RESTful Web Services on Standalone Disaggregated FPGAs

This paper was downloaded from TechRxiv (<https://www.techrxiv.org>).

LICENSE

CC BY 4.0

SUBMISSION DATE / POSTED DATE

12-01-2021 / 13-01-2021

CITATION

Polig, Raphael; Weerasinghe, Jagath; Hagleitner, Christoph (2021): RESTful Web Services on Standalone Disaggregated FPGAs. TechRxiv. Preprint. <https://doi.org/10.36227/techrxiv.13560317.v1>

DOI

[10.36227/techrxiv.13560317.v1](https://doi.org/10.36227/techrxiv.13560317.v1)

RESTful Web Services on Standalone Disaggregated FPGAs

Raphael Polig
IBM Research - Zurich
Email: pol@zurich.ibm.com

Jagath Weerasinghe
IBM Research - Zurich
Email: wee@zurich.ibm.com

Christoph Hagleitner
IBM Research - Zurich
Email: hle@zurich.ibm.com

Abstract—We present an architecture for field-programmable gate arrays (FPGAs) to expose RESTful web services. This architecture allows clients to access accelerated web services from any platform and programming language that can perform RESTful API calls. By using this architecture, the client's application benefits from a high throughput and low latency web service interface.

Traditionally, FPGAs are deployed in CPU-centric infrastructures as worker devices in the form of accelerators. However, for FPGA-centric applications, the overhead of a host CPU diminishes the performance, scalability and energy efficiency. cloudFPGA solves these issues by deploying FPGAs as standalone, disaggregated resources in the DC.

Building on top of the cloudFPGA platform, the presented architecture simplifies the integration of FPGA-accelerated functions with cloud applications. A configurable hardware block that can be generated from an OpenAPI-based specification of the web service is used to deploy an FPGA-based application. We compare a natural language processing (NLP) application that is exposed as a web service using the traditional server infrastructure and our RESTful service layer. Measurements show an improvement of 20x in terms of throughput and 4x reduction in mean latency.

1. Introduction

To provide sufficient compute resources for the age of Big Data, the cloud computing environment is becoming a heterogeneous place [1]. New generations of general purpose processors (CPUs) cannot keep up with the performance requirements of some applications, causing new processing devices to be introduced to the server systems. Most prominent are general purpose graphics processing units (GPUs) which are well suited for highly parallel and math-oriented algorithms [2]. Another type of processing devices are field-programmable gate arrays (FPGAs). Because of their low power consumption, these devices can be integrated into compact sized servers and can accelerate a wide range of applications.

In today's cloud environments many services can be accessed via RESTful APIs [3]. Most commonly based on the hypertext transfer protocol (HTTP) the representational state transfer (REST) provides interoperability between computer platforms and programming languages. A RESTful API

uses the HTTP verbs (GET, POST, etc.) together with the uniform resource identifier (URI) to trigger an operation which will return a response in a pre-defined format, most commonly XML or JSON. Examples services are natural language processing (NLP) tasks such as speech-to-text, text-to-speech or sentiment analysis or image analysis.

Traditionally, these web services are provided via application servers that implement the HTTP protocol and then communicate via the common gateway interface (CGI) or similar protocols with the application that implements the service function. Such an application can in turn access an FPGA to utilize accelerated functions and serve requests faster.

We present a configurable intellectual property (IP) block that is implemented on an FPGA to allow these accelerated functions to be exposed to the HTTP client. The IP block implements the basic functionality of the HTTP protocol and decodes the requests according to the developers specification in OpenAPI format [4]. The block manages the connections for timeouts and inspects the HTTP header for all required fields and their respective values. The accelerated function remains in charge of processing the payload and generating a response payload.

Traditionally, FPGAs are deployed in CPU-centric infrastructures, being connected over the PCIe bus to the host. The overhead of the CPU diminishes the performance, scalability, and the energy efficiency gains for FPGA-centric applications. cloudFPGA platform solves these issues by deploying FPGAs as standalone, disaggregated resources in cloud DCs.

We demonstrate that using the cloudFPGA platform, the throughput of the above RESTful application can be improved by 20x compared to a CPU-centric accelerated function running on a high-end server. The main contributions of the paper are as follows:

- A re-usable and configurable intellectual property block that enables an FPGA to expose web services directly via HTTP without the need for a server system.
- An example application from the natural language processing domain which extracts entities from scientific documents.
- A comparison with state-of-the-art implementations running on a high-end enterprise server hosting an

FPGA.

