SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Heterogeneity-Aware Scheduling Algorithms for FPGA Workloads in Cloud Environments

Anand Krishna Rallabhandi

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Heterogeneity-Aware Scheduling Algorithms for FPGA Workloads in Cloud Environments

Heterogenitätsbewusste Planungsalgorithmen für FPGA-Workloads in Cloud Umgebungen

Author:	Anand Krishna Rallabhandi
Supervisor:	Prof. Dr.-Ing. Pramod Bhatotia
Advisor:	Dr. Ph.D. Atsushi Koshiba
Submission Date:	15.08.2023

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 15.08.2023 Anand Krishna Rallabhandi

Acknowledgments

I would like to express my sincere gratitude to my supervisor, Prof. Dr.-Ing. Pramod Bhatotia, for allowing me to conduct my thesis research at the Chair of Distributed and Operating Systems. His insightful suggestions and invaluable advice greatly enriched my thesis. It helped me gain deeper insights and a more comprehensive understanding of the topic. I feel privileged to have had the opportunity to work under Prof. Dr.-Ing. Bhatotia.

I also am deeply indebted to my advisor, Dr. Atsushi Koshiba, whose constant support, guidance, detailed research discussions, and insightful ideas have contributed significantly to the successful completion of this thesis. His constructive comments on the experiments and results have helped me to organize and present my work in a more systematic manner. His profound knowledge and unwavering encouragement were instrumental in shaping the success of my research.

I would like to express my gratitude to my family for their unconditional support in my academic pursuits. I am also grateful for the support and encouragement of my close friend Anurag Singh.

With the guidance of Prof. Dr.-Ing. Bhatotia's and Dr. Koshiba's support, along with the encouragement of my family and friends, I have been able to achieve a significant milestone in my academic career, and for that, I am truly grateful.

Abstract

The usage of Field Programmable Gate Array (FPGA) devices in cloud environments has gained traction in recent times as they are programmable and allow the user to program custom compute and data processing workloads and accelerate them. Vendors such as Amazon (AWS) offer instances that use FPGAs for the acceleration of such workloads. Typical workloads deal with a lot of data processing and computation such as machine learning, image processing, packet processing, etc. However, such offerings do not consider heterogeneity present among different devices. The same OpenCL kernel code compiled for different target FPGAs can exhibit different performance due to factors related to the hardware platform such as the number of look-up tables, memory bank size, and types of memory banks, and the application such as the frequency at which the application can run on the target device along with application characteristics (phases and the resource demand). Not considering such properties arising from heterogeneity would lead to lower throughput in workload execution in cloud environments.

In this work, we present and evaluate scheduling policies for cloud environments that support heterogeneous FPGAs. The policies take into consideration the performance difference among different FPGAs to increase the throughput and kernel execution time and make sure that the FPGAs are effectively utilized. Through empirical experiments, we establish that kernel clock frequency is an important parameter that affects the kernel execution time and Input Output (I/O) throughput and should be given special consideration while scheduling. Our results show that as the kernel clock frequency increases there is an increase in the kernel I/O throughput and a decrease in kernel execution time across different FPGAs and workloads.

Contents

1 Introduction

In the ever-evolving landscape of cloud computing, the integration of FPGA devices has emerged as a powerful and versatile solution for addressing the growing demands of high-performance workloads. Unlike traditional Application-Specific Integrated Circuits (ASIC), FPGAs are re-programmable semiconductor devices, allowing users to tailor their hardware functions according to specific computing needs. This inherent flexibility has made FPGA adoption increasingly prevalent in cloud environments, where optimizing performance and cost efficiency is critical. Since FPGAs can be reprogrammed on-the-fly to accommodate changing requirements, they provide a valuable advantage over fixed-function hardware accelerators. As businesses scale their operations, they can leverage FPGA-based acceleration to maintain high performance without the need to overhaul their entire hardware infrastructure. This capability aligns well with the core principles of cloud computing, where resource allocation and optimization are crucial for cost-effectiveness and seamless service delivery.

Cloud service providers include multiple types of FPGA with varying degrees of characteristics in the offerings. Determining which FPGA offering is suitable for a given application becomes important for the users so that they can optimize for performance (latency or throughput) or cost. The true performance of an application on a specific FPGA is only known after the application completes execution on the device. Predicting the performance prior to the execution is important, but remained challenging.

It is well established that bitstreams compiled for different types of FPGAs offer different performances because the FPGAs have different hardware components, even if compiled from the same kernel code. This thesis aims to predict the performance of a compiled kernel on a target FPGA prior to kernel execution using the information included in the compilation report provided by the synthesis flow.

A few parameters can be used to predict the performance of a kernel synthesized for a given FPGA. These parameters can include device-specific parameters such as the size of the FPGA, memory bandwidth available on the device, type of memory used, and parameters included in the synthesis report such as the number of LookUp Tables (LUTs) and RAMs used for the design, and the clock frequency of the design.

Our proposal entails the development of a novel scheduling algorithm that aims to enhance throughput, measured by the number of items processed within a given time period. To achieve this, our algorithm takes into account a parameter – kernel clock

frequency – which is determined by analyzing bitstream files through a parser. By considering the frequency parameter during scheduling decisions, we can intelligently allocate resources and prioritize tasks, thereby optimizing the overall throughput of the system.

As part of our approach, we are dealing with multiple bitstreams, each representing a different configuration for the FPGA. The challenge lies in selecting the most suitable bitstream for a specific task or workload. To streamline this process and simplify the user's experience, we have implemented an in-memory cache that supports housekeeping operations. This cache efficiently stores and manages relevant information about the available bitstreams, making it easier for the system to select and load the appropriate configuration for each task.

By leveraging this in-memory cache, we can avoid repetitive parsing of bitstream files and reduce the overhead associated with configuration changes. Consequently, the user benefits from improved system responsiveness and a smoother user experience. In forthcoming discussions, we will delve deeper into the specifics of our scheduling algorithm and the mechanisms behind our in-memory cache, and bitstream analyzer, providing a comprehensive understanding of how these components work together to maximize throughput and optimize FPGA utilization.

Experimental results by compiling applications in *Rosetta* and C++ Kernels in *Vitis Accel Examples* for two target FPGAs – *Xilinx Alveo u50* and *Xilinx Alveo u280* – and executing them show that frequency is a reliable predictor of performance.

2 Background

2.1 Field Programmable Gate Arrays

FPGAs are versatile integrated circuits with reconfigurable hardware capabilities, making them ideal for implementing digital circuits and systems. Unlike ASICs, FPGAs can be configured and reconfigured after manufacturing, allowing for rapid prototyping, design iteration, and customization. Their adaptability has led to their wide usage across various industries, including aerospace, telecommunications, data centers, and automotive electronics.

FPGAs consist of programmable logic blocks and configurable interconnects, enabling users to design custom digital circuits using an Hardware Description Language (HDL) like Verilog or Very High-Speed Integrated Circuit Hardware Description Language (VHDL). High-level synthesis tools can also convert higher-level programming languages such as C/C++ into FPGA configurations. The following two FPGAs are used for the research conducted in this thesis [Xil21; FPG21a].

Xilinx U50: The Xilinx U50 belongs to the Alveo™ family of data center accelerator cards, designed to accelerate diverse data center workloads, including machine learning, data analytics, and video transcoding. Based on the Xilinx UltraScale+ architecture, the U50 incorporates features like High Bandwidth Memory (HBM) and hardened Domain Specific Processor (DSP) blocks, delivering exceptional performance and power efficiency.

Xilinx U280: The Xilinx U280 is another member of the Alveo™ family, specifically tailored for demanding data center workloads. Built on the UltraScale+ architecture, the U280 offers larger FPGA resources and enhanced connectivity options. It integrates multiple HBM stacks, significantly boosting memory bandwidth for memory-intensive applications, such as machine learning training and inference, database acceleration, and financial computing.

Both the Xilinx U50 and U280 accelerator cards have the potential to greatly enhance data center efficiency and reduce time-to-market for innovative applications in the realm of high-performance computing. Table 6.1 in Chapter 6 shows the details of the two FPGA boards [AMD23a; AMD23b].

Our research focuses on enhancing the performance and flexibility of PCIe-connected FPGAs, a widely adopted deployment model. This FPGA architecture encompasses

two distinct regions: the dynamic region and the static region, also known as the Shell. The dynamic region offers the ability for users to create their hardware logic on-the-fly during runtime. This reconfigurability helps users to adapt their hardware accelerators to the specific requirements of their applications dynamically.

On the other hand, the static region, or Shell, plays a crucial role in facilitating the seamless interaction between the dynamic region and the external environment. Within the Shell, essential glue logic components are present which are used to connect accelerators residing in the dynamic region to the outside world. These glue logic elements comprise interfaces to the host system, interconnects to various onboard devices such as Double Data Rate (DDR) memory and network ports, and Direct Memory Access (DMA) controllers. These integral components ensure efficient data exchange and communication between the FPGA and the rest of the system.

Moreover, the Shell holds additional significance as it takes on the responsibility of providing a user-friendly interface for reconfiguring the dynamic region. Through this interface, users can easily modify the hardware logic in the dynamic region as per their changing application needs. As a result, the Shell empowers users to optimize FPGA resources, minimize overheads, and tailor the FPGA's functionality to suit the specific computational tasks at hand. Our work targets this Peripheral Component Interconnect Express (PCIe)-connected FPGA architecture to enhance performance and versatility in cloud-based computing environments.

2.2 Programming Model

The FPGA programming model is split between the host (Central Processing Unit (CPU)) and the device (FPGA). The host programming model enables task offloading on the device, whereas the device model represents the offloaded code.

Various technologies, such as oneAPI [Int23c], OmpSs [Bos+18], EngineCL [NBB20], Kokkos [Tro+22], OpenCL [Gro21], and OpenACC [Ope21], have undergone development or improvements to provide FPGA support. By utilizing these technologies, it becomes feasible to blend execution across different architectures, although some approaches may offer a higher level of ease compared to others. Our work targets OpenCL, an open standard for parallel programming of heterogeneous devices [Gro21]. OpenCL offers a platform-agnostic programming model and is widely adopted by major FPGA vendors [Xil21; FPG21b]. OpenCL code consists of two parts: the host code running on a CPU, and the kernel code on the device (*kernel*). OpenCL offers *OpenCL APIs*, which allow the host code to launch kernels on the target device and manage the device-side memory.

For the device (FPGA), we mainly target software-controllable kernels, which expose

their control registers to the host code. The other kernel types (e.g., data-driven) can also be supported. Besides this, we do not impose any limitations on kernel code programming—all platform-dependent languages are supported (e.g., HDL [TM21; Bha21], HLS [Sas+06; Aue+10; Gro21; Koe+18; Leb+05], and DSLs [Bac+12; CDL13; Koe+16; Leb+12])).

Xilinx, one of the leading FPGA manufacturers, offers a comprehensive suite of tools to program and configure their devices. The toolflow comprises several stages, each catering to specific aspects of FPGA design and implementation.

High-Level Synthesis (HLS): The process begins with high-level synthesis, where developers describe their design using a high-level programming language like C or C++. HLS tools, like Xilinx's Vivado HLS, translate these software descriptions into equivalent hardware descriptions, such as Register Transfer Level (RTL) code.

RTL Synthesis: The next step involves converting the RTL code generated from HLS into a gate-level representation. This process is called RTL synthesis. Xilinx's Vivado Design Suite contains the RTL synthesis tool, which optimizes the RTL code and maps it to the target FPGA's library of primitive elements.

Implementation: After RTL synthesis, the implementation phase starts. The tools in this phase map the gate-level netlist onto the target FPGA's specific resources, such as LUTs, flip-flops, and I/O blocks. Xilinx's Vivado Design Suite includes the implementation tools required for this stage.

Bitstream Generation: The implementation results in a bitstream file, which is a binary representation of the configured FPGA. The bitstream file is then loaded onto the FPGA device to make it operate according to the specified design. Xilinx's Vivado generates the bitstream file using configuration data extracted during the implementation process.

Synthesis Reports: During the FPGA development process, various synthesis reports are generated at different stages of the toolflow. These reports provide valuable insights into the design's performance, resource utilization, and potential issues. Some of the essential information included in the synthesis reports are: *Timing Reports*: Timing reports analyze the critical paths in the design, determining the maximum clock frequency the design can achieve without violating timing constraints. It highlights setup and hold violations, helping developers optimize their design for better performance.

