(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
26 November 2020 (26.11.2020)

WIPO | PCT

(10) International Publication Number
WO 2020/234792 A1

(51) International Patent Classification:
G06F 9/50 (2006.01)

(21) International Application Number:
PCT/IB2020/054775

(22) International Filing Date:
20 May 2020 (20.05.2020)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
19175546. 1 21 May 2019 (21.05.2019) EP

(71) Applicant: POLITECNICO DI MILANO [IT/IT]; Piazza Leonardo da Vinci, 32, 20133 Milano (IT).

(72) Inventors: SANTAMBROGIO, Marco Domenico; c/o Politecnico di Milano, Piazza Leonardo da Vinci, 32, 1-20133 Milano (IT). BRONDOLIN, Rolando; c/o Politecnico di Milano, Piazza Leonardo da Vinci, 32, 1-20133 Milano (IT). BACIS, Marco; c/o Politecnico di Milano, Piazza Leonardo da Vinci, 32, 1-20133 Milano (IT).

(74) Agent: DI BERNARDO, Antonio et al.; c/o THINX S.r.l., Piazzale Luigi Cadoma, 10, 1-20123 Milan (IT).

(81) Designated States *(unless otherwise indicated, for every kind of national protection available)*: AE, AG, AL, AM, AO, AT, AU, AZ, BA, BB, BG, BH, BN, BR, BW, BY, BZ, CA, CH, CL, CN, CO, CR, CU, CZ, DE, DJ, DK, DM, DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IR, IS, JO, JP, KE, KG, KH, KN, KP, KR, KW, KZ, LA, LC, LK, LR, LS, LU, LY, MA, MD, ME, MG, MK, MN, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PA, PE, PG, PH, PL, PT, QA, RO, RS, RU, RW, SA,

(54) **Title:** AN FPGA-AS-A-SERVICE SYSTEM FOR ACCELERATED SERVERLESS COMPUTING

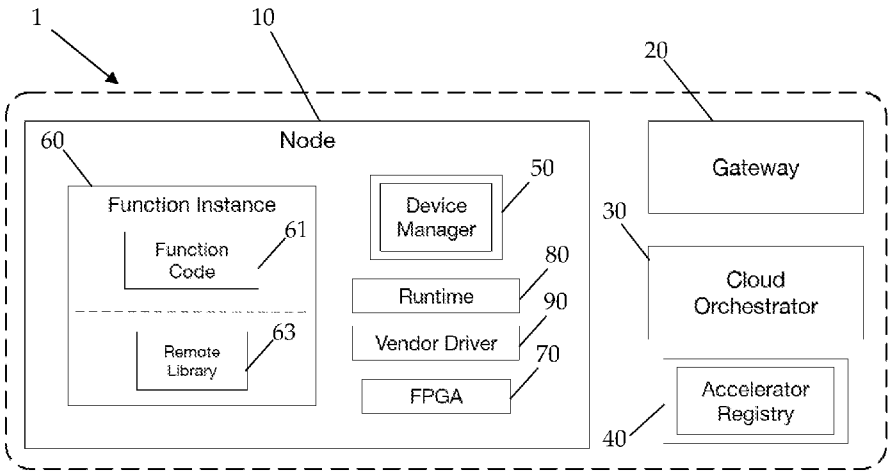


Fig-1

(57) Abstract: The present invention proposes a hardware accelerators management system **(1)** for containerized and serverless environments. The system (1) at least comprises a domain layer on which a plurality of application containers and functions (60, 61) are implemented, a hardware layer in which a set of hardware accelerators are implemented and a software layer configured for abstracting the application containers and the functions (60, 61) of the domain layer from the hardware layer, wherein the system (1) comprises a hardware interface (80, 90) to send tasks to and reconfigure at least a portion of the processing means (70) implemented in the hardware layer. The system (1) also comprises a software structure (40, 50, 63) that shares hardware accelerators of the hardware layer between application containers and functions (60, 61) of the domain layer. Advantageously, the software structure (40, 50, 63) performs scheduling and optimization algorithms on the resource allocations of the hardware accelerators of the hardware layer for the application containers and functions (60, 61) of the domain layer in terms of device time and/or space slot of utilization. In detail, the scheduling

[Continued on next page]

WO 2020/234792 A1

SC, SD, SE, SG, SK, SL, ST, SV, SY, TH, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, WS, ZA, ZM, ZW.

(84) Designated States *(unless otherwise indicated, for every kind of regional protection available)*: ARIPO (BW, GH, GM, KE, LR, LS, MW, MZ, NA, RW, SD, SL, ST, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, RU, TJ, TM), European (AL, AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HR, HU, IE, IS, IT, LT, LU, LV, MC, MK, MT, NL, NO, PL, PT, RO, RS, SE, SI, SK, SM, TR), OAPI (BF, BJ, CF, CG, Cl, CM, GA, GN, GQ, GW, KM, ML, MR, NE, SN, TD, TG).

Declarations under Rule 4.17:

— *of inventorship (Rule 4.17(iv))*

Published:

— *with international search report (Art. 21(3))*

— *before the expiration of the time limit for amending the claims and to be republished in the event of receipt of amendments (Rule 48.2(h))*

and optimization algorithms comprises a monitoring structure interfacing with processing means and with the software layer for reading performance metrics of at least one processing means (70). Advantageously, the software structure comprises at least one device manager (50) component connected with the hardware interface (80, 90) and at least one remote library (63) component to interface each application container and function (60, 61) with the at least one device manager (50) component concurrently.

AN FPGA-AS-A-SERVICE SYSTEM FOR ACCELERATED SERVERLESS COMPUTING

DESCRIPTION

TECHNICAL FIELD

The present invention refers to the field of software systems. Particularly, the present invention relates to the provision of computing power as a service. In detail, the present invention proposes a system for accelerated serverless computing based upon hardware resources such as Field-Programmable-Gate-Array (FPGA), Application-Specific-Integrated-Circuit (ASIC), Digital Signal Processor (DSP) and Graphic Processing Unit (GPU) boards.

STATE OF THE ART

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 consuming resources on-demand and designing web services following a cloud-native approach is fundamental to dynamically scale performance. However, some workloads may require computing power that current CPUs are not able to provide and, for this reason, heterogeneous computing is becoming an interesting solution to continue to meet Service Level Agreements (SLAs).

Workloads such as web search [1], image processing [2], compression [3], database operations [4], neural network inference [5] and many others can benefit from the use of specialized architectures and accelerators like Field Programmable Gate Arrays (FPGAs) Application-Specific-Integrated-Circuits (ASICs), Digital Signal Processors (DSPs) and Graphic Processing Units (GPUs) to timely react to the end users requests.

Particularly, the introduction of the AWS F1 instances [6] as well as project Catapult [7] and project Brainwave [8]from Microsoft demonstrates that FPGAs will play a key role in the cloud in the next years. To exploit FPGAs at their best in the cloud, hardware accelerators should be designed to optimize latency [7].

Known FPGAs cloud environments can be classified by communication method, sharing mechanism and computational model exploited.

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 virtualizes the access to the resources. This mechanism is used by pvFPGA [9]. The API Remoting mechanisms is the most used in the analyzed state of the art [10], [11], [12], [13], 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 [14], 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 [7]. 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 as disclosed in [15], [11], [13] employs FPGA virtualization using 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 as disclosed in [7], [9], [10], [12], [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 [9], [11], [13], [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 systems such as in [7], [10], [12], [14], instead, the FPGAs are continuously working by receiving and processing requests from the system or the application. Works such as [14] directly expose the underlying FPGA accelerator as a standalone service.

Regardless of how the FPGAs cloud environment is configured, requests from the outside network can come at unpredictable rates and they usually cannot be batched, thus minimizing latency becomes fundamental. The requests unpredictability can lead to an underutilization of the FPGAs, thus reserving one FPGA for each service that needs it can result in a waste of resources.

From a cloud provider perspective, sharing becomes then fundamental to improve time utilization of the FPGA and the serverless computing paradigm can be a promising approach [16].

Serverless computing 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.

Indeed, sharing the FPGA across different services seems like an interesting idea, as many compute intensive kernels may be accessed by more than one service. Moreover, FPGA sharing is a desirable feature not only for end users looking for cloud cost savings but also for cloud providers, because they can reduce the Total Cost of Ownership (TCO) while providing an hardware acceleration service to their customers.

For example, US 10,489,195 proposes a method for FPGA accelerated serverless computing, which comprises receiving, from a user, a definition of a serverless computing task comprising one or more functions to be executed. A task scheduler performs an initial placement of the serverless computing task to a first host determined to be a first optimal host for executing the serverless computing task. The task scheduler determines a supplemental placement of a first function to a second host determined to be a second optimal host for accelerating execution of the first function, wherein the first function is not able to accelerate by one or more FPGAs in the first host. The serverless computing task is executed on the first host and the second host according to the initial placement and the supplemental placement.

Moreover, [6] discloses that for an AWS F1 instance , in order to share an accelerator in a cloud-native application it is sufficient to build a Docker container [17] on top of it and expose Application Programming Interfaces (APIs) that other containers can consume. On the one hand, this approach provides maximum performance as the container has a direct link to the FPGA through PCI Express. On the other hand, this container has to be privileged, and this represents a security issue. It is clear that, from a cloud provider perspective, exposing an FPGA sharing service requires to isolate and decouple the code provided by the user from the actual execution of the accelerator onto the FPGA, and to allocate and deallocate the devices to serverless functions automatically in order to ensure the security of the data managed by the service and avoid malfunctioning thereof.

Although the previous works in the literature provide solutions to share consistently an FPGA across multiple workloads, to the best of the knowledge of the Applicant, a systematic way to manage FPGAs in a truly as-a-Service fashion is still missing, and this is key to achieve accelerated serverless computing.

Within this context, the Applicant proposes that compute-intensive kernels should be accelerated with shared FPGAs handled transparently by the serverless infrastructure: this will maximize utilization while reaching near-native execution latency.

Particularly, in order to exploit FPGAs in cloud-native workloads, kernels should be able to quickly react to unpredictable and spiky requests coming from the external network. For this reason, cloud native accelerators should be optimized also for latency and not only of throughput. On the contrary, FPGAs allocated exclusively to a single cloud service (typically hosted on a Virtual Machine (VM)) will often face idling periods and, in general, low utilization rates.

SUMMARY OF THE INVENTION

It is an objective of the present invention to overcome the drawbacks of the prior art.

Particularly, it is an object of the present invention to propose a serverless system for managing the provision of accelerated computing in an efficient manner.

It is a further object of the present invention to propose an accelerated computing serverless system

configured to share resources at the hardware layer - such as Field-Programmable-Gate-Array (FPGA), Application-Specific-Integrated-Circuits (ASIC) and Graphic Processing Unit (GPU) boards - among a plurality of function requiring acceleration.

A further object of the present invention is to propose an Accelerated Computing-as-a-Service, preferably a FPGA-as-a-Service, infrastructure that efficiently isolates accelerated kernels and provides a transparent way to access them remotely through a framework for writing programs that execute across heterogeneous platforms, such as OpenCL.

In a non-limiting manner, such an infrastructure provides for:

• a distributed architecture that supports serverless computing with FPGAs, ASICs and/or GPU along with a sharing mechanism (in time and space) able to increase the utilization of the accelerators and minimize the latency overhead w.r.t. native execution;

• an abstraction layer that isolates and decouples the physical accelerators from their CPU code, avoiding privileged access to the FPGAs, ASICs and/or GPU and enabling transparent offloading of compute-intensive tasks without code rewriting, and

• a central and device-level allocation and scheduling mechanism and components to enable functions autoscaling in the proposed serverless system

Particularly, it is an object of the present invention to propose a system for serverless accelerated computing - preferably FPGA-based. The system is substantially composed of a central component (i.e., Accelerator Registry), which integrates with an orchestrator to provide automatic allocation of devices and function instances. In addition, other components (i.e., Remote Library and Device Managers) are designed to isolate the user code from the actual hardware acceleration. The system is able to employ FPGA accelerator sharing in an isolated, automatic and transparent way.

These and further objects of the present invention will be more clear from the following description and from the annexed claims, which are an integral part of the present description.

According to a first aspect, the invention therefore relates to a hardware accelerators management system for containerized and serverless environments. Such a system at least comprises:

a domain layer on which a plurality of application containers and serverless functions are implemented,

a hardware layer in which a set of hardware accelerators are implemented and

a software layer configured for abstracting the application containers and the serverless functions of the domain layer from the hardware layer.

Moreover, the system comprises:

a hardware interface to send tasks to and reconfigure at least a portion of the processing means implemented in the hardware layer,

a software structure that shares hardware accelerators of the hardware layer between application containers and serverless functions of the domain layer.

Advantageously, the software structure performs scheduling and optimization algorithms on the resource allocations of the hardware accelerators of the hardware layer for the application containers and functions of the domain layer in terms of device time or space slot of utilization,

Particularly, the scheduling and optimization algorithm comprises a monitoring structure interfacing with processing means and with the software layer for reading performance metrics of at least one processing means.

Preferably, the software structure comprises at least one device manager component connected with the hardware interface and at least one remote library component to interface each application container and function with the at least one device manager components concurrently.

Thanks to such a system it is possible to efficiently isolate accelerated kernels and provides a transparent way to access them remotely through a framework for writing programs that execute across heterogeneous platforms.

In an embodiment, the hardware interface is configured to communicate with at least a portion of the processing means of the hardware layer, including Field-Programmable-Gate-Arrays (FPGAs), Application-Specific-Integrated-Circuits (ASICs), Digital Signal Processors (DSPs) and Graphic Processing Unit (GPUs).

In an embodiment, the at least one remote library is configured for receiving method calls performed by an application and/or a function implemented in the domain layer and forward such method call in an asynchronous manner to a service endpoint exposed by the at least one device manager.

In an embodiment, the at least one device manager receives a plurality of method calls that requires hardware accelerators to be performed from the at least one remote library associated with a corresponding application and/or function.

Moreover, the at least one device manager is configured to:

- create at least one task, the at least one task comprising a minimum sequence of called method to be performed in a predetermined order, and

- forward the at least one task to the hardware interface.

In an embodiment, the at least one device manager is configured to sequentially adding method to be performed in the at least one task until a blocking method or an explicit finish/flush/barrier command is added.

In an embodiment, the at least one device manager is configured to inserting the at least one task queue once created.

Advantageously, the device manager further comprises at least one worker thread configured to pull and execute on the hardware accelerator tasks comprised in the task queue.

