis, and Implementation." In *Proceedings of the 2007 ACM/SIGDA 15th international symposium on Field programmable gate arrays*, pp. 188–196. 2007.

[KJYH18] M. A. Kadi, B. Janssen, J. Yudi, and M. Huebner. "General-Purpose Computing with Soft GPUs on FPGAs." *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 11, no. 1, 5, 2018.

[KKL+09] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, et al. "Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs." *Proceedings of the VLDB Endowment*, vol. 2, no. 2, 1378–1389, 2009.

[KLMV12] T. Kaldewey, G. Lohman, R. Mueller, and P. Volk. "GPU Join Processing Revisited." In *Proceedings of the Eighth International Workshop on Data Management on New Hardware*, pp. 55–62. ACM, 2012.

[KNP15] A. Kumar, J. Naughton, and J. M. Patel. "Learning Generalized Linear Models over Normalized Data." In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pp. 1969–1984. ACM, 2015.

[KSH12] A. Krizhevsky, I. Sutskever, and G. E. Hinton. "Imagenet Classification with Deep Convolutional Neural Networks." In *Advances in neural information processing systems*, pp. 1097–1105. 2012.

[KTD+13] T. Kraska, A. Talwalkar, J. C. Duchi, R. Griffith, M. J. Franklin, and M. I. Jordan. "MLbase: A Distributed Machine-Learning System." In *Cidr*, vol. 1, pp. 2–1. 2013.

[KWZA] K. Kara, Z. Wang, C. Zhang, and G. Alonso. "doppioDB 2.0: Hardware Techniques for Improved Integration of Machine Learning into Databases." *Proceedings of the VLDB Endowment*, vol. 12, no. 12.

[KZ14] M. Kumm and P. Zipf. "Pipelined Compressor Tree Optimization Using Integer Linear Programming." In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–8. IEEE, 2014.

[KZF11] J. K. Kim, Z. Zhang, and J. A. Fessler. "Hardware Acceleration of Iterative Image Reconstruction for X-ray Computed Tomography." In *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 1697–1700. IEEE, 2011.

[KZTK15] M. Kandemir, H. Zhao, X. Tang, and M. Karakoy. "Memory Row Reuse Distance and Its Role in Optimizing Application Performance." In *ACM SIGMETRICS Performance Evaluation Review*, vol. 43, pp. 137–149. ACM, 2015.

[LAT18] B.-E. Laure, B. Angela, and M. Tova. "Machine Learning to Data Management: A Round Trip." In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pp. 1735–1738. IEEE, 2018.

[LBH+15] P.-Å. Larson, A. Birka, E. N. Hanson, W. Huang, M. Nowakiewicz, and V. Papadimos. "Real-time Analytical Processing with SQL Server." *Proceedings of the VLDB Endowment*, vol. 8, no. 12, 1740–1751, 2015.

[LLA+13] H. Lang, V. Leis, M.-C. Albutiu, T. Neumann, and A. Kemper. "Massively Parallel NUMA-aware Hash Joins." In *In Memory Data Management and Analysis*, pp. 3–14. Springer, 2013.

[LMG11] B. Lesser, M. Mücke, and W. N. Gansterer. "Effects of Reduced Precision on Floating-Point SVM Classification Accuracy." *Procedia Computer Science*, vol. 4, 508–517, 2011.

[LNS15] C. Liu, H.-C. Ng, and H. K.-H. So. "QuickDough: A Rapid FPGA Loop Accelerator Design Framework using Soft CGRA Overlay." In *2015 International Conference on Field Programmable Technology (FPT)*, pp. 56–63. IEEE, 2015.

[LP09] E. Lübbers and M. Platzner. "ReconOS: Multithreaded Programming for Reconfigurable Computers." *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 9, no. 1, 8, 2009.

[LP13] Y. Li and J. M. Patel. "BitWeaving: Fast Scans for Main Memory Data Processing." In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pp. 289–300. 2013.