Resource Utilization: Resource utilization reports detail how much of the FPGA's resources, such as LUTs, flip-flops, Block Random Access Memory (BRAM), and DSP slices, are utilized by the design. This information is crucial for evaluating the design's efficiency and identifying potential bottlenecks.

Area Reports: Area reports focus on the physical size of the design on the FPGA, providing insights into the chip area occupied by the synthesized design.

Power Reports: Power reports estimate the power consumption of the design when running on the FPGA. This information is essential for power-sensitive applications,

enabling developers to optimize the design for power efficiency.

Constraint Violations: The synthesis reports also flag any violations of constraints set by the developer, such as maximum operating frequency or specific resource limitations. These violations help the developers identify areas for improvement and debugging.

2.3 Scheduling Algorithms

In the context of FPGA-based applications, the performance characteristics of bitstreams generated for different FPGA targets can vary significantly. It is not uncommon for a bitstream generated for Xilinx U50 to exhibit faster execution for a particular application compared to the bitstream generated for U280, and vice versa depending on the workload. This variability emphasizes the importance of carefully selecting the appropriate target device to achieve optimal application execution speed.

In real-world execution environments, a series of applications need to be scheduled on a set of available FPGAs within the system. Given that different FPGAs offer varying capabilities, determining the schedule of execution becomes a critical task. Offline and online solutions can be employed to tackle this scheduling problem.

Online solutions handle the scheduling dynamically, without prior knowledge of each task's execution time or configuration time. This approach grants greater adaptivity to diverse applications and avoids the time-consuming step of profiling applications in advance. The online scheduler must make real-time decisions, assigning a specific time period to each arriving task. Within this allocated timeframe, the task can be loaded onto the FPGA and executed. The efficiency of the online scheduler directly influences the overall performance of the entire system, as it strives to make optimal runtime decisions based on available information.

On the other hand, an offline solution optimizes the scheduling decisions during the application's compilation phase. In this work, the focus is on addressing the offline scheduling problem, which involves precomputing the best scheduling decisions based on the characteristics of the generated bitstream and the reports provided as part of the bitstream generation process.

3 Overview

3.1 System Goals

The goal of this work is to build scheduling policies that increase the throughput of task execution and utilize the resources on a cluster effectively. To achieve this goal, we first establish a metric that can be used to effectively schedule workloads on heterogenous FPGAs. We find that kernel clock frequency is a simple yet effective metric for scheduling.

While we aim to maximize throughput (or performance) in this work, one can consider optimizing the system for an alternate metric. For example, the goal can be to minimize the total energy consumption of a data center and this may result in optimizing for energy per task with suboptimal throughput. Another metric can consider both energy and throughput by weighing them differently. An example metric can be energy-delay product that weighs them equally. Another example can be energy-delay-squared, which weighs throughput more than energy. In this work, we assign full weightage to throughput. Exploring other metrics is part of our future work.

Scheduling tasks to heterogenous FPGAs requires us to work with multiple bitstreams synthesized for different target FPGAs for which we use *Redis* to manage file storage.

3.2 System Design Overview

The system follows the *primary-worker* architecture consisting of a leader that schedules tasks to be deployed on worker nodes. The leader is designed as an event-driven harness that, in addition to scheduling tasks submitted by the users, has various components such as the bitstream database, and bitstream analyzer. The bitstream database is required for storing the different bitstreams the user submits after synthesizing them for different target FPGAs. The bitstream analyzer parses the bitstreams and extracts the kernel clock frequency value that will be used for ranking them. When the leader is initialized it waits for the user to either submit applications for execution or bitstreams to upload. Once a user submits the host code applications for execution, the harness looks for the bitstreams associated with these applications in the database. It picks the most appropriate bitstream and then deploys it for execution on the target FPGA,

on a worker, that is able to run this bitstream. The worker nodes maintain a list of FPGAs available on them and execute the applications deployed by the leader node. On initialization, they connect to the leader node and inform it of the list of FPGAs they manage.

Figure 3.1 provides an overview of the entire system. A *primary-worker* architecture that is event-driven, along with a database to handle multiple synthesized bitstreams, and a bitstream analyzer to rank the bitstreams makes it easy for users to accelerate their applications in heterogeneous environments.

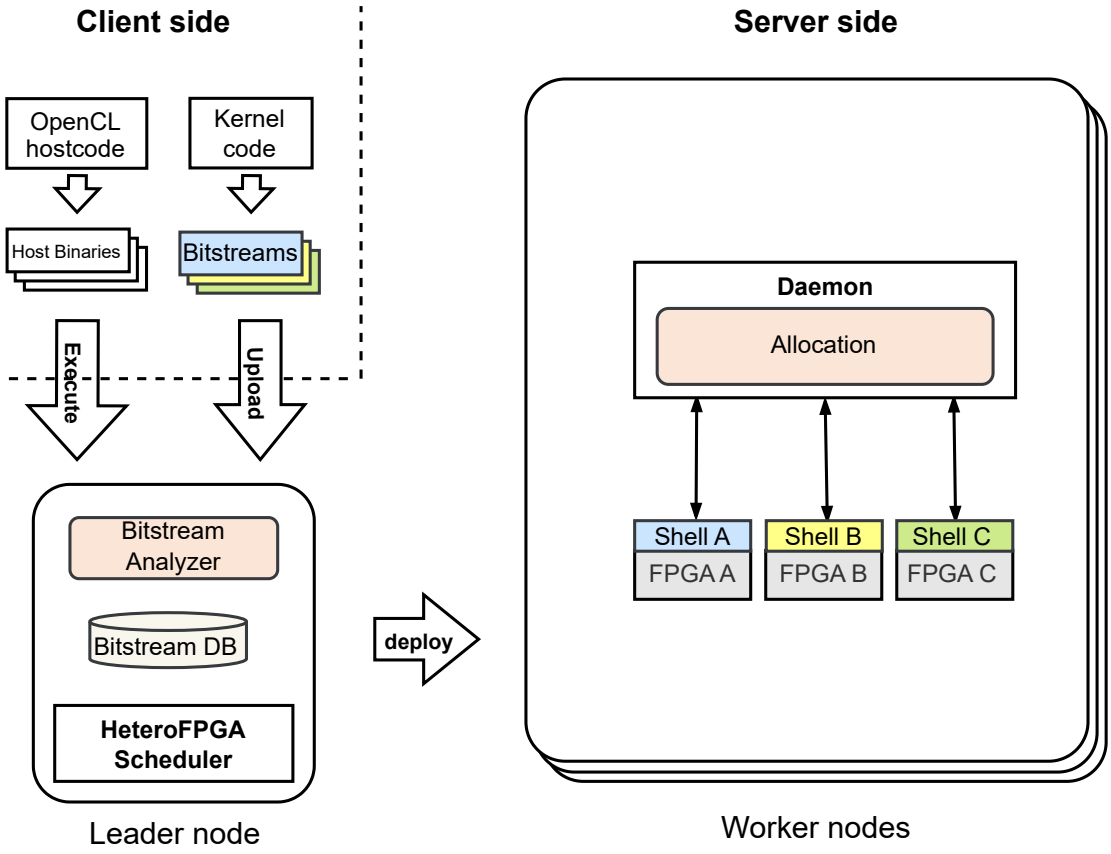


Figure 3.1: Design overview of a primary-worker system.

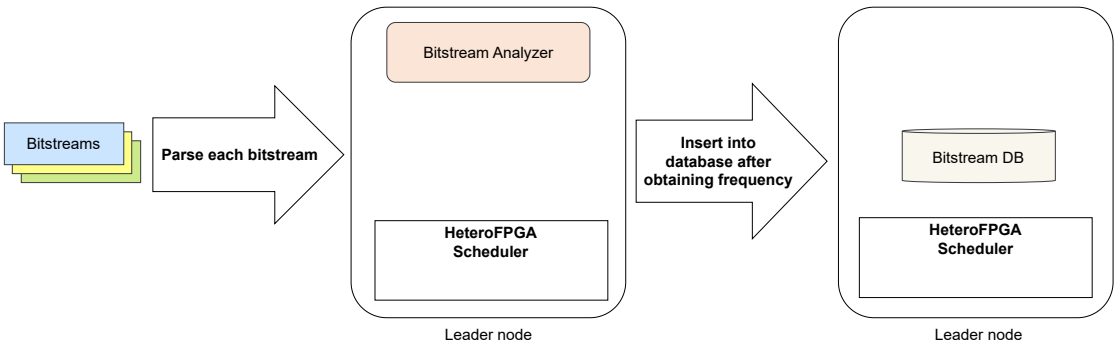


Figure 3.2: Workflow outlining the process of *Bitstreams* upload.

3.3 System Workflow

3.3.1 Bitstream upload and processing

Figure 3.2 demonstrates the workflow outlining the process of bitstreams upload to the bitstream database. The user, for an application, first synthesizes bitstreams for different targets at the maximum clock frequency and uploads them to the leader node.

The leader node consists of a bitstream analyzer, an FPGA scheduler, and a bitstream database. This leader node serves as a central control point for managing the deployment and execution of tasks for the FPGAs within a distributed system. The bitstream analyzer is a software module that interprets binary configuration files of the FPGAs. It extracts file type information, device details, and the kernel clock frequency used by the scheduling algorithm.

The box labeled HeteroFPGA scheduler represents the heterogeneous FPGAs scheduler. This scheduler manages the allocation and deployment of applications across a mix of different FPGA architectures within a system.

The bitstream database in a leader node is a centralized repository dedicated to managing and organizing bitstreams for different FPGAs. It stores multiple bitstream versions indexed in a manner that retrieval is fast and efficient.

3.3.2 Application execution

Figure 3.3 demonstrates the workflow that outlines the process of application execution. The user compiles the host code binaries corresponding to the bitstreams previously uploaded and submits them to the leader node. The leader node then executes the scheduling algorithm and picks a node that has the FPGA available to execute the application. The leader node then sends the host code binary along with the

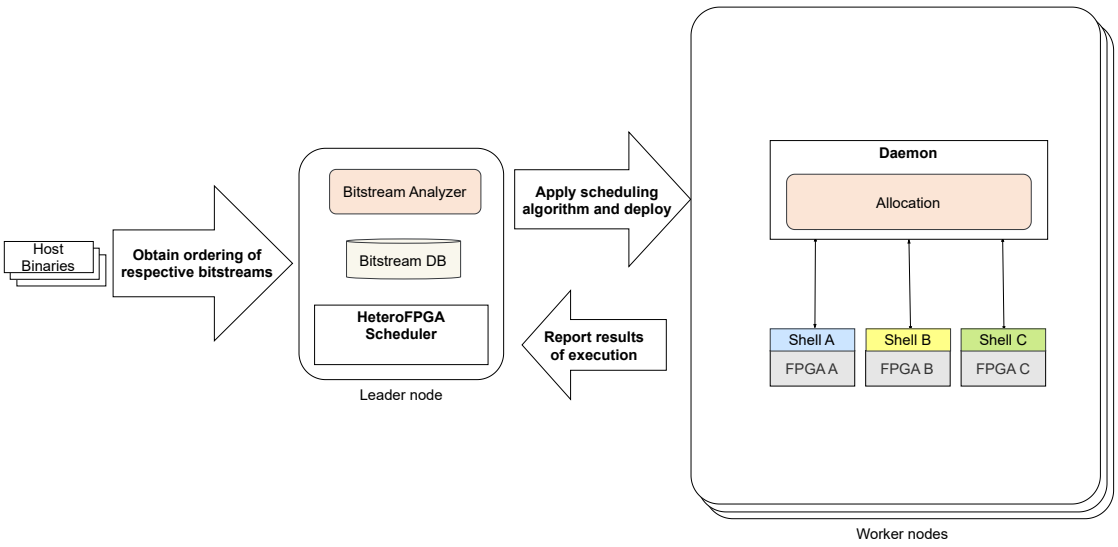


Figure 3.3: Workflow outlining the process of *Application* execution

corresponding bitstream to the selected worker node. The worker node employs a daemon process that executes applications on different FPGAs connected to it. Once the execution is complete the execution results are submitted back to the leader node.

4 Design

4.1 Leader Node

The leader node has an event-driven process that spawns ephemeral threads to handle execution requests and upload requests from the user. It also maintains persistent threads to handle worker node connections. It maintains a queue of applications submitted by the user for execution. It interacts with and utilizes the different components explained in the following sections.

