BlastFunction: an FPGA-as-a-Service system for Accelerated Serverless Computing

Marco Bacis
DEIB, *Politecnico di Milano*, Milano, IT
marco.bacis@mail.polimi.it

Rolando Brondolin
DEIB, *Politecnico di Milano*, Milano, IT
rolando.brondolin@polimi.it

Marco D. Santambrogio
DEIB, *Politecnico di Milano*, Milano, IT
marco.santambrogio@polimi.it

***Abstract*—Heterogeneous computing platforms are now a valuable solution to continue to meet Service Level Agreements (SLAs) for compute intensive cloud workloads. Field Programmable Gate Arrays (FPGAs) effectively accelerate cloud workloads, however, these workloads have a spiky behavior as well as long periods of underutilization. Sharing the FPGA with multiple tenants then helps to increase the board's time utilization. In this paper we present BlastFunction, a distributed FPGA sharing system for the acceleration of microservices and serverless applications in cloud environments. BlastFunction includes a *Remote OpenCL Library* to access the shared devices transparently; multiple *Device Managers* to time-share and monitor the FPGAs and a central *Accelerators Registry* to allocate the available devices. BlastFunction reaches higher utilization and throughput w.r.t. a native execution thanks to device sharing, with minimal differences in latency given by the concurrent accesses.**

***Index Terms*—FPGA sharing, Serverless, OpenCL**

I. Introduction

The last decade saw the exponential growth of cloud computing as the primary technology to develop, deploy and maintain complex infrastructures and services at scale. Cloud computing allows to consume resources on-demand in a multi-tenant fashion, thus design services with a cloud-native approach is fundamental to dynamically scale performance.

Compute intensive workloads may require performance that current CPUs are not able to provide and, for this reason, heterogeneous computing is becoming an interesting solution to continue to meet SLAs in the cloud. Within this context, in addition to GPU-enabled instances, cloud vendors offer FPGA-based compute instances (directly as in AWS F1 instances[1] or through managed services like Catapult [1] and Brainwave [2]). Cloud workloads such as web search [3], image processing [4], database operations [5], neural network inference [6], and many others can benefit from the use of FPGAs to timely react to the end-users' requests.

To exploit FPGAs at their best in the cloud, hardware accelerators should be designed to meet latency requirements while optimizing throughput [1]. Requests from the outside network can come at unpredictable rates and they usually cannot be batched, and for this reason minimizing latency becomes fundamental. Moreover, the unpredictability of the requests can lead to an underutilization of the FPGAs, thus reserve one FPGA for each service that needs it will result in a waste of resources. From a cloud provider perspective, a possible solution would be to share the FPGA across different tenants to improve its time utilization.

Within this context, the serverless computing paradigm can be leveraged to share an FPGA at a fine-grain level with other tenants to improve its time utilization. Serverless computing [7] is an architectural pattern for cloud applications where server management is delegated to the cloud provider. Each application functionality is deployed by the user as a function and scheduled, executed, scaled, and billed depending on the exact need of the moment. In this way, cloud-native applications can benefit from FPGAs to accelerate compute-intensive workloads such as neural network inference, web searches, and image processing in a seamless way. To support these goals, in this paper we propose BlastFunction, a distributed and *transparent* FPGA sharing system for the acceleration of microservices and serverless applications in cloud environments. We decided to focus on a *time sharing* approach to maximize the devices' utilization (in terms of accelerator execution time) and optimize the use of devices from the cloud provider perspective. The contribution of BlastFunction are:

- the design and implementation of a *scalable* system enabling *multi-tenancy* for cloud FPGAs in containerized and serverless environment,
- the design and implementation of a *transparent* layer that allows integrating applications and hardware accelerators written in OpenCL without code rewriting.

The rest of this paper is organized as follows: Section II describes the State of the Art in the field, Section III details the design and implementation of BlastFunction, Section IV evaluated the proposed design, while Section V draws the conclusion and the future work of this paper.

II. State of the Art

Here we present the analysis of the State of the Art of interest for this work, focusing on the integration of FPGAs in cloud environments. We classified the literature by *communication method*, *sharing mechanism* and *computational model*.