[LP19] C. Lattner and J. Pienaar. "MLIR Primer: A Compiler Infrastructure for the End of MooreâĂŹs Law." 2019.

[LSK+15] P. Li, J. L. Shin, G. Konstadinidis, F. Schumacher, V. Krishnaswamy, H. Cho, S. Dash, R. Masleid, C. Zheng, Y. D. Lin, et al. "A 20nm 32-Core 64MB L3 Cache SPARC M7 Processor." In *Solid-State Circuits Conference-(ISSCC), 2015 IEEE International*, pp. 1–3. IEEE, 2015.

[LW15] J. Liu and S. J. Wright. "Asynchronous Stochastic Coordinate Descent: Parallelism and Convergence Properties." *SIAM Journal on Optimization*, vol. 25, no. 1, 351–376, 2015.

[LZZ+18] Y. Liu, H. Zhang, L. Zeng, W. Wu, and C. Zhang. "MLBench: How Good Are Machine Learning Clouds for Binary Classification Tasks on Structured Data?" *Proceedings of the VLDB Endowment*, vol. 11, no. 10, 1220–1232, 2018.

[Mac15] C. Mack. "The Multiple Lives of Moore's Law." *IEEE Spectrum*, vol. 52, no. 4, 31–31, 2015.

[MBK02] S. Manegold, P. Boncz, and M. Kersten. "Optimizing Main-Memory Join on Modern Hardware." *IEEE TKDE*, vol. 14, no. 4, 709–730, 2002.

[MCB+12] A. Majumdar, S. Cadambi, M. Becchi, S. T. Chakradhar, and H. P. Graf. "A Massively Parallel, Energy Efficient Programmable Accelerator for Learning and Classification." *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 1, 1–30, 2012.

[MCC10] A. Majumdar, S. Cadambi, and S. T. Chakradhar. "An Energy-Efficient Heterogeneous System for Embedded Learning and Classification." *IEEE embedded systems letters*, vol. 3, no. 1, 42–45, 2010.

[MCMM06] R. P. McEvoy, F. M. Crowe, C. C. Murphy, and W. P. Marnane. "Optimisation of the SHA-2 Family of Hash Functions on FPGAs." In *IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures (ISVLSI'06)*, pp. 6–pp. IEEE, 2006.

[MD13] T. B. Murdoch and A. S. Detsky. "The Inevitable Application of Big Data to Health Care." *Jama*, vol. 309, no. 13, 1351–1352, 2013.

[MDCL+18] S. Markidis, S. W. Der Chien, E. Laure, I. B. Peng, and J. S. Vetter. "NVIDIA Tensor Core Programmability, Performance & Precision." In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 522–531. IEEE, 2018.

[MKFG15] N. Mirzadeh, Y. O. Koçberber, B. Falsafi, and B. Grot. "Sort vs. Hash Join Revisited for Near-Memory Execution." In *5th Workshop on Architectures and Systems for Big Data (ASBD 2015)*. 2015.

[MKS+18] D. Mahajan, J. K. Kim, J. Sacks, A. Ardalan, A. Kumar, and H. Esmaeilzadeh. "In-RDBMS Hardware Acceleration of Advanced Analytics." *Proceedings of the VLDB Endowment*, vol. 11, no. 11, 1317–1331, 2018.

[MLG10] M. Mücke, B. Lesser, and W. N. Gansterer. "Peak Performance Model for a Custom Precision Floating-Point Dot Product on FPGAs." In *European Conference on Parallel Processing*, pp. 399–406. Springer, 2010.

[MPA+16] D. Mahajan, J. Park, E. Amaro, H. Sharma, A. Yazdanbakhsh, J. K. Kim, and H. Esmaeilzadeh. "Tabla: A Unified Template-Based Framework for Accelerating Statistical Machine Learning." In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 14–26. IEEE, 2016.