4.1.1 Bitstream Analyzer

The bitstream analyzer is a component in the leader node that is used for parsing the synthesized bitstreams uploaded by the user. The parser supports bitstreams targetted for FPGAs from Intel and Xilinx. The parser is built to be aware of the difference between the bitstreams of different vendors. The bitstream generated by the intel compile is an ELF file whereas the bitstream generated by the Xilinx compiler is an AXLF file. The parser extracts the kernel clock frequency from the bitstreams which is then used to rank them. The flowchart in Figure 5.1 explains the internal working of the parser in more detail.

4.1.2 Bitstream Database

A bitstream database is used to handle multiple bitstreams efficiently. A *Redis* datastore is deployed alongside the event-driven process that handles requests from the user and manages workers. When the leader node recieves an upload request from the user, the bitstreams are parsed and their paths are stored inside the *Redis* datastore. These paths will be utilized later when deploying the application to a worker.

4.2 Worker Nodes

The daemon process is tasked with maintaining the FPGAs on a worker node and executing requests forwarded by the leader node and reporting back results to the

Algorithm 1: Scheduling Algorithm

```
Input  : Task list, Nodes map
Output: Picked Task, Picked Node, Index of Bitstream to execute
/* Pick a task, pick a bitstream in it, and pick a node to run the bitstream on. */
schedule(tasks, nodes)
begin
    picked_task, picked_node, bitstream_index ← NULL, NULL, -1;
    foreach task in tasks do
        if task.state == available then
            picked_task ← task;
            break;
    end
    if picked_task == NULL then
        // Nothing to execute as of now.
        return picked_task, picked_node, bitstreamIndex;
    for i = 1 to picked_task.num_bitstreams do
        // Get the target fpga on which this will be executed.  For e.g., Xilinx u50.
        target_type ← picked_task.bitstreams[i].target_type;
        foreach node in nodes[target_type] do
            // Pick the first node that can run this.
            if node.state == available then
                picked_node ← node;
                bitstream_index ← i;
                return picked_task, picked_node, bitstream_index;
        end
    end
    return picked_task, picked_node, bitstream_index;
end
```

leader node. It is designed to be event-driven responding to requests from the leader node and monitoring the state of execution of the applications.

4.3 Scheduling Algorithm

Next, we introduce the scheduling algorithm that takes into account the performance gaps among different FPGA platforms. As a consequence of ordering bitstreams by frequency earlier, sequential picks will always ensure that the scheduler picks the bitstream with a relatively better performance which results in better throughput. Even in cases where the worker nodes having the required FPGA to run the best bitstream are busy, we can pick another bitstream without losing the overall throughput as we do not have to consider their relative performance due to such ordering.

Algorithm 1 shows the proposed scheduling algorithm. Nodes is a hash map whose key is an enum indicating the type of FPGA and values indicate a list of nodes that contain that FPGA. (Just like an *std::map* with enum keys and a list of nodes as values.)

5 Implementation

5.1 Bitstream Analyzer

This section is dedicated to explaining the low-level details of how the bitstream analyzer deals with different bitstreams. As described in the previous chapter, the parser is used to process bitstreams. Our implementation of the parser supports two FPGA vendors, namely Intel and Xilinx, which makes it important to make a distinction between them in the implementation. As mentioned previously, the Intel bitstream is an Executable and Linkable Format (ELF) file and the Xilinx bitstream is an xclbin container format (AXLF) file. The parser reads the first eight bytes of the bitstream that contains a *magic word*. The magic words for ELF and AXLF file formats are `ELFMAG ("\177ELF")` and `"xclbin2\0"`, respectively. Using these markers, the parser makes a distinction between the two.

The parser next starts to look for the section in the file that contains information related to kernel clock frequency. For Xilinx bitstreams, the data related to kernel clock frequency is found in the section named `CLOCK_FREQ_TOPOLOGY`, and the value of the clock frequency is stored in the field with type `CLOCK_TYPE::CT_DATA`. Since Xilinx provides native XRT Application Programming Interface (API)s, it is easier to parse Xilinx bitstreams without much ad-hoc code. For Intel bitstreams, the data related to kernel clock frequency is found in the section named `.acl.quartus_json`, and the value of the clock frequency is stored in a field called `quartusFitClockSummary`. Due lack of API support from Intel to handle bitstream files, the parsing logic is involved and ad-hoc.

The workflow of the parser is summarized in the flowchart shown in Figure 5.1. The first step is to open the bitstream file and read the first eight bytes. Based on this, we determine whether it is an AXLF file. If yes, then we parse as per AXLF specifications. If it is not an AXLF file then we check whether it is an ELF file. If yes, we parse as per ELF specifications. If it is not, then we report an error message. After successfully parsing the data, we report it. Finally, we terminate after reporting.

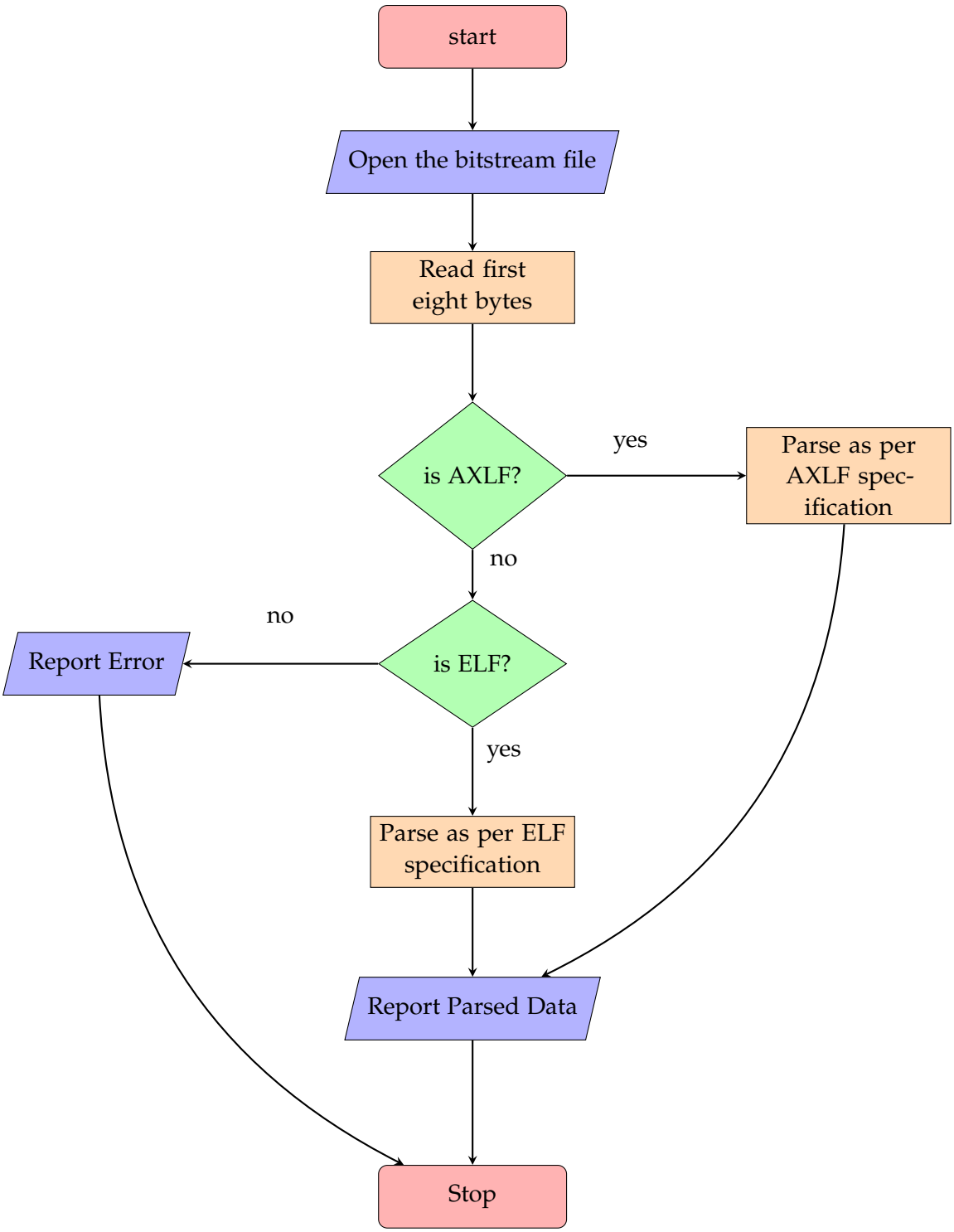


Figure 5.1: Flowchart describing the functionality of the parser.

5.2 Bitstream Storage

In this section, we describe the details of how the bitstreams are stored and managed within the system. To efficiently handle multiple bitstreams, the system leverages the capabilities of *Redis*, a popular in-memory data store known for its high-performance and low-latency characteristics.

To make operations in the scheduler easy, two different redis types are utilized. The first type is a *sorted set* in which the bitstreams of different target FPGAs are ranked according to the value of the kernel frequency, obtained earlier from the analyzer. The second type is a *hash* in which the path to the bitstreams is recorded as a *value* for each target FPGA type as the *field*. The usage of these two data types within Redis streamlines the bookkeeping process. The sorted set allows for automatic ranking based on kernel frequencies, eliminating the need for manual sorting. It provides an efficient way to identify the top-performing bitstreams quickly. On the other hand, the hash data structure simplifies the process of associating each target FPGA with its respective bitstream path, ensuring easy retrieval when required.

5.2.1 Leveraging Sorted Set

The *sorted set* type that we leverage ranks/orders the bitstreams by frequency for each target. The advantage of using a *sorted set* is that each time we want to obtain the ranking of target bitstreams, it is free (incurring no runtime overhead). Only when the sorted set is built by adding one bitstream at a time to the data structure after parsing, some runtime overhead is incurred. Adding a bitstream to the sorted set is an $\mathcal{O}(\log N)$ operation. This offers a favorable tradeoff as it scales for environments where the number of target devices and/or the number of bitstreams is large (1 bitstream per target $\Rightarrow$ N bitstreams for N targets). Figure 5.2 shows how the bitstreams for the task *face_detection* are stored, ranked by frequency for each target FPGA.

5.2.2 Leveraging Hash

We leverage *hash* to maintain a mapping between the target device and the actual path to the bitstream. It is used when the leader node deploys the task to a worker. The complexity of adding an element to a hash in redis is $\mathcal{O}(1)$, which means it is a constant-time operation regardless of the hash size. Similarly, reading an element from the hash also has a complexity of $\mathcal{O}(1)$, making these operations highly efficient. This offers a low-cost solution that scales well for environments with many target devices and bitstreams. Figure 5.3 shows how the paths to the bitstreams for the task *face_detection* are stored for each target FPGA.

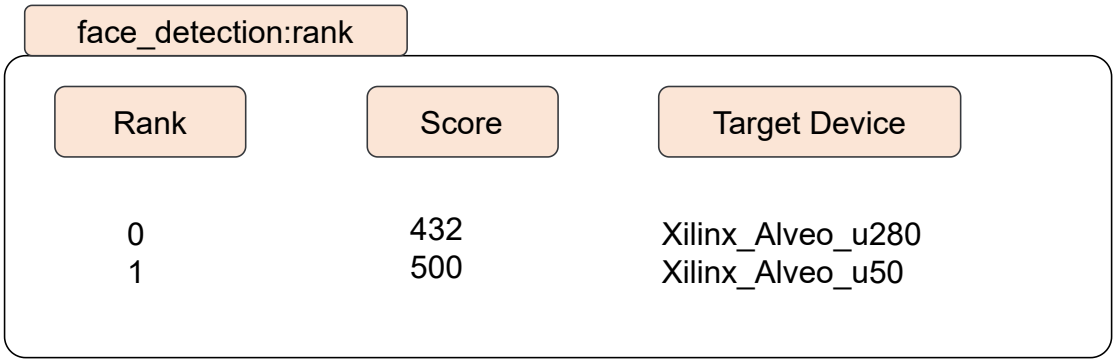


Figure 5.2: *Redis sorted set* to rank the uploaded *bitstreams* by kernel clock frequency for the task *face_detection*

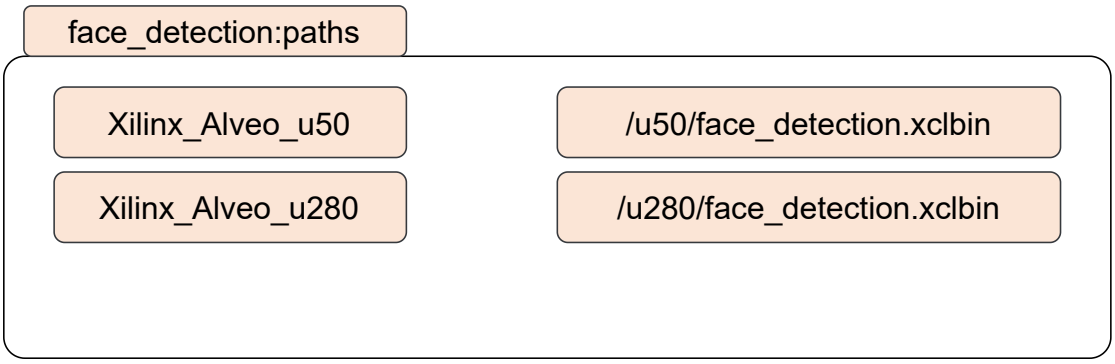


Figure 5.3: *Redis hash* to store the paths of the uploaded *bitstreams* for the task *face_detection*

5.3 Task Scheduling and Execution

The scheduling algorithm depicted in Algorithm 1 operates independently of specific parameters, such as the kernel clock frequency. This intentional design feature ensures that additional parameters can be easily incorporated without requiring modifications to the core algorithm. As previously discussed, the ordering of *Bitstreams* is determined by their scores, which correspond to the value of the kernel clock frequency. This separation of concerns effectively decouples the model definition and analysis from the scheduling algorithm. As a result, developers can focus on optimizing the model to enhance the target metric, such as improving throughput, and seamlessly integrate the updated model into the system. This process could be as straightforward as adjusting the score calculation for each bitstream, enabling smoother experimentation and model

refinement.