The remainder of the paper is organized as follows. Section 2 touches on the background of FPGAs and introduces the cloudFPGA platform. Section 3 explains in detail the configurable IP block and its use with the example NLP application. Section 5 outlines some related publications. Section 4 describes the experimental setup and the measurement results. Conclusion and outlook are given in Section 6.

2. Background

Both Moore's law and Dennard scaling continued to work until the last decade, however, now both of them have started to become less applicable [5]. This leads to innovation in two directions: (i) alternatives to CMOS technology and (ii) innovations in system stack. Although not much credible success has yet been achieved in the first approach, the second approach has resulted in improved extensions in the system stack. These extensions evolved in multiple paths: (i) low-level micro-architectural extensions, such as multi-core, SMP, SIMD, MIMD, and SoC in the processor road map, (ii) advanced memories, (iii) improved system software integration and cloud in the application space, and (iv) HW acceleration. However, the stringent physical limitations of CMOS technology inhibits exploiting increasingly more performance out of the CPUs except for the case of HW accelerators, which are built using customized HW.

Today, the domain of HW accelerators are dominated by graphics-processing units (GPUs), field programmable gate arrays (FPGAs), and application-specific integrated circuits (ASICs). While GPUs and ASICs can effectively accelerate DC tasks, the rapidly changing nature of the DC application algorithms can quickly render a dedicated accelerator obsolete. This is especially true in the case of ASICs, where the design, verification and bring-up might take several years while the applications change on a monthly basis. On the other hand, a GPU consumes an order of magnitude more power than an FPGA. Meanwhile, FPGAs deliver the performance advantages of a fixed-purpose accelerator in a low-power, flexible hardware platform that can enable faster time to innovation.

2.1. FPGAs

Because of the high flexibility of FPGAs, the scope of potential applications is broad. Microsoft showed one of the prominent use cases of FPGAs at large-scale in DCs, by improving the throughput and latency of their Bing web search service using FPGAs [6]. FPGAs have successfully been used in other application areas as well. A few examples are, big-data analytics [7], cryptography [8], HPC [9], deep learning [10], network security [11], and natural language processing [12].

However, one disadvantage of the FPGAs is their programmability. Traditionally, FPGA-based applications are developed using hardware description languages (HDL) such as VHDL or Verilog, which requires the programmer

to be knowledgeable on how the HW works at low-level. To alleviate these issues, one approach is to use HLS (High-Level Synthesis) tools, which offers a C- or C++-based programming environment. Another approach is to abstract the low-level HW details as much as possible by offering cloud-based FPGAs [13]. Typically, in these offerings, shell-role architectures are used [6] [13] [14], in which shell abstracts the FPGA's interaction between CPU and DRAM, and the role hosts the user application.

In DCs, traditionally, FPGAs are deployed in CPU-centric infrastructures, being connected to the host over the PCIe bus as worker devices [6] [13]. These infrastructures are suitable for CPU-centric applications, where most of the application processing is done in the CPU and the PCIe-attached FPGA works as an accelerator. However, for FPGA-centric applications, where most of the processing is done in the FPGA, the overhead of CPU-centric infrastructures diminishes the performance, scalability and power efficiency gains.

When offering FPGAs as compute resources to cloud users, existing CPU-centric solutions [15] face the following issues: (i) The CPU-centric approach requires the data path of the inter-FPGA communication to traverse a CPU for applications that use more FPGAs than a single CPU-centric unit can provide. (ii) The number of FPGAs that an application uses cannot be scaled independently from the number of CPU resources at the infrastructure level. (iii) the tight coupling of FPGAs to CPUs limits the number of FPGAs that can be deployed in a DC rack. Microsoft solved issue (i) by connecting PCIe-attached FPGAs in a mesh in their first implementation [6] and connecting each PCIe-attached FPGA directly to the DC network as a bump in the wire (host $\leftrightarrow$ NIC $\leftrightarrow$ FPGA $\leftrightarrow$ DC Network) in their second implementation [16]. Another approach to solving issue (i) without increasing the number of physical network connections to the host is to directly connect only the NIC to the DC network and the FPGA is connected to the NIC through PCIe (host $\leftrightarrow$ FPGA $\leftrightarrow$ NIC $\leftrightarrow$ DC Network), which is theoretically possible by using for example SR-IOV (Single Root IO Virtualization) based vNICs, but to the best of our knowledge there are no FPGA implementations available yet supporting this communication. Even if there are alternative solutions for issue (i), issues (ii) and (iii) can not be solved in the CPU-centric approach.

