HORIZON 2020
TOPIC FETHPC-02-2017
Transition to Exascale Computing

EPiGRAM-HS

Exascale Programming Models for Heterogeneous Systems
801039

D4.5
Report on experiences in using MPI and GASPI on systems with low-power microprocessors.

WP 4: Productive computing with FPGAs, GPUs and low-power microprocessors

Date of preparation (latest version): 30/11/2021

DOCUMENT INFORMATION

Deliverable Number	D4.5
Deliverable Name	Report on experiences in using MPI and GASPI on systems with low-power microprocessors.
Due Date	30/11/2021 (PM 39)
Deliverable lead	UEDIN
Authors	Michael Bareford (UEDIN) Valeria Bartsch (Fraunhofer) Oliver Thomson Brown (UEDIN) Nick Johnson (UEDIN) Arcesio Castaneda Medina (Fraunhofer)
Responsible Author	Oliver Thomson Brown (UEDIN) e-mail: `o.brown@epcc.ed.ac.uk`
Keywords	MPI GPI FPGA
WP/Task	WP4 / Task 4.3
Nature	R
Dissemination Level	PU
Planned Date	30/11/2021
Final Version Date	30/11/2021
Reviewed by	Martin Karp (KTH) Harvey Richardson (HPE)
MGT Board Approval	YES

DOCUMENT HISTORY

Partner	**Date**	**Comment**	**Version**
UEDIN	01/08/2021	Initial version modified from template	0.1
All partners	29/10/2021	Internal review version	0.2
All partners	26/11/2021	Final version	0.3

Executive Summary

In this document we present our experiences using both MPI and GASPI on low-power microprocessors, in particular FPGAs. They were a focus, as they are relatively widely available as accelerators in HPC machines, but still quite poorly supported from the software side making them challenging to use. This report is structured as follows. In section 1 we introduce FPGAs as well as ARM low-power processors which were initially considered, but later removed from the scope of our work. In section 2 we discuss our experiences in using MPI on FPGAs, and in section 3 we discuss our experiences in using GASPI on FPGAs. FPGAs are a challenging target for integration into traditional HPC programming models as the architecture is fundamentally different, however they are available in HPC systems, and can be directly networked together – opening the way for MPI and GASPI. Finally we conclude in section 4.

The overall conclusion is that the FPGA programming ecosystem is that it is possible to use MPI on FPGAs, but that support for doing so is still very immature, and consequently fragile. To better support FPGA usage, the MPI standard should continue to expand its persistent and one-sided communication interfaces, as these are better suited to the FPGA architecture. More success can be found using GASPI where the standard already supports FPGAs, however performance improvements should be possible with the introduction of the FPGADirect-RDMA API.

Contents

1 Low-power microprocessors

In this section we discuss in broad terms the two types of low-power microprocessors which were considered as being potentially relevant to distributed technologies such as MPI and GASPI during this project. Only FPGAs were used in the project in the end, and here we explain why.

1.1 FPGA

The Field-Programmable Gate Array was introduced in the mid-1980s. They were originally intended as a cheaper and more versatile, but less performant, complement to Application-Specific Integrated Circuits (ASICs) [1]. In a sense this is still the role they inhabit, except that design improvements have pushed the performance of FPGAs to a level where very high performance or volume is required for the ASIC to be preferable. FPGAs have grown from low-volume replacements for ASICs, to core components of networking hardware. It is in this way that they were introduced to the HPC ecosystem – as they appeared in high-speed interconnects to enable smart data routing, the question was asked: can they be used for more? This has lead to the development of larger FPGAs for use as accelerators in the HPC setting.

It is worth noting that the sort of FPGA which is of interest to HPC is not *necessarily* low-power. The Xilinx Alveo U280, which is installed in the Xilinx Adaptive Compute Cluster (XACC) hosted at ETH Zürich [2], for example can draw up to 215W [3]. Power consumption is determined by a combination of static usage from resources on the card, and dynamic power usage from actual operation of those resources, and dynamic power usage can be drastically lower than GPGPUs for example. This does mean that a carefully designed circuit using only a small portion of the card *can* be low-power, and the application developer does have the level of control required to make the necessary tradeoffs between power consumption and performance, unlike on a CPU.

