PipeArch: Generic and Context-Switch Capable Data Processing on FPGAs

KAAN KARA and GUSTAVO ALONSO, ETH Zurich, Switzerland

Data processing systems based on FPGAs offer high performance and energy efficiency for a variety of applications. However, these advantages are achieved through highly specialized designs. The high degree of specialization leads to accelerators with narrow functionality and designs adhering to a rigid execution flow. For multi-tenant systems this limits the scope of applicability of FPGA-based accelerators, because, first, supporting a single operation is unlikely to have any significant impact on the overall performance of the system, and, second, serving multiple users satisfactorily is difficult due to simplistic scheduling policies enforced when using the accelerator. Standard operating system and database management system features that would help address these limitations, such as context-switching, preemptive scheduling, and thread migration are practically non-existent in current FPGA accelerator efforts.

In this work, we propose PipeArch, an open-source project[1] for developing FPGA-based accelerators that combine the high efficiency of specialized hardware designs with the generality and functionality known from conventional CPU threads. PipeArch provides programmability and extensibility in the accelerator without losing the advantages of SIMD-parallelism and deep pipelining. PipeArch supports context-switching and thread migration, thereby enabling for the first time new capabilities such as preemptive scheduling in FPGA accelerators within a high-performance data processing setting. We have used PipeArch to implement a variety of machine learning methods for generalized linear model training and recommender systems showing empirically their advantages over a high-end CPU and even over fully specialized FPGA designs.

CCS Concepts: • **Computer systems organization** → **Reconfigurable computing**; **Data flow architectures**; Cloud computing; • **Hardware** → **Hardware accelerators**; • **Software and its engineering** → *Runtime environments*; • **Computing methodologies** → *Machine learning*; Shared memory algorithms;

Additional Key Words and Phrases: FPGA, generic architecture, programmable, context-switch, high-performance, machine learning, generalized linear models, training, matrix factorization, data processing

ACM Reference format:
Kaan Kara and Gustavo Alonso. 2020. PipeArch: Generic and Context-Switch Capable Data Processing on FPGAs. *ACM Trans. Reconfigurable Technol. Syst.* 14, 1, Article 3 (November 2020), 28 pages.
https://doi.org/10.1145/3418465

1 INTRODUCTION

The increasing demands from modern data processing workloads such as machine learning have opened up an opportunity for specialized hardware as a way to achieve faster and more energy

[1] https://github.com/fpgasystems/PipeArch.

Authors' address: K. Kara and G. Alonso, Systems Group, Department of Computer Science, ETH Zurich, Stampfenbachstrasse 114, 8092 Zürich, Switzerland; emails: kaan.kara@outlook.com, alonso@inf.ethz.ch.

1936-7406/2020/11-ART3 $15.00
https://doi.org/10.1145/3418465

efficient computing. Prominent examples include the TPU from Google for deep learning [38], the use of FPGAs by Microsoft for in-network data processing [62] and neural network inference [21], and the hybrid FPGA-CPU architectures offered by Intel [55]. FPGAs are interesting regarding the ongoing wave of hardware specialization, because they offer architectural flexibility without the costs associated to an ASIC solution. FPGAs are versatile in both the type of workloads supported (e.g., machine learning [42, 52, 58, 70, 72], database engines [35, 43, 45, 67, 68]), as well as how they can be integrated into a system (e.g., as a co-processor [2], network-attached [73], or in embedded-systems). Thanks to this flexibility and their potential advantages, we see FPGAs being deployed in datacenters of cloud providers such as Amazon [2], Microsoft [61], and Baidu [4]. However, FPGAs still remain a niche acceleration solution without wide applicability, for instance known from GPUs. We argue that the main reason for this is high specialization: The lower operating clock frequency of FPGAs requires highly specialized architectures 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 task.
(2) The designs enforce a rigid execution flow, i.e., only a start-done way of interacting with the accelerator.
(3) The level of difficulty in developing new accelerators and extending/modifying existing ones are high.

These limitations especially impact the usability of FPGAs in cloud infrastructures for data processing, because: (1) Cloud systems are based on multi-tenancy and rely on shared execution capabilities to ensure fair progress for users and response time guarantees for service-level-agreements. (2) Cloud workloads are multi-faceted. FPGAs have been shown to be useful regarding many aspects in this context, ranging from database workloads [45], to machine learning [42, 72], and integration of novel workloads to established software systems [44]. Therefore, if we could support more use-cases with one accelerator and if we could modify/extend an accelerator with ease, its usability will increase substantially. (3) FPGA vendors' highly specialized hardware/software interfaces force cloud providers to be closely tied to that vendor. For instance, AWS FPGA stack [3] depends on Xilinx stack; so customers already using Intel FPGAs might avoid moving their workloads due to the high porting effort. Therefore, easy portability via new abstraction layers is important.

To overcome these limitations and as part of a larger effort to use FPGA accelerators to extend and improve data processing in the cloud, in this work 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 cloud infrastructures.

In summary, the contributions of PipeArch are as follows:

- We show how a programmable hardware architecture can support a wide range of machine learning workloads while matching the performance of fully specialized designs.
- We introduce runtime scheduling 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.
- 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 evaluation highlights significant improvements such as up to 4× reduced median job runtime thanks to runtime scheduling policies, and up to 3.2× speedup for generalized linear model training workloads compared to a 14-core Xeon CPU.

2 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* that are able to perform a wide range of machine learning tasks as fast as highly specialized previous work [42, 44] and significantly faster than previous more generic solutions [52].

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, scratchpad memory) 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 FPGA. 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 [57, 67]. Second, FPGAs are increasingly being deployed in datacenters and the cloud [2, 62], 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 [2] 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 toward 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) [77] or stochastic coordinate descent (SCD) [66] as the optimization algorithm. Each optimization algorithm provides distinct benefits regarding the data access efficiency and convergence characteristics during training. LRMF [15] is widely used to create recommender systems, factorizing a large sparse matrix (e.g., users rating movies) into two smaller matrices with latent variables. We solve LRMF problems with SGD via alternating updates [63]. The goal behind these workloads is to showcase PipeArch's programmability, extensibility, and overall efficiency.

3 SYSTEM OVERVIEW

Figure 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).

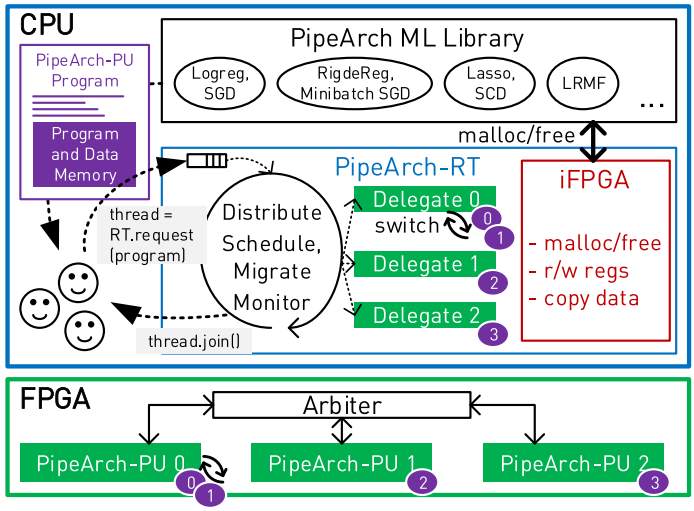


Fig. 1. PipeArch System Overview: Users submit programs to PipeArch-RT, which creates threads to execute them on available PipeArch-PU instances on the FPGA.

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. Each delegate object in software contains the necessary information about a PipeArch-PU instance on the FPGA, such as which threads are assigned to it and the status of the currently running thread. 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, similarly 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 1 on instance 0. Furthermore, a currently running thread can be preempted if a high-priority job is submitted, to ensure low response times for that high-priority job. PipeArch-RT also supports dynamic load balancing: As soon as an instance becomes free, a thread can be migrated to the free instance. Thanks to the thread management facilities available, 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.

