SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Utilizing dynamic partial reconfiguration in an FPGA-accelerated FaaS architecture

Martin Lambeck

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Utilizing dynamic partial reconfiguration in an FPGA-accelerated FaaS architecture

Nutzung von dynamischer partieller Rekonfiguration in einer FPGA-beschleunigten FaaS-Architektur

Author:	Martin Lambeck
Supervisor:	Prof. Dr.-Ing. Pramod Bhatotia
Advisor:	Dr. Atsushi Koshiba
Submission Date:	15.07.2024

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

Munich, 15.07.2024 Martin Lambeck

Acknowledgments

I want to thank my supervisor Prof. Dr.-Ing. Pramod Bhatotia for giving me the opportunity to work on this exciting research topic. I would like to extend my gratitude to my advisor Dr. Atsushi Koshiba for the insightful discussions we had and the quick response to any questions or hindrances. Lastly, I'd like to acknowledge Charalampos Mainas and everyone else involved in the project for their invaluable contributions.

On a personal note, I want to express my wholehearted appreciation and gratitude for my beloved family and my wonderful girlfriend. Your boundless compassion and kindness has shaped my life in countless ways. In challenging times, you have been a pillar of strength and a source of inspiration. I feel immeasurably blessed to have you all in my life.

Abstract

In the domain of cloud computing, serverless functions have become a popular deployment model. Its simplicity in resource management, cost-effectiveness and fast time-to-market makes it an attractive architecture for software developers. At the same time, computational workloads are growing unabated and demand for computing power in the cloud is expected to keep expanding. To meet this challenge, cloud providers have gradually started to arm compute nodes with hardware accelerators, such as field programmable gate arrays (FPGAs). Their capabilities in extreme parallelism and low-power characteristics render them an excellent option for a range of compute-intensive tasks. However, FPGA support in commercial cloud offerings is in its infancy and non-existent in the context of serverless computing.

In this thesis, we present a system design for a function-as-a-service (FaaS) framework, that supports task offloading to a host-attached FPGA device. Our design subdivides the FPGA into multiple, isolated regions in order to enable multi-tenancy on the FPGA. We employ dynamic partial reconfiguration to swap out accelerated functions on-demand. On the host-side, functions are confined into unikernel environments inside lightweight virtual machines, thus enforcing strong isolation. We present our system design proposal, and a set of benchmark applications, including data compression, image processing and cryptographic hashing, that we have ported to our framework. Finally, we conduct a performance evaluation of these benchmarks and compare key metrics between a traditional CPU-only compute node and our FPGA-accelerated co-processor setup.

Our experimental results suggest that an FPGA-accelerated FaaS platform is a feasible architecture. Our system design aligns well with the serverless computing paradigm in the sense that it liberates the developer from managing the attached FPGA device or its runtime. For some type of workloads, the design offers outstanding performance characteristics. In particular, if the computation allows parallelization and/or pipelining and the concrete implementation is optimized for it, then excellent throughput can be achieved.

Contents

1. Introduction

The popularity of function-as-a-service (FaaS) in cloud environments is ever-growing, largely due to its simplicity from the developer's perspective. At the same time, computational power is in high demand, no less due to the recent hype of artificial intelligence and the complexity of underlying machine learning applications. The deployment of graphic processing units (GPUs) in cloud environments is becoming increasingly prevalent, as many machine learning frameworks rely on them for reasonable performance. FPGAs represent an alternative, that is often overlooked. Due to their architectural design, they excel in highly parallelizable tasks and offer superb energy efficiency. In [20] the authors demonstrate a 34× speed-up when running a money-laundering detection application on FPGA hardware, instead of the CPU. Another study compared four workflows on FPGA versus CPU [84]. They observed speed-ups of 40×, 20×, 2× and 2× in favor of the FPGA. In [91] the authors measured up to 22× speed-up against CPU and up to 6× speed-up against GPU for basic linear algebra algorithms on the FPGA. The authors of [70] concluded that, for specific algorithms in an image processing pipeline, FPGAs are up to 22× and 8.8× times more energy efficient than CPUs and GPUs, respectively.

Although FPGAs offer benefits in certain applications, they are usually not the first choice when considering hardware acceleration. This is partly due to their difficulty to program and integrate into a running system. Furthermore, their absence in commodity hardware further impedes access to new developers. Cloud providers are reluctantly rolling out FPGA options to their catalogs. To this end, we present `serverless-fpga`, a FaaS-framework that extends to the FPGA. Our framework liberates the developer from tasks related to managing the FPGA, integrating it into existing infrastructure, and engineering a runtime for the FPGA. Developing and running accelerated functions becomes as "easy" as writing FPGA-compatible code.

Academic projects in that direction exist; however they often only support OpenCL kernels and lack support for applications written in fine-tuned, low-level code (e.g. hardware description language, HDL), thus throwing potential performance benefits overboard [62]. Other designs fall short on support for spatial or temporal multiplexing. In other words, these designs are unable to run multiple applications in parallel or efficiently reconfigure the FPGA, which is crucial in the serverless context, where it is typical to deal with a large number of invocations of short-lived functions.

In summary, our contributions are:

1. We present an end-to-end system design for a function-as-a-service platform with support for FPGA-accelerated functions. This includes:
   - A Kubernetes [69] setup, deployed with the OpenFaaS framework [56]
   - A unikernel environment for running the host applications, based on urunc [55], includeOS [11] and solo5 [32]. The solo5 monitor is extended by a set of new hypercalls for our purposes.
   - Our vFPGA manager, which acts as an intermediary between the unikernel applications and the FPGA device driver.
   - The Coyote shell [49] as runtime environment on the FPGA
2. We port and develop a set of benchmark applications to our system for the purpose of performance evaluation
3. We measure the performance characteristics of our design and compare them with a CPU-only system

2. Background

2.1. Serverless computing

Cloud computing has seen a tremendous rise in popularity in recent years and is becoming the standard deployment model for a growing number of mainstream software. Cloud services can be defined as a model for "ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers [...]) that can be rapidly provisioned and released with minimal management effort or service provider interaction" [61]. While it simplifies the process of bootstrapping the backend infrastructure, it does not per se relieve the developers from the burden of maintaining a potentially complex network of systems and resources. With serverless computing, the idea has evolved even further, as it rids the developer almost completely of the management of backend infrastructure. Applications are no longer deployed by the developer, instead, it becomes the cloud provider's responsibility to dynamically allocate machines, bootstrap applications, and scale resources up or down, based on demand [85].

Serverless computing is often associated with function-as-a-service (FaaS). In this deployment model, software developers publish their serverless applications in a set of independent packages. Each package contains the code and dependencies necessary to execute a certain function. Pre-defined triggers specify the events, that lead to the execution of the function. A typical trigger is an incoming request on an HTTP endpoint, with the request body serving as the function input and the function output constituting the HTTP response body. Upon invocation of such a function, the cloud provider allocates resources appropriately, deploys the package, and executes the function, all within seconds or fractions of a second and hidden from the developer. The billing plan is pay-per-use: Cost is derived from measured, actual resource consumption, such as the number of invocations, total runtime, allocated memory, and generated network traffic. Commercial cloud services complement the offer by providing frequently used backend services (Backend-as-a-Service, BaaS), such as managed databases or cloud storage [85].

A primary research goal regarding FaaS is the minimization of invocation latencies, i.e. the elapsed time between the event that triggers a function and its response. Two cases are distinguished. The first invocation of a function is referred to as *cold start*.

Here, the cloud provider must recruit compute resources and bootstrap the runtime environment. After the first invocation, the function can stand by for future requests. In this state, subsequent requests can be responded to more rapidly, as the environment is already fully prepared (warm start). In serverless computing, cloud providers virtualize physical machines. Applications from multiple customers can then run simultaneously on the same hardware (multi-tenancy). The systems are designed to enforce strong isolation between applications.

2.2. FPGA

Field Programmable Gate Arrays (FPGAs) are highly customizable integrated circuits, that can readily be reprogrammed to resemble any desired circuit and efficiently run computations based on that circuit. FPGAs are composed of an array of configurable logic blocks, memory elements, and interconnections between them. Among other use cases, FPGAs can be employed by data centers to accelerate compute-intensive tasks, such as image and video processing, deep learning, data compression, and other tasks that benefit from massive parallelization. Their low-latency characteristics are leveraged in high-frequency trading or programmable network interfaces. Not only do FPGAs provide significantly better runtime performance on a multitude of applications, they also exhibit superior energy efficiency, when compared to general-purpose CPUs [70] [9] [71].

Our work targets FPGAs deployed in a co-processor architecture. This deployment model describes a configuration where the FPGA accelerator is connected to, and managed by, the CPU in the host system. While the CPU can be used for general-purpose processing and administrative tasks, computation-intensive workflows are offloaded to the FPGA in an effort to achieve a substantial performance boost.

2.3. Dynamic, partial reconfiguration

By their nature, FPGAs have to be configured in order to execute the desired functionality. Traditionally, this process affected the entire chip and was followed by a device reset. Modern FPGAs have the ability to reconfigure selective subareas of the chip at runtime, without affecting other parts of the device [82]. This technology is known as *dynamic partial reconfiguration* (DPR). Sometimes it is simply referred to as *partial reconfiguration* (PR), but strictly speaking the term *dynamic* pertains to the fact that the reconfiguration does not halt or otherwise perturb the logic running outside the region being reconfigured [54]. The predefined chip area is referred to as *dynamic region*, and it is possible to have multiple and even nested dynamic regions in a design [2].

The available chip area has always been a limitation of FPGAs, and in the context of serverless functions, we can expect to see more FPGA-accelerated functions than would fit the device. DPR enables the execution of numerous functions one after another, by swapping them out as necessary (temporal multiplexing). As performance is a primary objective of our proposed system, it is essential to factor in the time overhead, that is incurred by the partial reconfiguration. In Coyote [49] the authors measured the time to reconfigure regions of varying sizes and observed a linear correlation between elapsed time and area of the dynamic region.

Floorplanning is the process of delineation of the positions and dimensions of the dynamic regions (see fig. 2.1). If regions are sized too small, complex user logic may not fit within the bounds of a region, rendering the design inadequate. By choosing larger dimensions, the number of dynamic regions that can be accommodated by the FPGA is reduced, thereby limiting the number of functions that can operate in parallel.

2.4. Coyote shell

The Coyote shell [49] constitutes the foundation for our system design, functioning as the runtime environment on the FPGA side. It divides the FPGA fabric into a static region and one or multiple dynamic regions. The *shell* is the logic that resides in the static region and constitutes the runtime. The Coyote shell carries out the functionalities that would be considered the responsibilities of a traditional operating system (OS), such as memory management, establishing I/O communication channels, and shared resource arbitration.

User logic can be regarded as the analog of an executable binary, although it is technically an integrated circuit design that can be programmed on the FPGA chip. In our case, the user logic behaves in a manner analogous to a stateless service, as it may be required to process multiple, independent requests over the course of its lifetime. User logic is programmed into one of the available dynamic regions and continues to serve incoming requests until it is swapped out for another user logic. This is coordinated by the shell through the use of a technology known as *dynamic partial reconfiguration* (see chapter 2.3). Coyote's design ensures strong isolation between multiple user logic modules running at the same time, thus paving the way for multi-tenancy on the FPGA. Each dynamic region can be considered a virtual FPGA (vFPGA). During the build process, one bitstream file is generated for each combination of user logic and dynamic region (partial bitstream), along with the single bitstream for the static region (full bitstream). In general, partial bitstreams are not interchangeable between builds; they are only compatible with the full bitstream from the same build.

Coyote adheres to the co-processor architecture. A minimal part of the design is

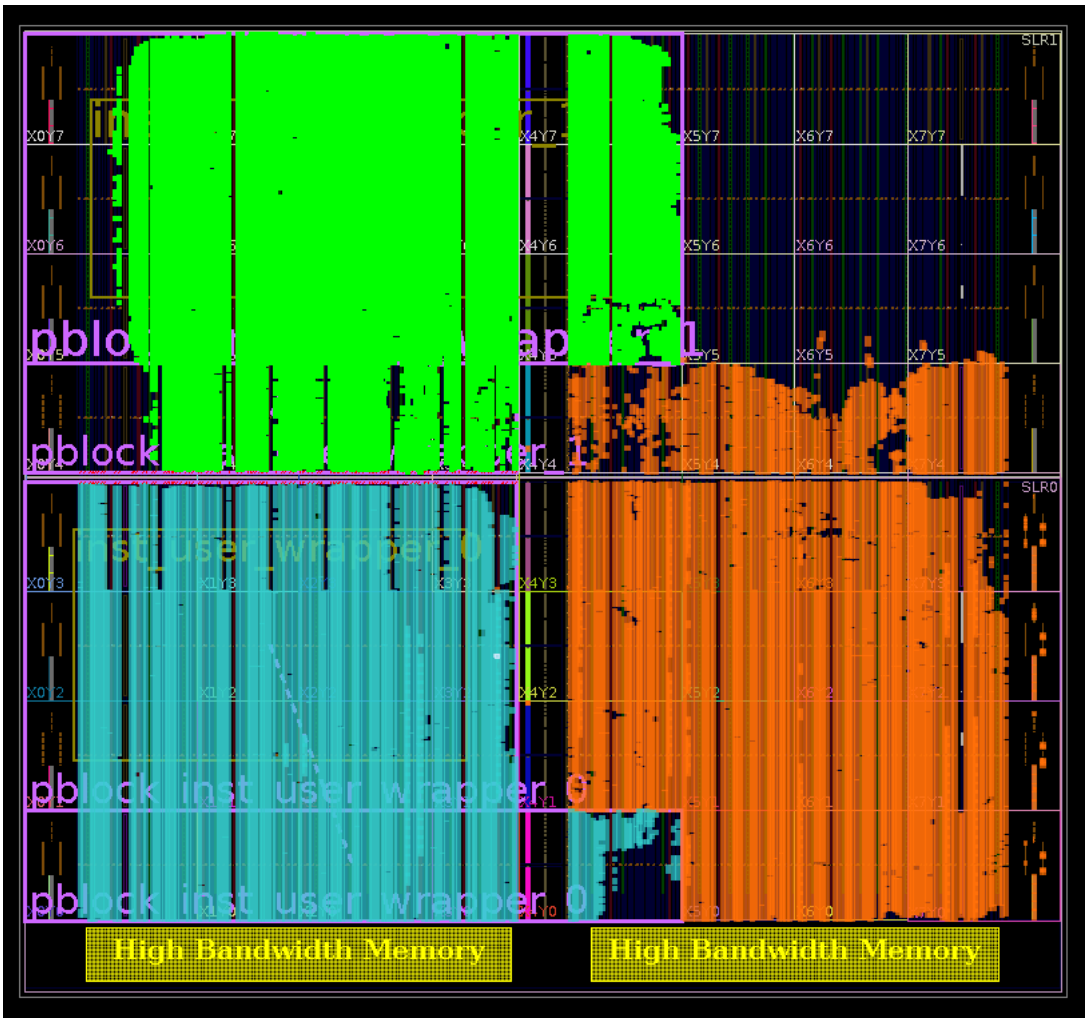


Figure 2.1.: Floorplanning layout of the Coyote shell with two dynamic regions on Xilinx Alveo U50. The static part of Coyote is highlighted in orange. The green and blue areas depict two user logic modules currently loaded, which are confined to the dynamic regions. The locations of the dynamic regions are manually configured by placing `pblocks` in the layout (purple outline).

running on the host CPU as a device driver in the Linux kernel. Communication with the shell is established via Peripheral Component Interconnect Express (PCI Express, PCIe). User logic on the FPGA has access to the host memory via direct memory access (DMA), managed by the Coyote shell.

2.5. Coyote execution flow

Coyote supports two modes of I/O between host memory and FPGA: Streaming data from and to the host memory directly, as well as transferring data between host memory and the FPGA's on-board memory. Coyote also has support for network setups, in which case data can be transferred over the FPGA card's built-in network interface via RDMA (remote direct memory access). The Coyote shell has the capacity to access either memory location and will establish a data channel with the user logic accordingly; this is concealed from the perspective of the user logic. Each user logic exposes two AXI4-Stream interfaces, one for accepting input data, and one for submitting output data. By default, the bandwidth of each port is 512 bits (64 bytes). In addition, there is an AXI4-Lite interface, which, in contrast to the streaming interfaces, operates bidirectionally and supports random access. This port is intended for Control and Status Register (CSR) exchange. However, this interface is not required to be serviced by the application, and tying it off can help reduce boilerplate code. An example user logic module definition is shown in listing 2.2.