[MSK+11] D. Mahmoodi, A. Soleimani, H. Khosravi, M. Taghizadeh, et al. "FPGA Simulation of Linear and Nonlinear Support Vector Machine." *Journal of Software Engineering and Applications*, vol. 4, no. 05, 320, 2011.

[MTA10] R. Mueller, J. Teubner, and G. Alonso. "Glacier: A Query-to-Hardware Compiler." In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pp. 1159–1162. ACM, 2010.

[mus] "Music (Audio Features) Dataset." `https://labrosa.ee.columbia.edu/millionsong`.

[MVKGR16] A. Morales-Villanueva, R. Kumar, and A. Gordon-Ross. "Configuration Prefetching and Reuse for Preemptive Hardware Multitasking on Partially Reconfigurable FPGAs." In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1505–1508. IEEE, 2016.

[NO14] C. Noel and S. Osindero. "Dogwild!-Distributed Hogwild for CPU & GPU." In *NIPS Workshop on Distributed Machine Learning and Matrix Computations*. 2014.

[NSBK01] R. Nagarajan, K. Sankaralingam, D. Burger, and S. W. Keckler. "A Design Space Evaluation of Grid Processor Architectures." In *Proceedings. 34th ACM/IEEE International Symposium on Microarchitecture. MICRO-34*, pp. 40–51. IEEE, 2001.

[NVS+17] E. Nurvitadhi, G. Venkatesh, J. Sim, D. Marr, R. Huang, J. Ong Gee Hock, Y. T. Liew, K. Srivatsan, D. Moss, S. Subhaschandra, et al. "Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks?" In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 5–14. 2017.

[OAF+19] M. Owaida, G. Alonso, L. Fogliarini, A. Hock-Koon, and P.-E. Melet. "Lowering the Latency of Data Processing Pipelines through FPGA based Hardware Acceleration." *Proceedings of the VLDB Endowment*, vol. 13, no. 1, 71–85, 2019.

[oca] "OpenCAPI OC-Accel." `https://github.com/OpenCAPI/oc-accel`.

[opa] "Intel OPAE Framework." `opae.github.io`.

[OSC+11] N. Oliver, R. R. Sharma, S. Chang, B. Chitlur, E. Garcia, J. Grecco, A. Grier, N. Ijih, Y. Liu, P. Marolia, et al. "A Reconfigurable Computing System Based on a Cache-Coherent Fabric." In *Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on*, pp. 80–85. IEEE, 2011.

[OSKA17] M. Owaida, D. Sidler, K. Kara, and G. Alonso. "Centaur: A Framework for Hybrid CPU-FPGA Databases." In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 211–218. IEEE, 2017.

[OZZA17] M. Owaida, H. Zhang, C. Zhang, and G. Alonso. "Scalable Inference of Decision Tree Ensembles: Flexible Design for CPU-FPGA Platforms." In *Field Programmable Logic and Applications (FPL), 2017 27th International Conference on*, pp. 1–8. IEEE, 2017.

[PCC+14] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, et al. "A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services." *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, 13–24, 2014.

[PDM12] K. Paul, C. Dash, and M. S. Moghaddam. "reMORPH: A Runtime Reconfigurable Architecture." In *2012 15th Euromicro Conference on Digital System Design*, pp. 26–33. IEEE, 2012.

[PGTXFN+14] A. N. Perez-Garcia, G. M. Tornez-Xavier, L. M. Flores-Nava, F. Gómez-Castañeda, and J. A. Moreno-Cadenas. "Multilayer Perceptron Network with Integrated Training Algorithm in FPGA." In *2014 11th International Conference on Electrical Engineering, Computing Science and Automatic Control (CCE)*, pp. 1–6. IEEE, 2014.

[PMK14] H. Pirk, S. Manegold, and M. Kersten. "Waste Not... Efficient Co-Processing of Relational Data." In *2014 IEEE 30th International Conference on Data Engineering*, pp. 508–519. IEEE, 2014.