4 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 [9, 34, 36, 51, 57]. Earlier studies in this line of work, ReconOS [51] and hthreads [9], expose hardware accelerators as threads; however, they assume system-on-chip platforms (FPGA is next to an embedded CPU). Centaur [57] focuses on integrating accelerated operators into databases in a shared-memory CPU-FPGA system [55]. 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 [51], which consider only embedded CPUs or FUSE [34] 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 [10, 14, 17, 76, 78]. Vaishnav et al. [71] provide an extensive survey of this field. The studies focus on partial reconfiguration (PR); 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 [31, 46, 53] via PR to capture the logic and on-chip memory contents as a way of context saving and 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 us to use PipeArch-PU and allows context-switches only at certain points during program execution.

A further method to support context-switching on FPGA-based accelerators is including scan-chains to extract data from flip-flops and memory elements [12, 47]. Bourge et al. [12] 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 caused hardware overhead (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. In PipeArch, we do not rely on scan-chains and avoid this type of resource overhead; the context that needs to be stored is mainly execution related information from the register machine (Section 6.1) and intermediate data stored in on-chip memory (Section 6.2).

A general difference to the previous line work in this domain is that in PipeArch we use 512-bit vectorization and target acceleration compared to high-end CPUs, so employing the type of context-switching mechanisms in related work discussed here 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 what the previous work targeted in this domain.
3. FPGA-based soft vector processors have been proposed [16, 20, 39, 40, 64, 65, 75] as a way of making FPGA-based processing more programmable while keeping the benefits of spatial parallelism. Chou et al. [20] propose the VEGAS architecture as an improvement over earlier work [75], thanks to features such as scratchpad memory and address registers. The main difference in PipeArch is our use of subroutines that can be pipelined compared to the use of vectorized instructions in VEGAS. Severance et al. [64] 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 tradeoff 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 benchmarks on multi-core CPUs or GPUs.

4. Dataflow-optimized architectures have been studied extensively [13, 24, 28] 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) [24, 30, 32, 37] 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 [28] 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) [13, 27, 54] 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 6.3. Furthermore, also due to its proximity to fully specialized solutions, in PipeArch data are processed in bulk and there is no concept of register files, but rather only scratchpad memory.

5. Overlays and networks-on-chip (NoC) solutions are additional layers of abstraction on top of the FPGA fabric to increase the runtime flexibility [22, 23, 29, 49, 60] and to make the communication of spatially laid-out processing units more streamlined [25, 41]. Govindaraju et al. [29] 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 [41] 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 [38, 42, 52, 72] and inference [18, 58, 70] mainly because general purpose processing has difficulties satisfying the compute/data intensive nature of machine learning. Mahajan et al. [52] propose a framework (DAnA) for generating FPGA-based accelerators to perform in-database machine learning. Their ISA-based MIMD 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 9.1). Our runtime scheduling capabilities are also interesting from a

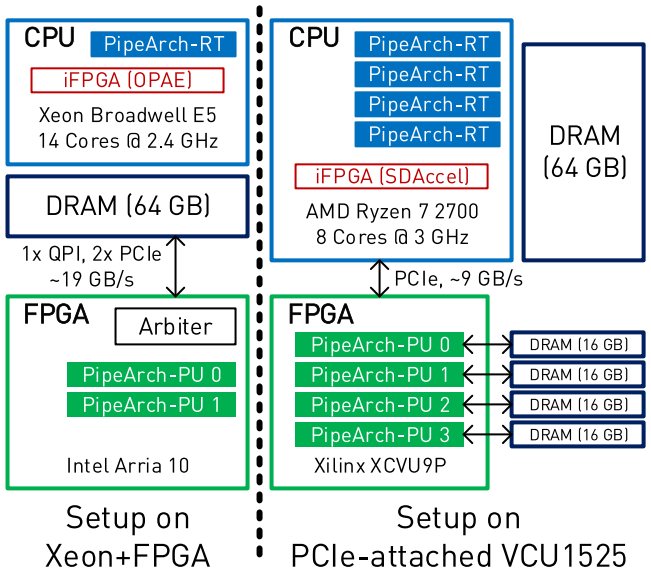


Fig. 2. PipeArch setup on both target platforms: Intel Xeon+FPGA and PCIe-attached VCU1525.

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. [21] 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 [62], and showcase the architectural flexibility of FPGAs in customizing the numerical format for the core computation.

5 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 2: Intel Xeon+FPGA and PCIe-attached Xilinx VCU1525. 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 1), which uses Open Programmable Acceleration Engine (OPAE) [5] for the Xeon+FPGA, and SDAccel [74] for VCU1525.

Intel Xeon+FPGA v2 [55] 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 [8]. The FPGA has four DRAM channels, 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 channel 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 are accessed iteratively; so the data are copied once but used multiple times, making the effect of copying less problematic.

Differences. The setup on the two target platforms is different, because of the non-uniform (channelwise) memory access on VCU1525. A PipeArch-RT instance manages threads running

Table 1. Resource Consumption on Both Target Platforms

Intel Xeon+FPGA (300 MHz)		
Resource	1 PipeArch-PU	Total
ALM	39,984 (9%)	143,158 (34%)
Registers	61,722 (7%)	221,930 (26%)
M20K	631 (23%)	1,679 (62%)
DSP	238 (15%)	476 (31%)

Xilinx VCU1525 (208 MHz)		
Resource	1 PipeArch-PU	Total
CLB (LUT)	92,823 (8%)	422,896 (36%)
CLB (Regs)	113,499 (5%)	535,600 (23%)
BRAM	333 (15%)	1480 (69%)
DSP	156 (2%)	624 (8%)

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 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 channel. Since there are four DRAM channels, there are also four 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, given in Table 1. The limiting resource in both cases is the amount of on-chip memory (BRAM), allowing us to put two instances on Xeon+FPGA and 4 instances on VCU1525. Because of the non-uniform memory access on VCU1525, different combinations of how we connect four available instances to four memory channels are available: For instance, as opposed to connecting each PipeArch-PU instance to its own channel (Figure 2), we could connect all four instances to just one channel 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 channel, optimizing for bandwidth.

The achieved frequency on both platforms also differs: On the Intel FPGA we achieve 300 MHz, whereas on the Xilinx FPGA we achieve 208 MHz. The main reason for the lower clock frequency on the Xilinx FPGA is the necessity to cross the super-logic-regions (SLR), which are separate dies put into the same package resulting in a larger FPGA. Crossing SLRs is more costly due to longer signal propagation times between separate dies. Although we constrain a PipeArch-PU to one SLR during placement, the crossings are unavoidable due to the data movement from the PCIe-endpoint to the local DRAM banks. At this stage, we have not performed fine-grained floorplanning on any FPGAs, which might improve the achieved frequency at the cost of tying the design to the particular FPGA at hand.

6 PIPEARCH PROCESSING UNIT

In this section, we explain the design principles behind PipeArch Processing Unit (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.

6.1 PipeArch-PU Register Machine

At the heart of a PipeArch-PU instance is a Register Machine (RM). The RM can be seen as a primitive CPU capable of integer arithmetic by itself and executing PipeArch programs instruction-by-instruction, controlling complex computation-oriented subroutines. Its important features is the capability to issue subroutine instructions asynchronously and to perform context-switches, as we explain in this section. Figure 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) context-switch capability.

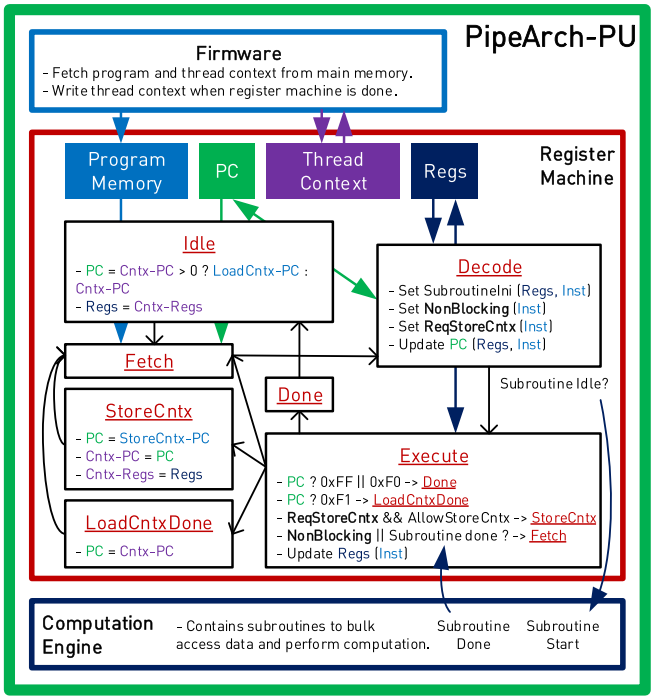


Fig. 3. A PipeArch-PU instance with a focus on the Register Machine, its states, transitions, and data structures.