In this work, we focus on the streaming I/O mode and ignore the FPGA's on-board memory. This is a logical choice given that we receive data from the FaaS infrastructure, apply the function on this data, and send the results back to the invoker. Since none of the data is re-used at a later stage, buffering in FPGA-attached memory is not necessary. One notable exception to this is the case of function chaining, where the on-board memory can bring significant benefits. This optimization is discussed superficially in future work (chapter 8.1), as it lies beyond the scope of this work. It is also worth noting that user logic still has access to certain memory facilities, such as Distributed RAM or Block RAM. These may be required for non-streaming algorithms to function. Invocations of FPGA user logic are always associated with a client process on the host machine. Input and output data is exchanged via shared memory buffers in host memory. These buffers are allocated by the client process; they do not have to be declared as shared memory to the operating system. However, Coyote requires these locations to be page-aligned. The client process will write all input data to the first memory region. At this point, the software side of Coyote can be instructed to trigger the start of the user logic, as shown in listing 2.1.

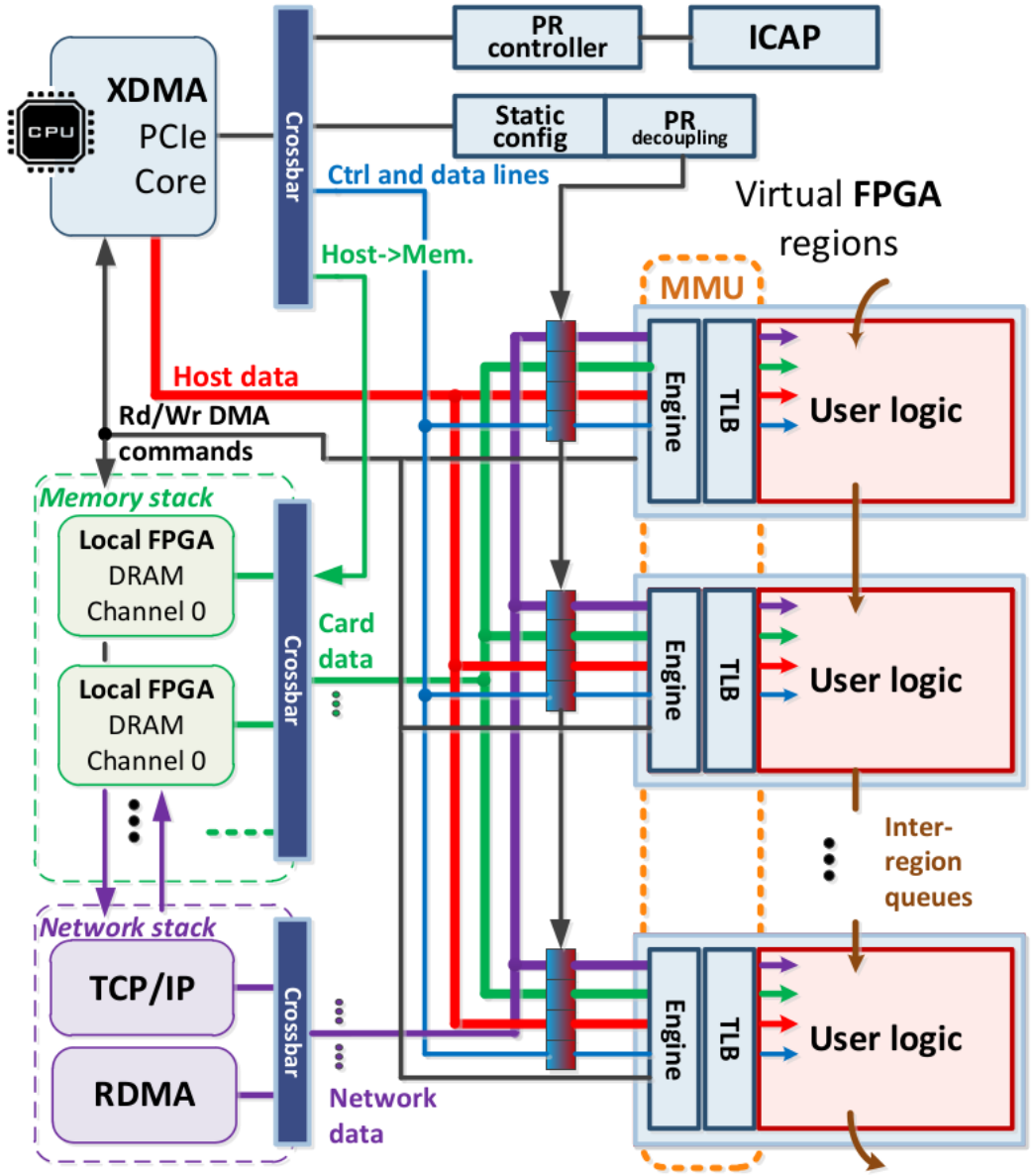


Figure 2.2.: Coyote shell architecture from [49]. The XDMA block (top left) establishes the host connection via PCIe and manages all access to host memory. One device can host multiple, isolated, virtual FPGAs (right). The PR controller and ICAP (top) enable dynamic partial reconfiguration. Coyote also supports on-board memory (green) and networking (purple), which are not used in this project.

```
cproc.invoke({fpga::CoyoteOper::TRANSFER,
              inputMemAddr, outputMemAddr,
              inputMemLen, outputMemLen});
```

Listing 2.1: Coyote library call for function invocation

The first argument pertains to the mode of operation, here we want to stream input data to the user logic on the FPGA, and we also expect output data back from the FPGA. The remaining arguments are addresses and lengths (in bytes) of the input and output memory regions, respectively. This information will be passed to the Coyote device driver, which will subsequently relay it to the Coyote shell in the static region on the FPGA. It should be noted, that the input and output memory locations are virtual addresses relative to the virtual address space of the host process. Consequently, the kernel device driver, prior to passing along memory locations, will translate virtual addresses into physical addresses. The input and output memory addresses are allowed to coincide, in which case the output data will overwrite the input data.

Upon receipt of the appropriate signals to execute a user logic, the Coyote shell will begin to stream the input data from the host memory to the user logic. This transfer is accomplished over the PCIe bus via DMA (Direct Memory Access), utilizing the Xilinx `DMA/Bridge Subsystem for PCI Express` (or simply `XDMA IP`). Output is also handled via DMA and is directly written to the designated memory location. The user logic will receive and send data over the two AXI4-stream interfaces previously mentioned. Only when the last byte of the output memory region has been written will the device driver consider the invocation complete and return control to the calling host process.

```
module design_user_logic_c0_0 (
    // AXI4-Lite control slave interface
    AXI4L.s             axi_ctrl,

    // AXI4-Stream host streams, input (sink) and output (src)
    AXI4SR.s            axis_host_0_sink,
    AXI4SR.m            axis_host_0_src,

    // Clock and reset
    input  wire         aclk,
    input  wire[0:0]    aresetn
);
endmodule
```

Listing 2.2: Verilog definition of a user logic module in Coyote

3. Design

3.1. vFPGA manager

Our system design employs a division of a single physical FPGA into a set of virtual FPGAs (vFPGAs). Multiple user logic modules can run in parallel, on spatially isolated regions on the device. One such virtual region has a fixed location and boundary on the chip. Reprogramming a virtual region is possible, without interrupting the operation of other virtual regions, by utilization of dynamic partial reconfiguration. We have designed and implemented the `vFPGA manager` – a daemon on the host machine, responsible for coordinating the user logic modules running on each of the vFPGAs. The functionality of `vFPGA manager` includes:

- Accept and queue invocation requests from clients
- Schedule invocation requests for execution
- Reconfiguration of vFPGAs, if necessary
- Invocation of user logic on the vFPGA
- Return signal to the invoker, as request completes
- Collect telemetry and forward it to the metrics collector

3.1.1. Accepting invocation requests

The `vFPGA manager` fulfills the role of a server in a client/server-architecture. Clients submit requests and the server will complete them, as soon as compute resources become available. Upon start-up, the `vFPGA manager` registers a UNIX domain socket in the file system at `/tmp/serverless.sock`. Clients may then connect to the socket in order to submit requests. The communication follows a simple, text-based protocol, as illustrated in figure 3.1. In detail, the steps involved are as follows:

1. The client connects to the `vFPGA manager` through the UNIX domain socket. The `vFPGA manager` queries and stores the process id (pid) through the operating system. The pid is required in a later step to resolve virtual addresses.

2. The client specifies the user logic it wants executed via the bitstream id. The dynamic region to use for the execution is determined by the `vFPGA manager` and not visible to the client. The combination of dynamic region and bitstream id uniquely identifies the partial bitstream file to be used.

3. Optionally, one or multiple control-and-status registers (CSR) can be defined in advance to the user logic being invoked. Registers are numbered, and each register can hold a 64-bit value. This message is repeatable, thus allowing multiple CSRs to be defined.

4. Apart from CSRs, the primary means of passing input and output is via shared host memory. The client prepares a region in main memory, which must be sufficiently large to accommodate both input and output data. It then passes the address and size of the memory buffer, as well as the input size and output size, to the `vFPGA manager`. The input size defines the amount of data, starting at the first address of the memory region, that will be streamed to the user logic. The output size specifies the amount of data to be expected back from the user logic. The output will be stored, beginning at the first address of the memory region, thus overwriting the input data. It should be noted that the memory address is a virtual address in the address space of the client.

5. Finally, the client sends the command to launch the invocation. The time until scheduling and completion of this command depends on the current FPGA utilization and latency of the user logic. Once the task begins execution on the FPGA, input data will be streamed from main memory to the user logic, and the output data will accumulate in the memory buffer, that was specified in the previous messages. Upon completion of the user logic's processing, the `vFPGA manager` will send a response to the client, to notify it about the completion. The client can then read the user logic's results from the memory buffer.

3.1.2. Scheduling

Our prototype implements a rather straightforward scheduling policy. Two types of resources are considered: tasks and vFPGA slots. Tasks are submitted at certain points in time, whose occurrences are not known a priori (online scheduling), and run for an unknown duration. Tasks are to be assigned to a slot by the scheduler, and each slot can execute at most one task at a time. In our model, tasks run to completion once started. An important property of each task is the bitstream id, which is analogous to a function name. In general, the number of different functions exceeds the available vFPGA slots

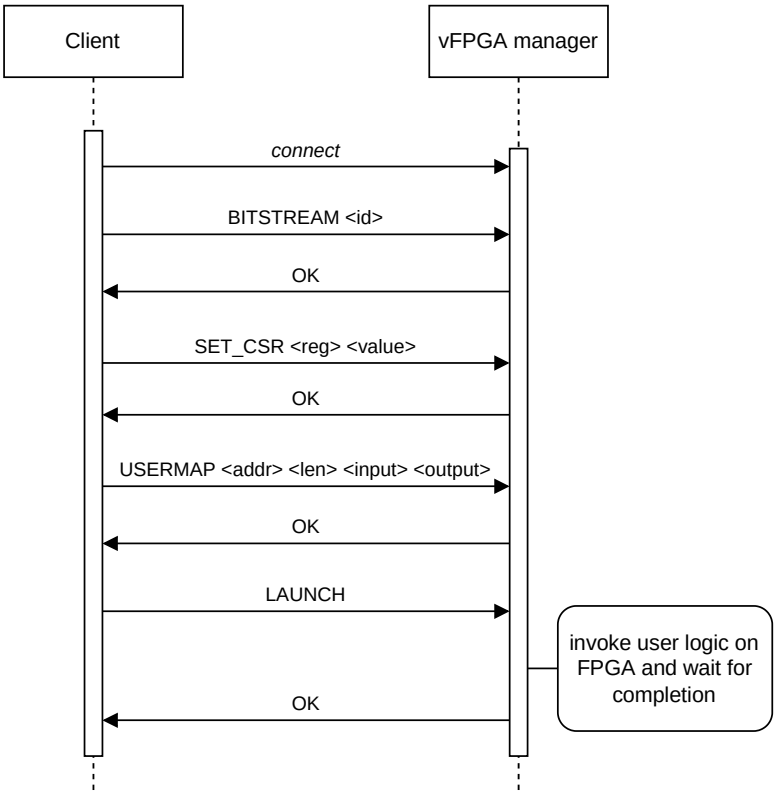


Figure 3.1.: Sequence diagram of the interaction between the vFPGA manager and a client.

on the device. In order to service functions, that are not currently configured on the FPGA, we leverage dynamic partial reconfiguration. However, this introduces a time factor that must be considered, as reconfiguring a vFPGA slot can take up to tens of milliseconds, depending on the bitstream size. We refer to the case where additional time is spent on configuring the correct bitstream on the designated vFPGA slot as a *cold start*. Ideally, a task can be assigned to a vFPGA region, that already has the correct bitstream programmed, which we refer to as *warm start*.

The scheduler maintains a concurrent, global queue of pending tasks. When a task is submitted by one of the clients, it first enters the scheduler queue. Each vFPGA region is embodied by a consumer, which attempts to retrieve the first element from the queue. If the queue is currently empty, then the consumer blocks, until a task is eventually submitted to the queue. As previously discussed, warm starts are preferable for improved performance. Upon submission of a task to the queue, if the bitstream id is already configured on one of the vFPGA regions, then the task is immediately assigned as the next task for that region, thus avoiding reconfiguration. However, the number of tasks that can be shortlisted in this manner is limited. This is advisable to avoid congestion in a situation with a high number of requests for a single function. In such a scenario, scheduling all tasks to the same vFPGA would be detrimental, given that other vFPGA regions may be idle. Instead, our scheduler will configure the highly-requested bitstream to multiple regions simultaneously, so that tasks can be processed in parallel.

During cold start, a reconfiguration is necessary to execute the requested user logic. This necessitates reprogramming one of the vFPGAs, thereby evicting the user logic currently configured on it. It is preferable to select a user logic that will not be used in the immediate future. To address this, an LRU (least recently used) policy has been implemented for handling cold starts. Specifically, the region selected for reconfiguration is the one where the most time has passed since the last execution.

3.1.3. Kernel device driver communication

When a task is ready to be executed, the vFPGA manager initiates a transaction with the Coyote kernel driver. Each vFPGA is made available by the driver through a separate device file, e.g. `/dev/fpga0`, `/dev/fpga1`, and so forth. The vFPGA manager then executes the following steps:

1. Acquire ownership of the virtual region through a system-wide lock.

2. If necessary, reconfigure the virtual region with the requested bitstream. The bitstream file may already reside in memory if it was used before. Otherwise, it is

loaded from disk into memory. The memory location is then passed to the kernel driver.

3. Set CSR registers, if the invoker specified them.
4. Map the memory region(s), reserved for input and output data. This grants the FPGA read and write access to the associated memory pages.
5. Initiate the user logic on the FPGA. This is a blocking call; control is returned once the task has finished. Coyote will start streaming the input data from the mapped memory to the user logic. Similarly, output data generated by the user logic is written back to the mapped memory.
6. Unmap the memory region, thus revoking FPGA access to the memory pages.
7. Release ownership of the virtual region.