The first distinction is the *communication method* used to access the shared or virtualized device. *PCIe-Passthrough* is the lowest-level method available, as it works by directly connecting a single Virtual Machine (VM) or container to the FPGA device. This communication level is used by the AWS F1 instances. With *Paravirtualization* the requesting application VM is connected to a host device driver which

[1] https://aws.amazon.com/ec2/instance-types/f1/

978-3-9819263-4-7/DATE20/©2020 EDAA

virtualizes the access to the resources. This mechanism is used by *pvFPGA* [8]. The *API Remoting* mechanisms is the most used in the analyzed state of the art [9], [10], [11], [12], and it works by defining a custom API to remotely access the device. It allows multiple applications to control the shared device and to perform both space and time sharing. A special API Remoting technique is represented by the work in [13], as in this case the system exposes a microservice for each accelerator, and not a general API for the entire system. Finally, *Direct Network Access* is used by *Catapult* [14]. This method works by exposing the FPGA through its network interface, thus enabling a low-latency access.

The second classification is based on the *sharing mechanism*. *Space-Sharing* [15], [10], [12] employs FPGA virtualization through the use of *Partial Dynamic Reconfiguration (PDR)* or *Overlays* to run multiple accelerators on the same FPGA, which are used by different applications. *Space-sharing* allows to use the entire resources on the device (in terms of logic blocks), but requires careful handling of the accelerators to minimize the reconfiguration time. *Time-Sharing* [8], [9], [11], [13], [14] works by *multiplexing* multiple requests from different applications on the same accelerator in the FPGA board. In this case, the challenge is to efficiently schedule the incoming requests to minimize latency on the application side and managing memory accesses to fit in the I/O bandwidth.

The last classification is related to the *computational model*. In a *Batch System* [8], [10], [12], [15], the workloads (and the connection to the FPGA) are seen as limited in time. Therefore the scheduling and allocation of workloads are computed based on the *lifetime* of each job. In *Service-Based* [9], [11], [13], [14] systems, instead, the FPGAs are continuously working by receiving and processing requests from the system or the application. Works such as [13] directly expose the underlying FPGA accelerator as a standalone service.

Within the analyzed literature, BlastFunction leverages *API Remoting* techniques in a transparent way with OpenCL, focusing on *Time-Sharing* of FPGAs for *Service-Based* workloads like micro-services and serverless functions.

III. System design

BlastFunction is an FPGA sharing system for the acceleration of microservices and serverless applications in cloud environments. The system allows multiple applications to concurrently execute kernels on the same FPGA without changing the underlying host code. This is achieved leveraging OpenCL as the accelerators' runtime support system. The system allocates the available devices as requested by the applications leveraging runtime metrics collected by the system itself.

Figure 1 shows the main components of BlastFunction: the *Remote OpenCL Library*, the *Device Manager* and the *Accelerators Registry*. The *Remote OpenCL Library* allows the client application (microservice or serverless function) to integrate with BlastFunction; it is a custom OpenCL implementation that transparently abstracts the remote device access protocol and the communication with the *Accelerators Registry* and the *Device Managers*. Each *Device Manager* is

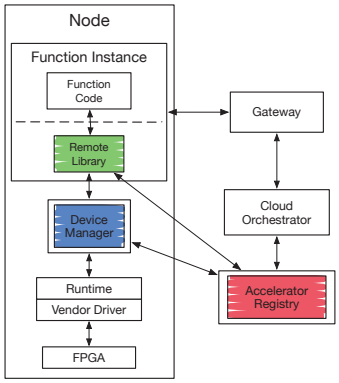


Fig. 1: High Level Overview of BlastFunction components (Remote Library in green, Device Manager in blue, and Accelerator Registry in red) and their connections.

connected to a FPGA in the system and provides the time-sharing mechanism. It exposes a service to remotely access the device functionalities, providing isolated access for multiple application containers. Finally, the *Accelerators Registry* is the central controller of the system. It tackles the allocation and reconfiguration of the available devices using runtime performance metrics. We use network protocols like *gRPC*[2] for control and *shared memory* for data transfers between the *Remote OpenCL Library* and the *Device Manager*. Data exchange between the *Accelerators Registry* and the other components is done through gRPC.

BlastFunction integrates with other external components to reach its goals. The *Cloud Orchestrator* (Kubernetes[16] in our case) is used by the *Accelerators Registry* to control the cluster resources and their allocation to the nodes. The *Gateway* is the serverless (OpenFaaS[3]) system's endpoint, which forwards the requests to the functions and handles autoscaling.

A. Remote OpenCL Library

The *Remote OpenCL Library* is a custom implementation of an OpenCL host library developed to integrate the applications with BlastFunction and to isolate them from the execution of the hardware accelerator. In particular, the *Remote OpenCL Library* implements most of the methods used to control an FPGA accelerator and can be linked both statically and dynamically to the application. The Remote OpenCL Library implements a central router component, which keeps the list of the available platforms. In particular, it gets the address of the selected *Device Manager* (or managers if multiple addresses are provided) and creates a connection to it through gRPC.