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 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 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. Since we use 8 bits for the PC, the maximum number of instructions a PipeArch-PU program can have is 256. This is more than enough for current programs—the longest using 34 instructions (LRMF in Section 8). However, not being a fundamental limitation, the width of the PC can be increased with ease if deemed necessary.

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

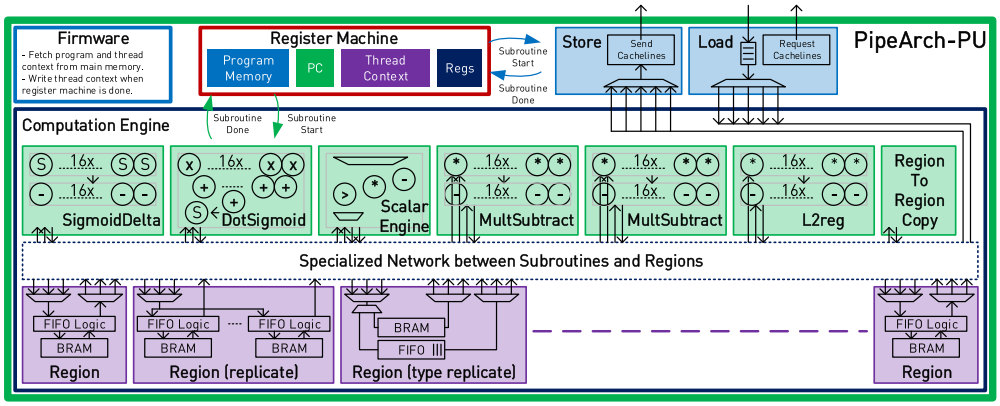


Fig. 4. A PipeArch-PU instance with a focus on the Computation Engine, its subroutines and regions. With the shown engine a wide variety of machine learning tasks can be completed with high performance.

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 8.

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, then 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, as in when the current program yields (examples in Section 8). 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. Following, the firmware will store RM and execution-related information such as the current PC. This completes the context-storing, allowing the start of another thread.

6.2 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 4.

There are two subroutines to access main memory: Load reads data from the main memory 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 six 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.

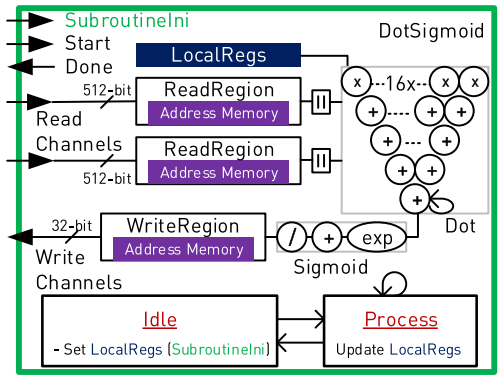


Fig. 5. DotSigmoid subroutine design, with its interfaces, computation pipeline, and internal state.

6.2.1 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 6.1. The DotSigmoid subroutine is shown in Figure 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 bits 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 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 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 two channels.

6.2.2 Regions. 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 4.

Table 2. Examples of Subroutine Instructions

Subroutine	Instruction (regionaccess)	Instruction (specific)
Load	out[6]	offset; length; localoffset
Store	in[3]	offset; length; localoffset
SigmoidDelta, DotSigmoid	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

Table 3. 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]

(1) In the regular mode, physically only one BRAM is used that can be interfaced either in a *FIFO-mode*, thanks to optionally used FIFO logic, or in *BRAM-mode*, as a regular 64 B addressed memory. The advantage of the regular mode 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 by multiple subroutines.
(2) In the replicated mode, as many BRAMs are used as read channels, so parallel reads by multiple subroutines are possible. 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. Similarly to the regular mode, all read/write channels can be used either in *BRAM-mode* or *FIFO-mode*.
(3) In the type replicated mode, we have separate BRAM and FIFO instantiations (spatially separate memory blocks), allowing read and write channels to use the region in both *FIFO-mode* and *BRAM-mode* simultaneously. This is useful when the result produced by one subroutine will be used by multiple subroutines, but one subroutine is consuming in *FIFO-mode* and the other is consuming in *BRAM-mode*. Compared to the replicated mode, we save FIFO logic here.

6.3 Programming PipeArch-PU

PipeArch programs consist of a set of subroutine and control instructions. Examples of subroutine instructions are in Table 2. Subroutine instructions consist of two parts: (1) directives for data access from regions via *regionaccess* and (2) subroutine-*specific* information.

The first part, controlling the data access from regions, is determined with a set of 32-bit words: *regionaccess* as shown in Table 3 determines whether the region is to be accessed in a RAM or FIFO mode, length and offset of accesses, whether the offset should be incremented after iterations, and whether the address memory (used for sparse access patterns) is to be utilized. The number of *regionaccess* words in an instruction depends on the number of read/write interfaces of the particular subroutine. For instance, Load requires six *regionaccess* words determining write behavior, while DotSigmoid requires three *regionaccess* words, two for read and one for write behavior.

The second part of the instruction provides specific information to the subroutine. For instance, for Load we provide offset and length of the data to be accessed from the main memory and the offset for writing that data to the regions. For DotSigmoid, we provide the dimensions of the vectors, whether to perform the sigmoid operation, and the number of iterations to repeat the operation.

Each instruction can be made either blocking or non-blocking. Correctness needs to be ensured by the program; hazards that might happen due to asynchronous execution has to be avoided while writing programs via blocking calls when necessary.

7 PIPEARCH RUNTIME MANAGER (PIPEARCH-RT)

PipeArch-RT is a two-threaded application running 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 consumes the requests from the queue and 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.

7.1 Scheduling Policies

Three scheduling policies are implemented by PipeArch-RT: first-come-first-served (FCFS), round-robin (RR), and shortest-job-first (SJF). Although it is possible for PipeArch-RT to serve jobs with different priorities as we show later in Section 9.2, for these policies we assume all jobs have the same initial priority at submission. With the capabilities already in-place in PipeArch, it is in fact possible to implement more complicated policies (e.g., one combining initial priority with runtime-based priority) with software-only modifications.

- With FCFS, PipeArch-RT serves incoming jobs *in-order* and *until completion*. Therefore context-switching is not required and is not used.
- With RR, PipeArch-RT tries to serve jobs fairly in dedicated time slots and in a round-robin fashion. An active thread is preempted as soon as possible and the next one in line is started. The initial order is determined by the arrival order.
- With SJF, PipeArch-RT gives the job with the shortest runtime the internal highest priority. Whenever a new job is submitted with a shorter total runtime than the remaining runtime of the active job, the new one will have the highest internal priority and the active thread will be preempted.

As we can deduce, from these policies RR and SJF take advantage of context-switching capabilities. As we explain in Section 8, for context-switches to occur, the programs actually need to *yield*. Although having yields resembles cooperative scheduling, we use the term preemptive, since the context-switch is indeed triggered by the PipeArch-RT (as explained in detail in Section 6.1) as opposed to PipeArch-PU programs deciding to yield by themselves. To be more explicit, we might categorize the policies implemented here as having a combination of preemptive and cooperative scheduling features.