3.1.4. Metrics collection

During each invocation, telemetry is collected regarding FPGA utilization. After the task has completed, the telemetry is relayed to a central metrics collector component. The rationale is to be able to route future workloads to other machines if the FPGA is over-utilized. The submitted information contains the following:

1. The current system time
2. Only in case of a cold start: Elapsed time for reconfiguring the virtual region
3. Elapsed time from starting the user logic until completion
4. Currently configured bitstream ids of all vFPGAs

The information is forwarded via HTTP POST requests (see listing 3.1). In the event that a reconfiguration was necessary, the first report submits reconfiguration time. After user logic has completed, the execution time is reported in a separate request. Both requests contain the bitstream ids, which are currently configured on the FPGA. Further details on the processing of the information can be found in [73].

```
http://MAIN_NODE_IP:PORT/?timestamp_ms=xxx
  &bitstream_identifiers=x,y,...,z
  &reconfiguration_ms=xxx

http://MAIN_NODE_IP:PORT/?timestamp_ms=xxx
  &bitstream_identifiers=x,y,...,z
  &duration_ms=xxx
```

Listing 3.1: HTTP POST requests to the central metrics collector

3.2. OpenFaaS

Although the heavy lifting occurs on the FPGA, our design requires a minimal, function-specific application running on the host. Its purpose is to receive the input data of the serverless function, make API calls to invoke the appropriate function on the FPGA, and finally to return the output data from the FPGA back to the invoker. This application runs in a unikernel environment on the host machine. The developer of the unikernel application packages it as an OCI-compliant image [41] and publishes it through an OCI container registry [40]. OpenFaaS [56] is used for the deployment and management of the unikernel applications in the cluster. OpenFaaS is a function-as-a-service framework, deployed on top of the Kubernetes [69] stack. This architectural approach enables a landscape with multiple compute nodes, as is typical for cloud deployments. A central component of OpenFaaS, the gateway, is responsible for listening for invocation requests and forwarding them to one of the compute nodes in the cluster. Figure 3.2 illustrates the flow of an invocation request in our design. First, the user contacts the OpenFaaS gateway, which is accessible under a known API endpoint. The request, including input data, is forwarded to one of the worker nodes; more specifically, an HTTP call is issued to the unikernel application for the requested function. The input data is contained in the request body of the HTTP request. The unikernel is a minimal application, that communicates to the API provided by the vFPGA manager. The interaction can be initiated from within the VM by issuing the appropriate hypercalls. This will instruct the `vFPGA manager` to schedule the accelerated function on the FPGA. Once the function has been scheduled and executed, the unikernel application will send the results back in the response body of the HTTP request. This overview of OpenFaaS shall be sufficient, as it is not the main focus of this thesis. Readers interested in learning more about the cluster setup, OpenFaaS, and scheduling on the Kubernetes scale may consult [73].

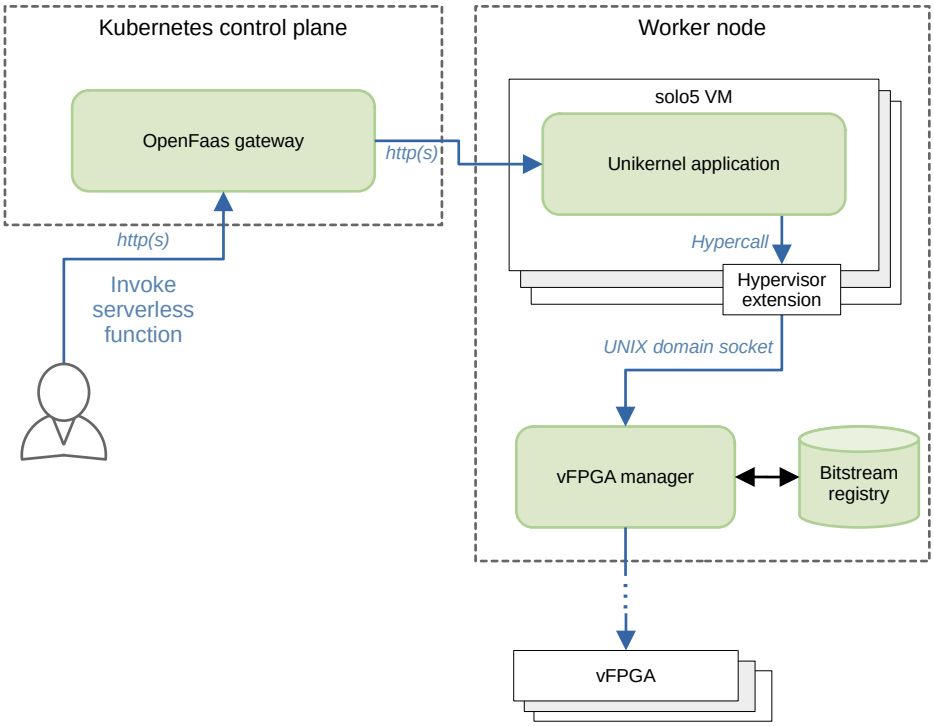


Figure 3.2.: Function invocation through OpenFaaS. The user calls the endpoint exposed by the OpenFaaS gateway. The request, including input data from the user, is forwarded to the unikernel application, which issues appropriate API calls to the vFPGA manager. The vFPGA manager invokes the user logic on the vFPGA. After completion, the result is returned back to the user.

3.3. Solo5

Serverless computing landscapes are typically multi-tenant configurations, wherein physical resources are shared by multiple users. This allows for better resource utilization of hardware, but calls for a robust security model with encapsulated runtimes for each tenant. To enforce the isolation on the host side, unikernel applications are running in confined, virtual machines. As the unikernel applications are user-provided executables, they must be considered as potentially malicious.

As performance is a critical metric in our context, we have opted to run the host code in a unikernel environment within a lightweight virtual machine. A unikernel is an application that is statically linked with a (heavily reduced) operating system kernel into a single binary. The binary can run directly on top of a virtual machine, obviating the need for an underlying operating system or kernel. Library operating systems [11] provide often-used functionality, such as a network stack with support for DNS, TCP/IP, HTTP, and so forth. The result is a minimal application, devoid of unnecessary overheads and exhibiting excellent performance characteristics in comparison to an ordinary application running on top of a full operating system. As unikernels have such a modest set of requirements on the surrounding environment, the hypervisor and virtual machine can be stripped down to a minimal configuration – just enough to run the unikernel. This concept allows for fast boot times and minimal memory footprints while profiting from strong encapsulation through virtualization [81] [87].

In our design, lightweight VMs are managed by the solo5 hypervisor [32], on top of the Linux Kernel-based virtual machine (KVM) infrastructure. We propose includeOS [11] as a library operating system and extend it with our API. However, there is no technical means (or purpose) of forcing developers to use a specific library operating system. This means that any compatible binary will be able to run in the virtualized environment.

As the unikernel application has to interact with the vFPGA manager, it is necessary to establish a communication channel between the two components. The unikernel runs in an isolated space, therefore all communication must necessarily pass through the hypervisor. Consequently, communication with the vFPGA manager can be divided into two steps. First, the unikernel will issue an API call, in the form of a hypercall to the hypervisor. Subsequently, the hypervisor forwards the API call to the vFPGA manager. This is accomplished via the previously mentioned UNIX domain socket (section 3.1.1).

We have extended the solo5 hypervisor and includeOS by implementing hypercalls as specified in listing 3.2.

```
/*
Sets one control-and-status register, immediately before invocation of
the user logic. CSRs can be thought of as function arguments and it
depends solely on the user logic, if/which CSRs have to be written.
Calling this method is optional.

@input offset The id of the register that will be written
@input value  The value that will be written into the register.
*/
solo5_result_t solo5_serverless_set_csr(uint32_t offset, uint64_t value);

/*
Specify the bitstream to be executed.
Calling this method is mandatory.

@input config The identifier of the bitstream
*/
solo5_result_t solo5_serverless_load_bitstream(uint32_t config);

/*
Makes the specified memory region available for read/write from
the user logic on the FPGA.
Calling this method is mandatory.

@input addr       The virtual address of the memory region to share.
@input len        The size of the memory region in bytes.
@input input_len  The number of bytes, that constitute the input data of
                  this invocation. The bytes will be read starting at
                  the first address of the memory buffer.
@input output_len The number of bytes to expect back from the user logic.
                  The bytes will be written starting at the first
                  address of the memory buffer.
*/
solo5_result_t solo5_serverless_map_memory(void *addr,
                                           size_t len,
                                           size_t input_len,
                                           size_t output_len);


```

```
40 /*
41 Schedules the execution of the user logic. It can be placed on any
42 of the available vFPGAs, as the scheduler sees fit. The call is blocking
43 and returns when the function has completed on the FPGA. The
44 output data is then available in the shared memory region.
45 */
46 solo5_result_t solo5_serverless_exec(void);
```

Listing 3.2: Implemented hypercalls

4. Benchmark

For the purpose of evaluating our design, we have developed a set of benchmark applications based on open-source implementations of popular algorithms. For each application, we have prepared one implementation to run on the FPGA and another for running on the CPU. The latter is used as a baseline in the performance evaluation.

4.1. AES-128-ECB

AES (Advanced Encryption Standard [78]) is a data encryption standard, established by NIST (US National Institute of Standards and Technology). AES operates on blocks of 128 bits of data and key lengths of 128, 192, or 256 bits, depending on the selected AES variant. In cryptography terminology, the unencrypted input data is referred to as *plain text*, while the encrypted output data is known as the *cipher text*. AES is a symmetric encryption algorithm, which means that both the encryption and decryption use the same secret key. To accommodate plain text input sizes that are not divisible by the block size, padding algorithms are used to extend the data size to the next multiple of 128 bits [45]. Consequently, cipher text is always a multiple of 128 bits. Different modes of operations exist for AES, here we focus only on Electronic Codebook (ECB) mode, in which each block of data is encrypted individually. Thus, each cipher block is dependent solely on the corresponding plain text block and the secret key. Albeit insecure in almost any scenario, it is the simplest mode of operation. In particular, the implementation in this mode allows for the parallel computation of individual blocks, which is beneficial for an FPGA implementation.

Coyote comes with a VHDL implementation of AES-128 encryption in ECB mode. The 128-bit encryption key is transferred from the host program to the accelerator via two 64-bit CSR registers. The implementation drives four identical encryption cores, each operating on one block of 128 bits of data. This allows processing at the full bandwidth of the AXI4-streaming interface (512 bits).

Later, we intend to compare the FPGA implementation against a CPU baseline. OpenSSL [67] is a popular cryptography library, and we base our CPU baseline on OpenSSL's implementation of AES. It should be noted that typical CPUs are bolstered with hardware extensions, that can efficiently encrypt and decrypt AES at the clock speed of the CPU [42].

4.2. SHA3-512

Hashing is the deterministic transformation of a potentially large amount of input data to a small, fixed-size word, known as the *digest*. It is an essential property of cryptographic hash functions, that the transformation cannot be reversed. Furthermore, given the digest, the only way to make assumptions about the original input data is by exhaustive search over all possible inputs. SHA3 [21] is a family of cryptographic hash functions, established by the US NIST in 2015. We consider the variant SHA3-512 here, which operates on blocks of 576 bits and produces a digest of 512 bits. Padding is required to handle input data that is not an exact multiple of the block length.

We have ported an open-source Verilog implementation of SHA3-512 by H. Hsing [37] to Coyote. The core accepts input words of 32 bits and, after signalling the end of the stream, produces a single output word of 512 bits. The throughput of the core is 2.4 Gbit per second on a 100 MHz clock frequency per the specification [38].

Coyote provides 512-bit wide AXI4 streams to feed input data to the user logic, which does not match the interface of this core (32-bit words). We used Xilinx' `Axis Datawidth Converter` IP to transform between the two. In addition, the core has a custom (non-AXI) interface to signal the terminal input word. The requisite logic was implemented to translate between the AXI4-stream end-of-data signals `TLAST`, `TKEEP` and the core's interface. Since the result is a single word of 512 bits, passing it to the output AXI4-stream interface is straightforward. The CPU baseline implementation is based on the OpenSSL library [67].

4.3. SHA-256

SHA-2 is another family of widely-used cryptographic hash functions [16]. SHA-256 is the variant that produces a 256-bit message digest. The standard was approved by the US NIST in 2002. Even though SHA-3 has succeeded the standard, SHA-2 remains prevalent in a variety of contemporary technologies, such as secure email communication, TLS-based protocols like HTTPS, and cryptocurrencies. Some modern CPUs have hardware extensions for the efficient computation of SHA-256 hashing [43].

We have ported the free SHA-256 Verilog implementation from J. Strömbergson et al. [80] to Coyote. Their implementation includes a 512-bit-wide AXI4-stream wrapper, which simplifies the integration. Since the core lacks padding on the last block of the data input stream, we implement the proper padding [22] on the host side, before passing input data forward to the core. The result is a single 256-bit message digest, returned via the output AXI4-stream. As before, the CPU baseline implementation utilizes the OpenSSL library [67].

4.4. AddMul

AddMul is one of the examples that comes with Coyote to demonstrate a simple user logic, that makes use of the AXI4-stream input and output, as well as the AXI4-lite interface for passing data via CSR (control and status registers). The input is a stream of 32-bit integers. Given that the streaming interface is 512-bits wide, each word contains 16 (= 512/32) integers. Each integer is binary-shifted left by x bits, then incremented by the value of y. Here, x and y are the two parameters that are passed in via CSR registers at the beginning of the program. The results are written back to the output AXI4-stream, again, partitioned in words of 16 integers. The implementation has a two cycle latency, i.e. for any integer passed on the input stream, the computation result is available through the output stream two clock cycles later.

4.5. Needleman-Wunsch

A recurring problem in bioinformatics is the optimal alignment of sequences, such as DNA or protein sequences. In 1970, Needleman and Wunsch published a dynamic programming algorithm for finding the best global alignment between two sequences, with respect to a scoring matrix [63]. The scoring matrix assigns a value to each pair of characters, from the alphabet of the sequences. Additionally, scores are assigned for introducing gaps in the sequence alignment. Figure 4.1 illustrates an example of the global alignment between two sequences.

We have ported a benchmark from FSRF [52], which computes the Needleman-Wunsch (NW) alignment score on a large corpus of sequences. The input consists of two arrays of sequences, where each sequence is a string of 64 characters or 128 bits. Each character is encoded in two bits, as DNA sequences are represented using one of four (= two bits) nucleotides. The benchmark code computes the score of the optimal global alignment for all pairs of sequences in the cartesian product of the two input arrays. In other words, the first sequence from the first input array is aligned with all sequences from the second input array, then the second sequence from the first array is aligned with all sequences from the second array, and so on. The output of the code is the optimal score for each pair of sequences.

FSRF implements a virtual memory model with an AXI4 interface exposing "5 independent channels: two for initiating R/W transactions, two for R/W data, and one for write completion meta-data" [52]. Host and FPGA exchange input and output data over this shared, virtual memory, which supports random access reads and writes by design. In Coyote, however, only two streaming interfaces are available for exchange of input and output data.

In our forked code, we re-implemented the memory management in parts and wrote a wrapper module, which reads from the input AXI4-stream the two input arrays and stores them in a buffer. The buffer was realized using Xilinx' `AXI BRAM Controller`. This IP utilizes block random access memory (BRAM) on the FPGA and can be configured to provide an AXI4 interface to the memory. We exploit the fact that AXI4 has separate channels for reading and writing access. Since our input handler only requires write-access to the BRAM, we connect the write channels to our input handler, while we forward the read channels to the ported NW code. In AXI4 terminology, the write channels are prefixed `AW`, `W` and `B`, while the channels for read access are prefixed `AR` and `R`. Once all input has been received and stored in BRAM, the memory addresses are passed to the NW code and the start signal is fired.

In the original NW implementation [52] the output scores are also written to random access memory. We modified the code to write to an AXI4-stream instead. In fact, all results are written directly to Coyote's output stream, where they are subsequently transferred back to the host.

At the core of the ported NW code, there is a logic module to compute the optimal alignment score according to the Needleman-Wunsch algorithm. We want to explain the implementation in some detail here, as it is a striking demonstration of the parallelization and pipelining paradigm, that FPGAs excel at. We won't analyze the algorithm itself in depth here nor prove its correctness, and instead focus on the runtime aspects of the implementation. NW is a dynamic programming algorithm, the pseudocode is given in listing 4.1. The input are two sequences, here they have the same length `L`. The two-dimensional dynamic programming matrix is denoted `dp`. The algorithm iterates over all pairs of `(i, j)` and computes the value of the current cell according to the dynamic programming recurrence (lines 11-16). The score is incremented, if two matching characters are aligned, otherwise a penalty is subtracted from the score (line 13). The result will be `dp[L][L]` after the matrix has been completely filled. Note that `dp[i][j] (i,j > 0)` is defined in terms of three previous cells (line 14). If the matrix is drawn on paper, it can be observed that each cell is dependent on the cells immediately above, immediately to the left, and diagonally top-left of it (cf. figure 4.1). A classical CPU implementation computes one cell after another, thus running in $\mathcal{O}(L * L)$ time complexity.