The system allows for both synchronous/blocking OpenCL calls to the remote runtime as well as asynchronous/non-blocking calls. Both the synchronous and asynchronous flows are designed with *asynchronous events*. An event in the system is composed by a set of subsequent asynchronous calls to the device manager service, a state machine to control the steps that the event must follow and an OpenCL status for the event that is updated while the event is processed. In this way, the remote library supports event polling (e.g. clWaitForEvents, clGetEventInfo) like the standard OpenCL specification. We

[2] https://grpc.io/ [3] https://www.openfaas.com/

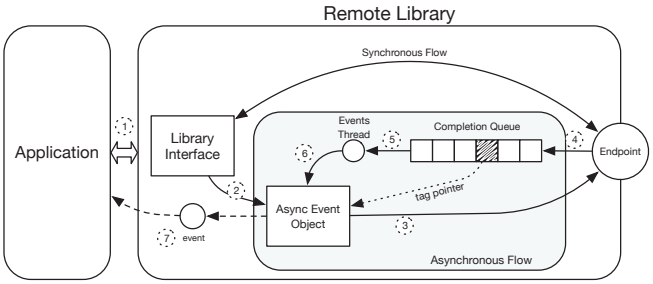


Fig. 2: OpenCL Remote Library Architecture, highlighting the steps performed in the asynchronous flow (dotted lines represent asynchronous responses).

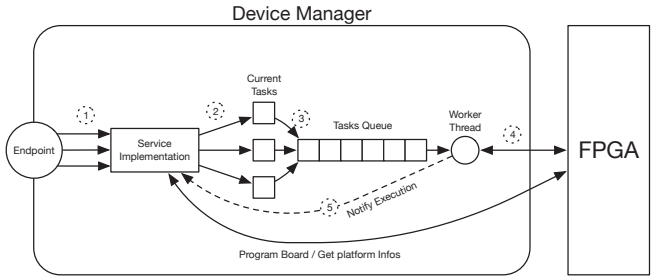


Fig. 3: Device Manager Architecture, with the command queue methods flow highlighted (dotted lines represent asynchronous responses).

will explain the asynchronous flow by following Figure 2, which shows the main components of the Remote Library.

When the Remote library receives an asynchronous OpenCL call like a clEnqueueReadBuffer from the application (step 1), it creates an event (step 2) and performs a first asynchronous request through the network stack (step 3). The request encapsulates a *tag*, which is the pointer to the newly created event. When the device manager responds (step 4), the network runtime pushes the tag into the completion queue of the client. Then, the connection thread pulls the tag and retrieves the corresponding event (step 5). The thread then calls the event state machine and updates its state and the OpenCL event status (step 6). Finally, the application is notified when the event changes the OpenCL status (step 7). For instance, to perform the clEnqueueReadBuffer function, the event state machine contains 4 states. The INIT state sends the call metadata (buffer size, buffer id, offset); the FIRST step waits for the command to be enqueued by the manager; the BUFFER step actually sends the buffer data when the manager is available, and the COMPLETE step signals the call completion.

B. Device Manager

The *Device Manager* shown in Figure 3 controls and manages a single board inside BlastFunction. In particular, along with the Remote OpenCL Library, it is the basic block of the *sharing mechanism* presented in this work, allowing many services to access the FPGA concurrently. The *Device Manager* separately controls each client's resources pool to enforce isolation between multiple clients.

There are two kind of methods exposed by the service: *context and information methods*, and *command-queue methods*. The *context and information methods* are executed synchronously as they do not involve execution on the FPGA. This group of requests includes the creation (on the client side) of kernels, the creation of platforms and contexts, the request of information related to the device and the buffer management requests. The *board reconfiguration request* represents the only exception in this group, as it blocks the execution of other operations to reprogram the board with the given bitstream. The other group of requests is represented by *command-queue methods*, which are composed by operations that must be executed in the order decided by the client application and might require to use the FPGA exclusively. An example is the kernel execution request, which might be interleaved with buffer reads and writes on one or multiple queues. For this kind of requests, if any operation is received or executed in the wrong order by the *Device Manager*, the results of the execution will change breaking the application consistency.