As required by SJF, the runtime of programs is approximated by their content and how much data they consume, since this is statically available knowledge. As an example, consider an SGD job: We know how many epochs (a complete pass over the entire dataset) the job has, how much data will be consumed by one epoch, and the approximate data consumption rate for the specific algorithm. The only approximate metric is the data consumption rate, which can be accurately modelled thanks to deterministic nature of FPGA-based computing. For the purposes of this work, we omit accurate modelling and use experimental results in Section 9.1 as the data consumption rate, when determining job runtime.

Table 4. 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

7.2 Context-Switch

Runtime scheduling requires the capability to store and restore the context of the threads running on PipeArch-PU. In Section 6.1 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. The context storing is carried out by the PipeArch-PU in steps: First, the execution jumps to the program's StoreContext part (example in Section 8, Algorithm 1), executing the instructions there to transfer data from on-chip memory (Regions as in Section 6.2) to the main memory. Following, the firmware in PipeArch-PU stores execution related information such as the current program counter (Section 6.1) to the main memory, completing context-storing. The PipeArch-RT is notified of this and a new thread can be started. The load of a context follows a similar logic: PipeArch-RT triggers the PipeArch-PU, followed by first the firmware fetching execution related information from the main memory and second the execution of LoadContext part of the program to load relevant data from the main memory, completing the context-loading. Afterward, the thread can continue execution where it had stopped.

The overhead caused by the context-switch depends mainly on the size of the intermediate data a program keeps, determining the duration of StoreContext and LoadContext parts of a program. For the machine learning algorithms we implement, this is usually the model being trained.

7.3 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 dynamically. 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.

We emphasize that neither partial reconfiguration nor any other FPGA-fabric runtime reconfiguration is required for this capability. We rely only on in-place logic on the running bitstream to save/restore context and all other thread-based functionality, as explained in Section 6. With context-switch capability in-place, thread migration is as simple as starting a previously paused thread, say, that was running on PU-0, on a new PU, say, PU-1. The essential point is that, when a thread is paused and its context is saved, nothing related to that thread (execution-related or intermediate data) remains on the PU that it was running on, so it can be started on an entirely separate PU by restoring its context on that new PU.

8 MACHINE LEARNING ON PIPEARCH

We implement generalized linear model training and low-rank matrix factorization (Table 4) using the capabilities of the PipeArch-PU presented in Section 6.

ALGORITHM 1: SGD for Rigdereg and LogregL2

Initialize:
- $\mathbf{x} = 0$, step size α, partition size P
- $S(\mathbf{z}) = \begin{cases} \mathbf{z} & \text{for Ridgereg} \\ 1/(1+\exp(-\mathbf{z})) & \text{for LogregL2} \end{cases}$

Load(xAddr, **x**)
for *epoch = 1, 2, ...* **do**
  for $p = 1, ..., m/P$ **do**
    $offset = p \cdot m/P$
    Load($\mathbf{a}_{offset+1...P}$)
    Load(bAddr, $b_{offset+1...P}$)
    for *i = offset + shuffle(1, ..., P)* **do**
      $dot = S(\langle \mathbf{x}^t, \mathbf{a}_i \rangle)$ { Load(aAddr, $\mathbf{a}_i$) / DotSig(dot,xAddr,aAddr,...) / Block() }
      $dot = \alpha(dot - b_i)$ { ScalarEngine(dot,α,bAddr,...)
      $\mathbf{x}^{t+1} = \mathbf{x}^t - dot \cdot \mathbf{a}_i^T$ { MultSub(xAddr,dot,aAddr,...)
      $\mathbf{x}^{t+1} = \mathbf{x}^{t+1} - 2\alpha\lambda_2\mathbf{x}^t$ { L2Reg(xAddr,α, λ_2,...)
    Yield()
  Store(**x**, xAddr)
StoreContext: Store(**x**, xAddr)
LoadContext: Load(xAddr, **x**)

8.1 Generalized Linear Models

GLMs are widely used in regression and classification tasks, also applied often in a transfer learning setting [59], 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:

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

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

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

where $(\mathbf{a}_1, b_1), \ldots, (\mathbf{a}_m, b_m) \in ([-1,1]^n \times \mathbb{R})$ 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 SCD [66], since these models are regularized with the L1-norm and favor sparsity, which SCD is efficient at optimizing. RigdeReg and LogregL2 are solved using SGD [77] with the ability to vary minibatch sizes.

SGD implementation for Rigdereg and LogregL2 is shown in Algorithm 1. 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

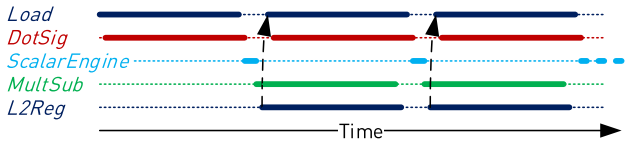


Fig. 6. SGD inner-loop execution

ALGORITHM 2: SCD for Lasso and LogregL1

Initialize:
- $\mathbf{x} = 0$, $\mathbf{z} = 0$, step size α, partition size P
- $S(\mathbf{z})$ as in Algorithm 1
- $T(x_j, g_j) = \begin{cases} \alpha g_j + \alpha\lambda_1 & x_j - \alpha g_j > \alpha\lambda_1 \\ \alpha g_j - \alpha\lambda_1 & x_j - \alpha g_j < -\alpha\lambda_1 \\ x_j & \text{else (to set } x_j = 0) \end{cases}$

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(μ,α,g,xAddr,...)
      $\mathbf{z}_p = \mathbf{z}_p - \mu\mathbf{a}_{p,j}$ { MultSub(zAddr,μ,aAddr,...)
    Store($\mathbf{x}$, xAddr)
    Store($\mathbf{z}_p$, zAddr)
    Yield()

so-called partitions of size P. This is required, because the size of the on-chip memory regions are limited. For instance, from the $b_{1\ldots m}$ 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 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: 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 2, as opposed to updating the entire model for each sample as in SGD. This leads to a columnwise access of the dataset as opposed to row-wise access in SGD. The columnwise access can be beneficial in systems that store data in columnar format such as in-memory databases [33]; being able to run both SGD

ALGORITHM 3: Tiled Alternating SGD for LRMF

Initialize:
· $\mathbf{M} = rand^{m \times d}$, $\mathbf{U} = 0$, step size α, 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 $x_{row}, x_{col}, x_{val}$*: non-zero entries in* $\mathbf{X}_{t_m, t_u}$ **do**
        $e = \langle \mathbf{M}_{x_{row}}, \mathbf{U}_{x_{col}} \rangle$ {DotSig(MAddr,UAddr,...)
        $e = \alpha(e - x_{val})$ {ScalarEngine(e,XAddr,...)
        $\mathbf{M}_{x_{row}} = \mathbf{M}_{x_{row}} - (e \cdot \mathbf{U}_{x_{col}} + \lambda \mathbf{M}_{x_{row}})$ {MultSub(MAddr,e,UAddr,...)
        $\mathbf{U}_{x_{col}} = \mathbf{U}_{x_{col}} - (e \cdot \mathbf{M}_{x_{row}} + \lambda \mathbf{U}_{x_{col}})$ {MultSub(UAddr,e,MAddr,...)
      Block()
      Store($\mathbf{U}_{t_u}$, UAddr)
    Store($\mathbf{M}_{t_m}$, MAddr)
    Yield()

and SCD on the same PipeArch-PU instance showcases the programmability and flexibility in accessing the data. In Algorithm 2, the data are 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 point where we allow context-switching (end of a partition) all intermediate data have been already stored to the main memory.

8.2 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 [11]. The optimization problem is as follows:

$$\min_{\mathbf{M} \in \mathbb{R}^{m \times d}, \mathbf{U} \in \mathbb{R}^{u \times d}} \|\mathbf{M}\mathbf{U}^T - \mathbf{X}\|_F^2 + \lambda \|\mathbf{M}\|_F^2 + \lambda \|\mathbf{U}\|_F^2, \tag{3}$$

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 3 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 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.

Table 5. 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

For LRMF datasets, m, u, d are matrix dimensions as introduced in Section 8.2 and x is the number of non-zero entries in the sparse target matrix.