In an embodiment, the worker thread is configured to select which task pull from the task queue based on at least one of the following metrics associated with the hardware accelerator:

- number of requests received/executed by the device,

- number of in-flight requests,

- allocated memory,

- number of allocated buffers,

- hardware accelerator utilization,

- number of connected applications and instances.

In an embodiment, the device manager component interfaces with the hardware interface to send multiple tasks in parallel to different hardware accelerators and/or to reconfigure the processing means.

In an embodiment, a respective remote library is implemented in each application container or function implemented in the domain layer.

In an embodiment, the at least one device manager and the at least one remote library are configured to communicate via a network connection. Alternatively or in addition, the at least one device manager and the at least one remote library are configured to communicate via a shared memory area of the hardware layer on which is deployed the software layer implementing both the device manager and the at least one remote library.

In an embodiment of the invention, the at least one device manager and the at least one remote library are configured to are configured to expose at least one of the following service:

- application containers and functions registration and disconnection;
- hardware accelerator information gathering;
- reconfiguration requests;
- buffers manipulation;
- accelerator-related methods, and
- command queue operations.

In an embodiment, the software structure may comprise a central management component

interfacing with the at least one device manager and the at least one remote library components to perform scheduling and optimization algorithms on the resource allocations of the hardware accelerators of the hardware layer.

In an embodiment, the central management component is configured to:

- receive request of instantiation of functions and/or applications, and

for each function or application:

- assign a domain layer resource for instantiating the function or application, and
- assign at least one device manager to the function or application, the device manger interfacing with the hardware interface associated with a hardware accelerator requested by the function or application.

In an embodiment, the scheduling and optimization algorithms performed by the central management component use system runtime performance indicators to efficiently allocate the resources of the hardware layer to the application containers and functions, such runtime performance indicators comprising at least one among:

- number of requests received;
- number of in-flight requests;
- allocated memory, and
- current workload,

related to the hardware layer, one or more hardware accelerators implemented in the hardware layer or one or more processing means of the hardware layer.

In an embodiment, the at least one device manager and the central management component are configured to communicate via a network connection to exchange network message/methods called comprising at least one among:

- hardware accelerator registration and removal from the central management component;
- reconfiguration request and validation of the hardware accelerator;
- metrics pushing from the device manager to the central management component, and
- periodic polling from the device manager to the central management component and *viceversa.*

In an embodiment, the at least remote Library and the central management component are configured to communicate via a network connection to exchange network message/methods called comprising at least one among:

- application containers and functions registration and removal from the central management

component;

- Instances of the application containers and functions registration and removal from the central management component, and
- hardware accelerator reconfiguration request and validation.

Further characteristics and advantages of the present invention will be more clear from the following detailed description of some preferred embodiments thereof, with reference to the annexed drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

The invention will be described here below with reference to some examples provided by way of example and not as a limitation, and shown in the annexed drawings. These drawings show different aspects and embodiments of the present invention and, where appropriate, reference numerals showing like structures, components, materials and/or elements in different figures are indicated by like reference numerals.

Figure 1 is a schematic overview of the system components (without connections between them);

Figure 2 is a block diagram of a single node diagram of the system with connections between components: external API gateway and cloud orchestrator, Accelerators Registry, Functions instance (integrated with the Remote Library) and Device Manager (connected to the FPGA board);

Figures 3 shows three block diagrams representing different configurations for a node: (a) Single shared FPGA with multiple function instances; (b) Multiple FPGAs from the same vendor, used by one or more instances, and (c) Multiple FPGAs from different vendors, used by one or more instances (one instance may use FPGAs from multiple vendors at the same time);

Figure 4 is a schematic diagram of Device Manager Architecture, showing part of the Actions performed, which are numbered as such: 1 - Request received by the network service; 2 - Context and information related methods; 3 - Grouping of operations into tasks; 4 - Task execution on FPGA, and 5 - Execution notification to the service implementation;

Figure 5 is a flowchart of operation grouping in tasks on Device Manager;

Figure 6 is a schematic diagram of OpenCL Remote Library Architecture, numbered steps are described in Section F of the detailed description;

Figure 7 is a flow sequence diagram of Function Deployment;

Figure 8 is a logarithmic graph showing average latency measurements at increasing requests/sec of Sobel on CPU, FPGA and with our shared library for different numbers of instances sharing the same host and board - lower is better;

Figure 9a is a logarithmic graph showing average latency measurements for BFS executed on

CPU, FPGA with and without the proposed shared library for different numbers of instances sharing the same host and board, and at increasing requests/sec sent to BFS; benchmark saturation depends on network bandwidth - lower the better;

Figure 9b is a logarithmic graph showing average latency measurements for BFS executed on CPU, FPGA with the proposed shared library for different numbers of instances sharing the same host and board, and at increasing requests/sec sent to BFS, isolating kernel execution network data movement - lower the better;

Figure 10a is a logarithmic graph showing average latency measurements for MM executed on CPU, FPGA with and without the proposed shared library for different numbers of instances sharing the same host and board, and at increasing requests/sec sent to the MM application; benchmark saturation depends on network bandwidth - lower is better,

Figure 10b is a logarithmic graph showing average latency measurements for MM executed on CPU, FPGA and with our shared library for different numbers of instances sharing the same host and board with and without the proposed shared library for different numbers of instances sharing the same host and board, and at increasing requests/sec sent to the MM application, isolating kernel execution network data movement - lower is better;

Figure 11a is a linear graph showing the latency overhead w.r.t. input data size for R/W operations for systems according the present invention and a system known in the art;

Figure 11b is a linear graph showing the latency overhead w.r.t. input data size for sobel operator for systems according the present invention and a system known in the art, and

Figure 11c is a linear graph showing the latency overhead w.r.t. input data size for MM kernel for systems according the present invention and a system known in the art.

DETAILED DESCRIPTION OF THE INVENTION

While the invention is susceptible of various modifications and alternative constructions, some preferred embodiments are shown in the drawings and will be described in details herein below. It should be understood, however, that there is no intention to limit the invention to the specific disclosed embodiment but, on the contrary, the invention intends to cover all the modifications, alternative constructions and equivalents that fall within the scope of the invention as defined in the claims.

The use of “for example”, “etc.”, “of” denotes non-exclusive alternatives without limitation, unless otherwise noted. The use of “includes” means “includes, but not limited to”, unless otherwise noted.

A. System Overview

An overview of the proposed serverless system - simply indicated as 'system 1' in the following for the sake for brevity - can be seen in Figure 1, which highlights the main components of the system

1. A simple diagram of the system 1 (with a single node 10, while the system 1 may be composed of multiple nodes) with also the connections between the different components is shown in Figure 2. The described system 1 diagrams are intended to be a generic structural view of the system, as it might be extended according to future needs and additional components. Here we provide a brief overview of each component, while a detailed description is given in the next subsections.

The API Gateway 20 and Cloud Orchestrator 30 components are per sé known in the art and do not form a part of the present invention and are generally needed to connect, orchestrate and, in general, manage the system 1 according to the present invention as known in the art, apart for the differences described below. These known components may be custom-developed or taken from an existing external system/product with the needed features.

In detail, for API Gateway 20 we mean any component able to receive and forward requests (in a given network protocol such as HTTP/S and others) to the deployed function instances. The API Gateway 20 is the component of the system 1 exposed to an external network, and must operate securely and without impacting on the performance of the requested services.

Moreover, for Cloud Orchestrator 30 we mean any component able to schedule and allocate function instances and other components in the given serverless system 1 (which could be a bare-metal cluster, a set of provisioned VMs both in a cloud environment and/or on-premise, a set of containers in a cloud environment and/or on-premise or other) based on the user requests and other factors. Thus, the Cloud Orchestrator 30 includes one or more existing orchestrators for cloud systems (e.g. Kubernetes [18], Docker Swarm and others) and existing or custom serverless systems (such as OpenFaas [19], Kubeless and others). The Cloud Orchestrator 30 may include an autoscaling mechanism to deploy and deallocate function instances based on the number of requests or other factors. Moreover, the Cloud Orchestrator 30 exposes an interface and integrates with an Accelerator Registry 40 in order to allow a more precise allocation and scheduling of the other components (i.e., Device Manager 50 and Function Instances 60) for function initialization, autoscaling and deallocation.

For Function we mean the definition of a component - also indicated as 'serverless function'[1] in the following - used to respond to an incoming request - e.g. from a client entity connected to the system - or exposing a serverless service. As such, a Function can be declared as a snippet of code - e.g., implemented as Function Code 61 in the example of Figures 1 and 2 - used as a handler to manage incoming requests to an endpoint of the serverless architecture integrated with the proposed system 1 as described in the following, or a standalone application exposing a service to an internal network of the system 1 (i.e., connecting the various components of the system) and/or an external network (i.e., connecting the system 1 with external entities). The generic function may or may not require the use of a single accelerator, multiple accelerators or none, using a single FPGA 70 board or multiple boards. Particularly, the term 'accelerator' is herein used to

identify a piece or a portion of a hardware accelerator - e.g. the FPGA 70 board or a portion thereof in the example at issue - configured to perform a specific computation, typically labor-intensive.

In order to serve the user requests in a scalable way, the system 1 can instantiate from one to multiple Function Instances 60.

With Function Instance 60, we mean a specific instance of the Function implementation which serves a same serverless service or endpoint of the system 1 in a load-balanced manner, in order to increase the system 1 scalability. Thus, the Function Instance 60 may be represented by a standalone application, a VM or an application container including the Function implementation. The Function Instance 60 is the component that directly connects to the Device Managers 50 in order to access the shared accelerator in the proposed system 1 - e.g. an FPGA 70 in the example at issue. The connection between the Function Instance 60 and the Device Manager 50 or the central registry - i.e., the Accelerator Registry 40 - is done through the Remote Library 63, which abstracts the remote accelerator configuration and access. The Remote Library 63 should expose different interfaces to access the accelerator - e.g., such as an interface exploiting OpenCL [20] version 1.2 or higher or other languages suitable for the purpose.

In addition, the Remote Library 63 allows accessing multiple accelerators in a vendor-independent way or using vendor-specific functionalities if supported by the system 1.

The Device Manager 50 is in charge of managing the FPGA device 70 and the related runtime, and it is the only component that can directly communicate with the accelerator - i.e., the FPGA 70 - and control it. The Device Manager 50 is also the component that allows sharing the FPGA 70 with multiple functions and accelerators - i.e., a FPGAs 70, other hardware resources or portions thereof configured to perform a specific task - , thanks to its underlying sharing runtime. The Device Manager 50 may employ both time and space sharing, as explained in Section B. In both cases, the Device Manager 50 will arrange operations requested by a Remote Library 63 in higher-level tasks and use a device-level scheduler to decide which tasks to run on the board - i.e., the FPGA 70 - at a given time (or which different accelerator to configure). The Device Manager 50 also integrates with the Accelerator Registry 40 to manage the different priority policies to employ. Moreover, the Device Manager 50 integrates with the Accelerator Registry 40 to decide which operation requested by the Functions implemented via a corresponding Remote Library 63 to accept, register and remove at a given time, based on the system 1 performance and other factors (such as autoscaling operations which involve function instances).

A respective Device Manager 50 is deployed on every node 10 (for node we mean a physical or virtual machine with a FPGA 70 connected to it via a PCI express link or other means) for each distinct FPGA 70 connected to such node 10. The Device Manager 50 exposes a service to access functionalities of the respective FPGA 70.

Finally, the Accelerator Registry 40 is the central component of the system 1. Its role is to manage Functions, Function Instances 60 and devices - i.e., FPGAs 70 and/or CPUs comprised in the nodes 10 of the system 1, along with allocating devices to Functions implemented and, in general, managing the system 1 state.

The Accelerator Registry 40 component integrates with the Cloud Orchestrator 30 in order to transparently allocate resources –e.g., computing power, CPUs, FPGAs and GPUs - and schedule the deployment, scaling and deallocation of Function Instances 60. In fact, the Accelerator Registry 40 employs a component that continuously checks the runtime metrics of the system and dynamically allocates and deallocates devices 70 and Function Instances 60 based on those metrics. All the high-level components of the system 1 (the Remote Library 63 in Function Instances 60, the Device Manager 50 and the Orchestrator 30) communicate with the Accelerator Registry 40 in order to expose their functionalities.

In summary, the Accelerator Registry 40, the Device Manager 50 and the Remote Library 63 provide a software structure implemented at a software layer of the system 1 that allows efficiently and transparently allocating hardware resources - both accelerator and CPUs computing power - of a hardware layer of the system 1 to any Functions - applications instantiated in respective containers, serverless functions, etc. - implemented at a domain layer of the system 1.

B. FPGA Sharing method

The system 1 supports FPGA sharing in multiple ways. Here for sharing we mean that an FPGA 70 may be used by multiple Functions implemented in the system 1 - e.g., applications deployed in corresponding Function Instances 60- in different times or at the same time, with a single or multiple accelerators configured on the same FPGA 70.

A first sharing method is called time sharing, meaning that the FPGA 70 is programmed with a single accelerator and the configuration - i.e. is used by one Function Instance 60 at a time. An accelerator is composed of a single kernel or a set of kernels, which are used by the same Function Instance 60 for a common goal (e.g. machine learning inference, data compression, and other accelerated computations).

Time sharing itself can be implemented in the system in multiple ways. A first option is to program the FPGA 70 board with a single accelerator, and interleave the execution of operations and tasks by different client applications/serverless functions - implemented in Function Instances 60 - in time by switching between them with a given policy. The policy can be round-robin, priority-based, profiling-based or any other policy suitable for the system. In addition, the policies may be user-defined, meaning that the system should accept external policies defined in a formal way (e.g. by means of math or logic formulas, algorithms or external services/functions). A second option consists in using a different accelerator for each external implemented Function that needs it, and

interleaving the execution of tasks on the different bitstreams by reconfiguring the FPGA 70 board continuously. This option, like the first one, involves the use of a policy for the interleaving mechanism.

Another approach to the sharing problem that may be included in the system 1, in addition or as an alternative to the time sharing, is called space sharing. Space sharing consists in splitting the reconfigurable fabric of the FPGA 70 among multiple accelerators (and kernels inside the accelerators), thus allowing the concurrent and parallel usage of them by the users.

The split can be done by employing Partial Dynamic Reconfiguration (PDR) or by merging the accelerators in a single bitstream (called Overlay) or any other technique suitable for the system 1. In this way, the system 1 may expose multiple accelerator for the same board 70, in addition to the time sharing mechanism used for each accelerator.