5.4 Application Execution

When a user submits host code binaries to execute an application, the system creates a *Task* object by ranking the bitstreams retrieved from the *sorted set* and associating their paths from the *hash* corresponding to the application. This newly formed Task object is then added to a list of tasks, denoted as L, which the scheduler actively maintains. Subsequently, the scheduling algorithm is executed to select an available node and determine a task that is ready for execution. Once the node and the task are identified, the scheduler deploys the respective host code binary and bitstream pair for execution on the chosen node.

6 Evaluation Methodology

6.1 Evaluation Setup

Empirical experiments and scheduling policies are executed on the server *momiji* which has two Xilinx FPGA boards, namely, Xilinx Alveo u50 and Xilinx Alveo u280. It is configured as the leader and worker. The server is equipped with Intel(R) Xeon(R) Gold CPU with 28 cores and 256 GB Dynamic Random Access Memory (DRAM), running Ubuntu 20.04 with Linux kernel version 5.15.0. Two types of Xilinx FPGA cards, attached to the server through a PCIe gen3 x16 interface are used for the experiments. They are Xilinx Alveo U50 and Xilinx Alveo U280 and their details are described in Table 6.1.

Feature	U280	U50
Total electrical load	215W	75W
Thermal cooling solution	Active	Passive
Form factor	Full height, full length, dual width	Half height, half length
Weight	1187g	300-325g
HBM2 total capacity	8 GB	8 GB
HBM2 total bandwidth	460 GB/s	316GB/s
LUTs	1,304K	872K
Registers	2,607K	1,743K
DSP slices	9,024	5,952
PCIe interface	Gen3 x16, Gen4 x8, CCIX	Gen3 x16, Gen4 x8, CCIX
Network interface	2x QSFP28	1x QSFP28

Table 6.1: Comparison between Xilinx U280 and U50 FPGA Boards.

6.2 Benchmarks

6.2.1 Rosetta

Rosetta benchmark suite is a comprehensive collection of realistic HLS benchmarks designed for software programmable FPGAs. It introduces six fully developed applications, including machine learning workloads like logistic regression and neural network inference, as well as real-time video processing applications such as image rendering and face detection. Rosetta focuses on complete applications rather than small kernel programs and it specifies realistic design constraints for each benchmark.

The suite provides both unoptimized software versions and optimized HLS implementations written in either C++ or OpenCL. The benchmarks encompass compute-bound and memory-bound applications, featuring diverse sources of parallelism, which are effectively harnessed in the current HLS implementations. These implementations leverage Instruction-Level Parallelism (ILP) through fine-grained pipelining and Task-Level Parallelism (TLP) by overlapping the execution of different kernels. Additionally, each benchmark has specific design objectives tailored to the use-case scenario, where machine learning applications require either low latency or high throughput, while video processing applications must achieve a real-time throughput target of at least 30 frames per second.

The following is a brief description of the workload from Rosetta used in this work:

- *3D rendering:* The 3D rendering benchmark is a compute-bound application that generates 2D images from 3D triangle mesh models, achieving 30 frames per second throughput. It utilizes a typical image processing pipeline with four kernel functions and employs dataflow optimization to overlap pipeline stages, ensuring efficient hardware module utilization for processing 3192 triangles.

- *Face detection:* The face detection application adopts the Viola-Jones algorithm to detect human faces in a given 320x240 greyscale image. It utilizes an image pyramid and integral images to apply a set of cascaded classifiers to a fixed-size window, resulting in the detection of human face positions and sizes. The application aims for a throughput target of 30 frames per second and faces hardware constraints, including limited on-chip storage and routing resources. The major compute kernels are image scaling and cascaded classifiers, with optimizations applied to exploit data-level parallelism and irregular memory access patterns. Customized memory partitioning and modified window buffering techniques are utilized to enhance kernel frequency and efficiently construct the integral image.

- *Spam detection:* The spam detection application utilizes Stochastic Gradient Descent (SGD) to train a Logistic Regression (LR) model for classifying spam emails.

The input consists of 5000 emails represented as 1024-dimensional vectors containing relative word frequencies stored as 16-bit fixed-point numbers. The design focuses on minimizing training latency, and critical resource constraints include the number of hardened DSP blocks and on-chip storage size. To optimize performance, the design exploits datatype customization, approximation of complex arithmetic operations, and dataflow optimization for communication and compute overlap. Despite these optimizations, the application is classified as memory-bound due to the limitation of on-chip memory and the necessity for streaming training instances from off-chip memory.

- *Optical flow:* The optical flow application captures the motion pattern of objects between consecutive image frames using the Lucas-Kanade method. It generates a 2D vector field showing the movement of each pixel in the input image frames. The benchmark targets real-time throughput of 30 frames per second and faces limited on-chip storage, leading to the use of dataflow optimization and specialized memory structures like line buffers and window buffers to maximize memory efficiency. The current implementation is classified as memory-bound due to its reliance on off-chip memory bandwidth, but future optimizations are planned to improve throughput by exploiting data reuse between input frames.

6.2.2 Vitis Accel Examples

The Vitis Accel Examples provide a collection of examples to facilitate application optimization for Xilinx PCIe FPGA acceleration boards and Xilinx System-on-Chip (SoC) FPGA acceleration boards. These examples are designed to be readily compiled and executed on Vitis-supported boards and accelerated cloud service partners. The examples are provided with shells, and compilation and execution guides. The examples cover a wide range of topics, such as hello world, AIE kernels, host programming in C++, Python, and Open Computing Language (OpenCL), performance optimization, system-level design, validation, and more. Below is a brief description of the examples used in this work:

- *Array partition:* The example demonstrates how array partitioning in an HLS kernel can improve performance, specifically showcasing matrix multiplication. Two kernels are implemented: *matmul* for simple matrix multiplication and *matmul_partition* using array partitioning. The *#pragma HLS array partition* is used to partition an array into smaller memories, with the example utilizing the complete partitioning for one dimension of a local matrix array.
- *Burst read/write:* The example demonstrates the use of AXI4-master interface for burst read and write operations, aimed at increasing data throughput. The

code showcases how multiple items can be efficiently read from global memory to the kernel's local memory in a single burst, utilizing low memory access latency and maximizing the bandwidth provided by the *m_axi* interface. The *for* loops in the code need to be pipelined with an II (Initiation interval) of 1, and memory addresses for read/write operations should be contiguous for optimal performance.

- *BIND OP and STORAGE:* This example demonstrates vector addition, highlighting the use of the *bind_op* and *bind_storage* pragmas to specify hardware resources and their properties for a more efficient implementation style.
- *Critical Path:* This example illustrates two coding styles for processing image pixels in the *Apply_watermark* kernel. The normal coding style with a global variable update in each iteration leads to a critical path issue and degraded timing, while the better coding style avoids this problem, resulting in improved design timing. The focus is on avoiding critical paths in kernels for better performance.
- *Custom Data Type:* This example showcases RGB to HSV conversion and demonstrates the usage of custom data types in C-based kernels, allowing efficient burst transfers between kernel and global memory with Xilinx HLS Compiler's support for custom data types.
- *Dataflow Using HLS Stream:* This example illustrates vector addition using HLS Dataflow, which enables scheduling multiple tasks together for increased throughput. The implementation showcases the use of HLS Stream datatype and the #pragma HLS dataflow directive to achieve task-level parallelism, optimizing communication with FIFOs instead of RAMs for more efficient data processing.
- *Dataflow Using Array of HLS Stream:* This example showcases the utilization of HLS streams with varying depths for vector addition operations, efficiently transferring data between global memory and kernels using the overloaded $<<$ operator and HLS stream functions.
- *Loop Dependency Inter:* This example demonstrates how the #pragma HLS DEPENDENCE can be used to specify additional dependency details for variables in consecutive loop iterations, enabling the V++ compiler to optimize loop executions by recognizing dependencies and generating pipelines with lower II counts. The use of #pragma HLS DEPENDENCE helps in avoiding unnecessary interdependencies, leading to better performance compared to the default unoptimized design with larger II factors.

- *Global Memory Two Banks:* This example showcases the utilization of multiple DDRs to create buffers in different DDR (Global memory) banks, allowing for improved performance by maximizing memory bandwidth for data transfers in the watermark application. By specifying memory banks using sp tags in the configuration file, the input image is placed in one DDR bank, and the kernel reads the input image from that bank and writes the output image to another DDR bank.

- *Stream Chain Matrix Multiplication:* This example showcases the use of *ap_ctrl_chain* in HLS kernels to improve performance by overlapping multiple enqueue calls of the kernel and starting the next kernel operation before completing the current one, demonstrated through cascaded matrix multiplication functionality. The *ap_ctrl_chain* interface mode provides block-level control ports for starting, continuing, and indicating when the design is idle, done, or ready for new input data, offering efficient chaining of Vivado HLS blocks.

- *Local Memory Two Parallel Read/Write:* This example showcases vector addition and how to effectively utilize both ports of local memory (2-port BRAM) in kernels for improved performance. By using the #pragma HLS UNROLL directive, loops with no dependencies between iterations can be unrolled, allowing multiple copies of the loop body to run in parallel, maximizing the concurrent utilization of the two ports.

- *Loop Pipelining:* This example illustrates the use of loop pipelining to enhance the performance of a kernel. By leveraging pragma HLS PIPELINE in read and write loops, the pipelined kernel achieves shorter execution times, enabling burst transfers and optimizing the usage of *m_axi* interface bandwidth. The comparison with the non-pipelined kernel (*kernel_vadd*) showcases the benefits of loop pipelining in improving overall performance.

- *Loop Reordering:* This example showcases matrix multiplication and highlights how reordering the loops can lead to better performance by improving the Initiation Interval (II) factor. By using #pragma HLS *ARRAY_PARTITION* and changing the loop order, the achieved II can be enhanced from 64.

- *Array Block and Cyclic Partitioning:* This example demonstrates how to enhance kernel performance using array block and cyclic partitioning techniques. By using #pragma HLS *ARRAY_PARTITION*, the array is divided into smaller arrays with increased read and write ports, improving throughput for repeated access operations in matrix multiplication, thereby reducing latency and enhancing performance. The three partitioning options (cyclic, block, and complete) allow

efficient decomposition of the array, further optimizing read/write operations for better design throughput.

- *Port Width Widening:* This example demonstrates how Vitis HLS allows users to resize the port width of kernel interface ports for improved resource utilization while maintaining performance. Users can configure the port width through pragma options, TCL settings, or interface pragmas, with certain rules to follow. Five kernels are provided, each showcasing different ways to set the port width, including default settings, auto port width widening, pragma-specified multiple bundles, explicit port width in a tcl file, and interface pragma-based port width allocation.

- *PLRAM Memory Access:* This example illustrates the usage of PLRAM (Private Local RAM) and its integration with the Vitis memory subsystem in a simple matrix multiplication application. PLRAM is a small shared memory constructed using FPGA fabric on-chip memory resources, offering rapid data storage with low latency and behaving similarly to DDR memory. By using sp tags in the mmult.cfg file, PLRAM can be assigned to a buffer for efficient data sharing between application kernels with minimal access latency.