To ensure the in-order execution of *command-queue* methods, the *Device Manager* employs *multi-operation tasks*. We define a *task* as the atomic unit of execution of BlastFunction, composed of a sequence of operations that should execute atomically on the FPGA. Whenever the *Device Manager* receives a *command-queue call* (step 1 in Figure 3), the requested operations are added to the task related to that particular client (step 2). After that, if the client sends a flush command (either by calling a blocking method or clFinish/Flush/EnqueueBarrier), the current task is sent to the central queue of the manager (step 3). Once the task arrives to the central queue of the manager, a worker thread pulls and executes it on the FPGA in a First-In-First-Out order (step 4). Each operation in the task is linked with a OpenCL event and when the operation is completed the event is notified to the caller (step 5). In this way, the client is notified punctually, even if the operations are executed in groups.

For what concerns data exchange with the *Remote OpenCL Library*, the *Device Manager* allows two different mechanisms: network based (using gRPC) or shared memory. The Device Manager employs gRPC if the client application is not on the same node, or if it is not possible to create a shared memory area. Although gRPC is a powerful protocol for data exchange over network, we found performance issues utilizing it locally due to serialization overhead and due to multiple data copies. For this reason we limit its use whenever possible, leveraging instead shared memory. This improves performance and reduces additional data copies (from four to one) at the cost of having the client function together with the device manager on the same node with enough permissions. We still need one data copy to maintain full OpenCL compatibility, as a direct access to the shared memory would require to define additional functions not available in the OpenCL specification.

C. Accelerators Registry

The *Accelerators Registry* is the master component of the system: it registers functions and devices, it aggregates performance metrics, it allocates devices to functions and it validates reconfiguration operations. The *Registry* offers two endpoints, each backed by a different service. The *Devices Service* collects and manages information about the devices

(e.g. platform, configured bitstream and connected instances). The *Functions Service* contains data about the serverless functions (e.g. identifier, location, device, created instances).

Data collected through the Device and Functions Services are integrated by the *Metrics Gatherer*, which receives Device Managers performance metrics from a Prometheus[4] service. Data like the FPGA time utilization (defined as the time spent by the device computing OpenCL calls in a given amount of time) are used to improve allocation of functions.

To match function instances and available devices, the Registry performs an *online* allocation algorithm when a new instance is created. To do so, the Registry integrates with Kubernetes to intercept function creation and deletion in the cluster. When the cluster notifies the creation of a new function, the allocation algorithm patches the notified operation (e.g. adds environment variables, volumes for shared memory and forces the host allocation). The allocation algorithm is presented in Algorithm 1. It takes as input the function instance that must be matched, all the available devices in the system and a list of metrics to be taken into account. First, the procedure filters the devices based on their compatibility with the application requests (in terms of vendor, platform and accelerator) and the performance metrics (e.g. filtering out highly utilized devices). The devices are then sorted by metrics and by accelerator compatibility to ensure an optimal and consistent allocation. The metrics priority can be chosen depending on the system and applications SLA (e.g. device utilization, connected functions, latencies). The *accelerator compatibility* instead checks if the device should be reconfigured by looking at the currently configured bitstream. When compatible accelerators are missing, the algorithm checks which workloads can be redistributed to other compatible devices. If at least one device is found, it is flagged for reconfiguration and the Registry allocates it to the requesting function instance.

When a reconfiguration is required, BlastFunction checks the redistribution of instances and then *migrates* them with the Kubernetes API if necessary. In particular, when a function instance sends a reconfiguration request, the Registry verifies the allocation of the requesting function instance and checks if the device needs to be reconfigured. In that case, it deletes all the functions connected to that device. Kubernetes creates new instances before deleting the previous ones: in this way the Registry can patch and schedule them on a different node.

Algorithm 1 Devices allocation algorithm

```
1: procedure ALLOCATE(instance, devs, metrics_order, metrics_filters)
2:     devs ← filterby_compatibility(devs, instance.devicequery)
3:     devs ← filterby_metrics(devs, metrics_filters)
4:     devs ← orderby_metrics_and_acc(devs, metrics_order)
5:     i ← 0
6:     if not_compatible(devs(i)) then
7:         while not_redistributable(devs(i)) do
8:             i ← i + 1
9:     if i < len(devs) then
10:        chosen_device ← devs(i)
11:    else
12:        raise error "device not found"
13:    instance.devs ← {chosen_device}
14:    if instance.node == "" then
15:        instance.node ← chosen_device.node
16:    return
```