Alternatively, the creation of a multi-accelerator bitstream (or a partial reconfigurable area) can be done based on size and performance information for each kernel, without the use of PDR. This mechanism may require the system 1 to be in possession of the source code (or the design files) of the different accelerators, in order to estimate performances and sizes and to perform the actual sharing (e.g. by merging the designs or tuning them in order to fit into the reconfigurable areas of the FPGA 70).

C Communication protocol

All the components of the system 1 must be able to communicate through a network connection (stylized by arrows in Figure 2) either virtual or physical depending on the components and their deployment. The network protocol used may be different for each distinct connection between components, but the type of messages/methods sent between components is fixed and described below.

In particular, the specification of a network service between the Device Manager 50 and the Remote Library 63 used by the generic Function instantiated and associated with the considered Remote Library 63 - includes:

- Function Instance 60 registration and disconnection;
- Platform/Device information gathering;
- Reconfiguration requests (with bitstream hash, accelerator name or other means, including sending the entire binary);
- Buffers manipulation (create/write/read/release);
- Accelerator-related methods (create, run, set arguments, release), and
- Command queue operations (e.g. flush of current operations).

The network message/methods called between the Device Manager 50 and the Accelerator Registry 40 includes:

- Device registration and removal from the Accelerator Registry 40;
- Reconfiguration request and validation;
- Periodic metrics pushing from the Device Manager 50 to the Accelerator Registry 40;
- Periodic polling from the Device Manager 50 to the Accelerator Registry 40 and *viceversa,* and
- Other commands to be executed by the Device Manager 50.

The network message/methods called between the Remote Library 63 (including the deployed function as a whole or its distinct instances) and Accelerator Registry 40 includes:

- Function registration and removal from Accelerator Registry 40;
- Function instance 60 (e.g. executable/container/VM) registration and removal from Accelerator Registry 40
- Reconfiguration request and validation (pre-Device Manager)
- Other polling messages and commands to be executed by the Function Instance 60 or by the Accelerator Registry 40.

These lists represent the minimum set of interfaces and services to be exposed in the system 1, but each interface/service may be extended based on future requirements and extensions of the system 1, and new interfaces and services may be added. In addition, the communication may happen using any compatible network protocol and framework (e.g. TCP/UDP, UNIX sockets, message passing or RPC/gRPC protocols).

In an alternative embodiment, the Device Manager 50 and the Remote Library 63 deployed on a same node 10 can communicate - i.e., expose services one another - by exploiting at least a shared memory area of a memory component (either physical or virtual) of the node 10. This improves performance and reduces additional data copies (from four to one w.r.t. network protocols such as gRPC) at the cost of having the Function instance 60 together with the Device Manager 50 on the same node 10 with permissions to create and/or access in read/write the shared memory area. When the system 1 is implemented according to OpenCL, one data copy is required to maintain full OpenCL compatibility, as a direct access to the shared memory area would require to define additional functions not available in the OpenCL specification.

For example, the Device Manager 50 may be configured to employ gRPC if the Function is not instantiated on the same node 10 - or if it is not possible to create a shared memory area - and, conversely, employ a shared memory area whether the Function is instantiated on the same node 10. 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, shared memory is preferable.

D. Node

A node 10 of the system 1 is represented - i.e., is implemented - by a computing device, which should include a uniprocessor or multiprocessor system (e.g., single/multicore, with any ISA such as x86, PowerPC, SPARC, MIPS or any other suitable system). The node 10 includes components such as memory, network interfaces, storage devices and computational devices (such as GPUs, FPGAs and processors), along with other devices needed by the system. Figure 3 shows how the system components may be located on a given node 10. In particular, the system 1 may include a single FPGA 70 (Figure 3 a)) or multiple FPGAs 70 (Figure 3 b) and c)), from the same vendor or from different vendors. According to an embodiment, the Device Manager 50 is preferably associated with a corresponding FPGA 70 - even though nothing prevents from associating a Device Manager 50 with multiple FPGAs 70 in different embodiments of the invention (not shown in the drawings). Finally, one or multiple Function Instances 60 may be instantiated onto the same node 10, each connecting to one or multiple Device Managers 50, and each Device Manager 50 receives connections and requests from one or multiple Function Instances 60, in this way allowing the sharing of the underlying FPGA 70 to which the device manager 50 is connected.

The runtime 80 component shown in Figure 3 is a software component that allows accessing the FPGA 70 functionalities from the host system - i.e., the client of the serverless service. The runtime 80 component may be a standard interface shared by multiple vendor drivers - as shown in Figure 3 b) - or may be diversified for each different FPGA 70 and FPGA vendor - as shown in Figure 3 c). The Vendor Driver 90 is a kernel-level software component that directly manages the physical signals between the FPGA 70 and the system processor and memory of the node 10, exposing an API to be used by the runtime 80 component.

£ *Device Manager*

The Device Manager 50 is the component that makes direct use of the FPGA 70 board. In fact, for each board present in the cluster, the system 1 runs an instance of the Device Manager 50 to control and use such FPGA 70. The Device Manager 50 exposes the network services described previously in Section C. The service implementation is integrated with the Device Manager 50 logic and it uses directly the underlying components - i.e., the runtime 80 and the Vendor driver 90 - to control the FPGA 70. The FPGA 70 may be accessed through different means, for example the vendor specific OpenCL library, a board device driver or any other compatible component.

Figure 4 shows the key features and components inside the Device Manager 50, along with the flow followed by operations that are to be executed on the FPGA 70. The first component of the Device Manager 50 is a service endpoint 51, which receives all the method calls by the Function

Instance 60 - i.e., which are provided by the Remote Library 63 as described in the following section. Method calls are forwarded to a Service Implementation 53 (circled number 1 in Figure 4), which is a multithreaded component with a set of worker threads, of which only a single worker thread 55 is shown in Figure 4, managed by the network protocol implementation that operates as described in the following.

In the considered embodiment, there are two kinds of methods exposed by the Service Implementation 53: context and information methods and command-related methods, as should be clear to the skilled person further methods type can be added depending on the computing language and runtime support of the proposed system 1. In the present disclosure, the term 'method' generally refers to an operation that has to be performed by a component of the hardware layer of the system 1, e.g. a processing element such as a CPU of the node 10 in which the Device Manager 50 is implemented and/or a FPGA 70 configured to provide one or more accelerators to which the Device Manager 50 is connected. For example, methods can be defined according to OpenCL specification or any other suitable computing language.

The context and information methods are executed synchronously as they do not require the execution of one or more tasks by the FPGA 70 (circled number 2 in Figure 4). More generally, all the methods which do not involve the FPGA 70 directly (e.g. platform, device, program and arguments information gathering, release of objects) may be executed synchronously and without concurrency issues. There may be exceptions, such as the programBoard operation in OpenCL, which is used to modify the configuration of the FPGA 70. In fact, reprogramming the FPGA 70 board with a given bitstream could block the execution of other operations in order to be performed, and it may require to synchronize with other methods, thus creating concurrency issues.

The command-related methods, instead, should be executed in a predetermined order (from the single Function point of view - in other words, the order has to be maintained to have the called methods executed as intended and have the Function instance 60 receiving a correct output) and require the use of the FPGA 70 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 50, the results of the execution will change, thus breaking the Instantiated Function consistency.

The Device Manager 50 employs procedures to avoid conflicts between kernels and/or accelerators, by either employing a single Task Queue 57 - as shown in Figure 4 - , or by using other methods to ensure the commands order, consistency and synchronization. On the Function instance 60 side, the system 1 virtualizes the creation and use of command queues - in case of OpenCL - (or other components depending on the computing language used) based on the client identifier, or Function identifier, obtained when registering to the system 1.

Advantageously, inside the Device Manager 50, i.e. by the Service Implementation 53 component,

multiple operations - i.e. methods - from a same starting point (e.g. a Function Instance 60, a command queue implemented in a serverless function or application, etc.) may be grouped together in atomic tasks (circled number 3 in Figure 4). An atomic task consists of one operation or multiple in-order operations, such as buffer read/write or kernel execution. In the case of multiple in-order operations, the atomic task comprises the minimum set of operations that have to be performed in-order to obtain a useful output from the FPGA 70.

A process 500 of creation of the generic atomic task, according to an embodiment of the present invention, is schematically shown in the flowchart of Figure 5.

Each operation sent by the Function Instance 60 and received at the service endpoint 51 is added to an object 531 of the Service Implementation 53 in the Device Manager 50 containing the operations that are being combined in a corresponding atomic task and atomic tasks that have yet to be flushed and executed by the system 1 - i.e., not yet forwarded towards the FPGA 70 - (block 501).

Particularly, operations - i.e. methods - from a same Function Instance 60 are inserted in an atomic task associated with such client (block 503).

Operations received are checked in order to detect a flush command - either by calling a blocking method or an explicit finish/flush/barrier command - (decision block 505), and as long as no flush command is detected (exit branch N from decision block 505) the process 500 is reiterated starting from the step described above with respect to block 501 .

Otherwise, when the a Function Instance 60, sends such a flush command (exit branch Y from decision block 505), the current task is sent to a Task Queue 57 of the Device Manager 50 (block 507) in order to be forwarded to the FPGA 70. Afterwards, a new (empty) task is generated to receive further methods calls, by reiterating the process 500 starting from the step described above with respect to block 501 . This mechanism - i.e., the creation of atomic tasks - avoids conflicts between different clients and lowers the resource usage on the board 70 (e.g., memory buffers can be deallocated while not used or when the data has to be overwritten by the next task).

In one embodiment, the process could be extended in order to not require explicit flushing from the client-side, by adding a speculation mechanism in order to flush tasks based on their structure or past history of the methods called - and their sequence - by the implemented Functions.

Once the atomic task arrives to the Task Queue 57 of the Device Manager 50, one or multiple worker threads 55 pull and execute the atomic task on the FPGA 70 (circled number 4 in Figure 4). The policy used to decide which atomic task to execute may be included in the Device Manager 50, or decided by the system 1 through the Accelerators Registry 40. In addition, the policy may be based on the system 1 performances and other relevant metrics related to each registered Function, the corresponding one or more Function Instances 60 or the FPGA 70 device in use.

Each operation of the executed atomic task is linked with an asynchronous event and when the method is done the corresponding event is notified to the method caller - i.e., the corresponding Function Instance 60 that requested the operation - (circled number 5 in Figure 4). In this way, the client is notified punctually, even if the methods are executed in groups, i.e. the atomic tasks generated by the Device Manager 50. It should be obvious to the skilled person that, in addition to the software scheduling employed by the Device Manager 50, an additional component may be synthesized in hardware to perform tasks and operations scheduling on the FPGA 70 board.

A further feature of the Device Manager 50 is the ability to expose device-related metrics - e.g., metrics related to the FPGA 70 - to the system 1 - particularly, to the Accelerator Registry 40 and/ the Remote Library 63 - , which may be used by the system 1 to enhance the scheduling and allocation performances, or to monitor the state of each device and component of the system 1.

A metric is defined as a measurement regarding a particular component of the system 1, which is of interest to the system itself or other systems. Possible metrics exposed by the Device Manager 50 include the number of requests received/executed by the device, number of in-flight (not yet executed, but flushed) requests, allocated memory, number of allocated buffers, device utilization (in terms of space and time slice/percentage), number of connected Function Instance 60 and all other possible runtime and offline information about the device, the configured accelerators or other connected components.

Also, the metrics may be related to the actual physical device, the node 10 on which the Device Manager 50 is, or for each distinct accelerator configured on the device, i.e. the FPGA 70 in the example at issue. Each metric may be represented by a single number, a sum, a percentage, a vector or any other needed data type. The metrics may be exposed by the Device Manager 50 with different mechanisms, such as Push (the Device Manager 50 sends the metrics to a specified endpoint), Pull (the Device Manager 50 exposes a service that external components might contact to grab metrics) or event-based. Finally, the Device Manager 50 may store the metrics in memory or on storage devices, either locally or remotely, for a needed period of time.

F. Remote Library

In order to integrate client applications and serverless functions with the system 1, and to isolate them from the actual execution thereof in the implemented hardware acceleration, the Remote Library 63 component is deployed in the system 1 according to the embodiments of the present invention.

In particular, the Remote Library 63 implements most of the methods used to control an accelerator implemented on the FPGA 70, without presenting the Function implemented in the Function Instance 60 lower level details - for example the FPGA 70 board position and the sharing mechanisms included into the system 1 - along with other details not directly connected to the

accelerator usage. A particular feature of the Remote Library 63 is that it can be easily integrated with any external user component (which can be a client application, a serverless function or other) by overwriting the already installed library, or by adding Remote Library 63 to the file-system and setting the LD_UBRARY_PATH variable accordingly. If the user code - e.g. the Function code 61 in the Function instance 60 - is linked dynamically, this allows integrating the user code with the system 1 without changing any line of code.

In an embodiment, the Remote Library 63 interface mimics the OpenCL specification, as it represents the most general interface to communicate with heterogeneous accelerators (including FPGAs) and is supported by the main vendors on the market. The Remote Library 63, may also implement other methods in addition to the OpenCL specification, in order to meet additional requirements of the system (e.g. the possibility to use a central bitstreams/accelerators repository instead of using bitstream or kernel files directly), or to use vendor specific functionalities.

We will now provide a description of the main mechanisms and components included in the Remote Library 63.

The Remote Library 63 implements a central router component - not shown in the Figures -, which acts as a singleton, and it is responsible for keeping a list of the available platforms and devices. When created, the router component connects to the Accelerator Registry 30 to obtain the allocation of the devices - i.e., the node 10 or the FPGA 70 - to the client, along with a list of addresses of the Device Managers 50 to which it must connect. If the Accelerator Registry 30 is not available (e.g. in local or testing scenarios) the router can connect directly to a given Device Manager 50 address. For state-changing operations (i.e., any method call that does not simply entails information - e.g., metrics - as output) the Remote Library 63 calls the registration method exposed by the Device Manager 50 to ensure that the connection is established and that the Device Manager 50 has reserved the needed resources for the Function Instance 60.

Moreover, in case the connection to the Accelerator Registry 30 or any one of the Device Managers 50 is lost, the Remote Library 63 should either crash the Function code 61, or try to restore the connection and the previous state of the virtualized device or devices.

After having checked the registration, the proper method is called based on the upper-level call by the Function Code 61. The system 1 allows for both synchronous/blocking calls to the Device Manager 50 and asynchronous/non-blocking calls. Both the synchronous and asynchronous flows rely on asynchronous events.

An event in the system 1 is composed by a set of subsequent asynchronous calls to the network services provided by the Device Manager 50 - or alternatively, by accessing the shared memory, a state machine of the Remote Library 63 - not shown in the drawings - is configured to control the steps that each event must follow and a status which is updated while the event is processed. In

this way, the Remote Library 63 supports event polling, which may be required by some interfaces (e.g. clWaitForEvents, clGetEventInfo in the standard OpenCL specification).