[PR14] O. Polychroniou and K. A. Ross. "A Comprehensive Study of Main-Memory Partitioning and Its Application to Large-Scale Comparison and

Radix-Sort." In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pp. 755–766. ACM, 2014.

[PRR06] N. Pramstaller, C. Rechberger, and V. Rijmen. "A Compact FPGA Implementation of the Hash Function Whirlpool." In *Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays*, pp. 159–166. ACM, 2006.

[PRR15] O. Polychroniou, A. Raghavan, and K. A. Ross. "Rethinking SIMD Vectorization for In-Memory Databases." In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pp. 1493–1508. ACM, 2015.

[PT12] M. PÇŐtraşcu and M. Thorup. "The Power of Simple Tabulation Hashing." *Journal of the ACM (JACM)*, vol. 59, no. 3, 1–50, 2012.

[Put14] A. Putnam. "Large-Scale Reconfigurable Computing in a Microsoft Datacenter." In *Hot Chips 26 Symposium (HCS), 2014 IEEE*, pp. 1–38. IEEE, 2014.

[PY09] S. J. Pan and Q. Yang. "A Survey on Transfer Learning." *IEEE Transactions on knowledge and data engineering*, vol. 22, no. 10, 1345–1359, 2009.

[RAD15] S. Richter, V. Alvarez, and J. Dittrich. "A Seven-Dimensional Analysis of Hashing Methods and Its Implications on Query Processing." *Proceedings of the VLDB Endowment*, vol. 9, no. 3, 96–107, 2015.

[RB16] M. B. Rabieah and C.-S. Bouganis. "FPGASVM: A Framework for Accelerating Kernelized Support Vector Machine." In *Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications*, pp. 68–84. 2016.

[RBK07] S. Rigler, W. Bishop, and A. Kennings. "FPGA-Based Lossless Data Compression using Huffman and LZ77 Algorithms." In *Electrical and Computer Engineering, 2007. CCECE 2007. Canadian Conference on*, pp. 1235–1238. IEEE, 2007.

[RC10] A. Roldao and G. A. Constantinides. "A High Throughput FPGA-Based Floating Point Conjugate Gradient Implementation for Dense Matrices." *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 3, no. 1, 1–19, 2010.

[RKRS07] L. Renganarayanan, D. Kim, S. Rajopadhye, and M. M. Strout. "Parameterized Tiled Loops for Free." In *ACM SIGPLAN Notices*, vol. 42, pp. 405–414. ACM, 2007.

[RPD+18] C. Ramesh, S. B. Patil, S. N. Dhanuskodi, G. Provelengios, S. Pillement, D. Holcomb, and R. Tessier. "FPGA Side Channel Attacks Without Physical Access." In *International Symposium on Field-Programmable Custom Computing Machines*, pp. paper–116. 2018.

[RR12] B. Recht and C. Ré. "Toward a Noncommutative Arithmetic-Geometric Mean Inequality: Conjectures, Case-Studies, and Consequences." In *Conference on Learning Theory*, pp. 11–1. 2012.

[RR13] B. Recht and C. Ré. "Parallel Stochastic Gradient Algorithms for Large-Scale Matrix Completion." *Mathematical Programming Computation*, vol. 5, no. 2, 201–226, 2013.

[RRWN11] B. Recht, C. Re, S. Wright, and F. Niu. "Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent." In *Advances in neural information processing systems*, pp. 693–701. 2011.

[RT16] P. Richtárik and M. Takáč. "Parallel Coordinate Descent Methods for Big Data Optimization." *Mathematical Programming*, vol. 156, no. 1-2, 433–484, 2016.

[SAB+15] D. Sidler, G. Alonso, M. Blott, K. Karras, K. Vissers, and R. Carley. "Scalable 10Gbps TCP/IP Stack Architecture for Reconfigurable Hardware." In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pp. 36–43. IEEE, 2015.