As noted, FPGAs were originally used in fields that made use of ASICs, and the programming models reflect this. Using an FPGA can be much more like designing a circuit than writing code. Xilinx Vitis and Intel HLS (High-Level Synthesis) compiler tools exist to help the code developer to compile C/C++ down to an FPGA bit-stream. However getting the best performance often requires significant redesign of the code, particularly since the FPGA's dataflow architecture is radically different to traditional CPU von Neumann architecture [4].

FPGAs were chosen to be the primary target device for the low-power microprocessor work in EPiGRAM-HS, in part because they are widely available in the HPC setting. During the course of this project we were able to access FPGA accelerated HPC systems hosted at ETH Zürich [2], Forschungszentrum Jülich [5] [1], and EPCC [7].

On top of being relatively widely available in HPC facilities, it is possible to interconnect FPGAs enabling distributed computations so they are potentially of relevance to large-scale HPC systems. Furthermore, they present an interesting challenge from a programming model perspective, as programming even a single FPGA for performance

[1] Unfortunately, the work involving the DEEP-EST platform had to be abandoned due to the unmaintained nature of the software stack as of December 2020 [6].

can be challenging. Good performance typically relies on setting up parallel pipelines and streaming data in to keep them supplied.

1.2 ARM

ARM are a Cambridge based semiconductor and software design company, known for their low-power CPU architectures. Their adoption has been widespread in mobile devices such as phones, games consoles, and single-board computers, however ARM design processors for a range of applications, and with a range of power consumption. For example, the latest generation of Apple's MacBook computers uses CPUs based on the ARM64 architecture.

One possibility that was considered at the start of EPiGRAM-HS was investigating the use of ARM big.LITTLE heterogeneous processors, which pair higher power compute cores with a number of lower power cores which can either work on less computationally intensive tasks in tandem with the higher-powered processors, or take over entirely in low-power contexts (such as a battery saving mode)[8]. Both types of core use the same ISA so can run the same machine code, but the LITTLE cores support simpler, in-order pipelines and operate at lower clock frequencies. The big.LITTLE approach has, as with low-power ARM processors generally, been widely adopted by CPUs intended for mobile devices. Such an architecture is easier to handle in such devices, which make use of system-on-a-chip (SoC) boards loaded with ASICs for specific tasks such as digital signal processing, so the programming models readily support offloading of specific tasks to specific hardware. The HPC community's interest in ARM has gone in a different direction, instead focusing on high-power server-grade ARM processors such as the Cavium (now Marvell) ThunderX2, used in EPCC's Fulhame [9] and GW4's Isambard [10] HPC facilities. These facilities were not included in the work of EPiGRAM-HS because from an application perspective they look largely the same as x86 architectures [11, 9]. Taking advantage of some low-level features of the ARM server processor architectures requires minor code changes, but in general already existing libraries and programming models can be used just as they are on Intel- or AMD-based supercomputers. It is not entirely surprising that big.LITTLE has failed to take hold in HPC, since the 'big' component still targets low-power mobile devices.

2 MPI on FPGAs

Modern FPGAs are capable of communication across the PCIe bus. Furthermore, such devices can now send/receive data directly to/from the network. It should be possible to distribute tasks across FPGAs even if the dependencies between those tasks must be resolved via communication of data.

The work presented in the forthcoming sections was done using the Xilinx Adaptive Compute Cluster (XACC) [12] hosted at ETH Zürich [2]. This cluster features four Xilinx Alveo servers controlling access to ten Xilinx Alveo FPGA boards. Access is managed via dedicated virtual machines running on the servers, e.g., the `alveo4b` VM running on the `alveo4` server controls access to a single Alveo U280 [13] board.

We investigated the feasibility of using FPGAs as an alternative environment for MPI-based workflows. Specifically, we looked at how easy it is to setup two FPGAs such that

they can exchange data. This work would give us an initial impression of how feasible it is to conduct halo swap operations whereby FPGAs communicate directly. Secondly, we explored how a pool of FPGAs could be managed as an independent computational pipeline. In this scenario, each FPGA operates on the data received before passing it on to the next FPGA; the last FPGA passes the transformed data back to the initiating CPU-based MPI rank.