9 EVALUATION

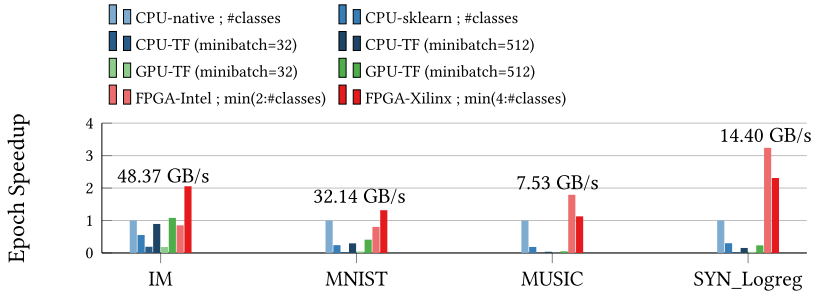
We evaluate our system on both target platforms with datasets presented in Table 5, covering a wide range of workloads including real-world use cases.

- AEA [1] (Amazon employee access/denial prediction) and KDD [6] (KDD Cup 2014) are datasets from *Kaggle* competitions. We use the features Liu et al. [50] 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 [59] in mind: We run *InceptionV3* [69] neural network on ImageNet to extract 2048 features from images of different classes. The task is to train the final layer responsible for the classification.
- MUSIC [7] is a regression task to predict the year a song was released in, given certain extracted features from them.
- NETFLIX [11] 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.

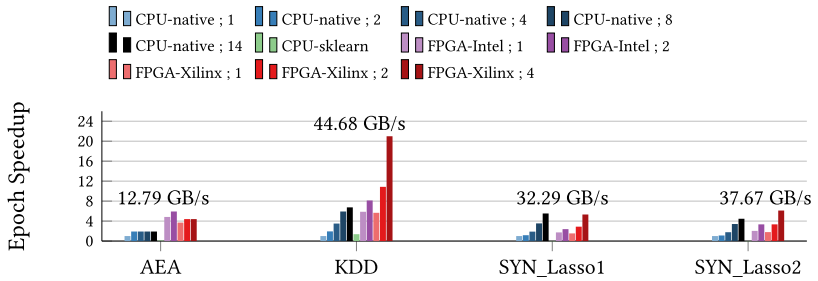
Evaluation Setup. For the baseline experiments we use (1) the 14-core Xeon Broadwell (in the Xeon+FPGA platform) and (2) the NVDIA QUADRO M6000 with 24 GB of memory attached to an 8-core Intel i7-5960X. On the CPU, we use *gcc 5.4* and compile with "*-O3 -march=native*." The frequency governor is set 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.

9.1 Individual Workload Performance

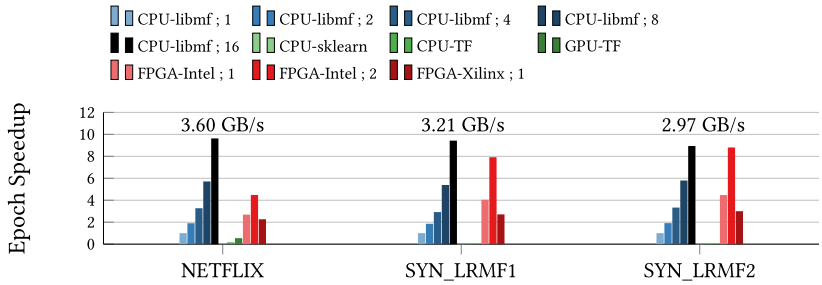
First, we analyze the individual workload performance categorizing them according to the optimization algorithm: SGD, SCD, and LRMF. For each algorithm we perform the same number



(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.

Fig. 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.

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 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 7(a), we observe for parallelizable problems (IM, MNIST), FPGA-Xilinx reaches peak performance thanks

Table 6. Comparison to Related Work: The Maximum Data Consumption Rate Achieved for Each Algorithm

Algorithm	[72]	[52]	[44]	PipeArch
SGD	15 GB/s	3.76 GB/s	—	48.47 GB/s
SCD	—	—	18 GB/s	44.68 GB/s
LRMF	—	0.79 GB/s	—	3.6 GB/s

to four 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 mini-batch 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 [44] and *LogisticRegression* function from scikit-learn with liblinear solver (Figure 7(b)). For KDD, FPGA-Xilinx reaches almost peak performance again thanks to utilizing 4 PipeArch instances, 2× 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 7(c)), we use the optimized multi-core implementation from LIBMF [19]. 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 [63]; 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 four 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. We show the maximum absolute data consumption rates achieved by the accelerators in Table 6. Wang et al. [72] 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 3.2× with 32-bit input data even with a more generic design, mainly thanks to our deployment on VCU1525. Mahajan et al. [52] implement SGD and LRMF on the same Xilinx FPGA (VU9P) as this work; however, the processing rate is not reported.

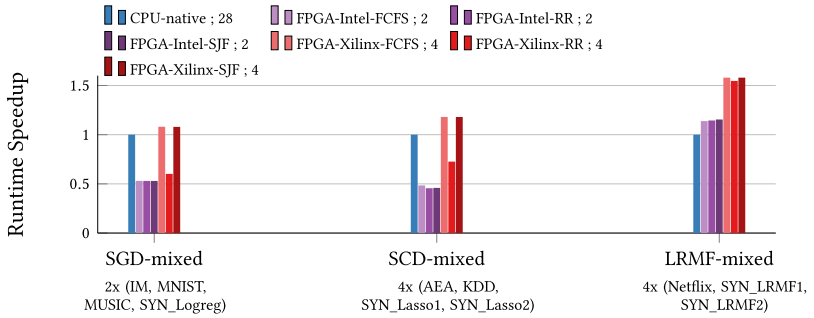


Fig. 8. Mixed workload runtime comparison. All jobs are submitted at the same time and total runtime is measured. For FPGA runs, different scheduling policies are used. Notation for legend labels: *platform; number of cores/instances.*

Thanks to the authors' response to our inquiry, we calculate the maximum processing rate they reach for SGD and LRMF to present a comparison as fair as possible: ~3.76 GB/s for SGD and ~0.79 GB/s for LRMF; we can surpass these up to 12.9× and 3.8×, respectively. Kara et al. [44] 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 2.5× using VCU1525.

9.2 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 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 2× slower compared to using the entire 14-core Xeon CPU (two threads per core) for SGD and SCD workloads. FPGA-Xilinx is slightly faster thanks to its higher aggregate bandwidth with four DRAM banks. A key observation is the slowdown at FPGA-Xilinx when 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 μs) caused mainly due to the memory model of OpenCL and PCIe-based communication is around 7× higher compared to the context-switch latency on the Intel platform (~117 μs), 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 four independent jobs can be executed in parallel on FPGA-Xilinx, we see a total runtime advantage of around 1.5× compared to the CPU. This could not be achieved for a single LRMF job due to the nonuniform access to four DRAM banks as explained in Section 9.1.

Effect of the Scheduling Policy. Figure 9 plots histograms showing the number jobs from a mixed workload completed in a certain time interval. The mixed workload consists of 48 jobs of three categories with varying lengths: Their stand-alone runtimes are plotted in the first plot in Figure 9. The experiment is performed by submitting all 48 jobs at the same time to the PipeArch-RT and individual runtimes are gathered.

The distribution of completion times observed in the histograms (Figure 9) depends on the scheduling policy: FCFS leads to an even distribution, because a started job is executed until it is finished, with no preemption. So, the distribution just depends on the order of arrival.