[SBJS15] J. Stuecheli, B. Blaner, C. Johns, and M. Siegel. "CAPI: A Coherent Accelerator Processor Interface." *IBM Journal of Research and Development*, vol. 59, no. 1, 7–1, 2015.

[SCD16] S. Schuh, X. Chen, and J. Dittrich. "An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory." In *Proceedings of the 2016 International Conference on Management of Data*, pp. 1961–1976. ACM, 2016.

[SDS+18] W. J. Starke, J. Dodson, J. Stuecheli, E. Retter, B. W. Michael, S. J. Powell, and J. A. Marcella. "IBM POWER9 Memory Architectures for Optimized Systems." *IBM Journal of Research and Development*, vol. 62, no. 4/5, 3–1, 2018.

[SEOL14] A. Severance, J. Edwards, H. Omidian, and G. Lemieux. "Soft Vector Processors with Streaming Pipelines." In *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*, pp. 117–126. ACM, 2014.

[SGA+19] I. Swarbrick, D. Gaitonde, S. Ahmad, B. Gaide, and Y. Arbel. "Network-on-Chip Programmable Platform in Versal TM ACAP Architecture." In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 212–221. ACM, 2019.

[Sha16] O. Shamir. "Without-Replacement Sampling for Stochastic Gradient Methods." In *Advances in Neural Information Processing Systems*, pp. 46–54. 2016.

[SIO+17] D. Sidler, Z. István, M. Owaida, K. Kara, and G. Alonso. "doppioDB: A Hardware Accelerated Database." In *Proceedings of the 2017 ACM International Conference on Management of Data*, pp. 1659–1662. ACM, 2017.

[SIOA17] D. Sidler, Z. István, M. Owaida, and G. Alonso. "Accelerating Pattern Matching Queries in Hybrid CPU-FPGA Architectures." In *Proceedings of the 2017 ACM International Conference on Management of Data*, pp. 403–415. ACM, 2017.

[SJ17] E. Stehle and H.-A. Jacobsen. "A Memory Bandwidth-Efficient Hybrid Radix Sort on GPUs." In *Proceedings of the 2017 ACM International Conference on Management of Data*, pp. 417–432. ACM, 2017.

[SK05] N. Sklavos and O. Koufopavlou. "Implementation of the SHA-2 Hash Family Standard using FPGAs." *The journal of Supercomputing*, vol. 31, no. 3, 227–248, 2005.

[SK10] N. Sklavos and P. Kitsos. "BLAKE HASH Function Family on FPGA: From the Fastest to the Smallest." In *2010 IEEE Computer Society Annual Symposium on VLSI*, pp. 139–142. IEEE, 2010.

[SKC+10] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey. "Fast Sort on CPUs and GPUs: A Case for Bandwidth Oblivious SIMD Sort." In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pp. 351–362. ACM, 2010.

[SKD15] F. M. Schuhknecht, P. Khanchandani, and J. Dittrich. "On the Surprising Difficulty of Simple Things: The Case of Radix Partitioning." *Proceedings of the VLDB Endowment*, vol. 8, no. 9, 934–937, 2015.

[SL11] M. Singh and B. Leonhardi. "Introduction to the IBM Netezza Warehouse Appliance." In *Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research*, pp. 385–386. IBM Corp., 2011.

[SL12] A. Severance and G. Lemieux. "VENICE: A Compact Vector Processor for FPGA Applications." In *2012 International Conference on Field-Programmable Technology*, pp. 261–268. IEEE, 2012.

[SMT+12] B. Sukhwani, H. Min, M. Thoennes, P. Dube, B. Iyer, B. Brezzo, D. Dillenberger, and S. Asaad. "Database Analytics Acceleration using FPGAs." In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pp. 411–420. ACM, 2012.

[spe] "Achronix Vectorpath with Speedster7t FPGAs." `https://www.achronix.com/vectorpath/`.