- *Vector Addition:* This example demonstrates a simple vector addition kernel using HLS Dataflow for improved throughput. The kernel has one *s_axilite* interface for host application configuration, but multiple master interfaces can be created for simultaneous memory access. The design uses STREAM pragma to use FIFOs instead of RAMs for more efficient communication, and the vector addition is divided into four sub-tasks, executed concurrently using Dataflow. The purpose of this code is to introduce users to application development in Vitis tools while showcasing efficient task scheduling and memory access techniques.

- *Shift Register:* This example showcases two implementations of an FIR filter discrete convolution operation, with one naive implementation using shift registers and another optimized implementation using #pragma HLS $ARRAY\_PARTITION$ to provide all array values simultaneously. The optimized implementation allows for efficient pipe-lining and loop unrolling, enabling the shifting of values in registers each clock cycle for improved performance. The FIR filter requires values of all elements for each output array element, and the implementation demonstrates effective kernel optimization techniques like loop unrolling and array partitioning.

- *Systolic Array:* This example showcases systolic array-based algorithm design for efficient matrix multiplication on FPGAs. The mmult kernel demonstrates how lo-

cal matrices A and B can be partitioned using #pragma HLS *ARRAY_PARTITION* along rows and columns to optimize the computation of elements in matrix C, resulting in efficient matrix operations using systolic array principles.

- *Wide Memory Read/Write:* This example showcases vector addition, utilizing $ap\_uint < 512 >$ datatype to enable V++ to determine the memory datawidth for transfers between the kernel and global memory, allowing concurrent transfer of up to 512 bits for improved efficiency. Including *ap_int.h* is necessary to use these datatypes, and this approach ensures wide memory access and efficient use of memory bandwidth for burst read and write operations.

6.2.3 Workload Categorization

Based on the description of each workload, we categorize the workload as (1) compute-bound, (2) memory-bound, (3) neither, or (4) balanced. Table 6.2 shows our categorization, which is qualitative. This categorization can be improved with further experiments. For example by running each workload at two different frequencies, one can quantify the performance sensitivity of the workload to change frequency.

Suite	Workload	Category
Rosetta	3D rendering	Balanced
	Face detection	Compute-bound
	Spam detection	Memory-bound
	Optical flow	Memory-bound
Vitis Accel Examples	Array partition	None
	Burst read/write	Memory-bound
	BIND OP and STORAGE	Compute-bound
	Critical path	Compute-bound
	Custom data type	Memory-bound
	Dataflow using HLS stream	Compute-bound
	Dataflow using array of HLS stream	Compute-bound
	Loop dependency inter	Compute-bound
	Global memory two banks	Memory-bound
	Stream chain matrix multiplication	Compute-bound
	Local memory two parallel read/write	Memory-bound
	Loop pipelining	Compute-bound
	Array block and cyclic partitioning	Balanced
	Port width widening	Memory-bound
	PLRAM memory access	Memory-bound
	Vector addition	Memory-bound
	Shift register	Compute-bound
	Systolic array	Compute-bound
	Wide memory read/write	Memory-bound

Table 6.2: Workload categorization.

7 Results

7.1 Outline

The objective of this work is to primarily answer two research questions:

1. How does the kernel clock frequency of the bitstream affect the runtime and I/O throughput of the kernel?
2. Does the I/O throughput increase when we use frequency to schedule?

To answer the first question, we perform empirical experiments where we synthesize bitstreams from two benchmarks Rosetta, and Vitis Accel Examples at maximum frequency and measure the kernel execution time and I/O throughput. Using the answer to the first question, we evaluate the scheduling algorithm 1 by testing it against different scenarios of application execution to answer the second question. We conclude that optimizing clock frequency for each FPGA is not easy, particularly for non-FPGA experts, and unexpected performance gaps between different FPGAs, which are more significant than the theoretical maximum freqs, can be caused.

The I/O throughput for each kernel is defined as:

$$IOThrp = (\sum_{i} sizeof(KernelInputBuffer_i))/KernelExecutionTime \tag{7.1}$$

7.2 Analysis

7.2.1 Emperical Analysis of Application Executions

Runtime performance of Rosetta workloads

Results obtained by running each of the four Rosetta workloads on Xilinx Alveo U50 and Xilinx Alveo U280 are shown in Figure 7.1. The figure shows the kernel execution time in milliseconds on the y-axis on a log scale. Each application is synthesized at the maximum kernel clock frequency. The frequency at which the workload is run is shown on the x-axis and the exact observed runtimes are shown on each of the bars.

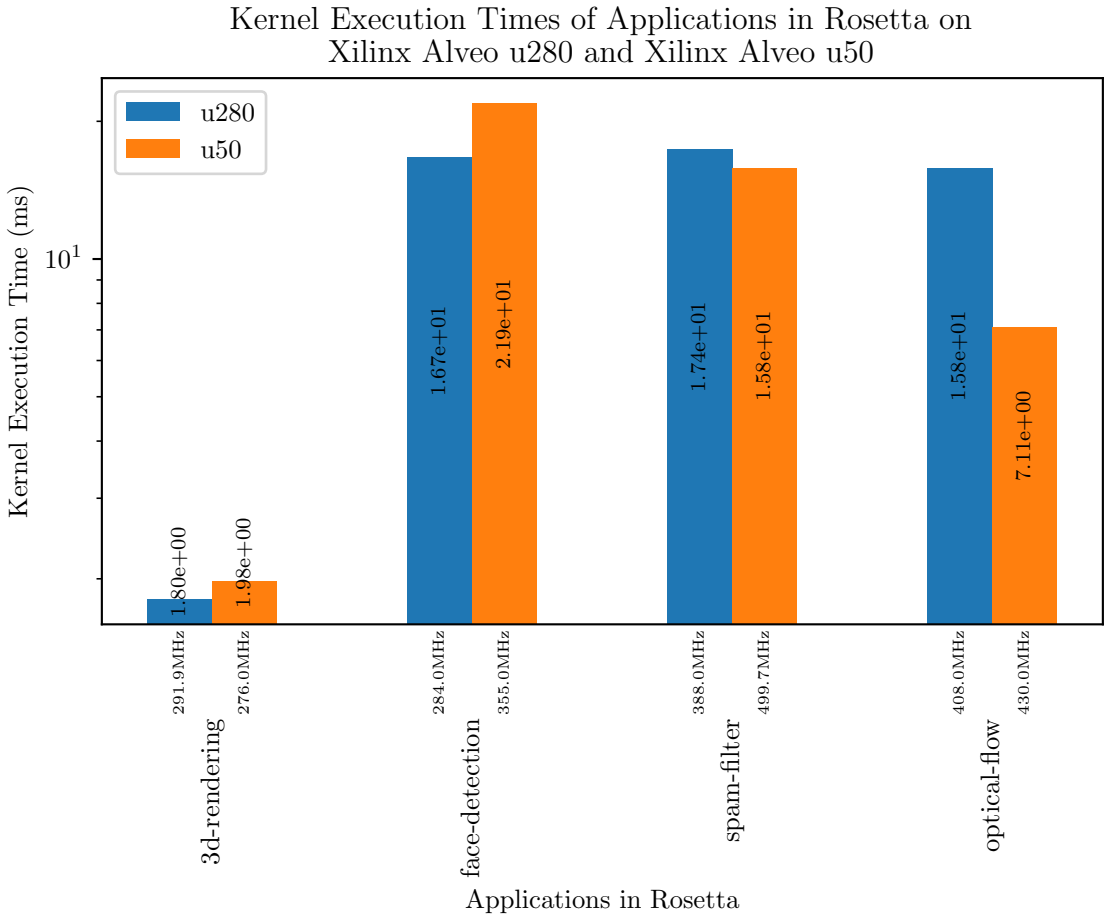
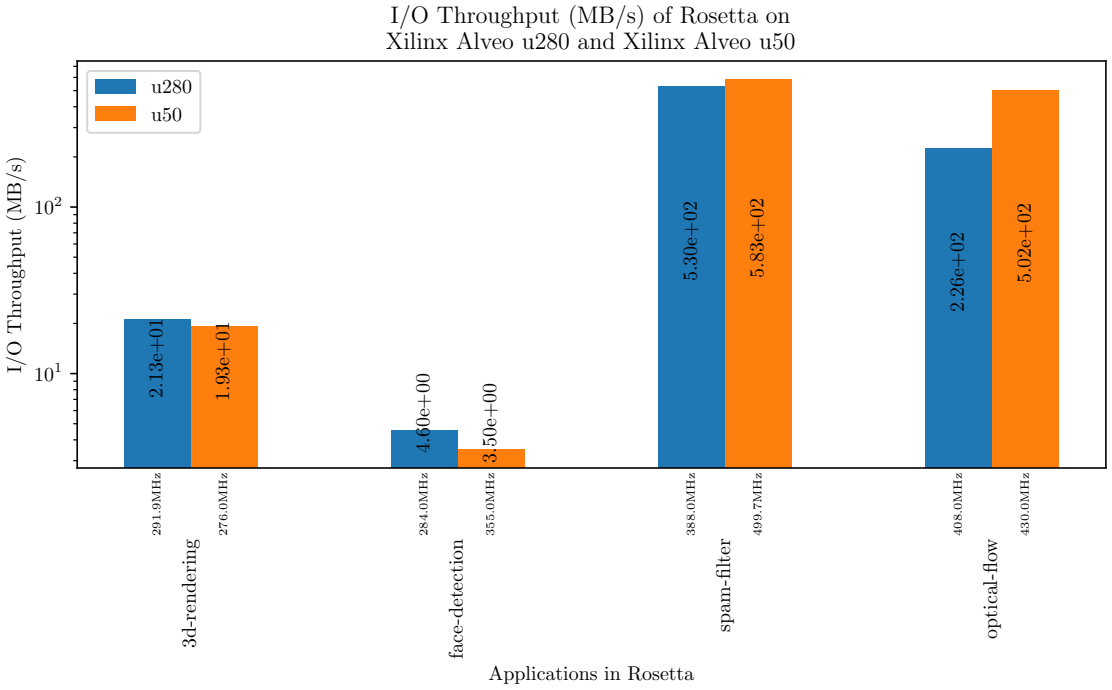


Figure 7.1: Kernel Execution Time of Applications in *Rosetta*

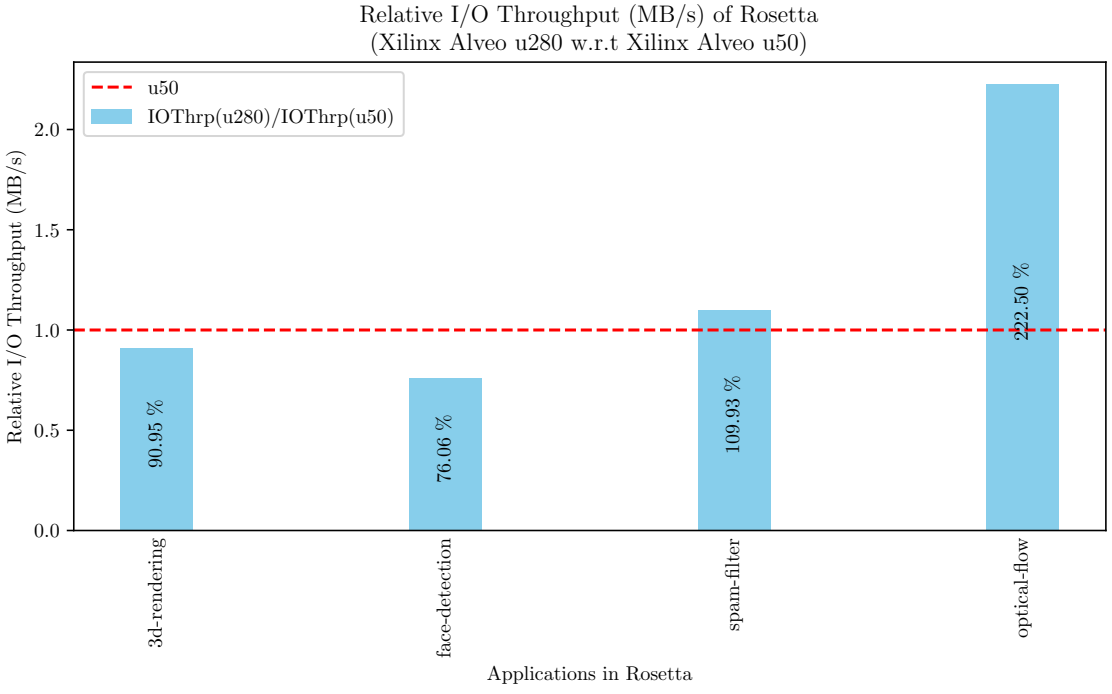
Apart from the *face-detection* application, the observed kernel execution times are lower on the FPGA that runs at the higher kernel clock frequency. For example, 3D rendering application runs 1.1× faster on U280 at 291.9MHz compared to U50 at 276MHz (1.06× faster clock frequency). Similarly, Spam filter and Optical flow applications run 1.1× and 2.2× faster on U50 compared to U280 with 1.28× and 1.05× higher clock frequency.