IV. Experimental evaluation

The goals of the experimental campaign are to assess whether BlastFunction introduces an acceptable overhead (Section IV-A) and if it can improve device's time utilization (Section IV-B) w.r.t. the maximum theoretical performance scenario represented by a native execution that has direct access to the FPGAs. We leveraged three accelerated cloud functions available in the State of the Art: the Sobel edge detector and the Matrix Multiply (MM) kernel from the *Spector* benchmark suite [17] and *PipeCNN* [18], which is an open-source implementation of an FPGA accelerator for Convolutional Neural Networks (CNNs). This benchmark calls several kernels iteratively with multiple parallel command queues to compute the CNN output. According to the Spector benchmark, we synthesized the Sobel edge detector (also called *Sobel operator*) with 32×8 blocks, 4×1 window with no SIMD applied and a single compute unit, as it results in the best latency performance. For MM, we found from [17] that the best design is with 1 compute unit, 8 work items for each unit, and a completely unrolled block of 16×16 elements. Finally, we synthesized PipeCNN with AlexNet as in [18].

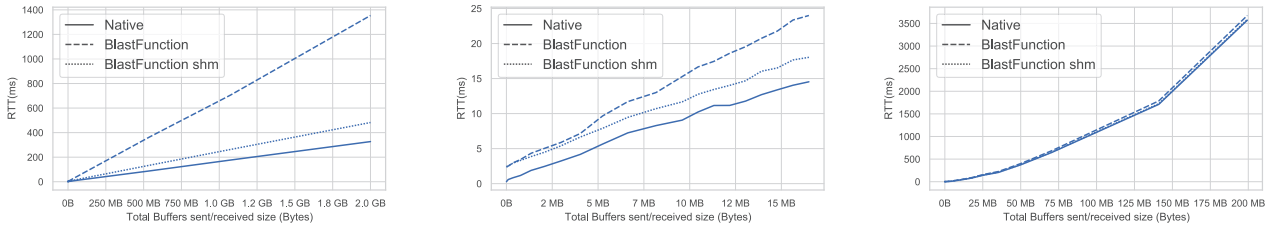
The experimental platform is composed of three nodes, one master and two workers. The master node (node A) contains a 2.80Ghz Intel® Xeon®W3530 CPU, with 8 threads (4 cores) and 24GB of DDR3 RAM. Each worker node (nodes B and C) is equipped with a 3.40Ghz Intel® Core™ i7-6700 CPU, with 8 threads (4 cores) and 32GB of DDR4 RAM. Each node is connected to the local network through a 1Gb/s ethernet link. Each node contains a Terasic DE5a-Net FPGA board with an Intel®Arria 10 GX FPGA (1150K logic elements), 8GB RAM over 2 DDR2 SODIMM sockets and a PCI Express x8 connector (version 3 for the workers, version 2 for the master).

A. System overhead

We evaluated the system overhead on a single node, deploying one instance of the Device Manager with a Docker Container connected to the FPGA. The host code was deployed on another Docker Container on the same node. Our system leverages the local virtual network stack + shared memory and PCI Express, while Native execution needs PCI Express only. We run each test by increasing the input and output size to see the impact of the Remote Library communication mechanism (both gRPC and shared memory) and the Device Manager queue. We tested each input size 40 times averaging results and waiting 200ms after each call to have independent measures. We skip PipeCNN here because it does not allow to change the input size. Results are presented in Figure 4.

Figure 4(a) shows the Round-Trip Time (RTT) for a write-read operation (first write, then read synchronously) with total size from 1KB to 2GB for Native, BlastFunction and BlastFunction with shared memory. The pure gRPC implementation ("BlastFunction" in Figure 4(a)) shows a total latency of four times w.r.t. the Native execution. This is due to

[4] https://prometheus.io



(a) Latency overhead for read and write operations. (b) Latency overhead for Sobel operator. (c) Latency overhead for MM accelerator.

Fig. 4: Latency overhead w.r.t. input data size for R/W operations, sobel operator and MM kernel, graphs are in linear scale. For MM Native overlaps with BlastFunction shm.

protobuf overheads and 3 copies of the data buffers. The shared memory implementation ("BlastFunction shm" label) shows an improvement in terms of latency and overhead, with a maximum overhead of 155ms when transferring 2GBs. Most of the overhead is composed by the single memory copy operation, while a smaller part ($\sim 2ms$) is given by the gRPC control signals, which are used in both systems.

Figure 4(b) shows the latency measurements for the Sobel operator. In all cases, the kernel has a linear behaviour w.r.t. the input size. The Native RTT starts from 0.27ms with a 10×10 image (800 bytes sent and received), up to 14.53ms for the largest image (1920×1080 pixels, read/write of $\sim 8MB$). BlastFunction starts with a overhead of 2.46ms and reaches 24ms with the largest image. BlastFunction shm, instead, has a constant $\sim 2ms$ overhead w.r.t. Native in all the experiments.