The investigations were done using the Alveo U280 boards accessible via Alveo servers 3 and 4 of the ETH Zürich XACC cluster. These tests were directed from a single lead VM, which necessitated the setting up of a Dask [14] cluster such that all VMs hosted a Dask worker with the lead VM hosting a Dask scheduler also. In this way, all FPGAs could be accessed from a python notebook running on a single VM.

The software developed for our work was adapted from that provided by two Xilinx Github repositories, XUP Vitis Network Example (VNx) [15] and Accelerated Collective Communication Library (ACCL) [16]. We will now introduce each of these two software collections before detailing our specific investigations.

2.1 VNx

VNx is a library for FPGAs [15], providing a kernel and Pynq overlay to send and receive network data via a UDP/IP (as opposed to the usual TCP/IP) stack. The network data is exposed to user kernels and the Pynq overlay allows Python-based co-ordination from the host such as retrieving results, kernel start/stop and basic debugging. Some exemplar kernels and workflows are provided: the "basic" kernel in which data can be sent and received to and from the network via the FPGA, but where processing is done on the host, and the more complex "benchmark" kernel. The benchmark kernel allows FPGAs to be configured in one of four modes: producer, consumer, loopback or latency. In the producer mode, the main finite state machine (FSM) creates 120 bit payloads which are pushed to the UDP stack at a user-selectable rate; the UDP packet size is variable and user-selectable, but is filled with dummy data, other than the payload header from the FSM, it is not designed to be used. In the consumer mode, the payload is received and Python-accessible counters updated; the data itself is not used beyond collection of performance metrics. In the loopback mode, the data is simply read from the network and then written to the network, either back to the original source, or to another destination. In the latency mode, packets are produced, as for the producer mode, but the packets themselves are particularly short and contain only timestamps. Packets are also received, both input and output streams operating simultaneously, as in the loopback mode. The timestamps are read and compared to the current time to allow determination of packet latency across the network, with results being presented to the Python interface as in the consumer mode.

2.1.1 Data exchange

Using the basic image-transfer example from the Xilinx VNx repository [17], one can see how data can be moved from one host to another by way of the FPGAs on both hosts acting as network interfaces. The example consists of two parts, a Python notebook (which can be easily converted to a Python script), and a set of kernels written in C++.

The image transfer example loads an image from file into application memory (in this case a host-based Python buffer) which is then pushed into a buffer in High Bandwidth Memory (HBM) on the FPGA. Once the `mm2s` kernel is started, it calls the VNx module to transfer the data to a pre-defined destination using UDP over IP. The VNx module handles all of the networking, exposing an API which extends Pynq [18] to allow users to set destination IP addresses and port numbers from Python code. At the receiving side, the VNx stack is given a buffer into which to write and once the sending kernel has begun, no application code intervention is required.

It is easy to envisage how this leads to a compute-offload scenario whereby the receiver waits for data to arrive, with some appropriate signal to indicate sufficient data has been received, performs some computational operation on the buffer of data, and sends this onward via the network. Given the FPGA can access both the host, and the network, it has the flexibility to pass some element of computation to the host, where it would be a poor fit for the FPGA (e.g. a highly-iterative kernel, or an algorithm requiring more memory than the card has available).

We modified the C++ kernels given in the example to test the difficulty of adapting them to perform more than a simple byte copy from HBM memory to the network stack, however, we were unable to reliably perform work in the kernel — compilation was successful in each case with no errors of timing, lack of resource (the design is small compared to the available space on the device) or runtime failure (i.e. causing the card to soft-lock). Further thought and investigation would be required to fully understand the reasons for failure, likely requiring use of network monitoring to verify emission of packets into network and in-design debugging to verify creation of packets and verify validity at each stage of the design.