The asynchronous flow steps are shown in Figure 6, which shows the main components of the Remote Library 63. For each distinct connection created with the Function Code 61 by means of a Library interface 633, the Remote Library 63 starts a thread, which uses a Completion Queue 635 filled with responses from the network service - i.e., provided by a corresponding Device Manager 50 of the node 10.

To perform a method call received from the Function code 61 at the Library Interface 631 (circled number 1 in Figure 6), the Remote Library 63 creates an event the Asynchronous Event Object 637 (circled number 2 in Figure 6).

A first asynchronous call is also created by the Remote Library 63 - at the Asynchronous Event Object 637 - and forwarded to the network service on the corresponding Device Manager 50 - by means of a service endpoint 639 (circled number 3 in Figure 6).

The first asynchronous call encapsulates a tag to identify the newly created event associated thereto. Whenever the Device Manager 50 responds - as described in section E - , the network runtime pushes the sent event identifier into the Completion Queue 635 of the Remote Library 63, where the event identifier is associated with the associated method call previously forwarded to the Device Manager 50 (circled number 4 in Figure 6).

Then, an Events Thread 639 pulls the event identifier - i.e., the tag - and retrieves the corresponding event (circled number 5 in Figure 6). The Event Thread calls the event state machine, which performs the needed operations and updates its state and the event status (circled number 6 in Figure 6). Finally, the application is notified when the event changes the status (circled number 7 in Figure 6).

For example, to perform a read buffer function, the event state machine contains four states (INIT, FIRST, BUFFER, COMPLETE). The INIT state sends the call metadata (buffer size, buffer id, offset); the FIRST step waits for the command to be enqueued by the Device Manager 50; the BUFFER step actually sends the buffer data when the Device Manager 50 is available, and the COMPLETE step signals the call completion. The states in the event state machine depend on the different kind of event created, and may be different from the ones described in the previous example.

G. Accelerator Registry

The Accelerator Registry 40 is the main component of the system 1. Its role is to manage and allocate both devices - e.g. one or more nodes 10 and/or FPGAs 70 in the example at issue - and user applications/serverless functions implemented in the system 1. The Accelerator Registry 40 is implemented as a standalone service, either in form of a native application running over the

underlying operating system, or as a containerized or virtualized component. All the other components of the system 1 may require connecting to the Accelerator Registry 40 in order to be work as intended.

We will now list and describe the functionalities of the Accelerator Registry 40:

- Contains information regarding all devices - i.e., one or more nodes 10 and FPGAs 70 in the example at issue –, client applications/functions - requesting services - and Function Instances 60 - deployed in the one or more nodes 10 of the system 1;
- Maps every Function instance 60 with the requested devices;
- Performs scheduling and allocation of devices - i.e. CPUs and FPGAs 70 in the example at issue - to Functions to be instantiated and/or Function Instances 60;
- Validates and allows the reconfiguration of devices based on different conditions;
- Gathers and aggregates metrics from the system devices and Functions Instances 60;
- Allows the runtime registration and removal of devices and Function Instances 60, and
- Integrates with the API of the Cloud Orchestrator 30 and the system 1 in general.

First, the Accelerator Registry 40 is responsible for the devices and Functions and Functions Instances 60 registration and removal at runtime. In fact, it contains multiple lists for all the other components of the system 1 and their information.

Example of information needed for the Device Managers 50 are the device path, type, vendor, current bitstream hash or complete binary, device manager address, node etc. With respect to the Functions and Function Instances 60 in the system 1, the Accelerator Registry 40 may store details about the node, address, required device, required kernel etc.

Furthermore, the Accelerator Registry 40 exposes on the network multiple endpoints (not detailed in the Figures) which allow the other components to register, update and remove their status throughout their lifetime in the system 1. Each system 1 component establishes a connection (directly or indirectly if possible) to the Accelerator Registry 40 at startup, in order to send an initial status. In fact, each component sends periodic updates to the Accelerator Registry 40 if needed, and removes its information when terminated (for example, when a Function Instance 60 is terminated by the user or to handle errors at runtime, or when it is moved on a different node 10 and must be reinitialized). The components information and registration are needed in order to perform online scheduling and allocation of the devices to fill subsequent requests, and to disconnect or move user functions and instances when needed (for example when the given device is reconfigured).

The Accelerator Registry 40 performs online scheduling and allocation of devices in the system at

runtime. Each Device Manager 50 registers itself with the Accelerator Registry 40 at the beginning of its execution, sending information about the managed device - e.g. one or more of the FPGAs 70 in the example at issue - (e.g. device identifier, vendor, hardware version). When a Function is created in the system 1 - i.e., a client forwards to the system 1 a function to be executed-, such Function is created along with a request for the Accelerator Registry 40. The Accelerator Registry 40 receives the Function instantiation request, checks it against multiple conditions, and selects the one or more nodes 10 on which the corresponding one or more Function Instances 60 may be allocated and scheduled. The conditions are related to the particular device or kernel/accelerator needed by the function, and may include its vendor, id, type, configured accelerator and in general any information of the device which may be stored in the Accelerator Registry 40, with a partial or complete match. Then, for each Function Instance 60 created, the Accelerator Registry 40 decides to which particular Device Manager 50 to connect the Function Instance 60 (i.e., performs the actual allocation/scheduling action).

The allocation is performed according to the runtime metrics gathered by the Accelerator Registry 40. Possible metrics include: number of requests received by the device, number of in-flight requests, allocated memory, current device utilization (in terms of space and time percentages) and all other possible runtime and offline information about the device, the configured accelerators or other connected components. Metrics may be related to the actual physical device, the node 10 on which the device and the Device Manager 50 are, or the distinct accelerator configured on the device.

Preferably, previous profiling results for the accelerator or the Function may be used in the allocation phase, along with node performance metrics (e.g., CPU performance, network bandwidth and latency). By collecting and using offline and runtime performance metrics, the Accelerator Registry 40 is able to perform device allocation to each single Function Instance 60 deployed in the system in order to maximize the connection performance between the Function Instance 60 and the Device Manager 50, to provide a fair share of the device/accelerator (in terms of time and space slices) and to avoid affecting the other components connected to the given device. Thus, even if overprovisioning of the device (meaning allocating it to more Functions than possible) may be allowed, the Accelerator Registry 40 constantly monitors the runtime metrics and acts accordingly. Possible actions may include terminating and rescheduling Function Instances 60, updating the Device Manager 50 policy regarding tasks execution, and other actions aimed at preventing performance issues over a single or multiple devices.

In an embodiment of the invention, the Accelerator Registry 40 offers two endpoints, each backed by a different service.

A first service is a Devices Service that collects and manages information about the devices (e.g. platform, configured bitstream and connected instances).

A second service is a Functions Service that contains data about the serverless functions (e.g. identifier, location, device, created instances).

Data collected through the Device and Functions Services are integrated by a Metrics Gatherer, which is an internal component of the Accelerator Registry 40. The Metrics Gatherer receives performance metrics of the Device Managers 50 from a suitable service - e.g. a Prometheus service [21]. Data as time utilization the FPGA 70 (defined as the time spent by the device computing method calls - e.g. OpenCL calls - in a given amount of time) are used to improve efficacy in allocating functions to the most appropriate node 10.

To match Function Instances 60 and available devices - e.g. FPGA 70 - , the Accelerator Registry 40 performs an online allocation algorithm when a new instance is created. For example, the Accelerator Registry 40 integrates with the Cloud Orchestrator 30 (preferably, Kubernetes [18]) to intercept function creation and deletion in the system 1. When the system 1 notifies the creation of a new Function Instance 60, the allocation algorithm patches the notified operation (e.g. adds environment variables, volumes for shared memory and forces the host allocation). An example of allocation algorithm, indicated as Algorithm 1, is presented in pseudo-code hereinbelow.