Figure 4(c) shows the latency measurements for the MM kernel. The MM accelerator is compute-intensive and the execution overhead between the Native and remote execution is low for both communication systems (still remaining lower in the shared memory system). The Native runtime shows a minimum RTT of 0.45ms for the smallest matrices (16×16 in both input and output matrices), but quickly rises up to 3.571s (for 4076×4096 matrices). As in the Sobel results, both BlastFunction and BlastFunction shm show a minimum RTT of $\sim 2ms$ given by the control signals. BlastFunction then reaches a maximum of 3.675s, while BlastFunction shm stops at 3.588s, which is only 17ms more than Native.

The results of Figure 4 show that the overall impact of our system depends on the complexity and operational intensity of the accelerator. When the majority of the execution time is spent in kernel execution the overall overhead is low (as in the MM example with a relative overhead of 0.27% for shared memory). Instead, with lower operational intensity, the I/O latency impacts more on the task even in the shared memory case (as in Sobel with a relative overhead of 24.04%). This derives from the fact that the Native system does not execute any additional data copy, while BlastFunction needs at least one copy to maintain full OpenCL compatibility.

B. FPGAs time utilization

To test the FPGAs utilization, we run a set of multi-application, multi-node experiments. The goal is to check if BlastFunction is able to increase the FPGAs time utilization and the number of total requests served without significant losses for the single tenant. Here we leverage BlastFunction with shared memory, wrapping each benchmark in a OpenFaaS function both for Native and BlastFunction. For each experiment (Sobel, MM and PipeCNN with AlexNet) we deployed 5 identical functions for *BlastFunction*, while we could deploy only 3 functions in the Native scenario (one for each device). We tested each function using Hey[5] (a tool for HTTP load testing), running the experiments multiple times with one connection per function. Table I shows the test configurations, where only the first 3 columns are used for the Native scenario.

Use-Case	Configuration	1st	2nd	3rd	4th	5th
Sobel	Low load	20 rq/s	15 rq/s	10 rq/s	5 rq/s	5 rq/s
	Medium Load	35 rq/s	30 rq/s	25 rq/s	20 rq/s	15 rq/s
	High Load	60 rq/s	50 rq/s	35 rq/s	30 rq/s	15 rq/s
MM	Low load	28 rq/s	21 rq/s	14 rq/s	7 rq/s	7 rq/s
	Medium Load	49 rq/s	42 rq/s	35 rq/s	28 rq/s	21 rq/s
	High Load	84 rq/s	70 rq/s	49 rq/s	42 rq/s	21 rq/s
AlexNet	Medium load	6 rq/s	3 rq/s	3 rq/s	3 rq/s	3 rq/s
	High Load	9 rq/s	9 rq/s	6 rq/s	6 rq/s	3 rq/s

TABLE I: Tests configurations overview, showing how many requests per second were sent to each function for each benchmark.

Type	Configuration	Function	Node	Util.	Latency	Processed	Target
BlastFunction	Low Load	sobel-1	B	21.95%	21.43 ms	17.25 rq/s	20.00 rq/s
		sobel-2	A	22.57%	24.23 ms	15.00 rq/s	15.00 rq/s
		sobel-3	C	13.22%	19.01 ms	10.00 rq/s	10.00 rq/s
		sobel-4	A	7.49%	31.98 ms	5.00 rq/s	5.00 rq/s
		sobel-5	B	6.48%	27.16 ms	5.00 rq/s	5.00 rq/s
	Medium Load	sobel-1	B	40.95%	19.45 ms	32.93 rq/s	35.00 rq/s
		sobel-2	A	39.40%	23.62 ms	26.30 rq/s	30.00 rq/s
		sobel-3	C	32.85%	18.28 ms	24.98 rq/s	25.00 rq/s
		sobel-4	A	29.85%	26.99 ms	19.98 rq/s	20.00 rq/s
		sobel-5	B	18.76%	22.94 ms	14.97 rq/s	15.00 rq/s
	High Load	sobel-1	B	60.31%	18.95 ms	49.58 rq/s	60.00 rq/s
		sobel-2	A	39.15%	32.05 ms	26.63 rq/s	50.00 rq/s
		sobel-3	C	45.75%	17.82 ms	34.96 rq/s	35.00 rq/s
		sobel-4	A	38.44%	22.56 ms	26.11 rq/s	30.00 rq/s
		sobel-5	B	18.39%	21.74 ms	15.00 rq/s	15.00 rq/s
Native	Low Load	sobel-1	A	30.41%	25.02 ms	19.49 rq/s	20.00 rq/s
		sobel-2	B	19.74%	21.50 ms	14.74 rq/s	15.00 rq/s
		sobel-3	C	13.73%	24.34 ms	9.75 rq/s	10.00 rq/s
	Medium Load	sobel-1	A	51.48%	26.04 ms	33.11 rq/s	35.00 rq/s
		sobel-2	B	37.19%	23.33 ms	27.95 rq/s	30.00 rq/s
		sobel-3	C	34.22%	23.48 ms	24.23 rq/s	25.00 rq/s
	High Load	sobel-1	A	58.10%	26.77 ms	38.36 rq/s	60.00 rq/s
		sobel-2	B	54.69%	23.95 ms	41.80 rq/s	50.00 rq/s
		sobel-3	C	44.81%	24.75 ms	32.61 rq/s	35.00 rq/s