2.2. cloudFPGA

In contrast to the traditional CPU-centric approach, our cloudFPGA platform [17] takes the FPGA-centric approach, where the FPGA is directly connected to the DC network as a standalone, disaggregated compute resource without the need of any hosting server (Figure 1). The standalone, disaggregated nature enables FPGA-centric applications to scale independently of the number of CPUs, while enabling an efficient communication path (low latency and jitter with high throughput) within FPGA-based distributed applications [18] [19].

Figure 1. Datacenter FPGA deployment: CPU-centric (PCIe-attached) approach [6] vs FPGA-centric (standalone, disaggregated) approach [17]

Figure 4. cloudFPGA in OpenStack-based cloud: CT = container, VM = virtual machine

Figure 2. cloudSHELL: The shell-role architecture of the cloudFPGA.

Figure 3. Architecture of the fabric agent

Similarly to other cloud-based FPGA approaches, we use a shell-role architecture in the cloudFPGA, in which the shell is called cloudSHELL (Figure 2). The cloudSHELL facilitates the necessary environment within the cloudFPGA to run user applications, abstracting CPU-FPGA, FPGA-FPGA, and FPGA-DRAM communication. It also relieves the user from other FPGA bring-up tasks, such as specification of board-specific pin assignments and clocks.

As shown in Figure 2, user application only interacts with the application interface in the cloudSHELL. The application interface in turn is connected to the DRAM controller, to the TCP/IP stack, and to the management module.

The application interface offers simple FIFO-based ports to the application to interact with DRAM and the TCP-based network. To manage each FPGA in the DC in a centralized manner, the management module consists of agents, which listens on a predetermined TCP port for commands from a centralized SW.

Interconnecting multiple-FPGAs for running distributed applications and changing the inter-FPGA data path on demand requires a new bit stream to be generated (full or partial bit stream) and the FPGA to be reconfigured, which disrupts the operation of the distributed applications and also introduce a high latency in the control path of the distributed application. This is not suitable for inherently dynamic cloud environments.

To address this issue, an agent is added to the management module of the cloudSHELL, which is called "fabric agent". The fabric agent listens for external commands on a predetermined TCP port when the FPGA is configured with the cloudSHELL. An external FPGA manager connects to the fabric agent over this TCP port and issues fabric commands to dynamically interconnect ROLES in multiple FPGAs over the DC network. After decoding the received command by the fabric command decoder, the relevant commands (TCP listen, TCP connect, TCP close) are sent to the TCP connection manager in the application interface. After executing the commands, the TCP connection manager updates an internal table, which consists of FIFO ID and network link ID. FIFO ID corresponds to the index of the FIFO exposed to the ROLE, whereas network link ID corresponds to a TCP connection between two FPGAs. By updating the FIFO ID that corresponds to a link ID, the data path between ROLES can be dynamically changed. Dynamic update of FIFO ID is also supported in fabric commands. This approach reduces the multi-FPGA fabric control path latency down to sub milliseconds from tens of minutes. To form a multi-FPGA fabric with 2 FPGAs took 0.754 ms, whereas the traditional approach of building a new bit-stream and invoking reconfiguration took 29 minutes.

The above cloudFPGA HW is integrated into an OpenStack-based cloud infrastructure. In OpenStack, usually one provider network is used in combination with port vlans in the openvswitch-based integration bridge to maintain multi-tenancy across multiple hypervisors. However,

cloudSHELL does not yet support openswitch extensions and we do not expect that to change in the near future due to resource constraints. Hence, to integrate cloudFPGA HW into openstack, while maintaining multi-tenancy, we add a secondary network interface to VMs and CTs, which is called the FPGA network. The FPGA network corresponding to each tenant is based on a unique provider network. Traditional openswitch-based data network is shown in blue in Figure 4, whereas the FPGA network is shown in green.