[SSI+18] J. Stuecheli, W. J. Starke, J. D. Irish, L. B. Arimilli, D. Dreps, B. Blaner, C. Wollbrink, and B. Allison. "IBM POWER9 Opens Up a New Era of Acceleration Enablement: OpenCAPI." *IBM Journal of Research and Development*, vol. 62, no. 4/5, 8–1, 2018.

[SSIS11] N. Sharma, P. Sharma, D. Irwin, and P. Shenoy. "Predicting Solar Generation from Weather Forecasts Using Machine Learning." In *2011 IEEE international conference on smart grid communications (SmartGridComm)*, pp. 528–533. IEEE, 2011.

[SST11] S. Shalev-Shwartz and A. Tewari. "Stochastic Methods for L1-regularized Loss Minimization." *Journal of Machine Learning Research*, vol. 12, no. Jun, 1865–1892, 2011.

[SVI+16] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. "Rethinking the Inception Architecture for Computer Vision." In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2818–2826. 2016.

[TBC+05] P. Tamayo, C. Berger, M. Campos, J. Yarmus, B. Milenova, A. Mozes, M. Taft, M. Hornick, R. Krishnan, S. Thomas, et al. "Oracle Data Mining." In *Data mining and knowledge discovery handbook*, pp. 1315–1329. Springer, 2005.

[Tib96] R. Tibshirani. "Regression Shrinkage and Selection via the Lasso." *Journal of the Royal Statistical Society. Series B (Methodological)*, pp. 267–288, 1996.

[u28] "Xilinx Alveo U280." `https://www.xilinx.com/products/boards-and-kits/alveo/u280.html`.

[UFG+17] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers. "FINN: A Framework for Fast, Scalable Binarized Neural Network Inference." In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 65–74. ACM, 2017.

[UIO15] T. Ueda, M. Ito, and M. Ohara. "A Dynamically Reconfigurable Equi-Joiner on FPGA." *IBM Tehnical Report RT0969*, 2015.

[vcu] "Xilinx VCU1525." `www.xilinx.com/products/boards-and-kits/vcu1525-a.html`.

[VPK18] A. Vaishnav, K. D. Pham, and D. Koch. "A Survey on FPGA Virtualization." In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 131–1317. IEEE, 2018.

[W+19] Z. Wang et al. "Accelerating Generalized Linear Models with MLWeaving: A One-Size-Fits-All System for Any-Precision Learning." *Proceedings of the VLDB Endowment*, vol. 12, no. 7, 807–821, 2019.

[WAHH15] J. Weerasinghe, F. Abel, C. Hagleitner, and A. Herkersdorf. "Enabling FPGAs in Hyperscale Data Centers." In *2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom)*, pp. 1078–1086. IEEE, 2015.

[WAT13] L. Woods, G. Alonso, and J. Teubner. "Parallel Computation of Skyline Queries." In *IEEE FCCM*. 2013.

[WBKR13] L. Wu, R. J. Barker, M. A. Kim, and K. A. Ross. "Navigating Big Data With High-Throughput, Energy-Efficient Data Partitioning." In *ACM SIGARCH Computer Architecture News*, vol. 41, pp. 249–260. ACM, 2013.

[WGLP13] S. Werner, S. Groppe, V. Linnemann, and T. Pionteck. "Hardware-Accelerated Join Processing in Large Semantic Web Databases with FPGAs." In *2013 International Conference on High Performance Computing & Simulation (HPCS)*, pp. 131–138. IEEE, 2013.

[WHZ15] Z. Wang, B. He, and W. Zhang. "A Study of Data Partitioning on OpenCL-based FPGAs." In *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–8. IEEE, 2015.

[WIA14] L. Woods, Z. István, and G. Alonso. "Ibex: An Intelligent Storage Engine with Support for Advanced SQL Offloading." *Proceedings of the VLDB Endowment*, vol. 7, no. 11, 963–974, 2014.

[Wir14] L. Wirbel. "Xilinx SDAccel Whitepaper.", 2014.