TABLE II: Multi-function test results for the Sobel acceleator in terms of average latency, FPGA time utilization (overall maximum 300%) and processed/target requests.

We show the per-function results for Sobel in Table II. The results are divided by scenario (BlastFunction vs Native), configuration, and tested function. In the low load configuration both runtimes keep up with the target throughput with latency between 20-30ms. Results are in line with the overhead results, with BlastFunction improving device's utilization. In the medium load configuration BlastFunction has better latency for *sobel-1*, *sobel-2* and *sobel-3* and the other two functions effectively increases the board's time utilization. Finally, in the high load configuration, BlastFunction still improved the

[5] https://github.com/rakyll/hey

Type	Configuration	Utilization	Latency	Processed	Target
BlastFunction	Low Load	43.49%	12.55 ms	76.96 rq/s	77 rq/s
	Medium Load	98.53%	11.57 ms	174.90 rq/s	175 rq/s
	High Load	144.18%	10.69 ms	262.73 rq/s	266 rq/s
Native	Low Load	50.87%	21.12 ms	60.49 rq/s	63 rq/s
	Medium Load	103.22%	22.81 ms	106.84 rq/s	126 rq/s
	High Load	122.97%	24.25 ms	121.85 rq/s	203 rq/s

TABLE III: Multi-function test aggregate results for MM in terms of average latency, FPGA time utilization (overall maximum 300%) and processed/target requests.

Type	Configuration	Utilization	Latency	Processed	Target
BlastFunction	Medium Load	124.68%	132.89ms	17.88 rq/s	18 rq/s
	High Load	202.08%	124.52ms	29.81 rq/s	33 rq/s
Native	Medium load	96.22%	94.29ms	11.91 rq/s	12 rq/s
	High Load	189.82%	91.74ms	23.57 rq/s	24 rq/s

TABLE IV: Multi-function test aggregate results for PipeCNN (AlexNet) with average latency, FPGA time utilization (overall maximum 300%) and processed/target requests.

overall FPGAs time utilization with comparable latency results for *sobel-1* and *sobel-3*. However, *Node A* saturated in both cases as it is not able to keep-up with the target throughput. Regarding the requests throughput, Native has a difference w.r.t. the target of 2.25% in the low load configuration, 5.23% and 22.22% for the medium and high load conditions respectively. BlastFunction has instead averages of 5.01%, 4.67% and 19.85% respectively. Although BlastFunction supports more load, the response of the two systems are still comparable.

Table III shows the aggregate results for MM. We do not show the detailed results for brevity as similar considerations can be made. The Native scenario presents a higher difference between target and processed requests w.r.t. BlastFunction, with slightly higher latencies and a similar utilization. The average difference for BlastFunction is of 0.04%, 0.05% and 1.22% for the low, medium and high load configurations. Meanwhile, Native reaches 3.97% with a low load, 15.19% and 39.97% in medium and high load conditions.

Finally, we show the aggregate results for AlexNet with the PipeCNN accelerator in Table IV. Because of the low number of requests that the accelerator is able to serve, we decided to test only two configurations, with medium and high load conditions. The results show that Native has an average latency of 94.29ms for medium load and 91.74ms for high load, while BlastFunction presents a higher latency (132.89ms for medium and 124.52ms for high load). This happens as the host code calls multiple times the kernels for each computation, increasing the overhead. Regarding the difference between sent and processed requests, we have 0.63% for BlastFunction and 0.68% for Native in medium load conditions, while in high load conditions Native behaves better (1.79% vs 9.64%). However, in both configurations, sharing allows BlastFunction to reach a higher utilization and number of processed requests.