3. REST IP Block

Any FPGA application that wants to expose its functionality as a RESTful web service must implement the HTTP protocol. Because it is a standard protocol many aspects of the communication can be implemented in an IP block that can be shared by all applications. A common FPGA design technique is to configure IP blocks which are then used by the application. Our REST IP block can be configured using an OpenAPI specification where a subset of features are currently supported.

3.1. OpenAPI Configuration

In order to configure the REST IP block we leverage the use of an OpenAPI specification (OAS). It defines machine-readable interface files for describing and documenting RESTful web services. It defines the various URI paths and HTTP verbs that are available for the API. Furthermore it defines required parameters for each API method and also specifies what MIME type each method consumes and produces.

An FPGA developer can write this specification according to the services that the application supports. The specification is then consumed by our generator tool which creates a set of customized Verilog files which implement the specification. The customized IP block can then be instantiated in the FPGA design and connected with the actual application. The generator tool will in turn create a mapping between URIs and binary command words that are used to communicate with the application. Figure 5 outlines the design flow for creating the REST IP block for a specific application

3.2. Architecture

The architecture of the REST IP block (Figure 6) is designed for use with the shell-role architecture of the cloudFPGA platform. The REST IP block resides in the ROLE and interfaces with the cloudSHELL over two FIFO-based AXI4 stream interfaces, which is an on-chip interconnect standard by ARM. There is one input stream and one output stream, which only carry the application layer data. The AXI4 stream standard defines an ID field which is used to indicate a connection ID and differentiate between different clients.

Figure 5. Design flow for designing with the REST IP block.

Figure 6. Main modules of the REST IP block in the ROLE of the cloudSHELL

The first stage of the REST IP block decodes the HTTP header of an incoming request. Because the TCP protocol transmits at a segment level, the REST IP block needs to re-assemble these segments to a complete HTTP request message. While decoding the HTTP header the IP block collects and stores the segments in DDR memory on the cloudFPGA. It uses the *Content-Length* field to determine the overall length of the message. If this field is missing in the HTTP header, the request is flushed and an error response is sent to the client (411 - Length required). Also, a limit can be set on the payload length; if that limit is exceeded, the request is flushed and the error response-413 is sent to the client, indicating that the payload is too large.

The decoding stage uses a set of finite-state machine that can consume the input stream at wire speed (10 Gbps). Each state machines is responsible for a specific header field and translates the decoded element into a binary integer identifier, which is later used by the top-level state machine. If only a segment of the message is received, the current states of the state machines are saved to a connection buffer on the FPGA. When the next segment of a connection arrives the states are restored and the decoding continues. This context swap requires two clock cycles on the FPGA which is roughly 15 ns for this implementation.

The request's type and URI are decoded using the OpenAPI specification provided by the developer. The specification was used by our generator tool to create a custom state machine which maps the various API methods to a binary instruction word and is supplied to the actual appli-

cation block. These commands are sent via an AXI4 stream interface once the complete payload has been received. The payload data is supplied to the application module via a second AXI4 stream interface. Any mismatching URIs or MIME types are immediately flushed and an error response is sent to the client without the application being involved.

Responses are generated by the response encoder of the REST IP block. It consists of two finite-state machines. One responsible for the overall response and one for generating the header information. A response can be triggered from two sources: the application or the internal request decoding logic. All internal responses are error codes or the informational *Continue* response and do not have a payload. Applications in turn may trigger the responses *OK* or *Internal Server Error* which can both have a payload. A third alternative for applications is the *No Content*, which indicates successful execution of the API call but the response requires no payload.

If the application provides a payload it has two options: either it knows the length of the payload at the time it submits the response command, or the payload length is unknown which is encoded as a zero-length payload with a generic OK response command. In this scenario the response encoder logic will send an HTTP response header with the transfer encoding set to *chunked*. In this mode the payload is sent via HTTP in chunks which are generated by the response encoder. Although this introduces some overhead to the processing performance of the FPGA it is a nice feature of the HTTP protocol because it avoids storing data on the local DDR memory. The AXI4 conformant *TLAST* signal indicates the last piece of data to be sent in this mode from the application and concludes the response payload.