4.5.1. Parallelization

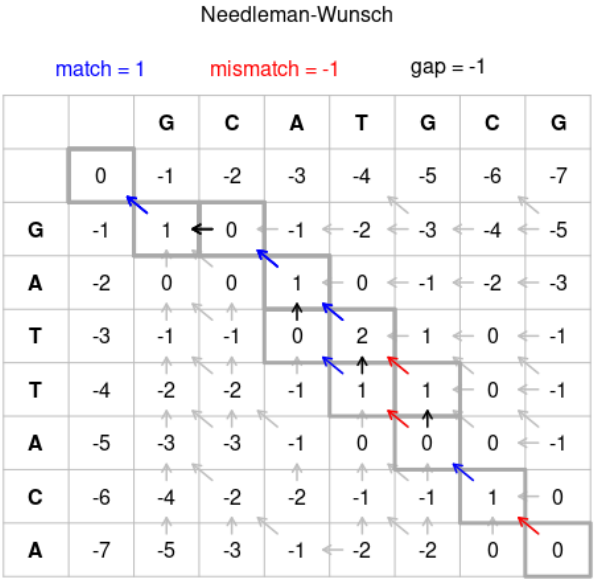
The FPGA implementation exploits the access pattern of each cell to compute multiple cells in parallel. Let d_* denote the diagonals in the `dp` matrix, running from top-right to bottom-left. The first diagonal is the cell `dp[0][0]`, the second diagonal contains the two cells `dp[0][1]` and `dp[1][0]`, and so on. It can be observed that all cells in

diagonal d_i depend exclusively on cells in d_{i-1} and d_{i-2}. By trading chip area for speed, an array of identical logic blocks can be implemented, each computing one cell of the same diagonal in the same clock cycle. Thus, by computing each diagonal in one clock cycle, the total runtime has been drastically improved. The number of diagonals is $|d_*| = 2 * L - 1$, thus the time complexity is reduced to $\mathcal{O}(L)$.

Further improvement is possible by computing two diagonals per clock cycle, instead of only one. Such optimization causes longer chains of logic gates on the FPGA to be driven in a single clock cycle. The maximal permissible length of a chain is primarily dependent on the clock frequency. A lower frequency allows for longer chains but causes an overall slowdown, hence a higher frequency is usually preferred. In order to verify compliance of logic gate paths with respect to the chosen clock frequency, an FPGA design suite such as Vivado will conduct a timing analysis and report violations.

4.5.2. Pipelining

The previous idea can be further developed. It has been demonstrated that each diagonal can be computed in terms of the previous two diagonals. Any diagonals preceding those are no longer necessary for the current computation and can be discarded. This fact can be exploited to initiate the computation on the next set of input data, as it will initially only occupy the top-left corner of the matrix. After a few more clock cycles, when the top-left corner becomes irrelevant again, yet another computation can commence. We pipeline successive executions of the same logic by utilizing resources as they become vacant, thus making another significant improvement on the total runtime. If k denotes the number of successive invocations of Needleman-Wunsch, then the ordinary CPU implementation has a time complexity of $\mathcal{O}(L * L * k)$. In contrast, the parallelized, pipelined FPGA implementation has a runtime of $\mathcal{O}(L + k)$. Note, however, this is only correct under the premise, that invocations occur in immediate succession, as is the case in the adopted NW benchmark code.



		G	C	A	T	G	C	G
	0	-1	-2	-3	-4	-5	-6	-7
G	-1	1	0	-1	-2	-3	-4	-5
A	-2	0	0	1	0	-1	-2	-3
T	-3	-1	-1	0	2	1	0	-1
T	-4	-2	-2	-1	1	1	0	-1
A	-5	-3	-3	-1	0	0	0	-1
C	-6	-4	-2	-2	-1	-1	1	0
A	-7	-5	-3	-1	-2	-2	0	0

Figure 4.1.: The dynamic programming matrix `dp`, after executing the Needleman-Wunsch algorithm on a pair of DNA sequences. An optimal alignment of sequences can be found by following the highlighted arrows, starting from the bottom-right cell. In this case, the optimal alignment is not unique. The scoring matrix used here is very simple: Pairs of characters are assigned a score of 1 if they match (blue arrows), and -1 on mismatch (red arrows). The gap penalty is -1 (black arrows). From [76].

```
int needleman_wunsch(string a, string b) {
  # We assume that a and b are both strings of the same length L
  L = |a| = |b|
  dp = new int[0..L][0..L] # Declare a 2D-array
  # Initialize base cases
  for(i = 0; i < L+1; i++) {
    dp[0][i] = -i # the gap penalty is -1 per unit
    dp[i][0] = -i
  }
  # The recursion
  for(i = 0; i < L; i++) {
    for(j = 0; j < L; j++) {
      score = if (a[i]==b[j]) then 1 else -1 # Do both characters match?
      dp[i+1][j+1] = max(dp[i][j+1]-1, dp[i+1][j]-1, dp[i][j]+score)
    }
  }
  return dp[L][L]
}
```

Listing 4.1: Needleman-Wunsch pseudocode

4.6. hls4ml CNN

Due to recent advancements, machine learning applications are becoming more and more popular. FPGAs can be harnessed for their low-latency, low-power characteristics as machine learning accelerators. `hls4ml` [24] is a framework for translating machine learning algorithms to high-level synthesis (HLS) code. HLS code can be further compiled and synthesized by Vivado or Vitis for use with FPGA devices. We utilized hls4ml to implement a convolutional neural network (CNN) for the detection of house numbers from images. The dataset consists of 32×32 pixel RGB images, captured from real-world street view footage, as illustrated in figure 4.2. The objective is to classify images based on the digit in the center of the image. Thus, each sample belongs to one of ten classes, one for each digit from zero through nine.

The hls4ml authors provide a tutorial with working code to build, train, optimize, and synthesize the neural network for house number recognition. The tutorial features two machine learning models, here we are using the improved version. This version is optimized by pruning dense layers, i.e. zeroing out a portion of weights. The second optimization, quantization, is the idea of avoiding floating point numbers and

Figure 4.2.: Three samples from the SVHN (Street view house numbers) data set [64]. Images are labeled with numbers zero through nine, representing the digit visible in the center of the image. Here: 4, 8, and 7, respectively.

instead using fixed-point decimals. This concept is not limited to FPGAs, but it is particularly useful here. FPGA devices do not come with native support for floating point operations, and thus, such functionality must be included in the logic design, consuming additional clock cycle and resources on the chip [44]. On the other hand, arithmetic on fixed-point decimals is fundamentally no different from arithmetic on integer numbers. Fortunately, FPGAs have built-in support for these operations through plenty of digital signal processing (DSP) logic slices on the chip. It can be shown that quantization has only moderate impact in terms of prediction performance [24] [14].

It should be noted that the model is trained only once during the development phase. The network weights are then hardwired into the synthesized logic so that the FPGA code contains only the neural network forward pass (inference), not the backward pass (training). This approach ensures that the logic remains relatively simple. A depiction of the layers in the neural network is included in the appendix figure A.1.

Once the model has been synthesized, it can be exported into an IP core. We integrate the IP core into our project using Vivado and implement a wrapper to mediate input and output handling. The hls4ml core exposes AXI4-streams for both input and output. The input data channel is a 48-bit wide bus, corresponding to one pixel of input, as each pixel is encoded into three 16-bit fixed point values, one per color channel. After passing in an entire image, 160 bits of output are produced. This corresponds to one 16-bit fixed point decimal per each classification category, i.e. the digits from zero through nine. The category with the highest score is the one inferred through the neural network.

4.7. Matrix Multiplication

A basic operation in linear algebra is the one of matrix multiplication. The naïve algorithm (listing 4.2) is very simple. Given $n \times m$ matrix A, and $m \times k$ matrix B, the product $C = AB$ is a $n \times k$ matrix, computed by:

```
for i from 0 to n:
  for j from 0 to k:
    C_ij ← 0
    for p from 0 to m:
      C_ij ← C_ij + A_ip * B_pj
```

Listing 4.2: Naive matrix multiplication algorithm

This implementation has a runtime of $\mathcal{O}(n * m * k)$ when assuming constant runtime for arithmetic addition and multiplication. In the case of A and B being square matrices with dimension $n \times n$ the complexity is simply $\mathcal{O}(n^3)$. Algorithms with better asymptotic runtime exist, however, not all of them are practical. For small matrices, the naïve algorithm is often faster, despite cubic time complexity [60].

For this project, we implemented a matrix multiplication algorithm from scratch. The algorithm is written in Verilog and expects two 64×64 matrices and returns one 64×64 matrix. The elements are signed 64-bit integers. Here we want to describe the implementation in some detail, as it exemplifies the time and effort that goes into writing efficient FPGA code. The implementation tallies up to approximately 300 lines of code, including input/output handling via AXI4-streams.

4.7.1. Structure

The main module instantiates three block memory modules. These are produced by the Xilinx `Block Memory Generator` IP. Each module stores elements of one of the two input matrices A, B, or the output matrix C. The data bus has a width of 4096 bits, which allows for the storage and retrieval of exactly one row of a matrix at each memory address (4096 = 64 × 64 = number of columns × integer precision). The block memory is configured to service read requests with one clock cycle latency, i.e. when the desired read address is presented to the core at the rising edge of the t-th clock cycle, the data is retrieved from memory and made available on the rising edge of the $(t + 1)$-th clock cycle.

Furthermore, an instance of the `addtree` module is created, which is responsible for the calculation of arithmetic operations. The module implementation is described in greater detail below.

4.7.2. State machine

The main module is organized as a state machine, which is traversed sequentially. Each state performs a specific task. Upon completion of a stage, the state machine transitions to the next state in accordance with the sequence outlined below.

I. Initialization

This is the initial state after a reset. All internal registers are initialized, and the process then continues to the next state.

II. Reading matrix A

The input stream now accepts data, specifically the elements of the first input matrix A. In total, 64×64×64 bits are transferred, as the matrix consists of 64^2 signed integers, each 64-bit wide. All values are written into the block memory reserved for matrix A. The block memory is addressed by 6 bits. The data at address i is the i-th row vector of A.

III. Reading matrix B

After matrix A has been received, the logic expects the elements of matrix B on the input stream. The values are stored in the memory reserved for B. Simultaneously, the first element of each row is extracted and stored in a 4096-bit register denoted `col0`. Ultimately, `col0` corresponds to $B_{*,0}$ – the first column vector of B. The column vectors of B are important for the next step.

IV. Vector multiplication

IV. (a) Fetch vectors
This is the state with the actual computation. The calculation of $C_{i,j}$ is performed in the order $C_{0,0}$, $C_{1,0}$, $C_{2,0}$ ... $C_{63,0}$, $C_{0,1}$, $C_{1,1}$, ... $C_{63,1}$, $C_{0,2}$ ... $C_{63,63}$. It should be noted that this column-order traversal differs from the CPU calculation. It corresponds to swapping the two outer loops of the algorithm in listing 4.2. The altered order is beneficial in terms of execution time. Recall that $C_{i,j}$ is the product of row vector $A_{i,*}$ multiplied with column vector $B_{*,j}$. Given that matrices A and B are stored in row-order in memory, retrieval of a row vector is as simple as a single read (fast), but retrieving a single column vector requires 64 reads (slow). Now that C is traversed in column-order, the cheap operation (increment row index) occurs frequently, while the expensive operation (increment column index) occurs rarely.

In each cycle, one element of $C_{i,j}$ can be processed. To compute its value, it is necessary to have both the row vector $A_{i,*}$ and the column vector $B_{*,j}$. The row vector can be retrieved from memory with a latency of one cycle, and the column vector is conveniently stored in register `col0`. Each vector contains 64 elements, and applying the formula for the product entails 64 integer multiplications and 63 integer additions. A thoughtless implementation would simply start calculating the result and 64 (or more) cycles later, write the result in the output memory, analogous to the naïve CPU algorithm. However, there is a better solution: We pass both vectors to the `addtree` module and – while `addtree` is computing the result – continue with the execution in the next cell of C.

IV. (b) Vector/vector multiplication

The `addtree` module accepts two vectors and computes their inner product within a few clock cycles. To achieve high performance, the module has a parallelized, pipelined design: The module accepts input at every clock cycle, even while previous computations are still in progress. The module is internally structured as a perfect binary tree, i.e. there is a root node with two children, and each child has another two children and so on, down to the lowest level, which consists of 64 leaf nodes. Each node contains a register to store a 64-bit signed integer.

Initially, when new input is passed into the module, the values are stored in the leaf nodes. Let's denote the two input vectors a and b and let u_i denote the i-th leaf node ($0 \leq i \leq 63$). Then, in the clock cycle in which input is passed in, the product of each pair of elements from a and b are computed and stored in the leaf node: $u_i \leftarrow a_i * b_i$, ($0 \leq i \leq 63$). Computing the product of two signed integers can be accomplished by utilizing the in-built DSP slices, with the result becoming available within the next clock cycle. At the end of the clock cycle, the multiplication product is written into the leaf node.

Furthermore, in each clock cycle, all non-leaf nodes simultaneously pull both values from their two children nodes and add them together. Analogous to the process of multiplication, computing the sum of two signed integers is featured by the on-board DSP slices. The sum is stored into the node, at the end of one clock cycle.

After eight clock cycles, the values from the two input vectors have ascended into the root node. In fact, the value of the root node is the sum of all pairwise products from the two vectors, or in other words, the inner product over the two input vectors. The computation is thus complete, and the result is passed back to the main module.

V. Wait for results

In the previous step, all the vectors to be multiplied were transferred to the `addtree`-module. A certain latency period must elapse before the results are available. In this stage, the logic simply awaits the final results computed by the `addtree` module. All results are stored in the block memory, reserved for matrix *C*. The order in which results are published by `addtree` is identical to the order in which the input was presented to it, namely in column-order of matrix *C*. However, a row-order memory layout is more favorable for the next step. Hence, memory addresses are computed such that the matrix is stored in row-order. `addtree` yields at most one matrix element per clock cycle, allowing the memory module to keep up with the pace of processing. Once the final result has been computed and stored, the transition to the next state is performed.

VI. Write output stream

Now that the result matrix *C* is available in block memory, it can be transmitted via Coyote's output stream. The memory component has a data bus of 4096 bits, which corresponds to a full row vector of matrix *C*. The output stream has a 512-bit bus, thus each row vector is split across 8 words. After the last value has been transmitted, the user logic resets itself, transitioning to the initial state and waiting for a potential next invocation.

4.8. Gzip compression

Gzip [26] is a popular file format for lossless compression on Unix-like operating systems. The format specification and reference implementations for compression and decompression are free and open-source, as part of the GNU project. Despite its initial release in 1992, the format continues to be widely used, e.g. to reduce data transfer of websites, images, and large files on the internet. The compression algorithm itself is known as DEFLATE, which is based on Lempel-Ziv compression followed by Huffman coding [19] [18].

Xilinx offers the `LOGICore Lossless Compression` IP for compression and decompression with the DEFLATE algorithm. The core also supports the gzip file format and AXI4-stream input and output interfaces. In order to integrate the gzip compression functionality into our project, we implemented a wrapper around the IP core. The purpose of the wrapper is to control the core and establish a conversion between the input/output stream of Coyote and the IP.

One peculiarity of the IP core is that, instead of a single clock, it requires multiple clock signals from the surrounding design. One of the clocks must operate at "twice

the frequency" of the other clock signal, and both clocks "should be phase-aligned and originate from [the] same source" [89]. Our first approach involved implementing a frequency multiplier inside our wrapper module using Xilinx' `Clocking Wizard`. The IP core depends on certain hardware primitives for clock signal generation, therefore the resources for the primitives must be present in the dynamic region designated for the user logic. This constraint was not met for our initial design, specifically the physical location we had reserved for the dynamic regions lacked the necessary resources. Consequently, we adjusted the placement in the floorplan, such that all virtual regions contained the resources necessary.

We also explored an alternative approach that does not impose restrictions on the virtual region placement. Coyote already employs a clocking wizard in the static region of the shell. It is responsible for generating independent clock circuits for the static region, all dynamic regions, and the network module. They are intentionally separated, such that the shell in the static region can operate at a clock frequency different from the user logic in the dynamic regions. The clocking wizard can be modified to synthesize an additional clock signal at half the frequency and phase-aligned to the dynamic region clock signal. The new clock signal needs to be wired and added to the user logic interface. Inside the user logic, the clock signal can then be connected to the compression IP core. As previously mentioned, the main advantage of this approach is that no additional constraints are imposed on the placement of virtual regions. Conversely, modifying Coyote and the user logic interface breaks compatibility with all existing user logic modules. Although the resolution is trivial, adding the extra clock signal pollutes the interface of all other user logic modules. For this reason, our final design incorporates the approach described in the previous paragraph instead.

The wrapper provides a conversion layer between the input/output stream of Coyote and the IP core. They operate on words of different lengths as well as different clock frequencies. To make the streams compatible with each other, our design exerts Xilinx-provided IP cores `Axis Dwidth Converter` and `Axis Clock Converter` for data width conversion and clock conversion, respectively.