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
```

Algorithm 1 takes as input the Function Instance 60 that must be matched to a device - e.g. a FPGA 70 - through a Device Manager 50, all the available devices in the system 1 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 1 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 Accelerator Registry 40 allocates it to the requesting function instance.

Finally, when a reconfiguration is required, the system 1 checks the redistribution of instances and then migrates them with API of the Cloud Orchestrator 30 if necessary. In particular, when a Function Instance 60 sends a reconfiguration request, the Accelerator Registry 40 verifies the allocation of the requesting Function Instance 60 and checks if the device needs to be reconfigured. In that case, it deletes any other Function Instances 60 connected to that device. Advantageously, the Cloud Orchestrator 30 creates new requests for instantiating the Function Instances 60 to be deleted before effectively deleting such Function Instances 60: in this way, the Accelerator Registry 40 can patch and schedule the deleted Function Instances 60 on a different node 10.

H. Function deployment flow

Figure 7 shows the possible flow of requests and responses between the different components of the system 1 when a Function Instance 60 is created. A Device Manager 50 may register (action 701) its presence to the Accelerator Registry 40 with an explicit request whenever necessary (e.g. at start-up time, when it is restarted or needs to reconnect to the system). As for Function instantiation, the process starts from a user request (action 703) to the system 1 (it may be made through an external severless orchestrator or a component of the system 1, i.e. the Cloud Orchestrator 30). The Cloud Orchestrator 30 will then create (action 705) the deployment related to the requested Function - i.e., by registering the client with the Accelerator Registry 40 as shown in Figure 7. In the meantime, the Cloud Orchestrator 30 sends a registration request (action 707) to the Accelerator Registry 40 for the one or more Function Instances 60 necessary to perform the function requested by the client. In detail, for each Function Instance 60 to be created (which may be an application container, Virtual Machine or standard application), the Cloud Orchestrator 30 sends a request to the Accelerator Registry 40 in order to obtain (action 709) from the latter the physical node 10 on which to deploy (action 711) the Function Instance 60. If no location is decided by the Accelerator Registry 40, the Cloud Orchestrator 30 may stop or delay the deployment of the Function, or deploy the Function Instance 60 on a node decided through another mechanism (such as according to an internal scheduler of the Cloud Orchestrator 20).

When the Function Instance 60 is deployed, the Remote Library 63 registers (action 713) the Function Instance 60 to the Application Registry 40, which will then respond (action 715) with a list of Device Managers 50 to which connect. Finally, for each address of a Device Manager 50 received, the Remote Library 63 will initialize a connection (action 717) to the corresponding Device Manager 50, then the Remote Library 63 will pass the control to the Function Code 61 contained in the Function Instance 60, which will begin its own execution flow forwarding operation requests - i.e., method calls - to the one or more connected Device Managers 50 to be processed as described above.

I. Design and implementation variations

The invention thus conceived is susceptible to numerous modifications and variations, all of which are within the scope of the inventive concept that characterizes it. Here we will list some variations and possible changes in the previously described system 1.

Regarding the overall system design, each component may be implemented using any programming language and technology, and deployed using different tools. For example, each component may be deployed on Virtual Machines, Containers or any existing host Operating System. The node 10 previously described should be a physical or virtual machine, on-premise or in a cloud environment (e.g. managed by an external entity such as a cloud/datacenter provider). The network connection between the components of the system may be performed using any physical mean of connection (wired or wireless) and any network protocol.

The system may not include some of the components described in previous sections: in particular, the system should work even without the presence of an Accelerator Registry 40, the API Gateway 20 and/or the Cloud Orchestrator 30 in case the system 1 comprises a single node 10. Alternatively, there may be more than one instance of the same component (e.g. Accelerator Registry 40, or the API Gateway 20) in the system 1, each configured for performing the same or different procedures and functionalities and deployed according to different arrangements (e.g. master-slave, master-master, load-balanced replicated instances) based on the specific constraints and/or resources availability.

Regarding the sharing methodology, the system may allow to share, connect and use different kind of hardware accelerators - e.g. FPGAs, Graphic Processing Units (GPUs), Digital Signal Processors (DSPs) or Application Specific Integrated Circuits (ASICs). The sharing may be performed in time (interleaving the execution and use of functionalities of the accelerator among different applications or functions) or in space (meaning that different functionalities and algorithms/circuits may be exposed by the shared accelerator at the same time).

Regarding the Device Manager 50 component, there may be variations related to the number and disposition of the subcomponents included, or additional sub-components. For example, there may

be more queues fetched by the working thread based on a pre-defined or custom policy decided by the Device Manager 50, or based on local runtime information (e.g. current functions connected to the Device Manager 50).

The operations grouping mechanisms may not require explicit flushing from the client, but be based also on other components and policies to choose which operations to run and in which order. Finally, the Device Manager 50 internal components may be implemented with any required technology and language.

Regarding the Remote Library 63, there may be variations related to the number and disposition of the sub-components included, or additional sub-components. The asynchronous flow components for the commands may change based on the underlying network protocol implementation or future requirements and extensions. In addition, the asynchronous events and state machines may change to include additional functionalities, such as procedures, tasks, methods etc. The component integration is not restricted to the method described in Section F above, but may be performed in any compatible way.

Additional components may be included in the Remote Library 63 based on future needs, and its interface may be extended to accommodate future requirements. The flow (sequence of steps and number and composition of the asynchronous event states) shown in Figure 6 and described in Section F is intended as an example implementation, and may be extended or implemented differently based on the specific application needs.

Regarding the Accelerator Registry 40, multiple instances may be deployed in different configurations to improve its scalability. There may also be variations related to the number and disposition of the sub-components included, or additional sub-components. The implementation of the component may be performed using any required technologies and is subject to change based on the overall system behavior and requirements. The different data structures included in the Accelerator Registry 40, and the related functionalities, may be extended or reduced based on future requirements. Finally, the scheduling performed by the Accelerator Registry 40 component may be executed with any compatible algorithm, based on current and future requirements of the overall system. This condition extends also to the processed information and metrics obtained by the Accelerator Registry 40 while offering its functionalities.

Finally, the communication protocol (here defined as the set of messages and procedures that may be used to perform communication between different components of the system) and the processes described (such as the deployment flow described in Section H) may change based on current and future functionalities and requirements to be included in the system 1.

In conclusion, the preferred embodiment of the invention provides FPGA-as-a-Service system able to automatically scale and allocate accelerators to the requesting functions. In addition, the system

allows to isolate and decouple the execution of accelerated kernels on a FPGA by means e.g. OpenCL, from the host codes that use them. This isolation layer is managed in a transparent way by creating a remote acceleration library, i.e. the Remote Library 63 (based on the OpenCL specification, or another compatible interface). Each FPGA is shared among different Function Instances 60 through the Device Manager 50 service, allowing the cloud provider to expose the given board - i.e., FPGA - and related accelerators as a service. Experimental results concerning the Remote Library 63 and Device Manager 50 show limited overhead in terms of latency w.r.t. a native execution, showing that the isolation mechanism exposes a reasonable cost in terms of execution overhead. Finally, the proposed system 1 can be integrated with existing orchestrators in order to provide an efficient scheduling and allocation of the shared devices among multiple services, enabling the use of FPGA accelerators in serverless scenarios.