3.3. Example Service

As an example service we choose a natural language processing (NLP) application which scans scientific documents for relevant entities. The service accepts plain text documents and returns a set of annotations which is returned as a JSON object. The application logic is implemented using the annotation query language (AQL) FPGA compiler framework from [20]. The logic is capable of processing the document in a single pass by evaluating it one byte per clock cycle, which is an eighth of the line rate. Using multiple instances of the processing logic this performance could be increased but for the initial evaluation this has not been implemented.

The RESTful API specification defines which annotations should be returned to the client. One examples is */annotate/*, which returns the types of annotations. Another example is */annotate/ValueUnit*, which returns only the annotations of the *ValueUnit* type. The text analytics application core generates all of these annotations in parallel. We filter those annotation results at the output of the core. The filtered results are then forwarded to the REST IP block where they are sent in *chunked* mode to the client.

Figure 7. Experimental setup.

4. Experiments

To evaluate our example web service application presented in Section 3 we compare it’s performance against a server-based implementation. The server version has been evaluated as a pure software implementation and as an accelerated service using an FPGA.

4.1. Setup

All components are connected via a 10 Gigabit Ethernet network. The application client runs on an x86-based server at 2.6 GHz and 32 GB of memory. The server machine is an IBM POWER8-based server, which runs at 2.92 GHz and has 512 GB of memory. The POWER8 processor has 20 physical cores and can run up to 160 hardware threads simultaneously. The server hosts a commercial-off-the-shelf (COTS) FPGA accelerator board based on a Xilinx Kintex UltraScale FPGA, which is connected via the CAPI interface to the processor. The cloudFPGA prototype is also implemented on the same FPGA card, which is powered by a PCIe extension chassis. The card is connected via SFP+ connectors to the 10 GbE network. Figure 7 illustrates the experimental setup.

We use Apache JMeter [21] as the client application to generate the API requests from the client. We use HTTP 1.1 as the communication protocol which defines all connections to be persistent. This means that the TCP connection is kept alive and multiple request/response pairs can be sent. This reduces the overhead of re-establishing a TCP connection for each call. JMeter uses multiple threads to create the requests and collects information about the processing performance.

On the server machine we run an nginx HTTP server [22] with four worker processes. The actual web service is a Python application hosted by the uWSGI [23] application server running 20 processes. The two servers communicate via the Python standard web server gateway interface (WSGI). The web service application can access the accelerated functions on the CAPI-attached FPGA via a Python extension and can run in accelerated or in standard mode.

To evaluate the impact of the individual components we ran the tests in five scenarios:

- **nginx:** nginx only serving a static file

Figure 8. Involved processes, modules and communication protocols on the POWER8 server and the cloudFPGA scenarios.

- **nginx+uwsgi:** nginx making a call to an WSGI app that immediately returns an empty result
- **nginx+uwsgi+app:** An actual API call to the application with all processing performed in software
- **nginx+uwsgi+app+fpga:** An actual API call where the application utilizes the accelerated function on the FPGA
- **cloudFPGA:** An actual API call where all processing is done on the cloudFPGA

4.2. Results

Apache JMeter reports the overall processing throughput in requests per second as well as the minimum, maximum and mean processing time of the individual API calls. Table 1 summarizes the results for all five testcase scenarios when there are 100 simultaneous requests in flight.

The pure nginx baseline performance is at 37,389 requests/s which is in line with numbers that can be found on the web. As there is no application processing involved at all, this is the highest rate at which the service could be operated on the POWER8 server. When adding a WSGI call to the picture, the throughput performance drops by nearly a factor of four but the jitter for the individual processing time is relatively small.

When running the full application API call, the performance significantly drops. The maximum throughput observed was 540 requests/s which results in a mean processing time of 175 ms for the client. Also the variation of the processing time becomes much wider. While the fastest call required 31 ms, the slowest took six times longer, 197 ms. When the application uses the CAPI-attached FPGA, the performance numbers improves again. The throughput increases to 7,917 requests/s and the processing times do not exceed 9 ms for the client.

TABLE 1. RESULTS FOR THE PERFORMANCE MEASUREMENTS RUNNING WITH 4 NGINX PROCESSES AND 20 UWSGI PROCESSES.