I/O throughput of Rosetta workloads

We next analyze the effect on I/O throughput. Since workloads consume input buffers to perform the work, the I/O throughput will observe an improvement if the kernel time is reduced. Figure 7.2a shows the I/O throughput (in MB/s) of each application



(a) I/O Throughput.



(b) Relative I/O Throughput.

Figure 7.2: I/O Throughput of Applications in *Rosetta*.

in *Rosetta* benchmark suite for *Xilinx Alveo U50* and *Xilinx Alveo U280*. The graph is formatted in the same way as the previous figure.

Apart from *face-detection*, the applications show that, across different FPGAs, as the kernel clock frequency increases the I/O throughput is higher. Since the I/O throughput is derived based on the kernel execution time, we note similar trends as the previous result.

The graph in Figure 7.2b shows the relative I/O throughput of each application in *Rosetta* on *Xilinx Alveo U280* w.r.t *Xilinx Alveo U50*.

Runtime performance of Vitis Accel Examples workloads

Results obtained by running each of the eighteen Vitis Accel Examples workloads on *Xilinx Alveo U50* and *Xilinx Alveo U280* are shown in Figure 7.3. The figure shows the kernel execution time in milliseconds on the y-axis on a log scale. Each application is synthesized at the maximum kernel clock frequency. The frequency at which the workload is run is shown on the x-axis and the exact observed runtimes are shown on each of the bars.

Apart from the *critical-path* application, the observed kernel execution times are lower on the FPGA that runs at the higher kernel clock frequency. For example, *array_partition* application runs 1.2× faster on U280 at 452.0MHz compared to U50 at 375.9MHz (1.20× faster clock frequency). Similarly, *loop_pipeline* and *kernel_chain* applications run 1.16× and 2.2× faster on U50 compared to U280 with 1.18× and 1.35× higher clock frequency.

I/O throughput of Vitis Accel Examples workloads

We next analyze the effect on I/O throughput. Since workloads consume input buffers to perform the work, the I/O throughput will observe an improvement if the kernel time is reduced. Figure 7.4a shows the I/O throughput (in MB/s) of each application in *Rosetta* benchmark suite for *Xilinx Alveo U50* and *Xilinx Alveo U280*. The graph is formatted in the same way as the previous figure.

Apart from *critical_path*, the applications show that, across different FPGAs, as the kernel clock frequency increases the I/O throughput is higher. Since the I/O throughput is derived based on the kernel execution time, we note similar trends as the previous result.

The graph in Figure 7.4b shows the relative I/O throughput of each application in *Rosetta* on *Xilinx Alveo U280* w.r.t *Xilinx Alveo U50*.

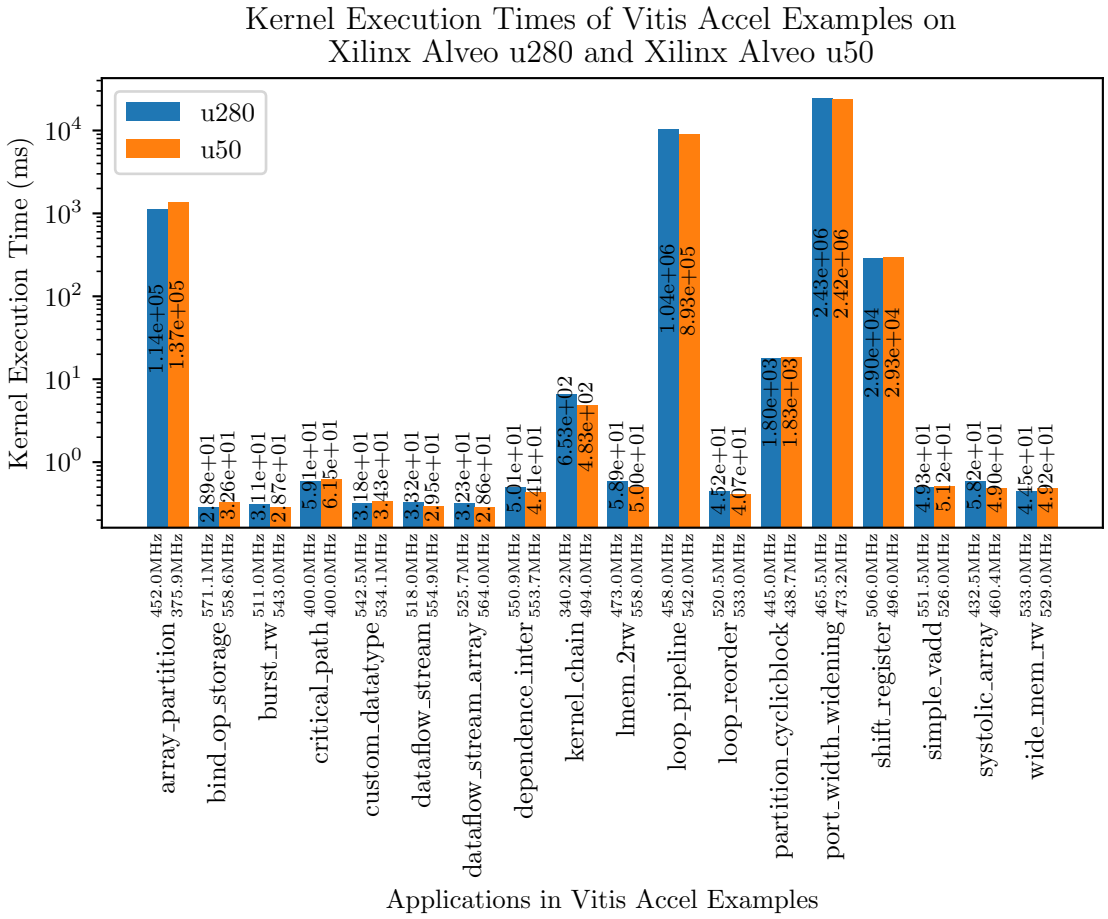
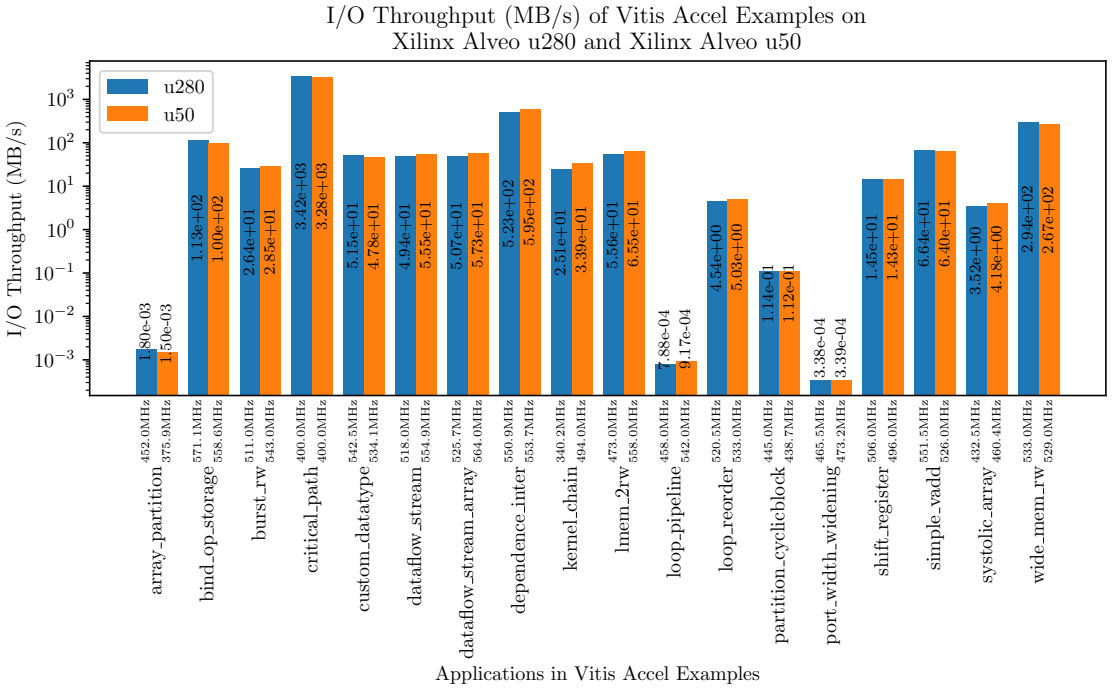


Figure 7.3: Kernel Execution Time of Applications in *Vitis Accel Examples*

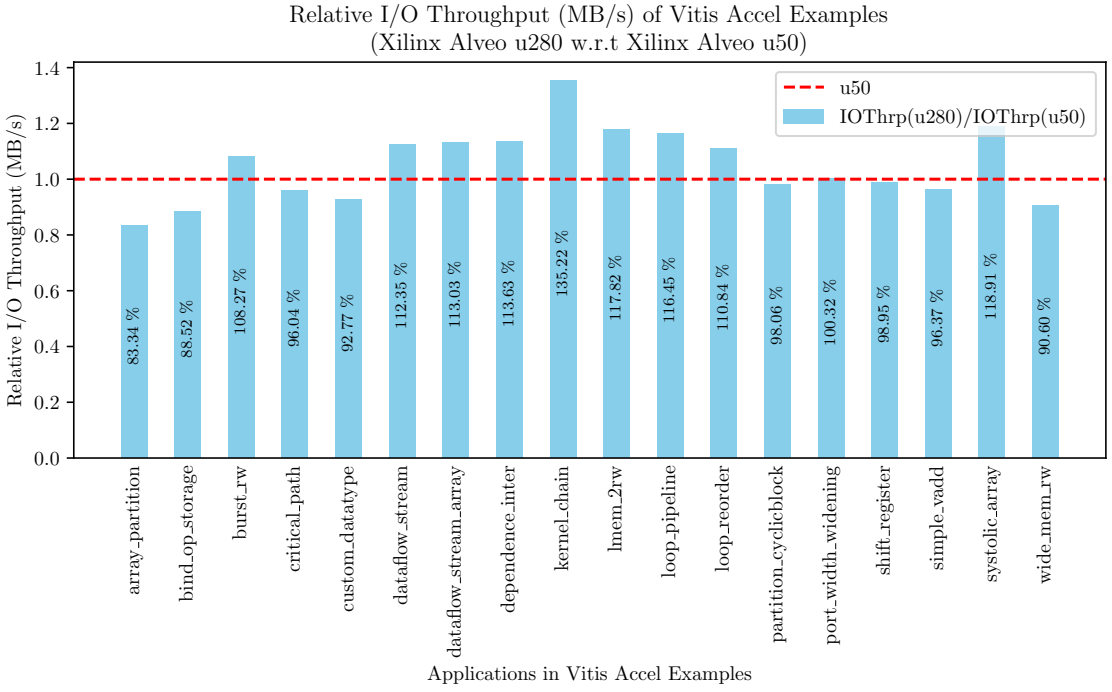
7.2.2 Analysis of the Scheduler

W evaluate two different policies, namely, *FCFS_FA* and *FCFS_RO* by executing different queues of applications of sizes $K \in \{5, 10, 15, 20\}$. These applications are a mixture of Vitis Accel Examples and Rosetta. Table 7.1 summarizes the differences between the two policies.

Results obtained by executing a mixture of the applications are shown in Figure 7.5. For both figures, applications is synthesized at the maximum kernel clock frequency. The number of applications in the queue K (or) the size of the scheduler queue is shown on the x-axis. Figure 7.5a shows the total kernel execution time in milliseconds on the y-axis and figure 7.5b shows the total kernel I/O throughput in MB/seconds on the y-axis.



(a) I/O Throughput.