Testing the image transfer code was done as shown below for the pipeline benchmark, including the use of Dask to co-ordinate the execution, except that only two nodes were required, `alveo4b` and `alveo4c`. The performance results were as expected, the image was successfully transferred from the host `alveo4b` to the FPGA attached to it, from there to the FPGA attached to `alveo4c` and then on to host `alveo4c` itself. Any network performance numbers are somewhat meaningless as the data was about 100$\tilde{\text{k}}$B in size and the network had 100$\tilde{\text{G}}$bps bandwidth giving a very short time, subject to jitter. The pipeline benchmark was designed for measuring such performance and results are below. Unexpectedly, we found that running the image transfer test multiple times failed, with only the first iteration completing successfully. Some short diagnoses found that the sticking point was transferring the data across the network, likely with the sending FPGA not correctly pushing the packets onto the network itself. Unfortunately, the restricted environment at ETHz did not allow more in-depth examination in the time-frame of the project.

2.1.2 Pipeline benchmark

It is not too difficult to adapt one of the VNx benchmark notebooks [19] such that data is passed through a 4-stage FPGA pipeline. We first checked out the VNx repo [15] and then built the CMAC and Network Layer kernels using Xilinx Vitis 2020.1 on the ETH Zurich XACC development server, `alveo0`. This process is detailed by the commands shown below.

```
git clone \
    https://github.com/Xilinx/xup_vitis_network_example.git --recursive
cd ./xup_vitis_network_example/NetLayers
export LIBRARY_PATH=/usr/lib/x86_64-linux-gnu
make all DEVICE=xilinx_u280_xdma_201920_3
```

The next stage involved building the VNx benchmark user kernel.

```
cd ..
make all DEVICE=xilinx_u280_xdma_201920_3 DESIGN=benchmark
```

A Python script was then developed [20], one that allows the user to set the operational modes for the user kernels running on specific Alveo U280 cards. For a 4-stage pipeline, this script can be called like so.

```
python benchmark-pipeline-switch.py -s 10.1.212.126 -p 8786 \
    -f 10.1.212.156 10.1.212.157 10.1.212.159 10.1.212.160 \
    -g 60512 62177 63842 65507 \
    -x ./vnx_benchmark_if0.xclbin
```

The first IP address is for the Dask scheduler (running on the `alveo3b` VM); this command line argument enables the python script to instruct the four Dask workers to download the correct binary image ("`vnx_benchmark_if0.xclbin`") to the FPGA cards controlled by the worker VMs. The IP addresses and port numbers for the FPGA cards are specified by the `-f` and `-g` options.

This new pipeline benchmark simply sets the `alveo3b` FPGA to be the producer and `alveo4c` to be the consumer. The user kernels of the other two FPGAs, `alveo3c` and `alveo4b` are set to loopback mode. The forwarding addresses are set such that data flows from `alveo3b` to `alveo3c` and then `alveo4b` before reaching `alveo4c`.

The pipeline was tested with 1 MB data streams split in to packets of various sizes. Some of the output produced by the pipeline benchmark script is reproduced below.

```
Sent 1,000,000 size: 1472-Byte done!
Got  1,000,000 took 0.1227 sec, throughput: 95.937 Gbps

Sent 1,000,000 size: 1408-Byte done!
Got  1,000,000 took 0.1176 sec, throughput: 95.750 Gbps

Sent 1,000,000 size: 1344-Byte done!
Got  1,000,000 took 0.1125 sec, throughput: 95.546 Gbps

...

Sent 1,000,000 size:  192-Byte done!
Got  1,000,000 took 0.0233 sec, throughput: 65.874 Gbps

Sent 1,000,000 size:  128-Byte done!
Got  1,000,000 took 0.0167 sec, throughput: 61.440 Gbps

Sent 1,000,000 size:   64-Byte done!
Got  1,000,000 took 0.0107 sec, throughput: 48.023 Gbps
```

The recorded throughput speeds match those seen for the original VNx benchmark [19], and, crucially, the pipeline tests were repeatable. The tests were also successful when running with a shorter pipeline, one with three FPGAs for example.