Case	requests/s	min (ms)	mean (ms)	max (ms)
nginx	37,389	1	1	2
nginx+uwsgi	15,930	2	6	8
nginx+uwsgi+app	540	31	175	197
nginx+uwsgi+app+fpga	7,917	1	3	9
cloudFPGA	166,093	0	0	1

With the proposed architecture the entire service is running on the cloudFPGA. The client makes a direct RESTful API call to the FPGA, thus the entire communication and application processing occurs on a single chip. Our architecture was able to provide a processing throughput of 166,093 requests/s which is more than 20 times higher than the accelerated version of the service on a high-end server node. All processing calls required a maximum of 1 ms to complete. In a real-life environment this number would depend on the actual distance in the network from the client to the server. The results show that application throughput is increased by 15x with the acceleration of application logic only, whereas the throughput is increased by 308x with the acceleration of the whole stack, including TCP, HTTP, REST, and the application. Therefore, compared to the traditional approach of acceleration using PCIe-attached FPGAs, cloudFPGA performs 20x better.

Another interesting effect was observed with the number of concurrent requests. The cloudFPGA performs significantly better than the server implementation in situations where there is little or very high load. Figure 9 shows the number of requests per second over the number of concurrent requests that are in flight. If there is only one request in flight the performance is mainly latency driven, therefore the pure hardware implementation on the cloudFPGA is much faster. With an increasing number of simultaneous requests the performance depends more on the overall processing power of the server. The cloudFPGA and nginx measurements reach their peak at 10 concurrent requests and then remain fairly constant. All measurements with uWSGI involved increase up to 100 simultaneous requests and then drop for 1,000.

4.3. Power Consumption

Power efficiency has always been a stronghold for FPGAs. With the cloudFPGA platform this efficiency can be fully exploited in a cloud environment. While the cloudFPGA requires at maximum 25 W to operate the web service, the fully equipped POWER8 server requires 340 W in idle mode and more than 360 W when running the web service, which is measured using the on-system sensors of the server [24]. This shows that the power consumption of a POWER8-based system is 13 times high compared to the cloudFPGA.

Figure 9. Number of requests served over different number of concurrent requests on a log scale.

4.4. System Cost

Adding an FPGA to each server in a DC environment significantly increases the cost of a server unit. Microsoft’s catapult implementation increased this cost by 30% [6]. For FPGA-centric applications the server that hosts the FPGA might not be efficiently used. Hence, by deploying standalone, disaggregated FPGAs, the cost of the server can be omitted from the total cost of the FPGA infrastructure. As an example, considering the application used in this paper, the system cost can be cut down by around \$5000 (according to prices on web), by completely eliminating a server to host the web application.

4.5. Resource Consumption

When developing on FPGAs the individual functional modules require resources like configurable logic blocks (CLBs) or internal memory elements (BlockRAM). Therefore it is important to keep the resource consumption low for the interface logic to allow the actual application to utilize the remainder. Together with the cloudFPGA’s network service layer the REST IP Block requires about 20 % of the overall logic resources of the FPGA and 30 % of the internal memory blocks. This is comparable with the resources required by the POWER service layer to implement the CAPI protocol. With newer generations of FPGAs more resources will be available for the application module.

5. Related Work

Cuenca-Asensi et al. [25] present the WS reconfigurable platform to support access to web services using the simple object access protocol (SOAP). As a communication layer HTTP is implemented in the protocol module which also allows communication with a service broker via the UDDI protocol. The example implementation shows a Wake on LAN over an Internet application which activates the device. The response times are up to four times lower compared to a software implementation on an Intel-based computer.

Yu et al. [26] present an FPGA implementation of a web server. It uses a MicroBlaze soft-core processor to perform some configuration and processing steps, while the main HTTP operations are performed in a web processing module in FPGA logic. Only the GET verb of the HTTP protocol is supported. The performance is actually worse for serving documents that are smaller than 10 kB compared to the nginx web server. The main benefit claimed is the power efficiency compared to an Intel Xeon-based server.

A rudimentary implementation for RESTful webservices is presented by Chang et al. [27]. The complete TCP/IP and HTTP stack is implemented to enable an example application for home device control. The system is limited to a single TCP message on a single connection. No performance numbers are provided, but the ability to implement such services on an embedded device is shown.

Alternative implementations of the HTTP protocol and services are often implemented using soft-core processors. Joshi et al. [28] demonstrate this with the Nios II processor provided by Altera for its FPGAs.

6. Conclusion

Heterogeneous computing has arrived at the cloud. With the ever increasing amounts of data stored and collected, the need for higher processing performance is inevitable. Using accelerator devices such as GPGUs or FPGAs this need can be satisfied for suitable applications.