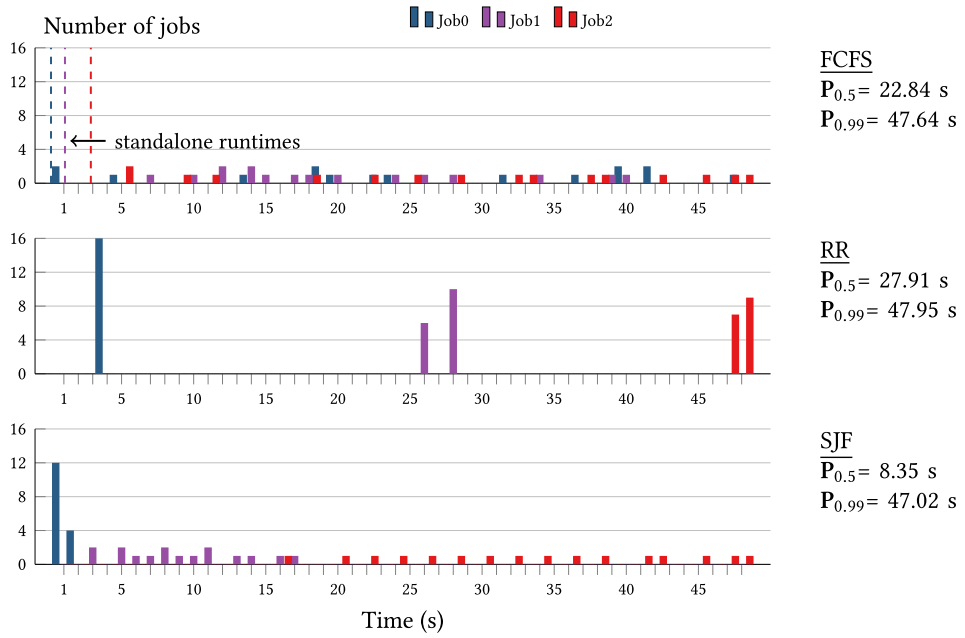


Fig. 9. Histograms of individual job runtimes, performed on FPGA-Intel with two PipeArch instances: Forty-eight jobs of three kinds are submitted to PipeArch at the same time and individual runtimes are gathered with different scheduling policies. Median ($P_{0.5}$) and 99th percentile ($P_{0.99}$) numbers are shown.

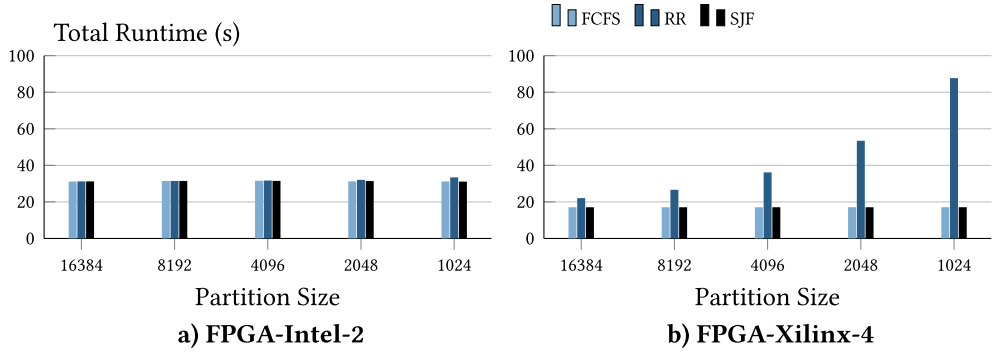


Fig. 10. Effect of context switch granularity on total completion time when 32 jobs with varying lengths are submitted to PipeArch-RT at the same time.

RR progresses all submitted jobs equally, leading to distinct points at which jobs are finished corresponding to their stand-alone runtimes. Note that the plots show only completion times; however, with RR all jobs are progressed equally, so intermediate results can be returned to the user as they become available, making the system feel more responsive. SJF prioritizes shorter jobs leading to much lower median completion time ($P_{0.5}$) compared to FCFS and RR, showing the advantage of the capability to perform different scheduling policies.

Context-Switch Granularity. Figure 10 shows how the total completion time is affected by the context-switch granularity. In this experiment we submit 32 jobs of mixed properties to the PipeArch-RT and wait until all the jobs are finished. We vary the scheduling policy and the

Table 7. Effect of Enabling Thread Migration on Workload Completion Times Shown for Different Scheduling Policies

	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

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 two PipeArch instances.

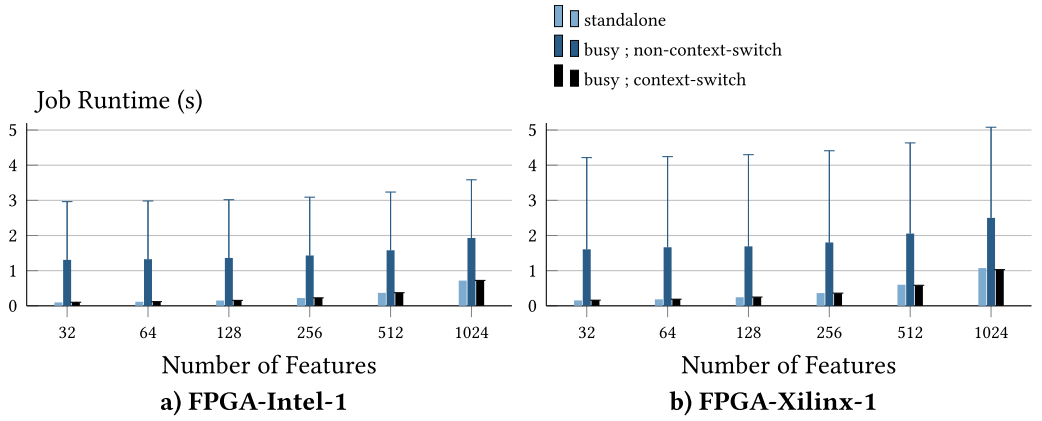


Fig. 11. High-priority job runtime on a busy system. The bars show the average runtime of five independent runs and the error bars show the maximum among these runs. The *x*-axis shows the number of features in the synthetic dataset used for the high-priority job, determining the model size and thus the context-switch cost.

partition size, the latter directly affecting the context-switch granularity by changing how often *Yield()* in PipeArch programs occur (Section 8). So 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 higher aggregate memory bandwidth.

Thread Migration Capability. Table 7 shows how the total runtime for a mixed workload is affected when thread migration between PipeArch instances is enabled. We submit 48 jobs of 2 categories (one short, one long) and measure individual runtimes to collect the presented statistics. 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 11 shows what happens if a high-priority job is submitted to a busy system depending on whether context-switching is enabled or not. We submit a high-priority job once stand-alone, once on a busy system without context-switching, and finally on

a busy system with context-switching. We vary the number of features in the dataset used for the high-priority job on the x-axis. The number of features directly determines the model size and thus the cost of context-switching. On both platforms, we observe that if context-switching is enabled, then the high-priority job is completed approximately at the same time as if it was running stand-alone in the system. When context-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. Furthermore, we observe that the context-switch cost (higher for increased number of features) remains negligible, observed by comparing the stand-alone runtime and the busy-system runtime with enabled context-switching.

10 DISCUSSION

An outcome of the evaluation is 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 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 [62] puts a network-facing FPGA in front of each server blade to offload computation [21].

Regarding the algorithms targeted with PipeArch, they are a base set for machine learning that focuses on training linear models 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 can offer over general purpose processors. More compute intensive workloads such as deep learning inference has been shown [26, 70] 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 [56]. Furthermore, thanks to their architectural flexibility and deep pipelining, computation on FPGAs can be extended: On-the-fly data transformation such as decompression/decryption can be added to existing modules to enable training machine learning models directly on compressed/encrypted data without reduced overall throughput [44]. 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.

Further aspects that can be addressed with future work include the following:

(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 [48], 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 [41] could be interesting to evaluate, to make the architecture more programmable at the expense of losing performance.

11 CONCLUSION

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. Finally, PipeArch is available as open-source.[2]

REFERENCES

[1] [n.d.]. Amazon Employee Access Dataset. https://github.com/owenzhang/Kaggle-AmazonChallenge2013.

[2] [n.d.]. Amazon F1 Instances. aws.amazon.com/ec2/instance-types/f1/.

[3] [n.d.]. AWS FPGA Stack Repository. Retrieved from https://github.com/aws/aws-fpga.

[4] [n.d.]. Baidu FPGA Instances. Retrieved from https://cloud.baidu.com/product/fpga.html.

[5] [n.d.]. Intel OPAE Framework. Retrieved from opae.github.io.