Another challenge was encountered with Coyote's design. In advance of invoking user logic, memory regions have to be reserved for the input data and the output data, respectively. When submitting a task to Coyote, the task structure contains the addresses and dimensions of both memory buffers. However, the dimension of the output memory must match exactly the amount of data that is expected from the user logic. Up until the very last byte of the output memory buffer has been written to, Coyote considers the task running. If the user logic never produces the amount of data, that Coyote expects from it, this causes a deadlock, and the vFPGA is no longer usable. In many cases, the amount of output to be expected is known in advance, so this is not an issue. However, it poses a major obstacle for the gzip compression module, as there

is no way to determine the compressed file size ahead of time. To overcome this issue, we prepared a set of test vectors, for which we have manually examined and recorded the compressed file size generated by the IP core. When using the compression module with one of the test vectors, the output memory region can be sized appropriately. It is evident that this approach is not a viable solution for a production-grade product, but it shall suffice for the scope of our academic project.

4.9. HyperLogLog

HyperLogLog [25] is a probabilistic data structure and algorithm. It answers the question of unique elements in a stream of data, that may contain duplicates. This could easily be answered by storing all elements in an ordinary data structure like a binary tree, hash set, or similar. Memory consumption is then linear in the number of (unique) elements. When dealing with very large data streams, the associated memory requirements can become prohibitively expensive for practical application. The HyperLogLog algorithm represents a trade-off between memory space and correctness. It consumes significantly fewer memory resources than other algorithms, at the cost of approximating the number of unique elements, instead of assessing the exact count. It can be shown that HyperLogLog has a low relative error and that it is "near optimal among algorithms that are based on order statistics" [35].

The FPGA implementation is part of the examples bundled with Coyote. The CPU counter-part we use for comparison is based on an open-source implementation of the HyperLogLog algorithm [29] and the compute-efficient hash function MurmurHash3 [4].

4.10. Harris Corner Detection

FPGAs are increasingly being used in computer vision applications. Detecting corners in images or video feeds is often a fundamental component of larger pipelines. A popular choice to solve this task is the Harris corner detection [33] algorithm. It is regarded as "one of the most precise corner detection algorithms. Although its operation is notably simple, the algorithm is computationally intensive" [75].

Xilinx offers an implementation of the Harris corner detection algorithm in their Vitis Vision Library [88]. We realized a wrapper module around the library function. It reads an 8-bit grayscale, 1920×1080 pixels input image from Coyote's input stream, and converts it into the appropriate matrix data structure. The matrix is passed to the Vitis Vision library function. The result is also encoded in a matrix, where each matrix element corresponds to a score for the pixel at that position. A higher score indicates a

higher probability of a corner being present at that location. The matrix is converted back and returned via Coyote's output stream.

As baseline, we rely on the popular OpenCV [10] computer vision library, which provides a CPU implementation of the Harris corner detection algorithm [65].

4.11. MD5

Another well-known message-digest algorithm is MD5. It was developed in 1991 and one year later specified as RFC 1321 [72]. MD5 processes blocks of 512 bits and generates a hash value of 128 bits in length. Analysis of its cryptographic properties has since revealed fatal weaknesses in the algorithm, rendering it unsuitable for cryptography applications [83] [47].

We have built a module around an open-sourced MD5 core [53]. Our module takes an MD5 hash as input and attempts to find the original text through brute force. The module iterates over all five-digit alphanumeric strings and generates the MD5 hash for each of them. If the hash matches the input, the module reports the five-digit sequence. A total of 64 possibilities are considered for each position in the string: numbers from zero to nine, Latin letters in upper and lower case, and two special characters. Therefore, the total number of combinations is $64^5 \approx 10^9$. The MD5 core is pipelined and can compute one hash per clock cycle (if the pipeline is fully utilized). The time required to examine all 64^5 combinations is less than 4.5 seconds at a clock frequency of 250 MHz. The CPU baseline uses the MD5 implementation from OpenSSL [67].

4.12. FFT

In the field of digital signal processing, discrete signals can be analyzed using *discrete auto-correlation*. The given sequence of samples is compared against time-shifted copies of itself to identify repeating patterns within the sequence. Formally, if x denotes a real, N-periodic sequence, then the auto-correlation $a_x(l)$ is defined as [77]:

$$a_x(l) := \frac{1}{N} \sum_{n=0}^{N-1} x(n)x(n+l) \tag{4.1}$$

A naive computation of $a_x(l)$ computes the sum of products for every value of l, thus leading to $\mathcal{O}(N^2)$ time complexity. A significantly more efficient computation can be achieved by harnessing FFT (Fast Fourier Transform) [86]:

$$a_x(l) = \mathsf{IDFT}\left(\mathsf{DFT}(x(n)) * \overline{\mathsf{DFT}(x(n))}\right) \tag{4.2}$$

Here, DFT and IDFT represent the discrete Fourier Transformation and its inverse, respectively. The bar denotes complex conjugation and $*$ is the element-wise product. Here, the time complexity of computing $a_x(l)$ is dominated by the discrete Fourier Transform, which is in $\mathcal{O}(N \log N)$ [77].

To compute the FFT and its inverse on an FPGA, we facilitated the `LogiCORE IP Fast Fourier Transform`, which is included in Vivado's IP library. The IP offers several choices for the concrete implementation and data types. We used the `Pipelined Streaming I/O` architecture with 32-bit floating point as the data type. Floating point operations are not natively supported by FPGA hardware and necessitate additional circuitry. For this purpose, we employed multiple instances of the `Floating-Point Operator IP`, offered by Vivado. These are required, for instance, in computing the element-wise product of two floating point values, according to equation 4.2.

For the CPU baseline implementation, we used the FFTW3 library [27] with single precision floating points.

5. Evaluation

In this chapter, we aim to empirically evaluate the performance characteristics of the running system. We differentiate between end-to-end evaluation and micro-benchmarks. The former will test the system as a whole, i.e. the client submits invocation requests to the OpenFaaS gateway, which will relay them to a compute node in the Kubernetes cluster. The client, the OpenFaaS gateway, and the compute node are located on three separate machines in the same network. Micro-benchmarks will single out certain aspects of our design and test them in isolation. In this case, we skip OpenFaaS, Kubernetes, and the virtualization layer. The unikernel applications, which would usually run inside a virtual machine, are started as ordinary processes on the host operating system. In lieu of network requests, data is passed over `stdin` and `stdout`.

5.1. Experimental setup

The compute node is a server equipped with an Intel Xeon Gold 6238R CPU and 256 GiB DRAM. The operating system is Ubuntu 20.04 with Linux kernel version 5.4.0. The machine is connected to a Xilinx Alveo U50 FPGA card, connected through a PCIe gen3 x16 interface. Unless otherwise noted, the experiments are conducted on a Coyote build with two virtual regions and with a clock frequency of 250 MHz (cf. listing 5.1). Vivado v2022.1 (64-bit) is used for the synthesis and implementation of FPGA code. To assess the performance across a range of workloads, the benchmarks from section 4 are used. Table 5.1 shows the resource usage of the Coyote shell and the benchmark applications. These numbers may vary depending on the Vivado version used, as well as the optimization strategies used for synthesis and implementation in Vivado. CPU code is compiled with the GNU compiler toolchain version 9.4.0. Optimization is enabled at the compiler level (listing 5.2). It is important to note that the CPU implementations are single-threaded. A fair comparison between the execution times of FPGA implementations and the CPU baseline may require further consideration.

```
cmake ../hw/ -DFDEV_NAME=u50 -DEXAMPLE=caribou -DN_REGIONS=2 \
  -DN_CONFIG=12 -DUCLK_F=250 -DACLK_F=250 -DCOMP_CORES=24
```

Listing 5.1: Coyote compilation flags

```
g++ -std=c++17 -O3 -march=native <source files> <libraries>
```

Listing 5.2: CPU compilation flags for C/C++

Table 5.1.: FPGA resource usage for different components and benchmark applications. Applications resource usage is per region.

Entity	**LUT**	**FF (Registers)**	**BRAM**	**URAM**
U50 capacity	872,000	1,743,000	1,344	640
Shell (static)	114,590	179,663	158	0
PCIe DMA	59,034	58,153	80	0
MMU	5,995	9,821	28	0
ADDMUL	2,207	3,849	0	0
AES	43,448	9,777	0	0
SHA256	2,459	2,883	0	0
SHA3	4,081	5,624	0	0
GZIP	44,134	35,630	42	6
MATMUL	12,562	26,048	191	0
NW	138,661	80,702	79	0
HLS4ML	72,770	31,987	20	0
HLL	16,979	23,781	48	0
HCD	23,820	23,192	33	0
MD5	8,242	8,412	0	0
FFT	15,866	29,273	354	8

5.2. Datasets

The majority of benchmarks have variable-sized inputs. In order to obtain a more representative evaluation, the applications were tested with three distinct datasets, classified as: small (S), medium (M), and large (L). `Corner` and `matmul64` have fixed-sized inputs, thus a single dataset was used for these two applications. The data sets are described in table 5.2.

Table 5.2.: Data sets used to benchmark applications. For each application, that accepts variably-sized input, three data sets of different sizes (small, medium, large) were prepared. `Corner` and `matmul64` accept only fixed-sized inputs.

Application	**Data set characterization**	**S**	**M**	**L**
aes128ecb, sha256, sha3	Random bytes	256 KiB	4 MiB	64 MiB
addmul, hyperloglog	32-bit integers in text representation	10^4 ints	10^5 ints	10^6 ints
nw	Number of words per array	16 words	64 words	256 words
hls4ml	Number of samples, each 32×32 pixels, three 8-bit color channels	10 samples	100 samples	10^3 samples
matmul64	Two 64×64 matrices of 64-bit integers in text representation	49 KB		
gzip	Uncompressed files from data compression corpus [5] [17]	alice29 152 KB	dickens 10 MB	mozilla 51 MB
corner	JPEG encoded HD image (1920×1080 pixels), 8-bit grayscale	39 KB		
md5x	Number of iterations until match is found	64^2	64^3	64^4
fft	Number of frames, each 32.768 samples of 32-bit floating points, binary encoded	1 frame	16 frames	256 frames

5.3. End-to-end network latency

Ideally, we would like to benchmark our system end-to-end and compare it to the CPU-only baseline. End-to-end here is characterized from the perspective of a client and considers the duration from starting the HTTP request against the OpenFaaS gateway until the complete response has arrived. However, we have experienced instabilities and extreme network latencies in the infrastructure, that was provided to us, cf. table 5.3. This is not a flaw in the system design, but rather a curable issue related to the unikernel framework or its integration in the Kubernetes pod network. To circumvent this issue, we have decided to only measure computation time for the remainder of the evaluation section. The computation time excludes network transfer overheads and measures only the duration from the moment the entire input data has been received until the moment the principal algorithm has computed the result for the input data. In the case of FPGA execution, this includes the communication overhead, in particular, the DMA to stream input and output data, between the unikernel application and the vFPGA.

Table 5.3.: Network latencies for transferring 64 KiB of data from a client to the application and back to the client

Container		Unikernel	
Mean	Stddev	Mean	Stddev
10.7 ms	1.1	11262 ms	3834

5.4. Computation time

Figure 5.1 shows the computation time, that we measured for our benchmark applications. We observe that the majority of applications execute faster on FPGA, with notable speed-up factors (table 5.4). In particular, the FPGA speed-ups for `nw`, `hls4ml`, and `md5x` are tremendous. The reason is that the FPGA user logic modules for these applications are implemented efficiently and leverage the pipelining principle, thus running many computations in parallel. The input format for `addmul`, `hyperloglog` and `matmul64` are integer values in text representation. Parsing the input values from human-readable into binary encoding makes up a major proportion of the computation time. In both execution environments, the conversion happens on the CPU. The algorithms themselves (integer arithmetic, hyperloglog insertion, or matrix multiplication, respectively) are relatively lightweight. Both the CPU and FPGA can compute the results quickly, but the FPGA execution incurs additional memory transfers. In total, the CPU execution is almost ten times faster for these three applications.

In the case of `aes128ecb` and `sha256` it is noticeable, that the CPU applications scale better than their FPGA counterparts. For instance, AES with the small data set exhibits faster execution on FPGA, whereas the large data set is faster on CPU. Upon further investigating into the reasons behind the observed slowdown for larger input sizes, we hypothesize that this phenomenon is related to memory transfers. In particular, Coyote maintains a translation lookaside buffer (TLB), which contains physical addresses for the memory pages that constitute the input and output buffers of the invocation. The larger the input and output buffers are, the more memory pages are required to store the data. Since the TLB in Coyote is relatively small, not all page addresses can fit in the TLB at once. During an invocation, every TLB miss stalls the FPGA execution. The host kernel driver receives an interrupt, adds new addresses to the TLB, and only then resumes FPGA execution. This interrupt handling can result in significant delays to the total execution time.

Table 5.4.: Speed-up factors for FPGA execution, as opposed to CPU execution. A number greater than one indicates FPGA computation time is reduced by that factor.

	sha3	**aes128ecb**	**addmul**	**sha256**	**nw**	**hls4ml**	**matmul64**	**hyperloglog**	**corner**	**md5x**	**fft**	**gzip**
S	4.5	3.7	0.12	2.4	6.9	91		0.12		11	1.3	4.2
M	2.9	0.76	0.13	1.2	6.7	29	0.11	0.1	4.6	140	1.4	7.8
L	4.0	0.6	0.12	1.0	6.9	7.2		0.11		220	1.1	8.0

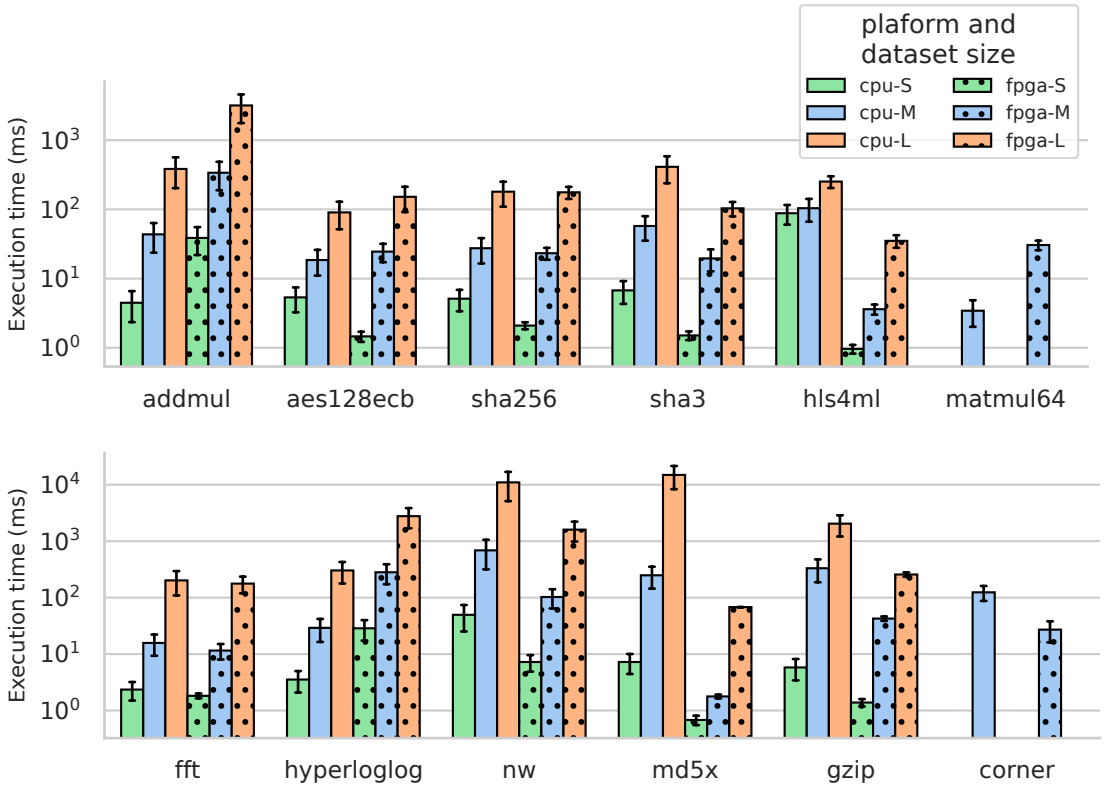


Figure 5.1.: End-to-end comparison of CPU vs. FPGA execution for our benchmark applications, according to their computation time (as defined in 5.3). Most applications accept variably-sized input and were measured with data sets of three sizes: small (S), medium (M), large (L), as defined in 5.2. Note the log-scale.