RESTful APIs have become a de-facto standard for web applications to communicate with each other. Web services can be easily accessed from anywhere and any platform that supports the HTTP protocol. But when using HTTP for accelerated web services the protocol handling on a conventional processor can become a bottleneck.

In this work we propose a reusable hardware IP block that can be used on the cloudFPGA platform to create standalone web services on an FPGA. The IP block can be generated from an OpenAPI specification for the implemented API and frees the application developer from dealing with the HTTP protocol and erroneous API calls. Our example application from the natural language processing domain demonstrates a 20x throughput improvement compared to the same accelerated service on a high-end server. The mean processing time is reduced by a factor of 4. This shows the potential for such an architecture in the cloud.

6.1. Future Directions

There are remaining features that need to be implemented on the platform to provide general usability.

Encryption: While annotating publicly available documents can be done with an unencrypted connection, many services require encrypted connections to provide secure communication. To enable this type of confidentiality the secure sockets layer (SSL) must be implemented on the platform.

Authorization: Users of the accelerated device must be identified and granted access to the service. Using an external authentication server and using a token-based access could be a solution.

Accountability: To know how often a specific user has used a web service it needs to be tracked. To keep the performance-levels, this accounting can be done in the FPGA's logic and periodically reported to a master server.

Scalability: Traditionally, services are scaled by running the same service on multiple nodes. These nodes may be accessed directly by load balancing at the DNS level or using HTTP load balancers [29]. Finding an adequate load balancing solution is essential to enable scalability of the presented platform. To support HTTP load balancing, the number of cloudFPGA instances must be dynamically scaled. For dynamic scaling, we plan to use the fabric agent of the cloudSHELL.