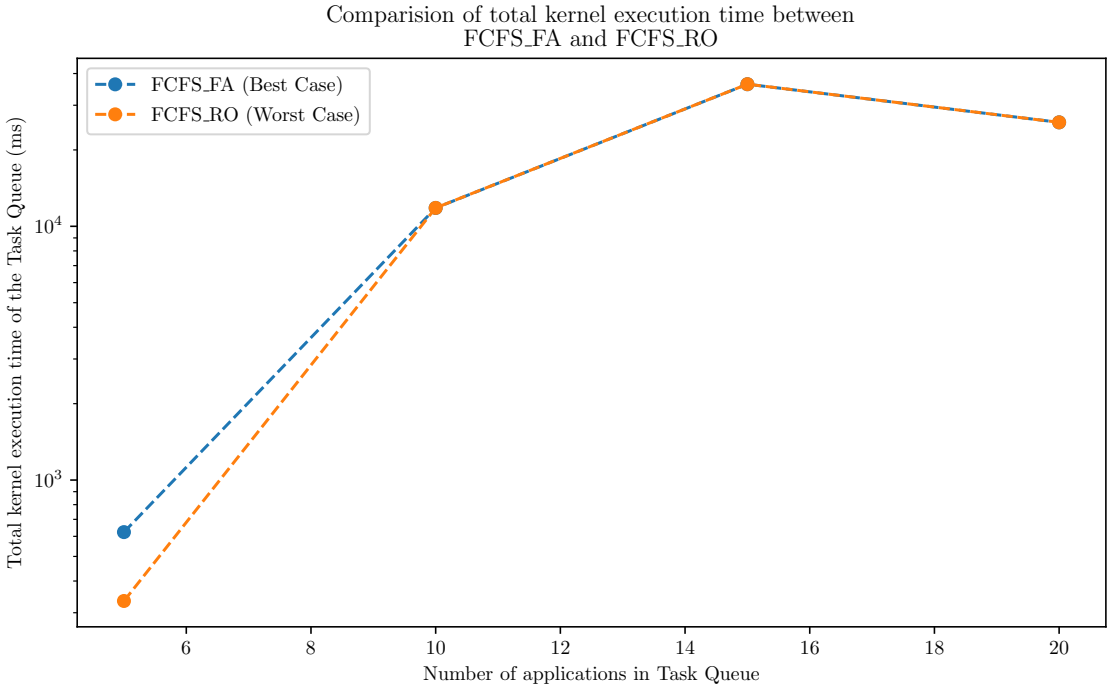
(b) Relative I/O Throughput.

Figure 7.4: I/O Throughput of Applications in *Vitis Accel Examples*.

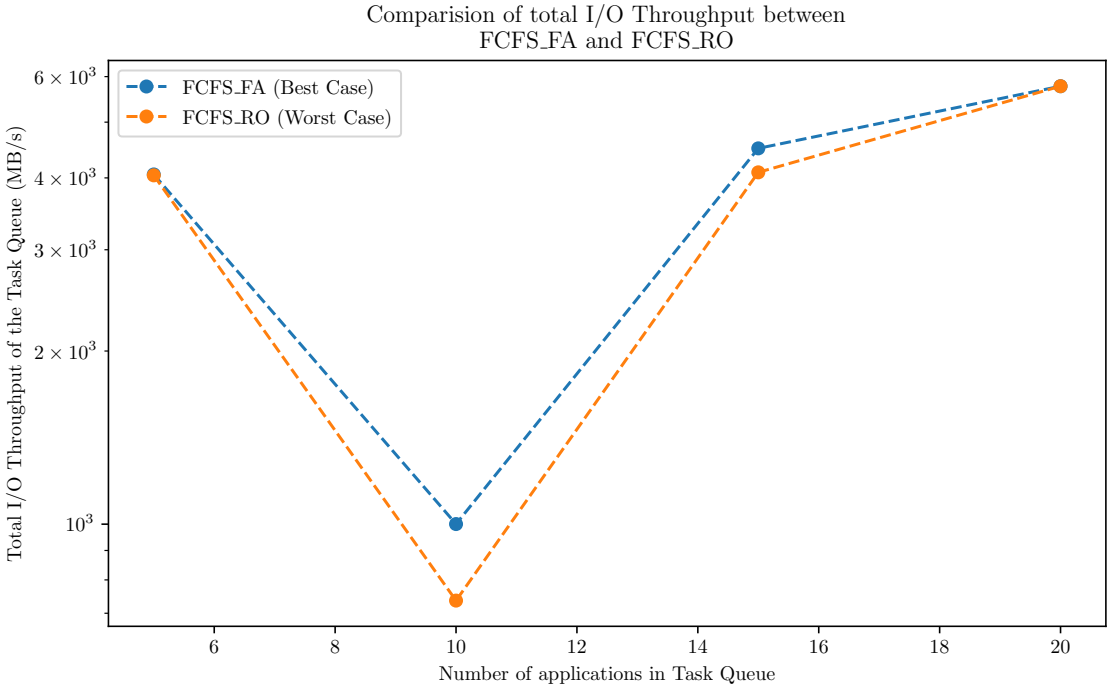
Policy	Expansion	Description
FCFS_FA	First Come First Server - Frequency Aware	Iterate over the bitstreams ranked by their kernel clock frequency value and select one
FCFS_RO	First Come First Server - Random Order	Iterate over the bitstreams and select one. The bitstreams are not ranked (no *sorted set* entry).

Table 7.1: Policies supported by the scheduler.

In each figure, the blue line corresponds to the policy that takes kernel clock frequency into consideration and the orange line corresponds to the policy that does not take kernel clock frequency into account while scheduling. We see that the I/O throughput increases is more for $K \in \{5, 10, 15\}$ and decreases slightly when $K = 20$.



(a) Kernel execution time comparison across policies.



(b) Kernel I/O throughput comparison across policies.

Figure 7.5: Comparision of *FCFS_FA* and *FCFS_RO*.

8 Related Work

8.1 FPGA workloads in Cloud Environments

With Dennard's scaling encountering limitations [Den+74], energy efficiency has emerged as a critical consideration in datacenters. As a result, there has been a notable exploration of alternative solutions, including the integration of accelerators like FPGAs, to mitigate power consumption.

One notable instance of this trend can be witnessed in Microsoft's data center, where they have embraced CPU-FPGA systems as a strategic approach to accelerate the performance of their renowned Bing search engine [Put+15]. By integrating FPGAs alongside traditional CPUs, Microsoft has harnessed the reconfigurable nature of FPGAs to accelerate specific search engine tasks efficiently. This combination of hardware allows the search engine to handle complex queries and process vast amounts of data with improved speed and energy efficiency.

Another prominent exemplar of this shift towards FPGA utilization is Amazon's introduction of the F1 instance [Ama23] in its commercial Elastic Compute Cloud (EC2). This innovative compute option is specially equipped with FPGA boards to bolster the computational capabilities of cloud-based applications. By offering FPGAs as part of their EC2 service, Amazon caters to customers seeking to enhance performance and optimize power consumption for specific workloads. The F1 instance enables users to leverage FPGA-accelerated functions, making it attractive for computationally intensive tasks such as machine learning, data analytics, and genomics research. The growing adoption of CPU-FPGA systems and FPGA-based compute instances underscore the industry's recognition of FPGA technology as a valuable tool for boosting performance while addressing energy efficiency concerns.

8.1.1 Programming FPGAs

Despite the growing adoption of FPGAs, they remain hard to program due to a steep learning curve for software programmers. APIs such as OpenCL [Gro21] that offer a low-level API for heterogeneous computing have evolved to support FPGAs. Our work targets OpenCL. OpenCL is widely adopted by major FPGA vendors [Xil21; FPG21b]. The Intel AOC compiler [Int23b] is used to compile OpenCL kernel programs into a

bitstream for FPGAs. This bitstream implements both lower-level utility (“shell”) logic and user kernel logic. A host program manages the creation and movement of buffers and tasks for execution.

AOC incorporates constant, private, and local memory, which can be implemented either in registers or embedded memory blocks (BRAM). On the other hand, global memory is implemented in external memory (DDR4). When accessing data stored in BRAM or external memory, Load-Store Unit (LSU) need to be generated. These LSUs are implemented in logic elements, with the possibility of utilizing BRAMs if caches are inferred.

The compiler infers various types of LSUs, which depend on the type of access and the array sizes in memory. These include coalesced, burst-coalesced, prefetching, and streaming pipelined LSUs [Int23a]. Each type exhibits different levels of performance and resource usage, with coalesced and burst-coalesced LSUs standing out as the most efficient options. These efficient LSUs are inferred when arrays are accessed in an aligned and consecutive manner.

Even with the help of HLS, accelerator designers still have to manually perform code reconstruction and cumbersome parameter tuning to achieve optimal performance. While many learning models have been leveraged by existing work to automate the design of efficient accelerators, the unpredictability of modern HLS tools becomes a major obstacle for them to maintain high accuracy. A recent work proposed AutoDSE, an automated design space exploration (DSE) framework for FPGA accelerators that leverages a bottleneck-guided coordinate optimizer to systematically find better design points [Soh+22]. It addresses the unpredictability of modern HLS tools by using a combination of manual and automatic pragma insertion techniques to explore solution space efficiently and converge on high-quality-of-results (QoR) designs. These advances are required to increase the FPGAs’ adoption rate.

8.2 Performance Difference in Bitstreams

It is reported that applications that have the potential of parallelism can benefit more from logical resources on the high-end FPGAs. However, applications that have more multistage computing kernels such as machine learning and cryptography benchmarks require a high frequency FPGA to increase the throughput. It is also observed that date-intensive kernels in applications can easily saturate the memory bandwidth. Hence, it is reported that to use medium or high-end FPGAs to prevent it becomes the bottleneck of the system for such applications. [Mak+19].

8.3 Scheduling in FPGAs

Andrés Rodríguez et al. [Rod+22] proposed a lightweight scheduling strategy named FastFi with a focus on FPGA accelerators with a new scheduler named MultiFastFit. This asynchronously handles heterogeneous systems consisting of different CPU cores and FPGAs. It is claimed that scheduling strategy along with the proposed scheduler significantly reduces the overhead to automatically compute the near-optimal chunk-sizes when compared to a previous state-of-the-art auto-tuned approach. The claimed approaches have been benchmarked by tuning for the low-power UltraScale+ platform. The results show the FastFit strategy always finds the near-optimal FPGA chunk size for any device configuration at a reasonable cost, even for fine-grained and irregular applications. Also it is shown that the heterogeneous CPU+FPGA co-executions are usually faster and energy efficient than the CPU-only and FPGA-only executions. It is claimed that MultiFastFit outperforms other auto-tuned approach up to 2x and is similar results to manually tuned schedulers without requiring an offline search of the ideal CPU-FPGA partition or FPGA chunk granularity.

Lin-Analyzer is a high-level performance estimation tool that rapidly predicts the performance of FPGA-based accelerators [Zho+16]. The tool relies on the Dynamic Data Depdence Graph (DDDG) to avoid false data dependences created by static analysis techniques used in most existing techniques, including commercial HLS tools. Lin-Analyzer offers different trade-offs via pragmas, which can help designers better understand the performance impact of different accelerator design choices. The tool can identify bottlenecks of different FPGA implementations when applying diverse optimizations and assist designers in evaluating different architectural options in the context of high-level synthesis. Results of experiments to evaluate the effectiveness of Lin-Analyzer show that Lin-Analyzer returns optimal recommendations while navigating complex design spaces within seconds to minutes. The experiments also demonstrate that Lin-Analyzer can accurately predict the performance of FPGA-based accelerators with an average error rate of 5%. Additionally, Lin-Analyzer can identify design bottlenecks and assist HLS developers in identifying potential limitations of the HLS tool. Lin-Analyzer uses a DDDG to capture dependencies between operations at runtime. The DDDG is then used to estimate the execution time for each operation and generate an overall performance estimate for the accelerator design. To explore different optimization configurations, Lin-Analyzer uses a set of pragmas that control loop unrolling, pipelining, and array partitioning. The proposed solution is shown to be effective in navigating complex design spaces and identifying bottlenecks while providing accurate estimates within seconds to minutes.