5.5. Huge pages

To assess the impact of using huge pages, our framework was modified to allocate memory buffers facilitating huge pages instead of regular pages. These memory pages are considerably larger (2 MiB) than regular pages (4 KiB), thus fewer TLB entries are required and TLB misses become less frequent. Listing 5.3 shows the Linux system calls for allocating regular pages or huge pages so that the memory buffer can be shared with the vFPGA. Unfortunately, the `mmap` call to allocate huge pages is not supported by our unikernel environment. Therefore, we executed all host applications as regular, non-virtualized processes for this experiment.

```
// Allocate regular pages
const size_t ALIGNMENT = 64;
bytes = roundup(bytes, ALIGNMENT);
void *ptr = aligned_alloc(ALIGNMENT, bytes);

// Allocate huge pages
const size_t ALIGNMENT = 2*1024*1024;
bytes = roundup(bytes, ALIGNMENT);
void *ptr = mmap(NULL, bytes, PROT_READ | PROT_WRITE,
                 MAP_PRIVATE | MAP_ANONYMOUS | MAP_HUGETLB, -1, 0);
```

Listing 5.3: Memory buffer allocation

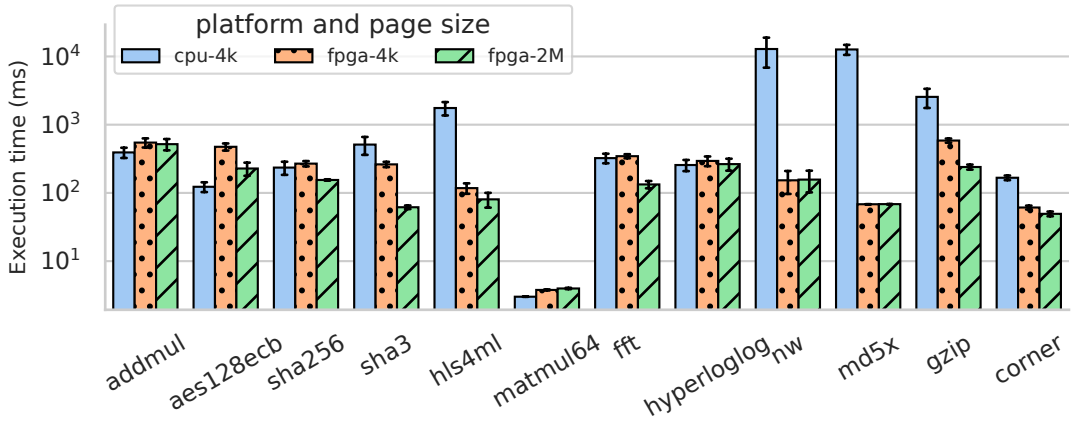


Figure 5.2.: Microbenchmark: Comparison of CPU and FPGA execution using regular pages (4k) and huge pages (2M). The large data set is used for all executions. Note the log-scale.

Table 5.5.: Speed-up factors for FPGA execution with huge pages, when compared with CPU and regular pages (first row) or FPGA execution with regular pages (second row).

baseline	sha3	aes128ecb	addmul	sha256	nw	hls4ml	matmul64	gzip	hyperloglog	corner	md5x	fft
cpu-4k	8.3	0.54	0.76	1.5	82	22	0.76	11	0.97	3.4	180	2.4
fpga-4k	4.2	2.1	1.1	1.7	0.98	1.5	0.95	2.4	1.1	1.2	1.0	2.6

5.6. Parallelism

The vFPGAs operate independently of each other and permit parallel execution on the hardware side. The number of virtual regions is not fixed in Coyote and can be set at design time. The number of regions determines the number of applications that can run in parallel on the FPGA. However, if the regions are too small, complex applications may not fit inside. The available space (or more precisely, the available resources) differs from one FPGA model to another. In our experiment, we used a Xilinx Alveo U50 FPGA card. With one or two equisized regions, we were able to accommodate all benchmark applications on the device. With four regions, some of the more complex benchmark applications were already too resource-intensive.

Given a list of parallelizable tasks, we expect the total duration to complete all tasks to scale approximately proportionally inverse to the number of virtual regions. To test this hypothesis, an experiment was conducted with one, two, and four available regions on the FPGA. Thirty clients concurrently submitted a task to execute `sha256` on 8 MiB of random input data. For each task, the time from submission to completion was measured, thus including time spent in the scheduler queue.

The results are presented in Figure 5.3 and Figure 5.4. A twofold and fourfold acceleration in speed is observed when two and four vFPGAs are utilized, respectively, compared to a single vFPGA. In the case of multiple vFPGAs, groups of two or four tasks, respectively, were finishing at nearly the same instant, indicating, that these tasks were in fact executed in parallel. The execution time per task is identical in all scenarios, the speed-up emanates solely from parallel execution. These results demonstrate the effectiveness of parallelization and the linear correlation between speed-up and the number of vFPGAs, assuming a set of parallelizable tasks.

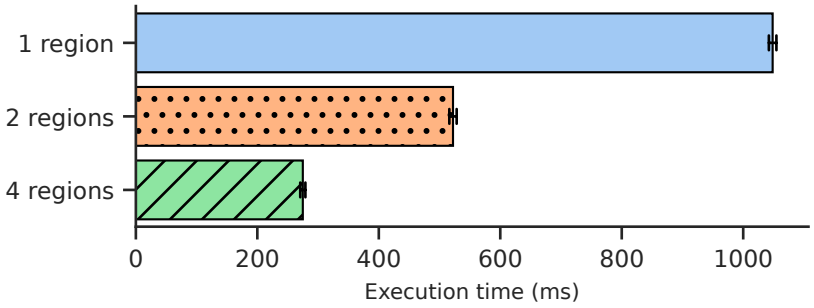


Figure 5.3.: Microbenchmark: Time to completion of 30 invocation requests, submitted at the same time.

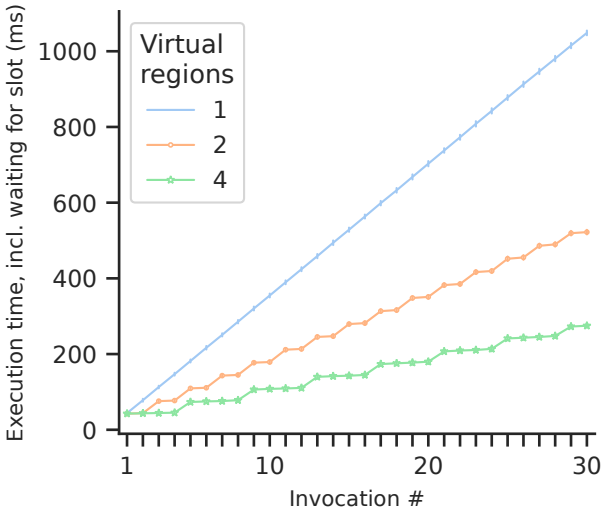


Figure 5.4.: Time to completion of 30 invocation requests, submitted at the same time. Requests are sorted by completion time on the x-axis.

5.7. Dynamic partial reconfiguration

Prior to invoking user logic on the designated vFPGA, it must be configured via dynamic partial reconfiguration (see chapter 2.3). Reconfiguration can be omitted if the vFPGA is already configured with the correct bitstream from the previous invocation. In this experiment, we seek to analyze the overhead, incurred by reconfiguring the vFPGA. We prepared four builds of the Coyote shell, each with a single dynamic region of a different size. The dynamic region spanned either 8, 16, 24, or 32 clock regions. The `sha256` benchmark was executed with 4 MiB of random data, either with a forced reconfiguration ahead of the invocation or by reusing the bitstream from the previous invocation. The results demonstrate that the bitstream size is dependent on the size of the dynamic region (see table 5.6). Furthermore, the overhead incurred by dynamic partial reconfiguration is proportional to the bitstream size (fig. 5.5).

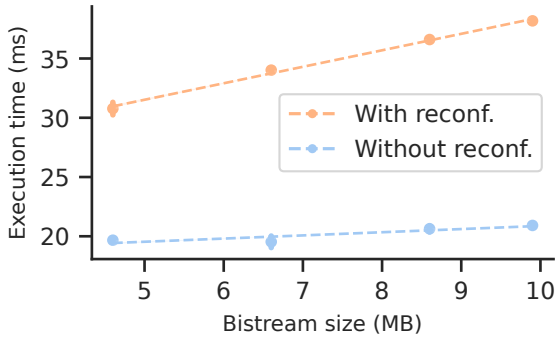


Figure 5.5.: Microbenchmark: Reconfiguration overhead.

Table 5.6.: Partial bitstream file size of `sha256` for four different dynamic region sizes. The Alveo U50 FPGA has 64 clock regions in total.

Clock regions	8	16	24	32
Bitstream size	4.6 MB	6.6 MB	8.6 MB	9.9 MB

6. Related work

Our approach is founded on existing software tools, libraries, and frameworks from a multitude of domains. Most of the building blocks have alternative approaches and implementations.

6.1. Serverless Functions Frameworks

Commercial cloud providers offer their customers easy to set up solutions for running serverless functions inside their infrastructure; however, these are proprietary systems, not available for self-hosting. AWS Lambda [74], being the most prominent platform, prices invocations primarily on runtime duration, allotted memory, and number of executions. Fixed limits are imposed on the maximum runtime and deployment package size. Microsoft's Azure functions [6] and Google cloud functions [13] are competitors with a similar feature set. All three platforms come with monitoring facilities built-in. For on-premise deployment, there are free, open-source alternatives, such as OpenWhisk [3], OpenLambda [34] and OpenFaaS [56]. While OpenFaaS is built upon Kubernetes and utilizes its scheduler for distributing workload, OpenWhisk utilizes Apache Kafka for forwarding requests to the compute nodes. OpenLambda integrates the nginx load balancer instead. While the former two are production-grade products, OpenLambda is more experimental and geared towards the exploration of new concepts in serverless computing.

All of these systems can be extended to wrap function invocation inside lightweight VMs and unikernels. As such, our approach is not limited to the use of OpenFaaS for function management.

6.2. Heterogeneous computing

With the recent surge of machine learning workloads, there is great demand for accelerators in cloud deployments. Molecule [20] provides a general abstraction for serverless computing on heterogeneous devices, such as FPGAs and DPUs. Programs are confined to sandboxed containers on accelerator units. The containers are interfaced with a distributed shim, which bridges the gap between platforms by managing the

global state and providing inter-process communication across heterogeneous devices. The authors also review and implement optimizations towards better communication latency and start-up times in their serverless runtime. Kernel-as-a-Service [68] demonstrates an alternative design for a broader variety of heterogeneous hardware. Developers can register device-specific, optimized code with their server. The server then exposes an invocation endpoint for the function over the network, upon which the device-specific code is deployed and executed on the accelerator. BlastFunction [7] shares FPGA devices among multiple users. Their OpenCL-compatible [79] interface enables transparent access to the shared accelerators. Like our design, container orchestration is based on Kubernetes [69] and OpenFaaS [56]. Another component manages time-sharing and monitoring of accelerator hardware. Funky [50] is another design for cloud-native FPGA deployment and is built on a similar technology stack as our design. FPGAs are virtualized, multi-tenant resources, implementing the OpenCL programming model. It facilitates cooperative, preemptive scheduling and task migration for improved resource management. TornadoVM [28] is a parallel API and JIT-compiler for Java bytecode. It can transform and optimize the code for the target platform and generates FPGA- and GPU-compatible programs using Intel's OpenCL SDK. In their co-processor design, a memory manager component organizes on-device memory and keeps data consistent between the different platforms.

Some designs attempt to abstract away heterogeneity with unified programming models, such as OpenCL. Other designs, including ours, opt for native languages to retain the complete performance potential.

6.3. FPGA multi-tenancy

Many designs allow multi-tenancy on FPGAs through time-multiplexing. Some approaches harness dynamic partial reconfiguration, to achieve spatial multiplexing and add multi-tenancy on another axis. Our approach leverages Coyote's [49] ability to subdivide the chip area into isolated regions of fixed size; number and size of these regions can be configured in the floorplan design. The uniform execution model provides access to main memory from host processes and FPGA user logic through the same address space. AmorphOS [46] packages user logic in dynamically sized *morphlets*, which alter their form to adjust the utilized chip area based on overall workload. It operates in one of two modes: In low-latency mode, applications are programmed onto separated bounding boxes inside the reconfigurable region. In high-throughput mode one big bitstream, made up of all active user logic modules, is programmed on the entire dynamic region, maxing out all available resources to achieve the highest, overall performance. Optimus [57] supports parallel task execution through spatial multiplex-

ing. Each user logic module can execute multiple, isolated tasks concurrently. Tasks are allotted time slices and context switches are managed through a preemptive interface, which stores and loads execution states to/from host memory. However, Optimus does not make use of dynamic partial reconfiguration, thus swapping user logic modules is not possible during runtime. The toolchain proposed by ViTAL [92] divides (clusters of) FPGAs into smaller virtual blocks. User logic is dynamically assigned and programmed onto one or multiple blocks, obviating the need for bitstream recompilation. The merit of using small blocks is the reduction in internal fragmentation of virtual regions, i.e. unused chip area that accrues by placing arbitrary-sized user logic in fixed-sized dynamic regions. As a single user logic can be divided across smaller blocks, and even across multiple FPGAs, communication channels are established between blocks. These interconnects, as well as the partitioning scheme, are automatically synthesized by the ViTAL toolchain.

Each of the aforementioned designs is focused on specific aspects of multi-tenancy and aims to alleviate certain pain points. Our work is based on Coyote, which offers a simple (in comparison), yet solid design, which is well-suited for serverless functions.

6.4. FPGA scheduling

We target FPGAs utilized in a co-processor system design. In a scenario with multiple nodes, resource availability and utilization are essential information for the orchestrator to make informed scheduling decisions. The Kubernetes extension in our design schedules serverless functions on an accelerator-equipped host node based on a scheduling policy that employs FPGA utilization as a key metric. The proposal by Chen et al. [12] assigns tenants to accelerated compute nodes, by establishing FPGAs as resource entities for the OpenStack scheduler. Dai et al. [15] builds on that work and implements a scheduler strategy where CPU-versus-FPGA speedup is the key metric and oversubscribed tasks are migrated to the CPU. In the design of Farhan et al. [23] execution time and resource usage of each task is assumed to be known a priori, allowing them to model the scheduling process as an optimization problem. The solution to this optimization problem is then used in their custom scheduler/resource manager to distribute tasks to FPGAs. Their evaluation analyzes the strategy under different scheduling objectives.

The aforementioned approaches are similar to ours. Unlike them, however, we specifically target Kubernetes and OpenFaaS in our work, and to the best of our knowledge no published work exists, that fits our context.

6.5. Unikernel

Unikernels are all-in-one binaries optimized towards minimal resource overhead. Our approach incorporates includeOS [11], a C++ library with low disk and memory footprint. Network I/O is established over a Virtio network driver. TCP/IP and other fundamental, low-level network protocols are supported by the inbuilt network stack. However, further development of the library seems to have been abandoned. MirageOS [58] builds unikernels from Ocaml sources. Unikraft [51] accommodates a greater variety of programming languages, including C++, Rust, and Python. Compatibility is one of the goals, and indeed complex applications, such as nginx and Redis, can be compiled to run in an unikernel built with Unikraft. Rumprun's [31] approach is stripping down the NetBSD kernel while maintaining device driver compatibility. It can be built with a POSIX interface, allowing many mainstream applications to run out-of-the-box.

We opted for includeOS, as it presents a simple and performant solution, which is easy to adopt for our use case. Compatibility with a wide variety of programming languages or existing applications is not a requirement in our setting.

6.6. Lightweight VMs

Our design runs unikernel applications on top of solo5 [32] and the Linux KVM infrastructure. Firecracker [1] is a production-grade virtualization technology optimized and geared towards workloads in AWS' serverless offerings. Firecracker implements a minimal device model, stripping away as much low-level functionality as possible in order to focus solely on serverless workloads. Compatibility is maintained through the Linux kernel, which is booted inside the VM. Yet, this kernel is freed of needless drivers and devoid of any kernel modules. A pool of pre-booted VMs is kept around to further reduce cold boot times. QEMU [8] has traditionally been used to virtualize full operating systems, but is also capable of running unikernels in a minimal configuration with low overhead. As it is a highly flexible solution with a plethora of features, it is not optimized towards serverless workloads. LightVM [59] is a derivative of the Xen hypervisor [36], optimized towards minimal boot times and high instance density. The system is tailored for two types of application images: A custom toolchain is provided to bundle applications with a minified Linux distribution. Alternatively, unikernel applications can harness the full performance potential of the design.