EXPERIMENTAL EVALUATION

In the following sections, we will describe the experimental campaigns we conducted to validate a specific initial implementation of a shared FPGA system for serverless computing according to an embodiment of the present invention.

The goal of the experimental campaign is to verify whether the FPGA sharing system according to an embodiment of the invention is introducing a limited and possibly negligible overhead w.r.t. a native execution.

Of course, if we consider a serverless scenario, the native execution represents the theoretical maximum performance that the FPGA sharing system can achieve.

J. First Experimental Setup

To test the FPGA sharing system, we set up a small Kubernetes cluster composed of two nodes. Both nodes are equipped with a single socket 3.40Ghz Intel® Core™ i7-6700 CPU, with 8 total threads (4 cores) and 32GB of DDR4 RAM. The nodes are connected on a local network with 1Gb/s Ethernet links.

Moreover, we installed a Terasic DE5a-Net FPGA board on the master node. The board includes an Intel Arria 10 GX FPGA (1150K logic elements) along with 8 GB of RAM over 2 DDR3 SODIMM sockets and a PCI Express x8 connector.

We packaged the Device Manager 50 in a Docker container to provision it easily on the cluster. The Device Manager 50 is then deployed using a Kubernetes DaemonSet with a custom label, based on the FPGA 70 available on each node.

We evaluated the FPGA sharing system 1 against three public available benchmarks written in OpenCL extracted from the Spector [22] benchmark suite. In particular, we chose to synthesize, integrate and test the Matrix Multiply (MM) kernel, the Sobel filter and the graph Breadth-First

Search (BFS) kernel for both CPU (leveraging the Portable OpenCL runtime [23]) and FPGA. For what concerns the Sobel and MM, these kernels are implemented in OpenCL by employing the tiling technique, which means to perform the kernel in parallel over multiple sub-blocks of the image/matrix. BFS is instead performed by iterating multiple times over the graph matrix using a mask.

For each kernel, we developed a micro-service that exposes a REST API to load data, apply the kernel and send back the output results after kernel execution. This represents the serverless function that our FPGA sharing system should be able to manage.

In all the cases, the kernels are suitable for both CPUs and FPGAs and allowed us to compare the two implementations on the same cluster in a transparent way, both from the application and from the client points of view.

We tested each function by deploying them on the cluster on the same node of the Device Manager. We placed the Function instance and the Device Manager on the same node to test the overhead of the isolation layer. Moreover, in any case, the accelerated kernel and its host code should be as near as possible as data transfers over network significantly reduce performance in our experimental setup. Finally, during the tests, we sent an increasing number of requests to the functions using Wrk [24], measuring the resulting latency.

K Experimental Results

Figure 8, 9 and 10 show the average latency of Sobel, BFS and MM respectively. If we consider the first set of experiments, the graph of Figure 8 shows that the average latency of Sobel is stable until a breaking point, which is reached when the service saturates. When the number of requests goes beyond the saturation point, the requests accumulates increasing the wait time in the network queues and thus the latency that a request experiences increases.

For Sobel, the inflection point is situated at 80 r/s for the altera native implementation, as the board can respond to at most 78.39 r/s without saturating. The CPU implementation reaches a maximum of 40 r/s with two instances on the same node. We did not add more instances as the Portable OpenCL implementation already uses all the available cores on the machine. Finally, we tested our sharing runtime using 1, 2, 4 and 6 instances on the same machine, all sharing the same accelerator through our system. By adding more instances to the service, we were able to increase the total utilization of the FPGA up to 96% and to improve the overall performance of the service. In fact, with just one instance the micro-service reached 43.13 r/s, while it reached 65.61 r/s with five instances and 66.4 r/s with six. Regarding the latency, our system added ~ 30ms delay w.r.t. the altera native: this is the overhead introduced by the gRPC layer over the loopback interface. Still, the system was able to outperform the CPU implementation.

We obtained similar results with the BFS and MM kernels. Figure 9(a) and 10(a) show the average

latency results for the BFS and MM applications respectively. In these cases the latencies of the altera native execution, the shared one and CPU ones are the same (or of comparable magnitude, as in the MM test). This happens as the 1Gb/s Ethernet link represents a bottleneck for the execution of the tests. For Sobel, this does not represent an issue because the application accepts a compressed format (jpeg) with size in the order of KBs, thus the bandwidth is not fully utilized even at a high throughput.

However, the average size of a matrix/graph for MM/BFS is in the order of MBs, and the transfer time represents a limit in the total latency of the application. To overcome this limitation, we decided to test the system without passing data between Wrk and the application micro-services. Data is passed just once at the beginning of the test, allowing to isolate the accelerator execution time from the network bottleneck. We show the results for these tests in Figure 9(b) and 10(b). In the BFS test, the latencies of the different implementations are similar. This depends mostly on the kernel algorithm, which is memory-bound in all cases. For the MM test, instead, we can notice a great difference between the altera native and shared implementations w.r.t. the CPU ones. In particular, our system added a -40ms delay on average when the service was saturated, while the CPU was two orders of magnitude worse in the same working condition (-2500ms for each requests, compared to 200ms with our system). As for the maximum performance, altera native reached 26 r/s, our system achieved 26.95 r/s while the CPU obtained 1.83 r/s.

L. Second Experimental Setup

In a second test, the setup of the FPGA sharing system according to an embodiment of the invention 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™ İ7-6700 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 Inte®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).

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 [22] and PipeCNN [25], 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 x8 blocks, 4 x1 window with no SIMD applied and a single compute unit, as it results in the best latency performance. For MM, we found from [22] that the best design is with 1 compute unit, 8 work items for each unit, and a completely

unrolled block of 16 x 16 elements. Finally, we synthesized PipeCNN with AlexNet as in [25].

K. Experimental Results

K.1 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 11(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, FPGA sharing system, System in the following, and the FPGA sharing system with shared memory, System shm in the following. The pure gRPC implementation (i.e., the System to which the dashed lines in Figure 11(a) refers) shows a total latency of four times w.r.t. the Native execution (to which the solid lines in Figure 11(a) refers). This is due to protobuf overheads and three copies of the data buffers. The shared memory implementation (i.e., System shm to which the dotted lines in Figure 11(a) refers) 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 (~ 2ms) is given by the gRPC control signals, which are used in both systems.

Figure 11(b) shows the latency measurements for the Sobel operator. In all cases, the kernel has a linear behavior w.r.t. the input size. The Native (solid line) RTT starts from 0.27ms with a 10 x10 image (800 bytes sent and received), up to 14.53ms for the largest image (1920 x 1080 pixels, read/write of ~ 8MB). System (dashed line) starts with an overhead of 2.46ms and reaches 24ms with the largest image. System shm (dotted line), instead, has a constant ~ 2ms overhead w.r.t. Native in all the experiments.

Figure 11(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, particularly, System shm substantially overlaps with Native in Figure 11 (c)). The Native runtime shows a minimum RTT of 0.45ms for the smallest matrices (16 x16 in both input and output matrices), but quickly rises up to 3.571s (for 4076 x 4096 matrices). As in the Sobel results, both System and System shm show a minimum RTT of ~ 2ms given by the control signals. System then reaches a maximum of 3.675s,

while System 1 shm stops at 3.588s, which is only 17ms more than Native.