[WS11] J. Wassenberg and P. Sanders. "Engineering a Multi-Core Radix Sort." In *European Conference on Parallel Processing*, pp. 160–169. Springer, 2011.

[WTMS94] Y.-L. Wu, S. Tsukiyama, and M. Marek-Sadowska. "On Computational Complexity of a Detailed Routing Problem in Two Dimensional FPGAs." In *Proceedings of 4th Great Lakes Symposium on VLSI*, pp. 70–75. IEEE, 1994.

[WZY+14] K. Wang, K. Zhang, Y. Yuan, S. Ma, R. Lee, X. Ding, and X. Zhang. "Concurrent Analytical Query Processing with GPUs." *Proceedings of the VLDB Endowment*, vol. 7, no. 11, 1011–1022, 2014.

[WZZY13] Q. Wang, X. Zhang, Y. Zhang, and Q. Yi. "AUGEM: Automatically Generate High Performance Dense Linear Algebra Kernels on x86 CPUs." In *SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pp. 1–12. IEEE, 2013.

[YHPC18] T. Young, D. Hazarika, S. Poria, and E. Cambria. "Recent Trends in Deep Learning Based Natural Language Processing." *ieee Computational intelligenCe magazine*, vol. 13, no. 3, 55–75, 2018.

[YLL+16] Y. You, X. Lian, J. Liu, H.-F. Yu, I. S. Dhillon, J. Demmel, and C.-J. Hsieh. "Asynchronous Parallel Greedy Coordinate Descent." In *Advances in Neural Information Processing Systems*, pp. 4682–4690. 2016.

[YSR08] P. Yiannacouras, J. G. Steffan, and J. Rose. "VESPA: Portable, Scalable, and Flexible FPGA-based Vector Processors." In *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, pp. 61–70. ACM, 2008.

[ZB17] Z. Zhang and M. Brand. "Convergent Block Coordinate Descent for Training Tikhonov Regularized Deep Neural Networks." In *Advances in Neural Information Processing Systems*, pp. 1719–1728. 2017.

[ZC17] F. Zhou and G. Cong. "On the Convergence Properties of a $K$-Step Averaging Stochastic Gradient Descent Algorithm for Nonconvex Optimization." *arXiv preprint arXiv:1708.01012*, 2017.

[Zha04] T. Zhang. "Solving Large Scale Linear Prediction Problems Using Stochastic Gradient Descent Algorithms." In *Proceedings of the twenty-first international conference on Machine learning*, p. 116. ACM, 2004.

[ZLK+17] H. Zhang, J. Li, K. Kara, D. Alistarh, J. Liu, and C. Zhang. "Zipml: Training Linear Models with End-to-End Low Precision, and a Little Bit of Deep Learning." In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp. 4035–4043. JMLR.org, 2017.

[ZLZC18] Z. Zhu, A. X. Liu, F. Zhang, and F. Chen. "FPGA Resource Pooling in Cloud Computing." *IEEE Transactions on Cloud Computing*, 2018.

[ZR14] C. Zhang and C. Ré. "Dimmwitted: A Study of Main-Memory Statistical Analytics." *Proceedings of the VLDB Endowment*, vol. 7, no. 12, 1283–1294, 2014.

[ZVdWB12] M. Zukowski, M. Van de Wiel, and P. A. Boncz. "Vectorwise: A Vectorized Analytical DBMS." In *ICDE*, pp. 1349–1350. 2012.

[ZXX+17] J. Zhang, Y. Xiong, N. Xu, R. Shu, B. Li, P. Cheng, G. Chen, and T. Moscibroda. "The Feniks FPGA Operating System for Cloud Computing." In *Proceedings of the 8th Asia-Pacific Workshop on Systems*, p. 22. ACM, 2017.

[ZYW+14] T. Zhao, M. Yu, Y. Wang, R. Arora, and H. Liu. "Accelerated Mini-Batch Randomized Block Coordinate Descent Method." In *Advances in neural information processing systems*, pp. 3329–3337. 2014.