V. Conclusion and future work

We presented BlastFunction, a distributed FPGA sharing system for the acceleration of microservices and serverless applications in cloud environments. The proposed system is based on *multi-tenancy* and *scalability*, with a focus on the *transparency* of the resulting software library. BlastFunction adds limited overhead and reaches higher utilization and throughput w.r.t. native execution thanks to device sharing. Future work will address the integration with AWS F1 for nodes autoscaling and the introduction of space-sharing techniques.

References

[1] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim *et al.*, "A cloud-scale acceleration architecture," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 2016, p. 7.

[2] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman *et al.*, "Serving dnns in real time at datacenter scale with project brainwave," *IEEE Micro*, vol. 38, no. 2, pp. 8–20, 2018.

[3] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 13–24, 2014.

[4] S. Asano, T. Maruyama, and Y. Yamaguchi, "Performance comparison of fpga, gpu and cpu in image processing," in *2009 international conference on field programmable logic and applications*. IEEE, 2009, pp. 126–131.

[5] P. Papaphilippou and W. Luk, "Accelerating database systems using fpgas: A survey," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2018, pp. 125–1255.

[6] X. Wei, C. H. Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, Y. Liang, and J. Cong, "Automated systolic array architecture synthesis for high throughput cnn inference on fpgas," in *Proceedings of the 54th Annual Design Automation Conference 2017*. ACM, 2017, p. 29.

[7] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar *et al.*, "Cloud programming simplified: A berkeley view on serverless computing," *arXiv preprint arXiv:1902.03383*, 2019.

[8] W. Wang, M. Bolic, and J. Parri, "PvFPGA: Accessing an FPGA-based hardware accelerator in a paravirtualized environment," *2013 International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2013*, 2013.

[9] A. Iordache, G. Pierre, P. Sanders, J. G. de F Coutinho, and M. Stillwell, "High performance in the cloud with fpga groups," in *Proceedings of the 9th International Conference on Utility and Cloud Computing*. ACM, 2016, pp. 1–10.

[10] M. Asiatici, N. George, K. Vipin, S. A. Fahmy, and P. Ienne, "Designing a virtual runtime for FPGA accelerators in the cloud," *FPL 2016 - 26th International Conference on Field-Programmable Logic and Applications*, 2016.

[11] S. Mavridis, M. Pavlidakis, I. Stamoulias, C. Kozanitis, N. Chrysos, C. Kachris, D. Soudris, and A. Bilas, "VineTalk: Simplifying software access and sharing of FPGAs in datacenters," *2017 27th International Conference on Field Programmable Logic and Applications, FPL 2017*, pp. 2–5, 2017.

[12] Z. Zhu, A. X. Liu, F. Zhang, and F. Chen, "FPGA Resource Pooling in Cloud Computing," *IEEE Transactions on Cloud Computing*, vol. PP, no. c, p. 1, 2018.

[13] S. Ojika, A. Gordon-Ross, H. Lam, B. Patel, G. Kaul, and J. Strayer, "Using fpgas as microservices: Technology, challenges and case study," in *9th Workshop on Big Data Benchmarks Performance, Optimization and Emerging Hardware (BPOE-9)*, 2018.

[14] A. M. Caulfield, Others, E. S. Chung, P. Kaur, J.-y. K. Daniel, L. Todd, and M. Kalin, "A cloud-scale acceleration architecture," *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–13, 2016.

[15] S. Byma, J. G. Steffan, H. Bannazadeh, A. Leon-Garcia, and P. Chow, "FPGAs in the cloud: Booting virtualized hardware accelerators with OpenStack," *Proceedings - 2014 IEEE 22nd International Symposium on Field-Programmable Custom Computing Machines, FCCM 2014*, pp. 109–116, 2014.

[16] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," 2016.

[17] Q. Gautier, A. Althoff, P. Meng, and R. Kastner, "Spector: An OpenCL FPGA benchmark suite," *Proceedings of the 2016 International Conference on Field-Programmable Technology, FPT 2016*, pp. 141–148, 2017.

[18] D. Wang, K. Xu, and D. Jiang, "Pipecnn: An opencl-based open-source fpga accelerator for convolution neural networks," in *2017 International Conference on Field Programmable Technology (ICFPT)*. IEEE, 2017, pp. 279–282.