The results of Figure 11 show that the overall impact of the System according to the embodiments of the present invention 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 affects 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 the System needs at least one copy to maintain full OpenCL compatibility.

K.2 FPGAs time utilization

To test the FPGAs utilization, we run a set of multi-application, multi-node experiments. The goal is to check if System 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 System function for both Native and System. For each experiment (Sobel, MM and PipeCNN with AlexNet), we deployed five identical Functions for System, while we could deploy only three Functions in the Native scenario (one for each device).

We tested each Function using Hey5 (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	10 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.

We show the per-function results for Sobel in Table II.

Type	Configuration	Function	Node	Util.	Latency	Processed	Target
System	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	18.75%	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	26.1 1 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	33.1 1 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 accelerator in terms of average latency.

The results are divided by scenario (System vs Native), configuration, and tested function. In the low load configuration, both runtimes keep up with the target throughput with latency between 20 and 30ms. Results are in line with the overhead results, with System improving device's utilization. In the medium load configuration System 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, System 1 still improved the 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. System 1 has instead averages of 5.01%, 4.67% and 19.85% respectively. Although System 1 supports more load, the response of the two systems are still comparable.

Table III shows the aggregate results for MM.

Type	Configuration	Utilization	Latency	Processed	Target
System	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.

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. System, with

slightly higher latencies and a similar utilization. The average difference for System 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.

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

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 System 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 System 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 System 1 to reach a higher utilization and number of processed requests.

REFERENCES

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

[2] S. Asano, T. Maruyama and a. Y. Yamaguchi, *"Performance comparison offpga, gpu ano cpu in image processing",* international conference on field programmable logic and applications. IEEE, 2009, pp. 126-131 , 2009.

[3] B. Sukhwani, B. B. B. Abali and a. S. Asaad, *"High-throughput, lossless data compresion on fpgas",* IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines. IEEE, 201 1, pp. 113-1 16, 201 1.

[4] P. Papaphilippou and a. W. Luk, *"Accelerating database systems using fpgas: A survey",* 28th

International Conference on Field Programmable Logic and Applications (FPL). IEEE, 2018, pp. 125-1255, 2018.

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

[6] *"Amazon EC2 F1 instances",* https://aws.amazon.com/ec2/instance-types/f1/ retrieved on 2019.05.15.

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

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

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

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

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

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

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

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

[15] S. Byma, J. G. Steffan, H. Bannazadeh, A. Leon-Garcia and a. 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.109-1 16, 2014.

[16] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth and N. Y. e. al., *"Cloud programming simplified: A berkeiey view on serveriess computing",* arXiv preprint arXiv:1 902.03383, 2019.

[17] *"Docker containers",* https://www.docker.com, retrieved on 2019.02.28.

[18] *"Kubernetes",* https://kubernetes.io retrieved on 2019.05.17.

[19] *"Openfaas serveriess system",* https://www.openfaas.com/ retrieved on 2019.05.17.

[20] *"Openci specification", version 1.2.*

[21] *"Prometheus",* https://prometheus.io retrieved on 2019.05.17.

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

[23] *"Portable OpenCL runtime",* http://portablecl.org/ retrieved on 2019.05.17.

[24] *"Wrk2 http benchmarking tool",* https://github.com/giltene/wrk2 retrieved on 2019.05.17.

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

CLAIMS

1. A hardware accelerators management system (1) for containerized and serverless environments at least comprising

a domain layer on which a plurality of application containers and functions (60, 61) are implemented,

a hardware layer in which a set of hardware accelerators are implemented and

a software layer configured for abstracting the application containers and the functions (60, 61) of the domain layer from the hardware layer,

wherein the system (1) comprises a hardware interface (80, 90) to send tasks to and reconfigure at least a portion of the processing means (70) implemented in the hardware layer,

wherein the system (1) comprises a software structure (40, 50, 63) that shares hardware accelerators of the hardware layer between application containers and functions (60, 61) of the domain layer,

wherein the software structure (40, 50, 63) performs scheduling and optimization algorithms on the resource allocations of the hardware accelerators of the hardware layer for the application containers and functions (60, 61) of the domain layer in terms of device time and/or space slot of utilization,

wherein the scheduling and optimization algorithms comprises a monitoring structure interfacing with processing means and with the software layer for reading performance metrics of at least one processing means (70), and

wherein the software structure comprises at least one device manager (50) component connected with the hardware interface (80, 90) and at least one remote library (63) component to interface each application container and function (60, 61) with the at least one device manager (50) component concurrently.

2. The hardware accelerators management system of claim 1, wherein the hardware interface is configured to communicate with at least a portion of the processing means of the hardware layer, including Field-Programmable-Gate-Array (FPGA), Application-Specific-Integrated-Circuits (ASIC), Digital Signal Processor (DSP) and Graphic Processing Unit (GPU) boards.

3. The hardware accelerators management system of claim 1 or 2, wherein the at least one remote library (63) is configured for receiving method calls performed by an application and/or a function (60, 61) implemented in the domain layer and forward such method call in an asynchronous manner to a service endpoint exposed by the at least one device manager (50).

4. The hardware accelerators management system of claim 3, wherein the at least one device manager (50) receives a plurality of method calls that requires hardware accelerators to be performed from the at least one Remote Library (63) associated with a corresponding application and/or function, and

wherein the at least one device manager (50) is configured to:

- create at least one task, the at least one task comprising a minimum sequence of called method to be performed in a predetermined order, and
- forward the at least one task to the hardware interface (80, 90).

5. The hardware accelerators management system of claim 4, wherein the at least one device manager (50) is configured to sequentially adding method to be performed in the at least one task until a blocking method or an explicit finish/flush/barrier command is added.

6. The hardware accelerators management system of claim 4 or 5, wherein the at least one device manager (50) is configured to inserting the at least one task queue (57) once created, and wherein the device manager (50) further comprises at least one worker thread (55) configured to pull and execute on the hardware accelerator tasks comprised in the task queue.

7. The hardware accelerators management system of claim 6, wherein the worker thread (55) is configured to select which task pull from the task queue (57) based on at least one of the following metrics associated with the hardware accelerator:

- number of requests received/executed by the device,
- number of in-flight requests,
- allocated memory,
- number of allocated buffers,
- hardware accelerator utilization,
- number of connected applications and instances.

8. The hardware accelerators management system of any one of the preceding claims 3 to 7, wherein the device manager (50) component interfaces with the hardware interface (80, 90) to send multiple tasks in parallel to different hardware accelerators and/or to reconfigure the processing means.

9. The hardware accelerators management system of any one of the preceding claims 3 to 8, wherein a respective remote library (63) is implemented in each application container or function (60, 61) implemented in the domain layer.