References

- [1] S. Crago, K. Dunn, P. Eads, L. Hochstein, D.-I. Kang, M. Kang, D. Modium, K. Singh, J. Suh, and J. P. Walters, "Heterogeneous cloud computing," in *Cluster Computing (CLUSTER)*, 2011 IEEE International Conference on. IEEE, 2011, pp. 378–385.
- [2] V. T. Ravi, M. Becchi, G. Agrawal, and S. Chakradhar, "Supporting gpu sharing in cloud environments with a transparent runtime consolidation framework," in *Proceedings of the 20th international symposium on High performance distributed computing*. ACM, 2011, pp. 217–228.
- [3] L. Richardson and S. Ruby, *RESTful web services*. " O'Reilly Media, Inc.", 2008.
- [4] "OpenAPI Initiative," <https://www.openapis.org/>, accessed: 2017-06-29.
- [5] M. B. Taylor, "A landscape of the new dark silicon design regime," *IEEE Micro*, vol. 33, no. 5, pp. 8–19, 2013.
- [6] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmailzadeh, J. Fowers, G. P. Gopal, J. Gray *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," in *Computer Architecture (ISCA)*, 2014 ACM/IEEE 41st International Symposium on. IEEE, 2014, pp. 13–24.
- [7] S.-W. Jun *et al.*, "BlueDBM: An Appliance for Big Data Analytics," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: ACM, 2015, pp. 1–13.
- [8] T. Güneysu *et al.*, "Cryptanalysis with copacabana," *IEEE TRANSACTIONS ON COMPUTERS*, vol. 57, no. 11, pp. 1498–1513, 2008.
- [9] F. Belletti, M. Cotallo, A. Cruz, L. A. Fernandez, A. Gordillo-Guerrero, M. Guidetti, A. Maiorano, F. Mantovani, E. Marinari, V. Martin-Mayor, A. Muñoz-Sudupe, D. Navarro, G. Parisi, S. Perez-Gaviro, M. Rossi, J. J. Ruiz-Lorenzo, S. F. Schifano, D. Sciretti, A. Tarancon, R. Tripiccone, J. L. Velasco, D. Yllanes, and G. Zanier, "Janus: An fpga-based system for high-performance scientific computing," *Computing in Science Engineering*, vol. 11, no. 1, pp. 48–58, Jan 2009.
- [10] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, 2015, pp. 1737–1746.
- [11] H. Song and J. W. Lockwood, "Efficient packet classification for network intrusion detection using fpga," in *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-programmable Gate Arrays*, ser. FPGA '05. New York, NY, USA: ACM, 2005, pp. 238–245. [Online]. Available: <http://doi.acm.org/10.1145/1046192.1046223>
- [12] C. Ciressan, E. Sanchez, M. Rajman, and J.-C. Chappelier, "An fpga-based coprocessor for the parsing of context-free grammars," in *Field-Programmable Custom Computing Machines, 2000 IEEE Symposium on*. IEEE, 2000, pp. 236–245.
- [13] D. Pellerin, "Announcing amazon ec2 f1 instances with custom fpgas hardware-accelerated computing on aws," December 2016. [Online]. Available: <https://www.slideshare.net/AmazonWebServices/announcing-amazon-ec2-f1-instances-with-custom-fpgas>
- [14] IBM, "Coherent accelerator processor interface user's manual," January 2015.
- [15] Amazon, "Amazon f1 instance," 2017. [Online]. Available: <https://aws.amazon.com/ec2/instance-types/f1/>
- [16] A. M. Caulfield *et al.*, "A cloud-scale acceleration architecture," in *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.
- [17] J. Weerasinghe, F. Abel, C. Hagleitner, and A. Herkersdorf, "Enabling FPGAs in hyperscale data centers," in *IEEE CBDCOM, Beijing, China*. IEEE, 2015.
- [18] —, "Disaggregated FPGAs: Network performance comparison against bare-metal servers, virtual machines and linux containers," in *Cloud Computing Technology and Science (CloudCom)*, 2016 IEEE International Conference on. IEEE, 2016, pp. 9–17.
- [19] J. Weerasinghe, R. Polig, F. Abel, and C. Hagleitner, "Network-attached fpgas for data center applications," in *2016 International Conference on Field-Programmable Technology (FPT)*, Dec 2016, pp. 36–43.
- [20] R. Polig, K. Atasu, H. Giefers, C. Hagleitner, L. Chiticariu, F. Reiss, H. Zhu, and P. Hofstee, "A hardware compilation framework for text analytics queries," *Journal of Parallel and Distributed Computing*, 2017.
- [21] E. H. Halili, *Apache JMeter: A practical beginner's guide to automated testing and performance measurement for your websites*. Packt Publishing Ltd, 2008.
- [22] W. Reese, "Nginx: the high-performance web server and reverse proxy," *Linux Journal*, vol. 2008, no. 173, p. 2, 2008.
- [23] "The uWSGI project," <https://uwsgi-docs.readthedocs.io/>, accessed: 2017-06-14.
- [24] H. Y. McCreary, M. A. Broyles, M. Floyd, A. Geissler, S. P. Hartman, F. Rawson, T. Rosedahl, J. Rubio, and M. Ware, "EnergyScale for IBM POWER6 microprocessor-based systems," *IBM Journal of Research and Development*, vol. 51, no. 6, pp. 775–786, Nov 2007.
- [25] S. Cuenca-Asensi, H. Ramos-Morillo, H. Llorens-Martinez, and F. Macia-Perez, "Reconfigurable architecture for embedding web services," in *Programmable Logic, 2008 4th Southern Conference on*. IEEE, 2008, pp. 119–124.
- [26] J. Yu, Y. Zhu, L. Xia, M. Qiu, Y. Fu, and G. Rong, "Grounding high efficiency cloud computing architecture: Hw-sw co-design and implementation of a stand-alone web server on fpga," in *Applications of Digital Information and Web Technologies (ICADIWT)*, 2011 Fourth International Conference on the. IEEE, 2011, pp. 124–129.
- [27] C. Chang, F. Mohd-Yasin, and A. Mustapha, "An implementation of embedded restful web services," in *Innovative Technologies in Intelligent Systems and Industrial Applications, 2009. CITISIA 2009*. IEEE, 2009, pp. 45–50.
- [28] N. N. Joshi, P. Dakhole, and P. Zode, "Embedded web server on nios ii embedded fpga platform," in *Emerging Trends in Engineering and Technology (ICETET)*, 2009 2nd International Conference on. IEEE, 2009, pp. 372–377.
- [29] A. Piórkowski, A. Kempny, A. Hajduk, and J. Strzelczyk, "Load balancing for heterogeneous web servers," in *International Conference on Computer Networks*. Springer, 2010, pp. 189–198.