A prior paper proposed an online task scheduling solution for FPGA-based partially reconfigurable systems [Lu+09]. The proposed solution uses a modified First-Fit

algorithm called mFS, which considers both task size and dependencies when selecting resources for allocation. The paper also introduces a "reuse and partial reuse" approach that can shorten the ACT and STRT times for new tasks. The "best fit" scheduling heuristic is used to further optimize allocation selection by considering both task size and dependencies. This approach helps ensure that tasks are placed in the most appropriate locations, which can lead to shorter completion times for the overall application running on these systems. The mFS algorithm works by first identifying all available MFRs that can be used for placing a new task. It then selects the MFR that is the best fit for the task based on its size and resource requirements. To determine the best fit MFR, the heuristic considers both the size of the MFR and its proximity to other allocated MFRs. The "reuse and partial reuse" approach involves reusing previously allocated resources whenever possible, rather than allocating new resources for each new task. This can help reduce ACT and STRT times by minimizing the amount of time required to configure new resources. Overall, the solutions in the paper aim to improve task scheduling efficiency in FPGA-based partially reconfigurable systems by considering the factors such as available space, resource distribution, time constraints, and task dependencies. While an interesting solution, our work targets offline scheduling and can be complementary to each other.

9 Summary and Conclusion

9.1 Impact of kernel clock frequency

Through empirical experiments, we have successfully established that the kernel clock frequency has a direct impact on kernel execution time and kernel I/O throughput across devices. We have also demonstrated that scheduling policies taking this into consideration perform better than those that don't.

9.2 Bitstream Analyzer and Database

We implemented two artifacts, namely, A bitstream analyzer, to analyze bitstreams from two FPGA vendors, and, a bitstream database for managing multiple bitstreams efficiently. These artifacts are useful in handling multiple bitstreams and help the scheduler operate in heterogeneous environments.

9.3 Scheduling Policies

We evaluated two policies based on the proposed algorithm 1 and have seen mixed results when taking kernel clock frequency into consideration. The total I/O throughput increases and the kernel execution time also increases.

To conclude, the results and artifacts provide a baseline framework to handle heterogeneous deployment environments. Further developments to the baseline framework laid out in the chapter 10 will make it production ready and robust.

10 Future Work

In this section, future directions of development, and improvements to the work are presented. Improvements are geared towards making the system more efficient and developments are geared towards integrating work that has not been dealt with.

10.1 Virtualization

The current framework presented primarily deals with scheduling applications. We have not dealt with the challenges of virtualization that are typically found in cloud environments. A potential enhancement would be to use Unikernels, such as IncludeOS [Bra+15] to achieve process-level FPGA virtualization.

10.2 Additional Parameters to Rank Bitstreams

The existing scheduling policies can take into other factors such as the memory bandwidth, which depends on the specs of the memory device on FPGA. Such kind of parameters are an important factor for memory-bound workloads. Even if kernels can be operated with high clock frequency, the actual throughput can be saturated due to the limited peak memory bandwidth.

Abbreviations

FPGA Field Programmable Gate Array

CPU Central Processing Unit

ASIC Application-Specific Integrated Circuits

DSP Domain Specific Processor

SoC System-on-Chip

DDDG Dynamic Data Depdence Graph

RTL Register Transfer Level

HDL Hardware Description Language

VHDL Very High-Speed Integrated Circuit Hardware Description Language

HLS High-Level Synthesis

LUTs LookUp Tables

BRAM Block Random Access Memory

DRAM Dynamic Random Access Memory

DDR Double Data Rate

HBM High Bandwidth Memory

DMA Direct Memory Access

LSU Load-Store Unit

PCIe Peripheral Component Interconnect Express

API Application Programming Interface

OpenCL Open Computing Language

ELF Executable and Linkable Format

ILP Instruction-Level Parallelism

TLP Task-Level Parallelism

SGD Stochastic Gradient Descent

LR Logistic Regression

AXLF xclbin container format

I/O Input Output

List of Figures

List of Tables

Bibliography

[Ama23] Amazon. *Amazon EC2 F1 Instance*. https://aws.amazon.com/ec2/instance-types/f1. Online. 2023.

[AMD23a] AMD. *Alveo U280 Data Center Accelerator Card Data Sheet (DS963)*. https://docs.xilinx.com/r/en-US/ds963-u280/Summary. Online. 2023.

[AMD23b] AMD. *Alveo U50 Data Center Accelerator Card Data Sheet: Alveo U50 Card Data Sheet (DS965)*. https://docs.xilinx.com/r/en-US/ds965-u50/Summary. Online. 2023.

[Aue+10] J. Auerbach, D. F. Bacon, P. Cheng, and R. Rabbah. "Lime: A Java-Compatible and Synthesizable Language for Heterogeneous Architectures." In: *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '10. Reno/Tahoe, Nevada, USA: Association for Computing Machinery, 2010, pp. 89–108. ISBN: 9781450302036. DOI: 10.1145/1869459.1869469.

[Bac+12] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović. "Chisel: Constructing hardware in a Scala embedded language." In: *DAC Design Automation Conference 2012*. 2012, pp. 1212–1221. DOI: 10.1145/2228360.2228584.

[Bha21] J. Bhasker. *A VHDL primer*. Prentice-Hall. 2021.

[Bos+18] J. Bosch, X. Tan, A. Filgueras, M. Vidal, M. Mateu, D. Jiménez-González, C. Álvarez, X. Martorell, E. Ayguade, and J. Labarta. "Application Acceleration on FPGAs with OmpSs@FPGA." In: *2018 International Conference on Field-Programmable Technology (FPT)*. 2018, pp. 70–77. DOI: 10.1109/FPT.2018.00021.

[Bra+15] A. Bratterud, A.-A. Walla, H. Haugerud, P. E. Engelstad, and K. Begnum. "IncludeOS: A Minimal, Resource Efficient Unikernel for Cloud Services." In: *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*. 2015, pp. 250–257. DOI: 10.1109/CloudCom.2015.89.

[CDL13] E. S. Chung, J. D. Davis, and J. Lee. "LINQits: Big Data on Little Clients." In: *Proceedings of the 40th Annual International Symposium on Computer Architecture*. ISCA '13. Tel-Aviv, Israel: Association for Computing Machinery, 2013, pp. 261–272. ISBN: 9781450320795. DOI: 10.1145/2485922.2485945.

[Den+74] R. Dennard, F. Gaensslen, H.-N. Yu, V. Rideout, E. Bassous, and A. LeBlanc. "Design of ion-implanted MOSFET's with very small physical dimensions." In: *IEEE Journal of Solid-State Circuits* 9.5 (1974), pp. 256–268. DOI: 10.1109/JSSC.1974.1050511.

[FPG21a] I. FPGA. *Accelerator Functional Unit Developer's Guide for Intel FPGA Programmable Acceleration Card*. https://www.intel.com/content/www/us/en/programmable/documentation/bfr1522087299048.html. 2021.

[FPG21b] I. FPGA. *Intel FPGA SDK for OpenCL Pro Edition: Programming Guide*. https://www.intel.com/content/www/us/en/programmable/documentation/mwh1391807965224.html. 2021.

[Gro21] K. Group. *The OpenCL Specification*. https://www.khronos.org/registry/OpenCL/specs/3.0-unified/html/OpenCL_API.html. 2021.

[Int23a] Intel. *Intel FPGA SDK for OpenCL Pro Edition: Best Practices Guide*. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/. Online. 2023.

[Int23b] Intel. *Intel FPGA SDK for OpenCL Pro Edition: Programming Guide*. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/. Online. 2023.

[Int23c] Intel. *Intel oneAPI*. https://www.oneapi.com/spec/. Online. 2023.

[Koe+16] D. Koeplinger, R. Prabhakar, Y. Zhang, C. Delimitrou, C. Kozyrakis, and K. Olukotun. "Automatic Generation of Efficient Accelerators for Reconfigurable Hardware." In: *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 2016, pp. 115–127. DOI: 10.1109/ISCA.2016.20.

[Koe+18] D. Koeplinger, M. Feldman, R. Prabhakar, Y. Zhang, S. Hadjis, R. Fiszel, T. Zhao, L. Nardi, A. Pedram, C. Kozyrakis, and K. Olukotun. "Spatial: A Language and Compiler for Application Accelerators." In: *SIGPLAN Not.* 53.4 (June 2018), pp. 296–311. ISSN: 0362-1340. DOI: 10.1145/3296979.3192379.

[Leb+05] J. Lebak, J. Kepner, H. Hoffmann, and E. Rutledge. "Parallel VSIPL++: An Open Standard Software Library for High-Performance Parallel Signal Processing." In: *Proceedings of the IEEE* 93.2 (2005), pp. 313–330. DOI: 10.1109/JPROC.2004.840303.

[Leb+12] I. Lebedev, C. Fletcher, S. Cheng, J. Martin, A. Doupnik, D. Burke, M. Lin, and J. Wawrzynek. "Exploring Many-Core Design Templates for FPGAs and ASICs." In: *Int. J. Reconfig. Comput.* 2012 (Jan. 2012). ISSN: 1687-7195. DOI: 10.1155/2012/439141.

[Lu+09] Y. Lu, T. Marconi, K. Bertels, and G. Gaydadjiev. "Online Task Scheduling for the FPGA-Based Partially Reconfigurable Systems." In: *Reconfigurable Computing: Architectures, Tools and Applications*. Ed. by J. Becker, R. Woods, P. Athanas, and F. Morgan. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 216–230. ISBN: 978-3-642-00641-8.

[Mak+19] H. M. Makrani, H. Sayadi, T. Mohsenin, S. rafatirad, A. Sasan, and H. Homayoun. "XPPE: Cross-Platform Performance Estimation of Hardware Accelerators Using Machine Learning." In: *Proceedings of the 24th Asia and South Pacific Design Automation Conference*. ASPDAC '19. Tokyo, Japan: Association for Computing Machinery, 2019, pp. 727–732. ISBN: 9781450360074. DOI: 10.1145/3287624.3288756.

[NBB20] R. Nozal, J. L. Bosque, and R. Beivide. "EngineCL: Usability and Performance in Heterogeneous Computing." In: *Future Generation Computer Systems* 107 (June 2020), pp. 522–537. DOI: 10.1016/j.future.2020.02.016.

[Ope21] OpenACC-Standard.org. *The OpenACC Application Programming Interface*. https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC-3.2-final.pdf. Online. 2021.

[Put+15] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. "A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services." In: *IEEE Micro* 35.3 (2015), pp. 10–22. DOI: 10.1109/MM.2015.42.

[Rod+22] A. Rodríguez, A. Navarro, K. Nikov, J. Nunez-Yanez, R. Gran, D. Suárez Gracia, and R. Asenjo. "Lightweight asynchronous scheduling in heterogeneous reconfigurable systems." In: *Journal of Systems Architecture* 124 (2022), p. 102398. ISSN: 1383-7621. DOI: https://doi.org/10.1016/j.sysarc.2022.102398.

[Sas+06] R. Sass, D. Andrews, E. Komp, W. Peck, F. Baijot, E. Anderson, J. Stevens, and J. Agron. "Enabling a Uniform Programming Model Across the Software/Hardware Boundary." In: *2006 14th Annual IEEE Symposium on Field Programmable Custom Computing Machines*. Los Alamitos, CA, USA: IEEE Computer Society, Apr. 2006, pp. 89–98. DOI: 10.1109/FCCM.2006.40.

[Soh+22] A. Sohrabizadeh, C. H. Yu, M. Gao, and J. Cong. "AutoDSE: Enabling Software Programmers to Design Efficient FPGA Accelerators." In: *ACM Trans. Des. Autom. Electron. Syst.* 27.4 (Feb. 2022). ISSN: 1084-4309. DOI: 10.1145/3494534.

[TM21] D. Thomas and P. Moorby. *The Verilog Hardware Description Language*. Springer Science & Business Media. 2021.

[Tro+22] C. R. Trott, D. Lebrun-Grandié, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. Ibanez, N. Liber, J. Madsen, J. Miles, D. Poliakoff, A. Powell, S. Rajamanickam, M. Simberg, D. Sunderland, B. Turcksin, and J. Wilke. "Kokkos 3: Programming Model Extensions for the Exascale Era." In: *IEEE Transactions on Parallel and Distributed Systems* 33.4 (2022), pp. 805–817. DOI: 10.1109/TPDS.2021.3097283.

[Xil21] Xilinx. *XRT and Vitis Platform Overview*. https://xilinx.github.io/XRT/master/html/platforms.html. 2021.

[Zho+16] G. Zhong, A. Prakash, Y. Liang, T. Mitra, and S. Niar. "Lin-Analyzer: A High-Level Performance Analysis Tool for FPGA-Based Accelerators." In: *Proceedings of the 53rd Annual Design Automation Conference*. DAC '16. Austin, Texas: Association for Computing Machinery, 2016. ISBN: 9781450342360. DOI: 10.1145/2897937.2898040.