10. The hardware accelerators management system according to any one of preceding claims, wherein the at least one device manager (50) and the at least one remote library (63) are configured to communicate via a network connection, or via a shared memory area of the hardware layer on which is deployed the software layer implementing both the device manager (50) and the at least one remote library (63).

11. The hardware accelerators management system according to claim 10, wherein the at least one device manager (50) and the at least one remote library (63) are configured to are configured to expose at least one of the following service:

- application containers and functions (60, 61) registration and disconnection;
- hardware accelerator information gathering;
- reconfiguration requests;
- buffers manipulation;
- accelerator-related methods, and
- command queue operations.

12. The hardware accelerators management system according to any one of preceding claims, wherein the software structure may comprise a central management component (40) interfacing with the at least one device manager (50) and the at least one remote library (63) components to perform scheduling and optimization algorithms on the resource allocations of the hardware accelerators of the hardware layer.

13. The hardware accelerators management system of claim 12, wherein the central management component (40) is configured to:

- receive request of instantiation of functions and/or applications, and

for each function or application:

- assign a domain layer resource for instantiating the function or application, and
- assign at least one device manager (50) to the function or application, the device manger (50) with the hardware interface (80, 90) associated with a hardware accelerator requested by the function or application.

14. The hardware accelerators management system of claim 12 or 113, wherein the scheduling and optimization algorithms performed by the central management component (40) use system runtime performance indicators to efficiently allocate the resources of the hardware layer to the application containers and functions (60, 61), such runtime performance indicators comprising at least one among:

- number of requests received;
- number of in-flight requests;
- allocated memory, and
- current workload,

related to the hardware layer, one or more hardware accelerators implemented in the hardware layer or one or more processing means of the hardware layer.

15. The hardware accelerators management system according to any one of preceding claims 12 to 14, wherein the at least one device manager (50) and the central management component (40) are configured to communicate via a network connection to exchange network message/methods called comprising at least one among:

- hardware accelerator registration and removal from the central management component (40);

- reconfiguration request and validation of the hardware accelerator;
- metrics pushing from the device manager (50) to the central management component (40), and
- periodic polling from the device manager (50) to the central management component (40) and *viceversa.*

16. The hardware accelerators management system according to any one of preceding claims 12 to 15, wherein the at least remote Library (63) and the central management component (40) are configured to communicate via a network connection to exchange network message/methods called comprising at least one among:

- application containers and functions (60, 61) registration and removal from the central management component (40);
- Instances of the application containers and functions (60, 61) registration and removal from the central management component (40), and
- hardware accelerator reconfiguration request and validation.

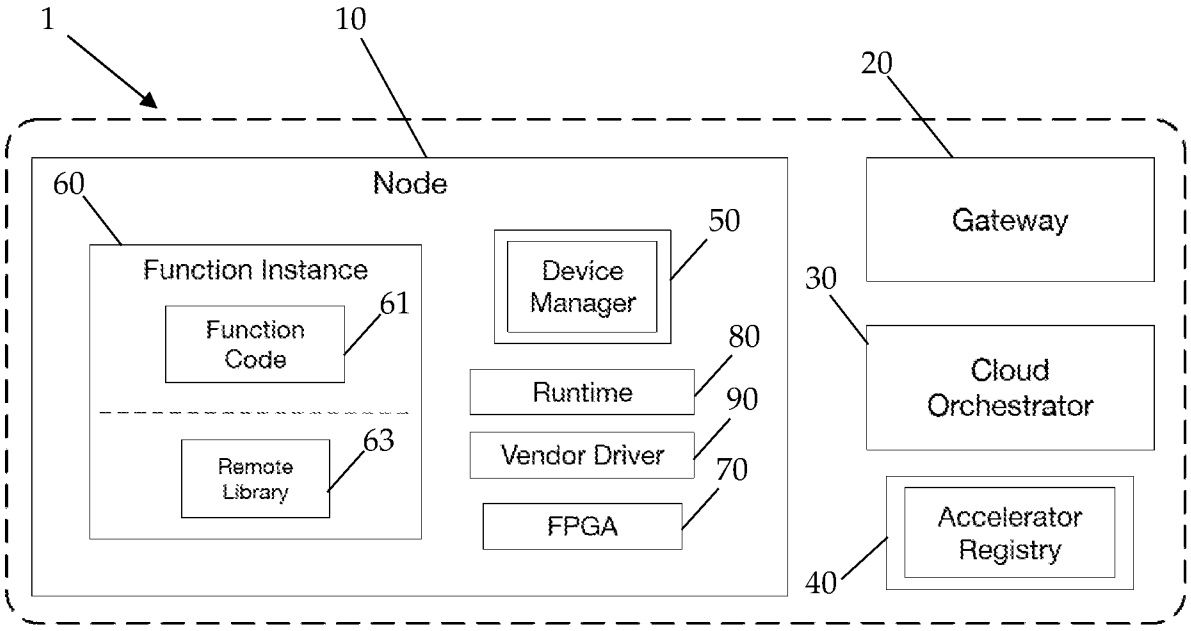

1
10
20
60
Node
Gateway
50
Function Instance
Device Manager
30
61
Function Code
80
Cloud Orchestrator
Runtime
90
63
Vendor Driver
Remote Library
70
FPGA
Accelerator Registry
40

Fig.1

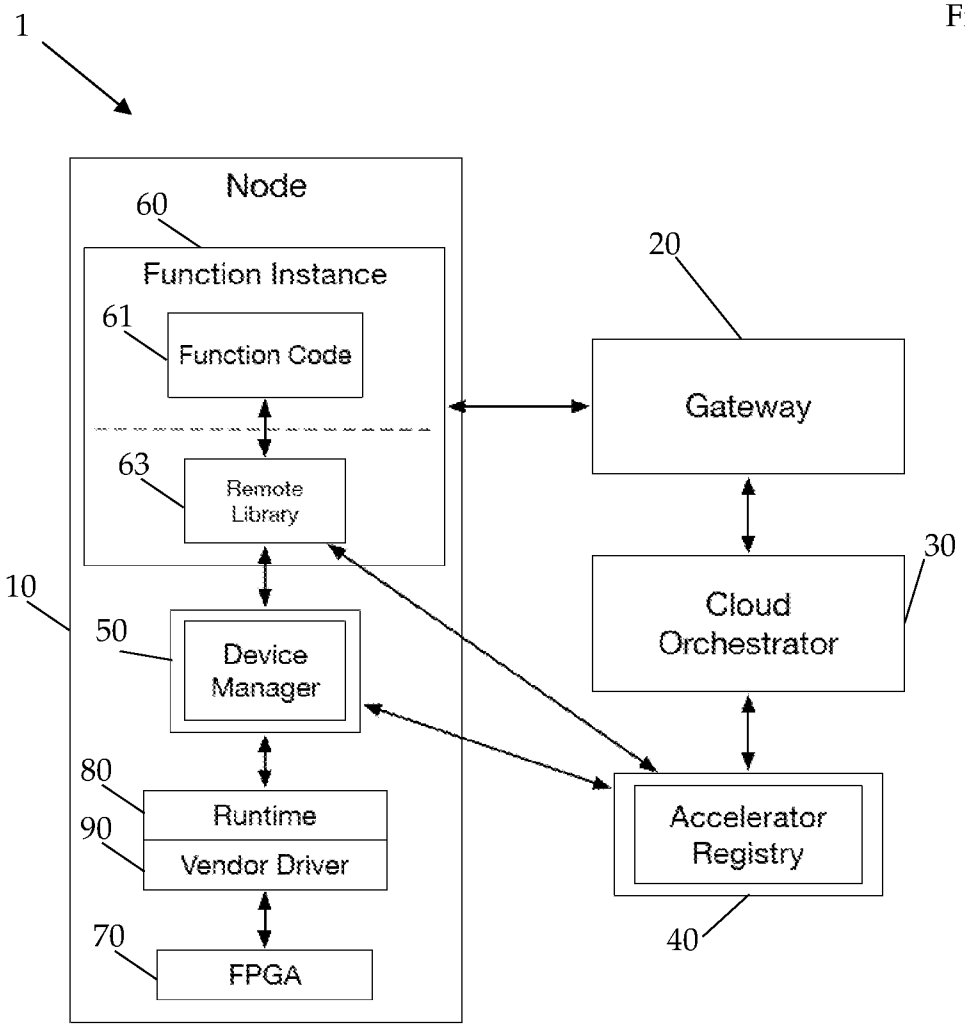
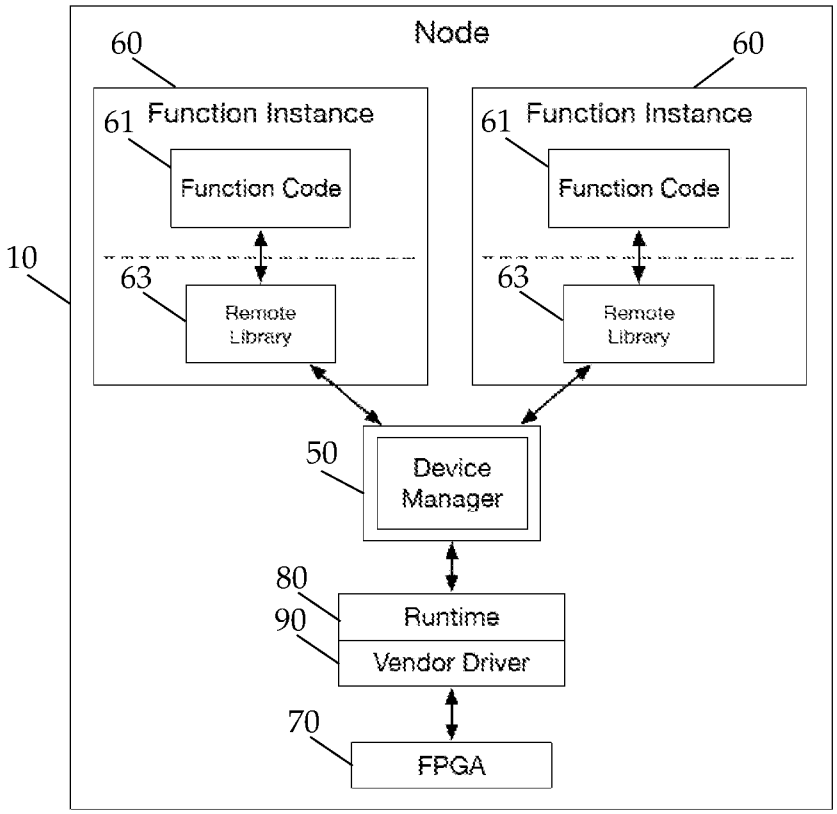
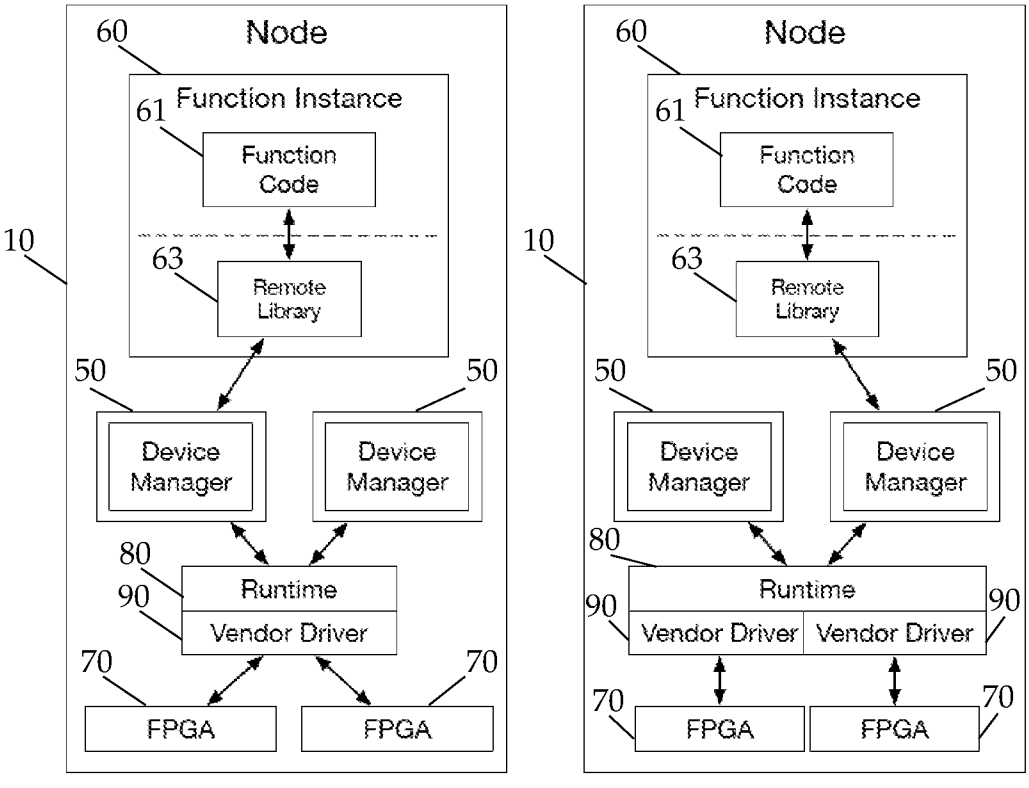

1
Node
60
Function Instance
61
Function Code
20
Gateway
63
Remote Library
30
10
Cloud Orchestrator
50
Device Manager
80
Runtime
90
Vendor Driver
Accelerator Registry
70
FPGA
40

Fig.2


Node
60
60
61 Function Instance
Function Code
63
Remote Library
61 Function Instance
Function Code
63
Remote Library
10
50
Device Manager
80
90
Runtime
Vendor Driver
70
FPGA

(a)


60
Node
61 Function Instance
Function Code
63
Remote Library
10
50
50
Device Manager
Device Manager
80
90
Runtime
Vendor Driver
70
70
FPGA
FPGA
60
Node
61 Function Instance
Function Code
63
Remote Library
10
50
50
Device Manager
Device Manager
80
90
Runtime
90
Vendor Driver
Vendor Driver
70
70
FPGA
FPGA

(b) (c)

Fig.3

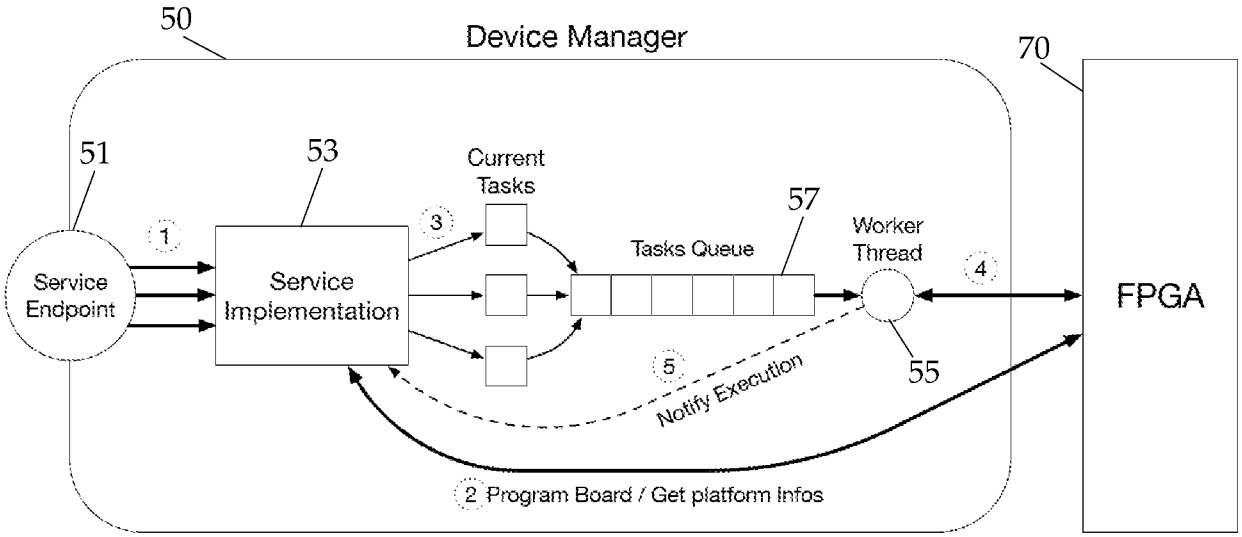

50
Device Manager
70
51
53
Current Tasks
57
Worker Thread
Tasks Queue
Service Endpoint
Service Implementation
FPGA
55
Notify Execution
2 Program Board / Get platform infos

Fig.4

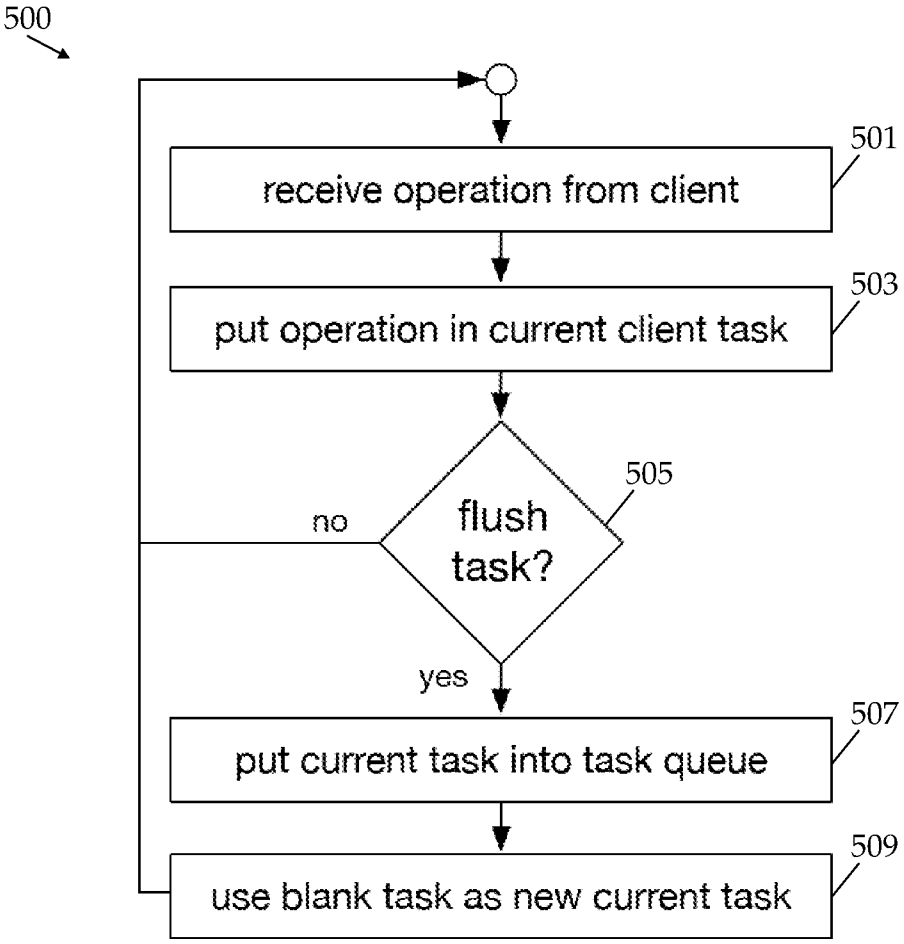

500
receive operation from client
501
put operation in current client task
503
flush task?
505
no
yes
put current task into task queue
507
use blank task as new current task
509

Fig.5

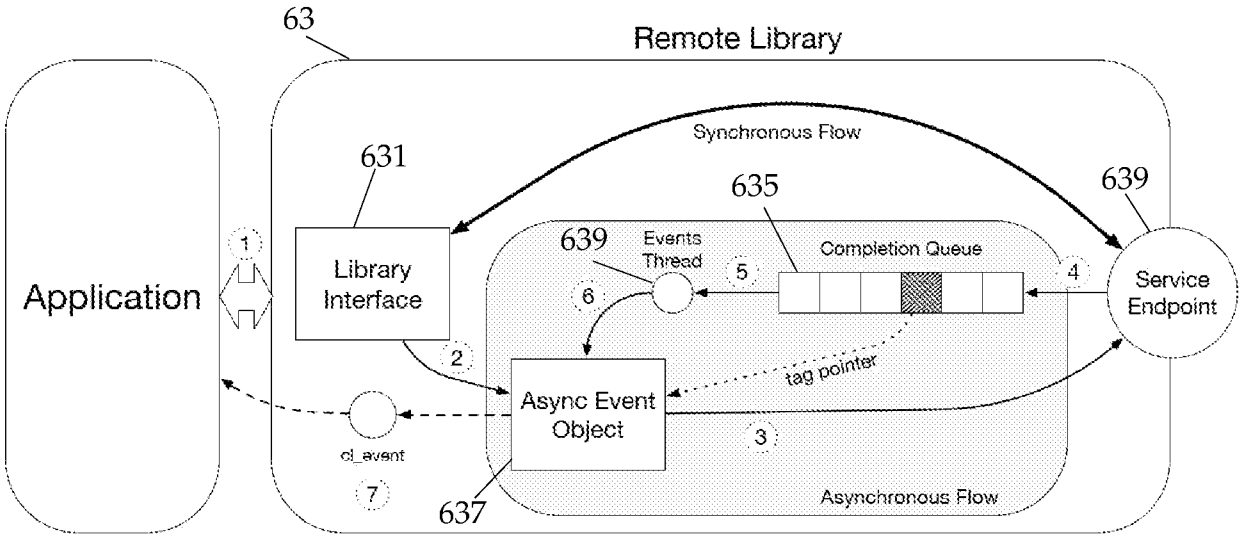

63
Remote Library
Application
631
Library Interface
Synchronous Flow
635
639
639
Events Thread
Completion Queue
Service Endpoint
Async Event Object
tag pointer
cl_event
637
Asynchronous Flow
1
2
3
4
5
6
7

Fig.6

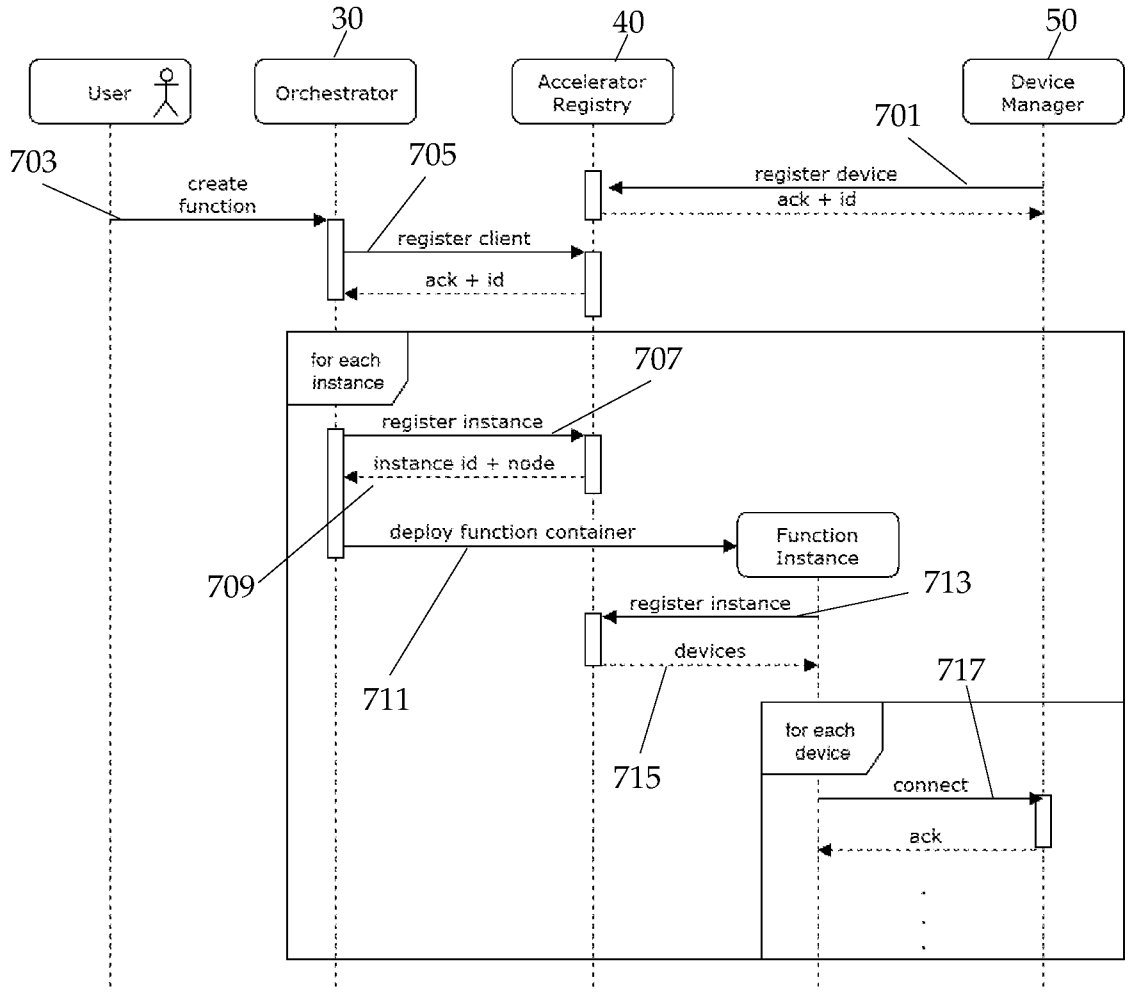

30
40
50
User
Orchestrator
Accelerator Registry
Device Manager
701
703
705
create function
register device
ack + id
register client
ack + id
for each instance
707
register instance
instance id + node
deploy function container
Function Instance
709
711
713
register instance
devices
715
717
for each device
connect
ack

Fig.7

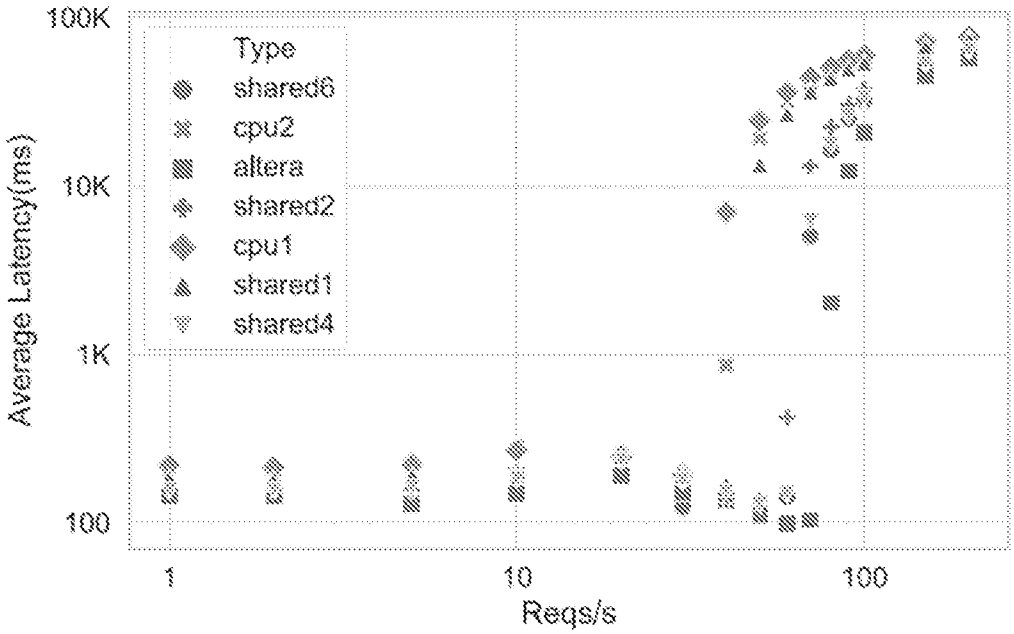

100K
10K
1K
100
Average Latency(ms)
1
10
100
Reqs/s
Type
shared6
cpu2
altera
shared2
cpu1
shared1
shared4

Fig.8

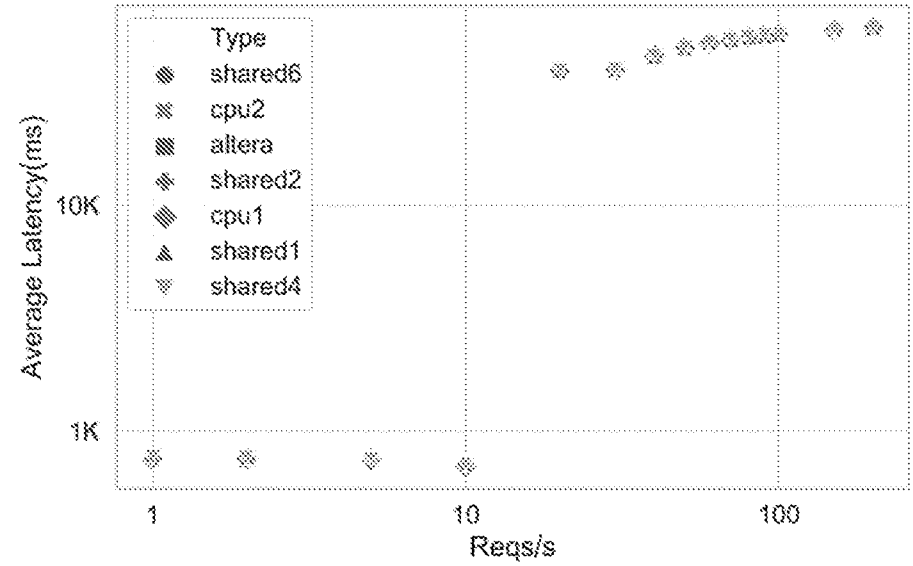

Type
shared6
cpu2
altera
shared2
cpu1
shared1
shared4
Average Latency(ms)
10K
1K
1
10
100
Reqs/s

Fig.9a

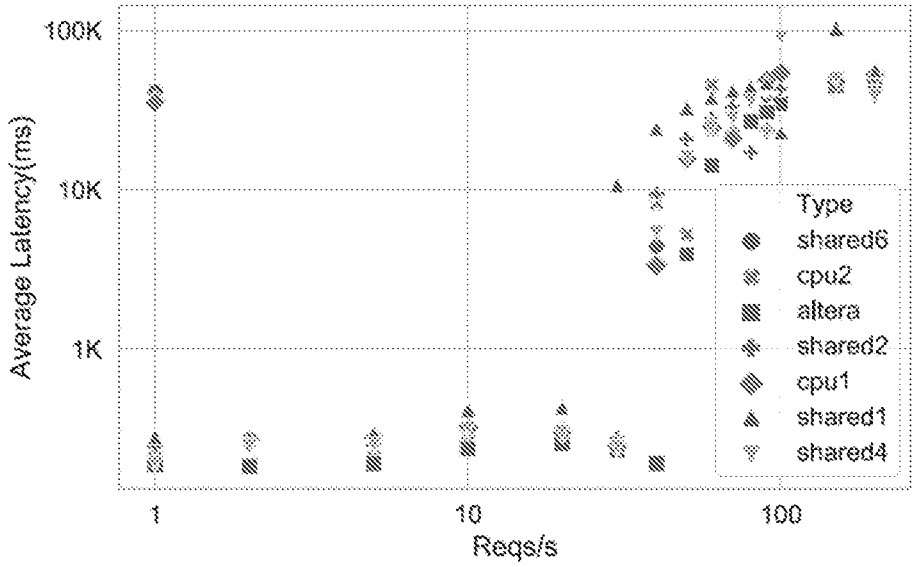

100K
10K
1K
Average Latency(ms)
Type
shared6
cpu2
altera
shared2
cpu1
shared1
shared4
1
10
100
Reqs/s

Fig.9b

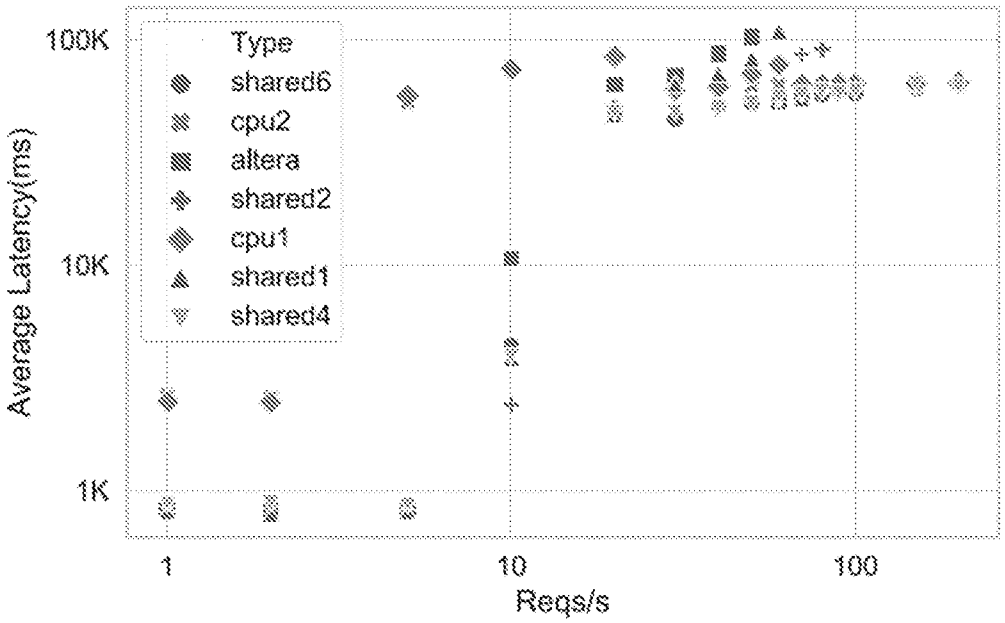


Fig.10a

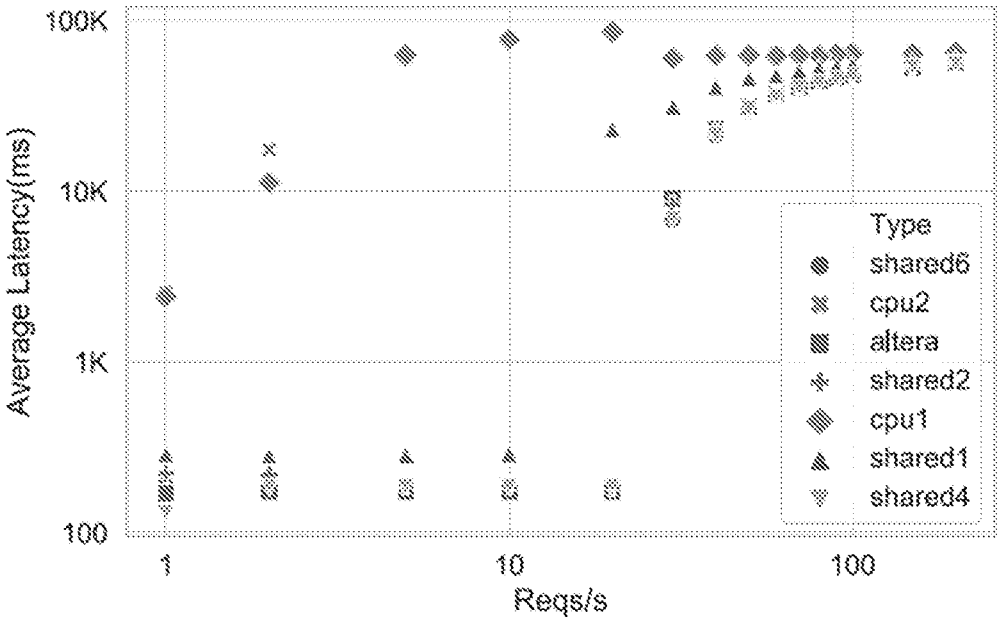


Fig.10b

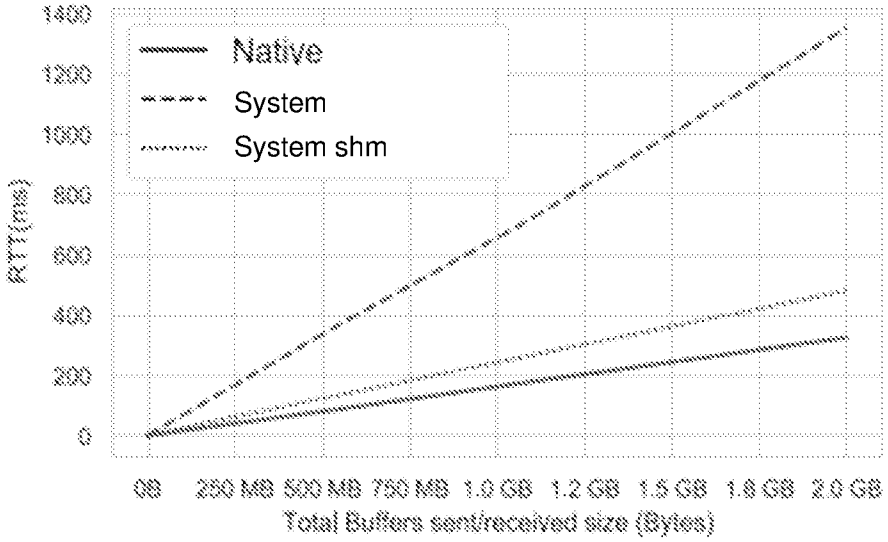


Fig.11a

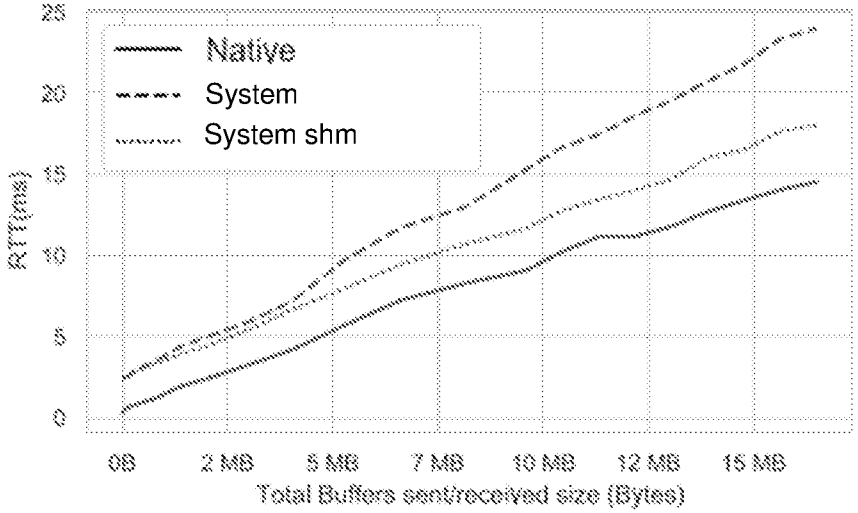


Fig.11b

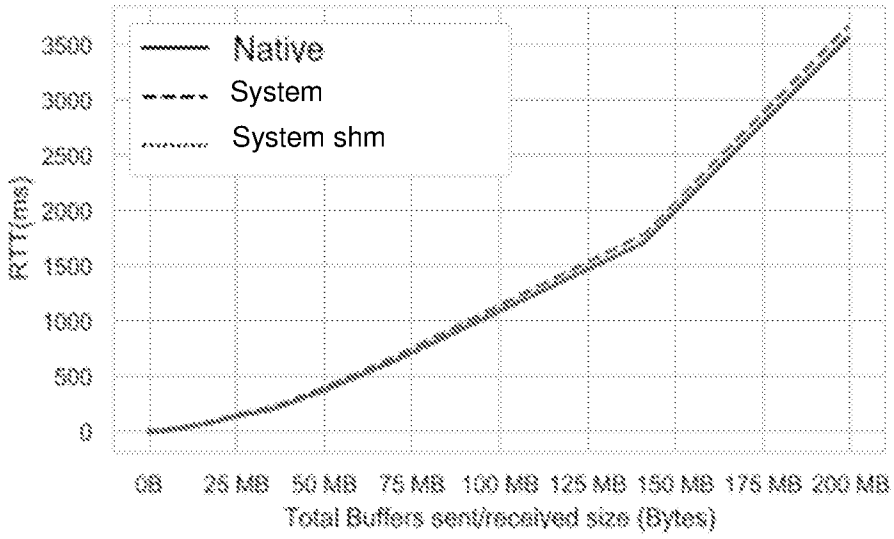


Fig.11c

INTERNATIONAL SEARCH REPORT

International application No
PCT/IB2020/054775

A. CLASSIFICATION OF SUBJECT MATTER
INV. G06F9/50
ADD.

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)
EPO-Internal, WPI Data, INSPEC, COMPENDEX

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
Y	US 10 109 030 B1 (SUN YIFAN [US] ET AL) 23 October 2018 (2018-10-23) abstract column 2, line 54 - column 4, line 60 column 5, line 65 - column 6, line 15 column 7, line 14 - column 9, line 50 figures 2,3	1-16
Y	US 2019/026150 A1 (SHIMAMURA KOMEI [GB] ET AL) 24 January 2019 (2019-01-24) abstract paragraph [0034]	1-16
A	US 2019/102238 A1 (CALDATO CLAUDIO [US] ET AL) 4 April 2019 (2019-04-04) the whole document	1-16

-/--

[X] Further documents are listed in the continuation of Box C.

[X] See patent family annex.

* Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier application or patent but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
16 September 2020	29/09/2020

Name and mailing address of the ISA/	Authorized officer
European Patent Office, P.B. 5818 Patentlaan 2 NL - 2280 HV Rijswijk Tel. (+31-70) 340-2040, Fax: (+31-70) 340-3016	Beltrán-Escavy, José

3

Form PCT/ISA/210 (second sheet) (April 2005)

C(Continuation). DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	US 2018/343567 A1 (ASHRAFI SOLYMAN [US]) 29 November 2018 (2018-11-29) the whole document	1-16
A	US 2018/376338 A1 (ASHRAFI SOLYMAN [US]) 27 December 2018 (2018-12-27) the whole document	1-16
A	US 10 275 851 B1 (ZHAO JUNPING [CN] ET AL) 30 April 2019 (2019-04-30) the whole document	1-16
A	OPEN FOG CONSORTIUM: "OpenFog Reference Architecture for Fog Computing", N/A, 1 February 2017 (2017-02-01), pages 1-162, XP055565373, USA the whole document	1-16

3

INTERNATIONAL SEARCH REPORT

Information on patent family members

International application No
PCT/IB2020/054775

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 10109030 B1	23-10-2018	NONE	
US 2019026150 A1	24-01-2019	US 2019026150 A1	24-01-2019
		US 2020089532 A1	19-03-2020
US 2019102238 A1	04-04-2019	CN 111263933 A	09-06-2020
		CN 111279309 A	12-06-2020
		CN 111279319 A	12-06-2020
		CN 111279320 A	12-06-2020
		CN 111279321 A	12-06-2020
		EP 3688579 A1	05-08-2020
		EP 3688592 A1	05-08-2020
		EP 3688593 A1	05-08-2020
		EP 3688594 A1	05-08-2020
		EP 3688595 A1	05-08-2020
		US 2019102157 A1	04-04-2019
		US 2019102226 A1	04-04-2019
		US 2019102238 A1	04-04-2019
		US 2019102239 A1	04-04-2019
		US 2019102280 A1	04-04-2019
		US 2020218589 A1	09-07-2020
		US 2020226013 A1	16-07-2020
		WO 2019068024 A1	04-04-2019
		WO 2019068029 A1	04-04-2019
		WO 2019068031 A1	04-04-2019
		WO 2019068036 A1	04-04-2019
		WO 2019068037 A1	04-04-2019
US 2018343567 A1	29-11-2018	NONE	
US 2018376338 A1	27-12-2018	NONE	
US 10275851 B1	30-04-2019	NONE	