Our focus on performance has led us to choose the simple and lightweight, yet robust, solution that solo5 presents. The broader compatibility that other approaches offer is less of a concern in our context.

7. Summary

In this thesis, we have presented a system design for an FPGA-accelerated cloud architecture. The deployment follows the function-as-a-service (FaaS) model, which relieves developers from the burden of managing the infrastructure components. The design is founded on an FPGA runtime which supports dynamic partial reconfiguration. This allows reconfiguring parts of the FPGA during runtime, to the end that the accelerated function, programmed on the FPGA, can be swapped out at any time. In contrast, a system without dynamic partial reconfiguration can only support a static, and strictly limited set of FPGA-accelerated functions, or requires a full device reconfiguration and reset, whenever the accelerated functions need to be replaced. To allow limitless scaling, we built on top of Kubernetes and OpenFaaS, so that multiple FPGA-equipped host machines can form a larger compute cluster. Each serverless function is associated with a small program on the host machine, which is responsible for relaying input and output data between the FaaS infrastructure and the accelerated function on the FPGA. For performance and security reasons, these programs are developed as unikernel applications and are confined to a lightweight virtual machine. We have extended the solo5 hypervisor with our API library to establish the necessary communication channel between the unikernel and the FPGA code. In order to prove the feasibility of our design, we have ported a set of benchmark applications for a variety of workloads, including encryption, cryptographic hashing, machine learning, image processing, and digital signal processing. The system allows applications to be written in low-level languages, so they can exploit the full performance potential that FPGAs have to offer. Finally, we have measured the performance of the proposed setup on a real cluster. For the majority of workloads, significant performance gains are exhibited by our FPGA-accelerated architecture, when compared to a CPU-only baseline.

In conclusion, we have demonstrated the feasibility and the excellent performance characteristics of an FPGA-accelerated FaaS architecture, which utilizes dynamic partial reconfiguration. The source code of our project is publicly available at [30].

8. Future work

The system developed in this work has the quality of a proof-of-concept. While it has demonstrated the feasibility of the proposed approach, there are several minor software-related issues that prevent it from being considered a production-grade solution. Apart from these issues, there are (at least) two major topics that warrant further research efforts. The first is function chaining, which is the concept of sequential execution of two or more user logic modules while passing the results from one function to the next. The second unsolved problem pertains to the compilation, distribution, and packaging of bitstream files at a cloud-scale level.

8.1. Optimized function chaining

In many scenarios, a well-designed serverless function can be considered to implement a specific algorithm as a transformation of the input data. Then, a workflow can be designed by consecutive execution of serverless functions, where the output of one function serves as the input of the successive function. For example, the face detection workflow described in [39] is composed of a total of eight steps, such as color correction, image segmentation, and face feature detection. A reasonable implementation of this algorithm models each step into an independent serverless function. Then, each component can be evaluated independently or even swapped out for an alternative implementation. However, such a design may be susceptible to extreme overheads, if the functions are placed far apart and data transfer between steps incurs large performance penalties. One notable example is from a blog post [48] regarding Amazon's Video on Demand service. The developers outline one of their pipelines, which detects corrupted frames in a video stream. Their original implementation was based on a chain of serverless functions using Amazon Lambda and Amazon Step functions. When the developers migrated away from a series of serverless functions to a single monolith, they were able to reduce infrastructure costs by over 90% and improve scaling capabilities. This example demonstrates the risk of a distributed system design that is not optimized for function chaining. Our current design suffers from similar imperfections. To illustrate, consider a scenario with OpenFaaS where two serverless functions are executed sequentially, and the second function uses the output of the first function. In this case, the intermediate result has to be transferred between the

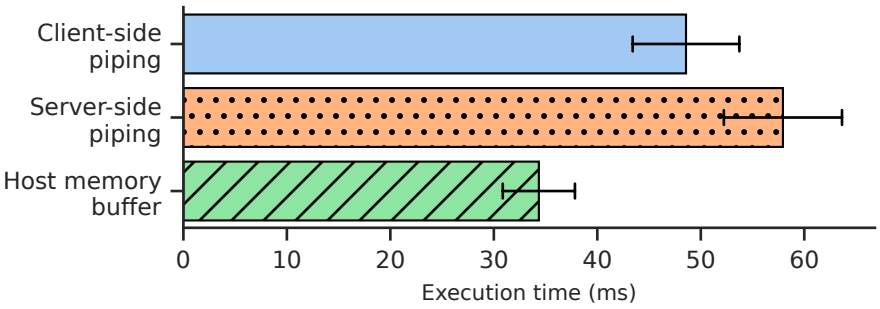


Figure 8.1.: End-to-end benchmark of chaining GZIP and AES. The input size is 150 KB before, and 70 KB after compression.

serverless functions. The two functions may even be located on different hosts, in which case a network transfer is inevitable. Even if both invocations happen to execute on the same host machine, our current design will cause an unnecessary network transfer of the intermediate result, since transmitting the data from the first function to the second function necessitates an invocation via the OpenFaaS gateway. An optimized system will avoid this network roundtrip, whenever possible. Possible approaches include buffering the intermediate result in host memory or the FPGA on-board memory. An even more optimal latency performance is attainable by pipelining the execution of the two user logic modules on the FPGA. This approach, however, requires both functions to be deployed at the same time and necessitates the existence of an interconnect between the modules over which the intermediate result is transmitted. Figure 8.1 illustrates the preliminary results of a comparison between function chaining as proposed by OpenFaaS [66] versus buffering in host memory. The latter avoids all unnecessary network roundtrips, resulting in a significantly lower latency.

8.2. Bitstream management

Our current approach supports only a static set of partial bitstreams. All user logic modules must be compiled in conjunction with the shell in a single run, and the produced partial bitstreams are only compatible with the full bitstream from the same build. Retrospectively adding user logic modules is not possible out-of-the-box. This is problematic for numerous reasons. Upgrading the user logic is not possible without recompiling the static shell and *all* other user logic modules. The same applies for any modifications (such as security bugfixes) to the static region. For a real-world scenario, this renders bitstream management quite challenging and makes a large-

scale deployment infeasible. One potential solution is to facilitate the intermediate build artifacts, which are generated by Vivado during the compilation of the static region. These build artifacts can be utilized in future builds to establish compatibility between the new partial bitstream and the original, full bitstream. A Vivado TCL script to accomplish this task can be found in appendix listing A.1. A better option may be to use the RapidWright [90] toolkit from Xilinx. It provides the means for programmatic, low-level access to bitstream files and other build artifacts. This could open the possibility of distributing a single artifact of each user logic module and have an automated pipeline in the cloud infrastructure transform the build artifact into a partial bitstream that is compatible with the currently deployed FPGA shell.

A. Appendix

```
# Open checkpoint from older build
open_checkpoint /home/user/Coyote/build1/cyt_top_routed_bb.dcp
lock_design -level routing

# Read checkpoint from new build into dynamic regions
set_property HD.RECONFIGURABLE true [get_cells inst_dynamic/
    inst_user_wrapper_0]
set_property HD.RECONFIGURABLE true [get_cells inst_dynamic/
    inst_user_wrapper_1]
read_checkpoint -cell inst_dynamic/inst_user_wrapper_0 /home/user/Coyote/
    build2/lynx/lynx.runs/design_user_wrapper_c5_0_synth_1/
    design_user_wrapper_0.dcp
read_checkpoint -cell inst_dynamic/inst_user_wrapper_1 /home/user/Coyote/
    build2/lynx/lynx.runs/design_user_wrapper_c5_1_synth_1/
    design_user_wrapper_1.dcp

# Implement design
opt_design
place_design
phys_opt_design
route_design

# Write partial bitstreams
write_bitstream -cell inst_dynamic/inst_user_wrapper_0 -force -bin_file /
    home/user/Coyote/build2/part_bstream_0
write_bitstream -cell inst_dynamic/inst_user_wrapper_1 -force -bin_file /
    home/user/Coyote/build2/part_bstream_1
```

Listing A.1: Vivado TCL script to establish compatibility of partial bitstreams with a full bitstream from a previous build

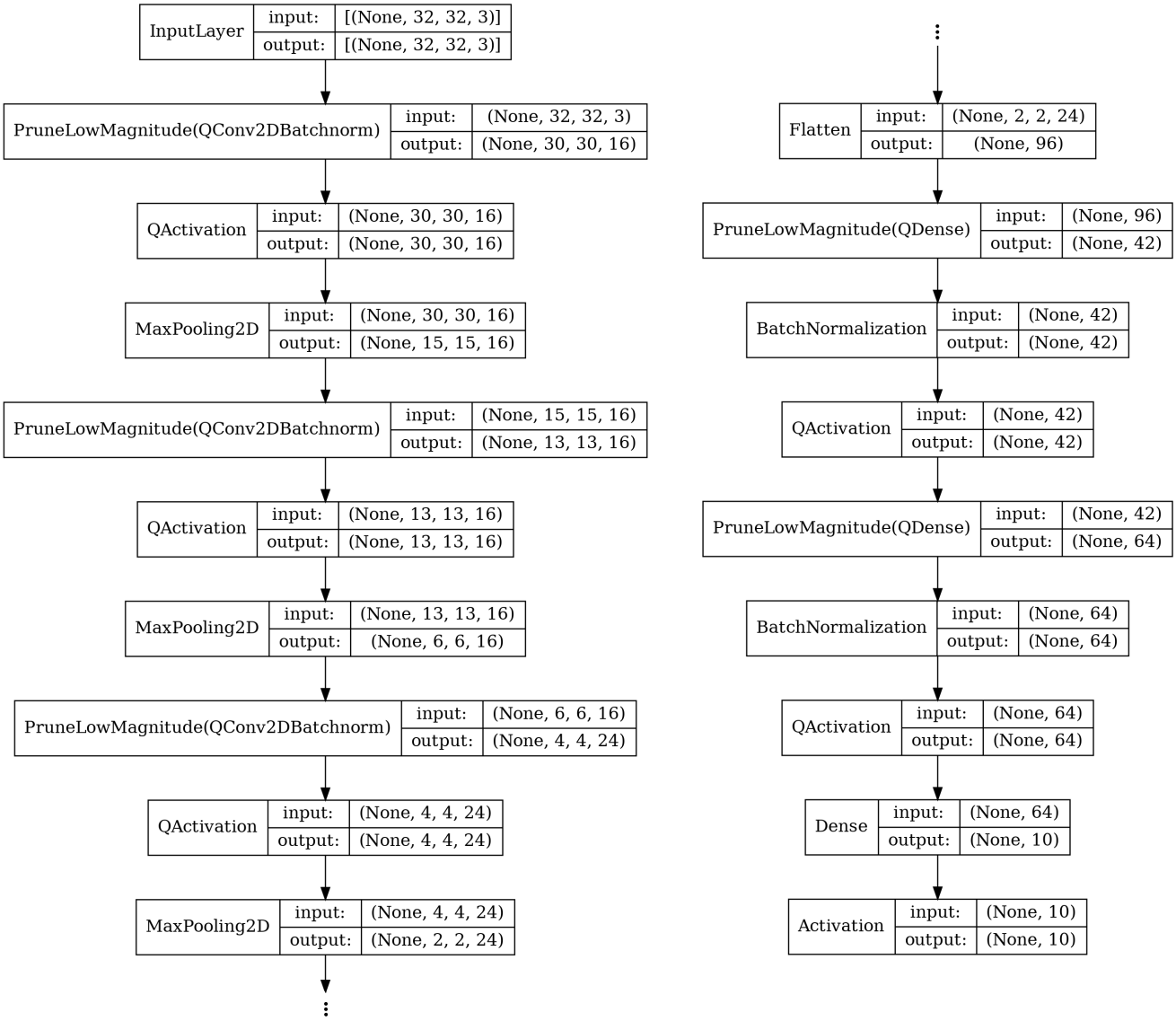


Figure A.1.: The layers of the convolutional neural network, used in the `hls4ml` benchmark. The input is a batch of images with dimensions of 32×32 pixels and three color channels. For each image in the input, there is an output vector with ten elements: the probability scores for each of the digits between zero and nine. The digit with the highest score is the one predicted by the neural network.

List of Figures

List of Tables

Listings

Bibliography

[1] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa. "Firecracker: Lightweight Virtualization for Serverless Applications." In: *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 419–434. ISBN: 978-1-939133-13-7.

[2] AMD. *AMD Technical Information Portal — docs.amd.com*. `https://docs.amd.com/r/en-US/ug909-vivado-partial-reconfiguration`. [Accessed 13-06-2024].

[3] *Apache OpenWhisk is a serverless, open source cloud platform — openwhisk.apache.org*. `https://openwhisk.apache.org/`. [Accessed 26-02-2024].

[4] A. Appleby. *smhasher/src/MurmurHash3.cpp at master · aappleby/smhasher — github.com*. `https://github.com/aappleby/smhasher/blob/master/src/MurmurHash3.cpp`. [Accessed 26-06-2024].

[5] R. Arnold and T. Bell. "A Corpus for the Evaluation of Lossless Compression Algorithms." In: DCC '97. USA: IEEE Computer Society, 1997, p. 201. ISBN: 0818677619.

[6] *Azure Functions – Serverless Functions in Computing | Microsoft Azure — azure.microsoft.com*. `https://azure.microsoft.com/en-us/products/functions/`. [Accessed 26-02-2024].

[7] M. Bacis, R. Brondolin, and M. D. Santambrogio. "BlastFunction: an FPGA-as-a-Service system for Accelerated Serverless Computing." In: *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2020, pp. 852–857. DOI: 10.23919/DATE48585.2020.9116333.

[8] F. Bellard. "QEMU, a fast and portable dynamic translator." In: *USENIX annual technical conference, FREENIX Track*. Vol. 41. 46. California, USA. 2005, pp. 10–5555.

[9] B. Betkaoui, D. B. Thomas, and W. Luk. "Comparing performance and energy efficiency of FPGAs and GPUs for high productivity computing." In: *2010 International Conference on Field-Programmable Technology*. 2010, pp. 94–101. DOI: 10.1109/FPT.2010.5681761.

[10] G. Bradski. "The OpenCV Library." In: *Dr. Dobb's Journal of Software Tools* (2000).

[11] 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.

[12] F. Chen, Y. Shan, Y. Zhang, Y. Wang, H. Franke, X. Chang, and K. Wang. "Enabling FPGAs in the cloud." In: *Proceedings of the 11th ACM Conference on Computing Frontiers*. CF '14. Cagliari, Italy: Association for Computing Machinery, 2014. ISBN: 9781450328708. DOI: 10.1145/2597917.2597929.

[13] *Cloud Functions | Google Cloud — cloud.google.com*. https://cloud.google.com/functions/. [Accessed 25-03-2024].

[14] C. N. Coelho, A. Kuusela, S. Li, H. Zhuang, J. Ngadiuba, T. K. Aarrestad, V. Loncar, M. Pierini, A. A. Pol, and S. Summers. "Automatic heterogeneous quantization of deep neural networks for low-latency inference on the edge for particle detectors." In: *Nature Machine Intelligence* 3.8 (June 2021), pp. 675–686. ISSN: 2522-5839. DOI: 10.1038/s42256-021-00356-5.

[15] G. Dai, Y. Shan, F. Chen, Y. Wang, K. Wang, and H. Yang. "Online scheduling for FPGA computation in the Cloud." In: *2014 International Conference on Field-Programmable Technology (FPT)*. 2014, pp. 330–333. DOI: 10.1109/FPT.2014.7082811.

[16] Q. H. Dang. *Secure Hash Standard*. July 2015. DOI: 10.6028/nist.fips.180-4.

[17] S. Deorowicz. *Silesia Corpus*. https://sun.aei.polsl.pl/~sdeor/index.php?page=silesia. [Accessed 23-06-2024].

[18] L. P. Deutsch. *DEFLATE Compressed Data Format Specification version 1.3*. RFC 1951. http://www.rfc-editor.org/rfc/rfc1951.txt. RFC Editor, May 1996.

[19] L. P. Deutsch, J.-L. Gailly, M. Adler, L. P. Deutsch, and G. Randers-Pehrson. *GZIP file format specification version 4.3*. RFC 1952. http://www.rfc-editor.org/rfc/rfc1952.txt. RFC Editor, May 1996.