[6] [n.d.]. KDD Dataset. Retrieved from https://www.datarobot.com/blog/datarobot-the-2014-kdd-cup.

[7] [n.d.]. Music (Audio Features) Dataset. Retrieved from https://labrosa.ee.columbia.edu/millionsong.

[8] [n.d.]. Xilinx VCU1525. Retrieved from www.xilinx.com/products/boards-and-kits/vcu1525-a.html.

[9] Jason Agron and David Andrews. 2009. Building heterogeneous reconfigurable systems with a hardware microkernel. In *Proceedings of the 7th IEEE/ACM International Conference on Hardware/software Codesign and System Synthesis*. ACM, 393–402.

[10] Mikhail Asiatici, Nithin George, Kizheppatt Vipin, Suhaib A. Fahmy, and Paolo Ienne. 2017. Virtualized execution runtime for FPGA accelerators in the cloud. *IEEE Access* 5 (2017), 1900–1910.

[11] James Bennett, Stan Lanning, et al. 2007. The Netflix prize. In *Proceedings of the KDD Cup and Workshop*, Vol. 2007. New York, NY, 35.

[12] Alban Bourge, Olivier Muller, and Frédéric Rousseau. 2016. Generating efficient context-switch capable circuits through autonomous design flow. *ACM Trans. Reconfig. Technol. Syst.* 10, 1 (2016), 1–23.

[13] Doug Burger, Stephen W. Keckler, Kathryn S. McKinley, Mike Dahlin, Lizy K. John, Calvin Lin, Charles R. Moore, James Burrill, Robert G. McDonald, and William Yoder. 2004. Scaling to the end of silicon with EDGE architectures. *Computer* 37, 7 (2004), 44–55.

[14] Stuart Byma, J. Gregory Steffan, Hadi Bannazadeh, Alberto Leon Garcia, and Paul Chow. 2014. FPGAs in the cloud: Booting virtualized hardware accelerators with OpenStack. In *Proceedings of the 2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 109–116.

[15] Emmanuel J. Candès and Benjamin Recht. 2009. Exact matrix completion via convex optimization. *Foundations of Computational Mathematics* 9, 6 (2009), 717.

[16] Hui Yan Cheah, Suhaib A. Fahmy, and Douglas L. Maskell. 2012. iDEA: A DSP block based FPGA soft processor. In *Proceedings of the 2012 International Conference on Field-Programmable Technology*. IEEE, 151–158.

[17] Fei Chen, Yi Shan, Yu Zhang, Yu Wang, Hubertus Franke, Xiaotao Chang, and Kun Wang. 2014. Enabling FPGAs in the cloud. In *Proceedings of the 11th ACM Conference on Computing Frontiers*. ACM, 3.

[2] https://github.com/fpgasystems/PipeArch.

[18] Yao Chen, Jiong He, Xiaofan Zhang, Cong Hao, and Deming Chen. 2019. 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*. ACM, 73–82.

[19] Wei-Sheng Chin, Yong Zhuang, Yu-Chin Juan, and Chih-Jen Lin. 2015. A fast parallel stochastic gradient method for matrix factorization in shared memory systems. *ACM Transactions on Intelligent Systems and Technology (TIST)* 6, 1 (2015), 2.

[20] Christopher H. Chou, Aaron Severance, Alex D. Brant, Zhiduo Liu, Saurabh Sant, and Guy G. F. Lemieux. 2011. VEGAS: Soft vector processor with scratchpad memory. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. ACM, 15–24.

[21] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, et al. 2018. Serving DNNs in real time at datacenter scale with project brainwave. *IEEE Micro* 38, 2 (2018), 8–20.

[22] Jason Cong, Hui Huang, Chiyuan Ma, Bingjun Xiao, and Peipei Zhou. 2014. A fully pipelined and dynamically composable architecture of CGRA. In *Proceedings of the 2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 9–16.

[23] James Coole and Greg Stitt. 2013. Fast, flexible high-level synthesis from OpenCL using reconfiguration contexts. *IEEE Micro* 34, 1 (2013), 42–53.

[24] Henk Corporaal. 1997. *Microprocessor Architectures: From VLIW to TTA*. John Wiley & Sons, Inc.

[25] Kermin Fleming, Hsin-Jung Yang, Michael Adler, and Joel Emer. 2014. The LEAP FPGA operating system. In *Proceedings of the 2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 1–8.

[26] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, et al. 2018. A configurable cloud-scale DNN processor for real-time AI. In *Proceedings of the 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 1–14.

[27] Mark Gebhart, Bertrand A. Maher, Katherine E. Coons, Jeff Diamond, Paul Gratz, Mario Marino, Nitya Ranganathan, Behnam Robatmili, Aaron Smith, James Burrill, et al. 2009. An evaluation of the TRIPS computer system. *ACM SIGARCH Computer Architecture News* 37, 1 (2009), 1–12.

[28] Seth Copen Goldstein, Herman Schmit, Matthew Moe, Mihai Budiu, Srihari Cadambi, R. Reed Taylor, and Ronald Laufer. 1999. PipeRench: A coprocessor for streaming multimedia acceleration. In *Proceedings of the 26th International Symposium on Computer Architecture (Cat. No. 99CB36367)*. IEEE, 28–39.

[29] Venkatraman Govindaraju, Chen-Han Ho, Tony Nowatzki, Jatin Chhugani, Nadathur Satish, Karthikeyan Sankaralingam, and Changkyu Kim. 2012. Dyser: Unifying functionality and parallelism specialization for energy-efficient computing. *IEEE Micro* 32, 5 (2012), 38–51.

[30] Panu Hamalainen, Jari Heikkinen, Marko Hannikainen, and Timo D. Hamalainen. 2005. Design of transport triggered architecture processors for wireless encryption. In *Proceedings of the 8th Euromicro Conference on Digital System Design (DSD'05)*. IEEE, 144–152.

[31] Markus Happe, Andreas Traber, and Ariane Keller. 2015. Preemptive hardware multitasking in ReconOS. In *Proceedings of the International Symposium on Applied Reconfigurable Computing*. Springer, 79–90.

[32] Jan Hoogerbrugge and Henk Corporaal. 1995. Automatic synthesis of transport triggered processors. In *Proceedings of the First Ann. Conf. Advanced School for Computing and Imaging, Heijen, The Netherlands*.

[33] S. Idreos, F. Groffen, N. Nes, S. Manegold, S. Mullender, and M. Kersten. 2012. MonetDB: Two decades of research in column-oriented database architectures. *Data Engineering* 40 (2012).

[34] Aws Ismail and Lesley Shannon. 2011. FUSE: Front-end user framework for O/S abstraction of hardware accelerators. In *Proceedings of the 2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 170–177.

[35] Zsolt István, David Sidler, and Gustavo Alonso. 2016. Runtime parameterizable regular expression operators for databases. In *Proceedings of the IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'16)*. IEEE, 204–211.

[36] Xabier Iturbe, Khaled Benkrid, Chuan Hong, Ali Ebrahim, Raul Torrego, Imanol Martinez, Tughrul Arslan, and Jon Perez. 2013. R3TOS: A novel reliable reconfigurable real-time operating system for highly adaptive, efficient, and dependable computing on FPGAs. *IEEE Transactions on Computers* 62, 8 (2013), 1542–1556.

[37] Pekka Jääskeläinen, Aleksi Tervo, Guillermo Payá Vayá, Timo Viitanen, Nicolai Behmann, Jarmo Takala, and Holger Blume. 2018. Transport-triggered soft cores. In *Proceedings of the 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 83–90.

[38] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 1–12.

[39] Muhammed Al Kadi, Benedikt Janssen, Jones Yudi, and Michael Huebner. 2018. General-purpose computing with soft GPUs on FPGAs. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 11, 1 (2018), 5.

[40] Nachiket Kapre. 2016. Optimizing soft vector processing in FPGA-based embedded systems. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 9, 3 (2016), 17.

[41] Nachiket Kapre and Jan Gray. 2015. Hoplite: Building austere overlay NOCs for FPGAs. In *Proceedings of the 2015 25th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 1–8.