[20] D. Du, Q. Liu, X. Jiang, Y. Xia, B. Zang, and H. Chen. "Serverless computing on heterogeneous computers." In: *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '22. Lausanne, Switzerland: Association for Computing Machinery, 2022, pp. 797–813. ISBN: 9781450392051. DOI: 10.1145/3503222.3507732.

[21] M. J. Dworkin. *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*. July 2015. DOI: 10.6028/nist.fips.202.

[22] D. Eastlake and T. Hansen. *US Secure Hash Algorithms (SHA and HMAC-SHA)*. RFC 4634. http://www.rfc-editor.org/rfc/rfc4634.txt. RFC Editor, July 2006.

[23] A. Farhan, R. Aburukba, A. Sagahyroon, M. Elnawawy, and K. El-Fakih. "Virtualizing and Scheduling FPGA Resources in Cloud Computing Datacenters." In: *IEEE Access* 10 (2022), pp. 96909–96929. DOI: 10.1109/ACCESS.2022.3204866.

[24] FastML Team. *fastmachinelearning/hls4ml*. Version v0.8.1. 2023. DOI: 10.5281/zenodo.1201549.

[25] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier. "HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm." In: *Discrete Mathematics & Theoretical Computer Science* (2007), pp. 137–156.

[26] I. Free Software Foundation. *Gzip - GNU Project - Free Software Foundation — gnu.org*. https://www.gnu.org/software/gzip/. [Accessed 28-06-2024].

[27] M. Frigo and S. G. Johnson. "The Design and Implementation of FFTW3." In: *Proceedings of the IEEE* 93.2 (2005). Special issue on "Program Generation, Optimization, and Platform Adaptation", pp. 216–231.

[28] J. Fumero, M. Papadimitriou, F. S. Zakkak, M. Xekalaki, J. Clarkson, and C. Kotselidis. "Dynamic application reconfiguration on heterogeneous hardware." In: VEE 2019. Providence, RI, USA: Association for Computing Machinery, 2019, pp. 165–178. ISBN: 9781450360203. DOI: 10.1145/3313808.3313819.

[29] *GitHub - armon/hlld: C network daemon for HyperLogLogs — github.com*. https://github.com/armon/hlld. [Accessed 26-06-2024].

[30] *GitHub - ml-tum/FPGA_serverless — github.com*. https://github.com/ml-tum/FPGA_serverless. [Accessed 01-07-2024].

[31] *GitHub - rumpkernel/rumprun: The Rumprun unikernel and toolchain for various platforms — github.com*. https://github.com/rumpkernel/rumprun/. [Accessed 26-02-2024].

[32] *GitHub - Solo5/solo5: A sandboxed execution environment for unikernels — github.com*. https://github.com/Solo5/solo5. [Accessed 13-06-2024].

[33] C. G. Harris and M. J. Stephens. "A Combined Corner and Edge Detector." In: *Alvey Vision Conference*. 1988.

[34] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. "Serverless computation with openLambda." In: *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing*. HotCloud'16. Denver, CO: USENIX Association, 2016, pp. 33–39.

[35] S. Heule, M. Nunkesser, and A. Hall. "HyperLogLog in Practice: Algorithmic Engineering of a State of The Art Cardinality Estimation Algorithm." In: *Proceedings of the EDBT 2013 Conference*. Genoa, Italy, 2013.

[36] *Home - Xen Project — xenproject.org*. https://xenproject.org/. [Accessed 25-03-2024].

[37] H. Hsing. *Overview :: SHA3 (KECCAK) :: OpenCores — opencores.org*. https://opencores.org/projects/sha3. [Accessed 11-06-2024].

[38] H. Hsing. *SHA3 Core Specification*. https://opencores.org/usercontent/doc/1359445372. [Accessed 14-06-2024].

[39] R.-L. Hsu, M. Abdel-Mottaleb, and A. Jain. "Face detection in color images." In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 24.5 (2002), pp. 696–706. DOI: 10.1109/34.1000242.

[40] O. C. Initiative. *The OpenContainers Distribution Spec — specs.opencontainers.org*. https://specs.opencontainers.org/distribution-spec. [Accessed 29-06-2024].

[41] O. C. Initiative. *The OpenContainers Image Spec — specs.opencontainers.org*. https://specs.opencontainers.org/image-spec/. [Accessed 29-06-2024].

[42] Intel. *Intel® Advanced Encryption Standard Instructions (AES-NI) — intel.com*. https://www.intel.com/content/www/us/en/developer/articles/technical/advanced-encryption-standard-instructions-aes-ni.html. [Accessed 14-06-2024].

[43] Intel. *Intel® SHA Extensions — intel.com*. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sha-extensions.html. [Accessed 25-06-2024].

[44] M. K. Jaiswal and H. K.-H. So. "DSP48E efficient floating point multiplier architectures on FPGA." In: *2017 30th International Conference on VLSI Design and 2017 16th International Conference on Embedded Systems (VLSID)*. 2017, pp. 1–6. DOI: 10.1109/ICVD.2017.7913322.

[45] B. Kaliski. *PKCS #7: Cryptographic Message Syntax Version 1.5*. RFC 2315. Mar. 1998. DOI: 10.17487/RFC2315.

[46] A. Khawaja, J. Landgraf, R. Prakash, M. Wei, E. Schkufza, and C. J. Rossbach. "Sharing, Protection, and Compatibility for Reconfigurable Fabric with AmorphOS." In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 107–127. ISBN: 978-1-939133-08-3.

[47] V. Klima. *Tunnels in Hash Functions: MD5 Collisions Within a Minute*. Cryptology ePrint Archive, Paper 2006/105. https://eprint.iacr.org/2006/105. 2006.

[48] M. Kolny. *Scaling up the Prime Video audio, video monitoring service and reducing costs by 90% — primevideotech.com*. https://www.primevideotech.com/video-streaming/scaling-up-the-prime-video-audio-video-monitoring-service-and-reducing-costs-by-90. [Accessed 28-06-2024].

[49] D. Korolija, T. Roscoe, and G. Alonso. "Do OS abstractions make sense on FPGAs?" In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 2020, pp. 991–1010.

[50] A. Koshiba, C. Mainas, and P. Bhatotia. "Cloud-Native FPGA Virtualization and Orchestration." [Submitted for publication].

[51] S. Kuenzer, V.-A. Bădoiu, H. Lefeuvre, S. Santhanam, A. Jung, G. Gain, C. Soldani, C. Lupu, Ş. Teodorescu, C. Răducanu, C. Banu, L. Mathy, R. Deaconescu, C. Raiciu, and F. Huici. "Unikraft: fast, specialized unikernels the easy way." In: *Proceedings of the Sixteenth European Conference on Computer Systems*. EuroSys '21. Online Event, United Kingdom: Association for Computing Machinery, 2021, pp. 376–394. ISBN: 9781450383349. DOI: 10.1145/3447786.3456248.

[52] J. Landgraf, M. Giordano, E. Yoon, and C. J. Rossbach. "Reconfigurable Virtual Memory for FPGA-Driven I/O." In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. ASPLOS 2023. <conf-loc>, <city>Vancouver</city>, <state>BC</state>, <country>Canada</country>, </conf-loc>: Association for Computing Machinery, 2023, pp. 556–571. ISBN: 9781450399180. DOI: 10.1145/3582016.3582048.

[53] J. Leitch. *Overview :: MD5 Pipelined :: OpenCores — opencores.org*. https://opencores.org/projects/md5_pipelined. [Accessed 11-06-2024].

[54] W. Lie and W. Feng-Yan. "Dynamic partial reconfiguration in FPGAs." In: *2009 Third International Symposium on Intelligent Information Technology Application*. Vol. 2. IEEE. 2009, pp. 445–448.

[55] N. Ltd. *GitHub - nubificus/urunc: a simple container runtime that aspires to become 'runc' for unikernels — github.com*. https://github.com/nubificus/urunc. [Accessed 13-06-2024].

[56] O. Ltd. *Home — openfaas.com*. https://www.openfaas.com/. [Accessed 26-02-2024].

[57] J. Ma, G. Zuo, K. Loughlin, X. Cheng, Y. Liu, A. M. Eneyew, Z. Qi, and B. Kasikci. "A Hypervisor for Shared-Memory FPGA Platforms." In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '20. Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 827–844. ISBN: 9781450371025. DOI: 10.1145/3373376.3378482.

[58] A. Madhavapeddy and D. J. Scott. "Unikernels: the rise of the virtual library operating system." In: *Commun. ACM* 57.1 (Jan. 2014), pp. 61–69. ISSN: 0001-0782. DOI: 10.1145/2541883.2541895.

[59] F. Manco, C. Lupu, F. Schmidt, J. Mendes, S. Kuenzer, S. Sati, K. Yasukata, C. Raiciu, and F. Huici. "My VM is Lighter (and Safer) than your Container." In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP '17. Shanghai, China: Association for Computing Machinery, 2017, pp. 218–233. ISBN: 9781450350853. DOI: 10.1145/3132747.3132763.

[60] D. Mathew. "Comparative Study of Strassen's Matrix Multiplication Algorithm." In: *International Journal of Computer Science and Technology* 3 (Mar. 2012), pp. 749–754.

[61] P. M. Mell and T. Grance. *SP 800-145. The NIST Definition of Cloud Computing*. Tech. rep. Gaithersburg, MD, USA, 2011.

[62] M. Moghaddamfar, C. Färber, W. Lehner, and N. May. "Comparative analysis of OpenCL and RTL for sort-merge primitives on FPGA." In: *Proceedings of the 16th International Workshop on Data Management on New Hardware*. DaMoN '20. Portland, Oregon: Association for Computing Machinery, 2020. ISBN: 9781450380249. DOI: 10.1145/3399666.3399897.

[63] S. B. Needleman and C. D. Wunsch. "A general method applicable to the search for similarities in the amino acid sequence of two proteins." In: *Journal of Molecular Biology* 48.3 (1970), pp. 443–453. ISSN: 0022-2836. DOI: https://doi.org/10.1016/0022-2836(70)90057-4.

[64] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng. "Reading Digits in Natural Images with Unsupervised Feature Learning." In: *NIPS Workshop on Deep Learning and Unsupervised Feature Learning 2011*. 2011.

[65] OpenCV-Team. *OpenCV: Harris Corner Detection — docs.opencv.org*. https://docs.opencv.org/4.x/dc/d0d/tutorial_py_features_harris.html. [Accessed 12-06-2024].

[66] OpenFaas. *OpenFaaS Function chaining*. https://github.com/openfaas/faas/blob/b32a3f30ea936215ba10628fb202f7dcb2415e96/guide/chaining_functions.md. [Accessed 28-06-2024].

[67] I. OpenSSL Foundation. *openssl.org*. https://www.openssl.org/. [Accessed 28-06-2024].

[68] T. Pfandzelter, A. Dhakal, E. Frachtenberg, S. R. Chalamalasetti, D. Emmot, N. Hogade, R. P. H. Enriquez, G. Rattihalli, D. Bermbach, and D. Milojicic. "Kernel-as-a-Service: A Serverless Programming Model for Heterogeneous Hardware Accelerators." In: *Proceedings of the 24th International Middleware Conference*. Middleware '23. <conf-loc>, <city>Bologna</city>, <country>Italy</country>, </conf-loc>: Association for Computing Machinery, 2023, pp. 192–206. ISBN: 9798400701771. DOI: 10.1145/3590140.3629115.

[69] *Production-Grade Container Orchestration — kubernetes.io*. https://kubernetes.io/. [Accessed 25-03-2024].

[70] M. Qasaimeh, K. Denolf, J. Lo, K. Vissers, J. Zambreno, and P. H. Jones. "Comparing Energy Efficiency of CPU, GPU and FPGA Implementations for Vision Kernels." In: *2019 IEEE International Conference on Embedded Software and Systems (ICESS)*. 2019, pp. 1–8. DOI: 10.1109/ICESS.2019.8782524.

[71] B. Ringlein, F. Abel, A. Ditter, B. Weiss, C. Hagleitner, and D. Fey. "Programming Reconfigurable Heterogeneous Computing Clusters Using MPI With Transpilation." In: *2020 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*. 2020, pp. 1–9. DOI: 10.1109/H2RC51942.2020.00006.

[72] R. L. Rivest. *The MD5 Message-Digest Algorithm*. RFC 1321. Apr. 1992. DOI: 10.17487/RFC1321.

[73] B. Scheufler. *Cloud-Native Scheduling for Serverless FPGAs*. 2023.

[74] *Serverless Function, FaaS Serverless - AWS Lambda - AWS — aws.amazon.com*. https://aws.amazon.com/lambda/. [Accessed 09-04-2024].

[75] P. Sikka, A. R. Asati, and C. Shekhar. "Real time FPGA implementation of a high speed and area optimized Harris corner detection algorithm." In: *Microprocessors and Microsystems* 80 (2021), p. 103514. ISSN: 0141-9331. DOI: https://doi.org/10.1016/j.micpro.2020.103514.

[76] Slowkow. *File:Needleman-Wunsch pairwise sequence alignment.png - Wikimedia Commons — commons.wikimedia.org*. https://commons.wikimedia.org/wiki/File:Needleman-Wunsch_pairwise_sequence_alignment.png. [Accessed 28-06-2024].

[77] J. O. Smith. *Mathematics of the discrete Fourier transform (DFT): With audio applications*. en. Julius Smith, 2008.

[78] *Specification for the Advanced Encryption Standard (AES)*. Federal Information Processing Standards Publication 197. 2001.

[79] J. E. Stone, D. Gohara, and G. Shi. "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems." In: *Computing in Science Engineering* 12.3 (2010), pp. 66–73. DOI: 10.1109/MCSE.2010.69.

[80] J. Strombergson. *GitHub - secworks/sha256: Hardware implementation of the SHA-256 cryptographic hash function — github.com*. https://github.com/secworks/sha256. [Accessed 25-06-2024].

[81] B. Tan, H. Liu, J. Rao, X. Liao, H. Jin, and Y. Zhang. "Towards Lightweight Serverless Computing via Unikernel as a Function." In: *2020 IEEE/ACM 28th International Symposium on Quality of Service (IWQoS)*. 2020, pp. 1–10. DOI: 10.1109/IWQoS49365.2020.9213020.

[82] K. Vipin and S. A. Fahmy. "FPGA Dynamic and Partial Reconfiguration: A Survey of Architectures, Methods, and Applications." In: *ACM Comput. Surv.* 51.4 (July 2018). ISSN: 0360-0300. DOI: 10.1145/3193827.

[83] X. Wang and H. Yu. "How to break MD5 and other hash functions." In: *Annual international conference on the theory and applications of cryptographic techniques*. Springer. 2005, pp. 19–35.

[84] X. Wang, Y. Niu, F. Liu, and Z. Xu. "When FPGA Meets Cloud: A First Look at Performance." In: *IEEE Transactions on Cloud Computing* 10.2 (2022), pp. 1344–1357. DOI: 10.1109/TCC.2020.2992548.

[85] J. Wen, Z. Chen, X. Jin, and X. Liu. *Rise of the Planet of Serverless Computing: A Systematic Review*. 2022. arXiv: 2206.12275.

[86] William H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical recipes in FORTRAN 77: Volume 1 of FORTRAN numerical recipes volume 1*. 2nd ed. Cambridge, England: Cambridge University Press, Sept. 1992.

[87] D. Williams and R. Koller. "Unikernel Monitors: Extending Minimalism Outside of the Box." In: *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. Denver, CO: USENIX Association, June 2016.

[88] Xilinx. *Harris Corner Detection - Vitis Libraries*. https://xilinx.github.io/Vitis_Libraries/vision/2021.2/harris-bm.html. [Accessed 26-06-2024].

[89] Xilinx. *Lossless Compression — xilinx.com*. https://www.xilinx.com/products/intellectual-property/ef-di-compress.html. [Accessed 26-06-2024].

[90] Xilinx. *RapidWright — rapidwright.io*. https://www.rapidwright.io/. [Accessed 28-06-2024].

[91] C. Xiong and N. Xu. "Performance Comparison of BLAS on CPU, GPU and FPGA." In: *2020 IEEE 9th Joint International Information Technology and Artificial Intelligence Conference (ITAIC)*. Vol. 9. 2020, pp. 193–197. DOI: 10.1109/ITAIC49862.2020.9338793.

[92] Y. Zha and J. Li. "Virtualizing FPGAs in the Cloud." In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '20. Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 845–858. ISBN: 9781450371025. DOI: 10.1145/3373376.3378491.