[42] Kaan Kara, Dan Alistarh, Gustavo Alonso, Onur Mutlu, and Ce Zhang. 2017. FPGA-accelerated dense linear machine learning: A precision-convergence trade-off. In *Proceedings of the IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'17)*. IEEE, 160–167.

[43] Kaan Kara and Gustavo Alonso. 2016. Fast and robust hashing for database operators. In *Proceedings of the 26th International Conference on Field Programmable Logic and Applications (FPL'16)*. IEEE, 1–4.

[44] Kaan Kara, Ken Eguro, Ce Zhang, and Gustavo Alonso. 2018. ColumnML: Column-store machine learning with on-the-fly data transformation. *Proceedings of the VLDB Endowment* 12, 4 (2018), 348–361.

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

[46] Oliver Knodel, Paul R. Genssler, and Rainer G. Spallek. 2017. Migration of long-running tasks between reconfigurable resources using virtualization. *ACM SIGARCH Computer Architecture News* 44, 4 (2017), 56–61.

[47] Dirk Koch, Christian Haubelt, and Jürgen Teich. 2007. Efficient hardware checkpointing: Concepts, overhead analysis, and implementation. In *Proceedings of the 2007 ACM/SIGDA 15th International Symposium on Field Programmable Gate Arrays*. 188–196.

[48] Chris Lattner and Jacques Pienaar. 2019. MLIR primer: A compiler infrastructure for the end of Moore's law. (2019).

[49] Cheng Liu, Ho-Cheung Ng, and Hayden Kwok-Hay So. 2015. QuickDough: A rapid FPGA loop accelerator design framework using soft CGRA overlay. In *Proceedings of the 2015 International Conference on Field Programmable Technology (FPT)*. IEEE, 56–63.

[50] Yu Liu, Hantian Zhang, Luyuan Zeng, Wentao Wu, and Ce Zhang. 2018. MLBench: How good are machine learning clouds for binary classification tasks on structured data? *Proceedings of the VLDB Endowment* 11, 10 (2018), 1220–1232.

[51] Enno Lübbers and Marco Platzner. 2009. ReconOS: Multithreaded programming for reconfigurable computers. *ACM Transactions on Embedded Computing Systems (TECS)* 9, 1 (2009), 8.

[52] Divya Mahajan, Joon Kyung Kim, Jacob Sacks, Adel Ardalan, Arun Kumar, and Hadi Esmaeilzadeh. 2018. In-RDBMS hardware acceleration of advanced analytics. *Proceedings of the VLDB Endowment* 11, 11 (2018), 1317–1331.

[53] Aurelio Morales-Villanueva, Rohit Kumar, and Ann Gordon-Ross. 2016. Configuration prefetching and reuse for preemptive hardware multitasking on partially reconfigurable FPGAs. In *Proceedings of the 2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1505–1508.

[54] Ramadass Nagarajan, Karthikeyan Sankaralingam, Doug Burger, and Stephen W. Keckler. 2001. A design space evaluation of grid processor architectures. In *Proceedings of the 34th ACM/IEEE International Symposium on Microarchitecture. MICRO-34*. IEEE, 40–51.

[55] Neal Oliver, Rahul R. Sharma, Stephen Chang, Bhushan Chitlur, Elkin Garcia, Joseph Grecco, Aaron Grier, Nelson Ijih, Yaping Liu, Pratik Marolia, et al. 2011. A reconfigurable computing system based on a cache-coherent fabric. In *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig'11)*. IEEE, 80–85.

[56] Muhsen Owaida, Gustavo Alonso, Laura Fogliarini, Anthony Hock-Koon, and Pierre-Etienne Melet. 2019. Lowering the latency of data processing pipelines through FPGA based hardware acceleration. *Proceedings of the VLDB Endowment* 13, 1 (2019), 71–85.

[57] Muhsen Owaida, David Sidler, Kaan Kara, and Gustavo Alonso. 2017. Centaur: A framework for hybrid CPU-FPGA databases. In *Proceedings of the 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 211–218.

[58] Muhsen Owaida, Hantian Zhang, Ce Zhang, and Gustavo Alonso. 2017. Scalable inference of decision tree ensembles: Flexible design for CPU-FPGA platforms. In *Proceedings of the 27th International Conference on Field Programmable Logic and Applications (FPL'17)*. IEEE, 1–8.

[59] Sinno Jialin Pan and Qiang Yang. 2009. A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering* 22, 10 (2009), 1345–1359.

[60] Kolin Paul, Chinmaya Dash, and Mansureh Shahraki Moghaddam. 2012. reMORPH: A runtime reconfigurable architecture. In *Proceedings of the 2012 15th Euromicro Conference on Digital System Design*. IEEE, 26–33.

[61] Andrew Putnam. 2014. Large-scale reconfigurable computing in a microsoft datacenter. In *Proceedings of the Hot Chips 26 Symposium (HCS), 2014 IEEE*. IEEE, 1–38.

[62] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. 2014. A reconfigurable fabric for accelerating large-scale datacenter services. *ACM SIGARCH Computer Architecture News* 42, 3 (2014), 13–24.

[63] Benjamin Recht and Christopher Ré. 2013. Parallel stochastic gradient algorithms for large-scale matrix completion. *Mathematical Programming Computation* 5, 2 (2013), 201–226.

[64] Aaron Severance, Joe Edwards, Hossein Omidian, and Guy Lemieux. 2014. Soft vector processors with streaming pipelines. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*. ACM, 117–126.

[65] Aaron Severance and Guy Lemieux. 2012. VENICE: A compact vector processor for FPGA applications. In *Proceedings of the 2012 International Conference on Field-Programmable Technology*. IEEE, 261–268.

[66] Shai Shalev-Shwartz and Ambuj Tewari. 2011. Stochastic methods for L1-regularized loss minimization. *Journal of Machine Learning Research* 12, Jun (2011), 1865–1892.

[67] David Sidler, Zsolt István, Muhsen Owaida, Kaan Kara, and Gustavo Alonso. 2017. doppioDB: A hardware accelerated database. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 1659–1662.

[68] Bharat Sukhwani, Hong Min, Mathew Thoennes, Parijat Dube, Balakrishna Iyer, Bernard Brezzo, Donna Dillenberger, and Sameh Asaad. 2012. Database analytics acceleration using FPGAs. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*. ACM, 411–420.

[69] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2818–2826.

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

[71] Anuj Vaishnav, Khoa Dang Pham, and Dirk Koch. 2018. A survey on FPGA virtualization. In *Proceedings of the 2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 131–1317.

[72] Zeke Wang et al. 2019. Accelerating generalized linear models with MLWeaving: A one-size-fits-all system for any-precision learning. *Proceedings of the VLDB Endowment* 12, 7 (2019), 807–821.

[73] Jagath Weerasinghe, Raphael Polig, Francois Abel, and Christoph Hagleitner. 2016. Network-attached FPGAs for data center applications. In *Proceedings of the 2016 International Conference on Field-Programmable Technology (FPT)*. IEEE, 36–43.

[74] Loring Wirbel. 2014. Xilinx SDAccel Whitepaper.

[75] Peter Yiannacouras, J. Gregory Steffan, and Jonathan Rose. 2008. VESPA: Portable, scalable, and flexible FPGA-based vector processors. In *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. ACM, 61–70.

[76] Jiansong Zhang, Yongqiang Xiong, Ningyi Xu, Ran Shu, Bojie Li, Peng Cheng, Guo Chen, and Thomas Moscibroda. 2017. The Feniks FPGA operating system for cloud computing. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*. ACM, 22.

[77] Tong Zhang. 2004. Solving large scale linear prediction problems using stochastic gradient descent algorithms. In *Proceedings of the Twenty-first International Conference on Machine Learning*. ACM, 116.

[78] Zhuangdi Zhu, Alex X. Liu, Fan Zhang, and Fei Chen. 2018. FPGA resource pooling in cloud computing. *IEEE Transactions on Cloud Computing* (2018).

Received April 2020; revised June 2020; accepted August 2020