One final point to note here is that the throughput tests had to be done in a specific order: the data was sent by descending order of packet size. Attempting to use the smallest packet size first would result in the last FPGA in the pipeline (the consumer) not receiving any data. As elaborated in Section 2.1.4, there is some fragility in the kernel code whereby small changes can mean the difference between success and failure.

We next explored how to amend the kernels such that some computation could be carried out on the incoming data stream. In this way, we move closer to the concept of an FPGA pipeline where each FPGA transforms the data.

The kernels are written in SystemVerilog[21] and implement a finite state machine (FSM) which handles packet creation, forwarding, consumption, and monitoring logic. The code is well written but the structure imposed by the design was difficult to adapt without large re-design.

Our first foray was to add extra bytes to the payload - it was already of a fixed size so we were careful to ensure that any additions did not push packet sizes to exceed 64 bytes which would have affected the UDP packet sizing and timing logic which expects 64 byte payloads to be delivered at a pre-determined rate. In the `segment_generator.v` file, the section `PRODUCE_PAYLOAD_HEADER` was adapted to be as shown below.

```
PRODUCE_PAYLOAD_HEADER : begin
    if (axis_producer_free) begin
    producer_tdata[ 39:  0]      <= packet_counter;
    producer_tdata[ 79: 40]      <= free_running_counter[39:0];
    producer_tdata[119: 80]      <= number_packets_1d;
    producer_tdata[151:120]      <= nick;
    producer_tkeep               <= {((AXIS_TDATA_WIDTH/8)){1'b1}};
____producer_tvalid_____________<=_1'b1;
```

The `nick` variable contains the value `32'hBEEF`, ie it is a 32 bit wide register with the hex value `BEEF` for ease of recognising it in any subsequent data processing steps, but could contain other data.

The interfaces to the segment generator module, which is called by the traffic generator class are of 512 bits, the width of the streaming AXI interfaces to the networking modules (the VNx parts), and 128 bits to the monitoring port used for summary data collection by the latency measurement benchmark. Extending the segment generator module to add additional interfaces was likely to require extensive design changes given the complexity of the state machine.

Instead, we tried to add additional bit-width to the `bandwidth_reg` module which monitors the bytes sent and received between the network kernels and the benchmark kernels to count packets, bytes and produce timing information. However, given the re-use of this module throughout the design, changing the interfaces to accommodate ex-filtration of additional data proved too complex.

The final option tried was to adapt the AXILite interface used to report such information and, in place of the byte counter, report the number of times the `BEEF` data had been seen in data passing the module. Unfortunately, this also did not function, likely due to the byte counter being used elsewhere in the design and being corrupted by this patch.

In hindsight, a more complex option of adding or replacing the `segment_generator` module with a version which directly included an AXILite port, or interface to the existing

debug port, would have been perhaps easier, although this would have required more work upfront to adapt the block design.

2.1.3 ACCL

ACCL[16] is a project providing an FPGA kernel, Python overlay (using the Pynq library) and other low-level drivers which aim to provide MPI functions (including collectives) on a Xilinx FPGA such as those used in the above examples.

ACCL is designed to enable compute kernels resident in FPGA fabric to communicate directly under host supervision, but crucially, without requiring data movement between the FPGA and host. Instead, ACCL uses Vitis-compatible TCP and UDP stacks (such as those provided by VNx) to connect FPGAs directly over Ethernet at up to 100 Gbps on Alveo cards.

This is an exciting proposition as it allows compute kernels to be resident in the FPGA fabric and the MPI interfacing to be very similar to that used on the host. Essentially, this provides a mechanism for FPGAs to be self-hosting in an MPI communicator with little host control, and potentially, with no host control, save for initial launch, reset etc, so long as access to host memory is not required.

Unfortunately, we were unable to get the ACCL code to run successfully. We do note however, that it is strictly tagged as under development and not yet suitable for production use, so future revisions may bring it to a functional state.

2.1.4 Fragility

As noted in the above sections, even examples supplied by vendors did not run smoothly in every case. There is a connection between the Xilinx Vivado Design Suite (i.e., the compiler) and the Xilinx Runtime Library (XRT) used at build time, and that used at runtime. In our experiences, small changes in benchmark version, changing compiler version or XRT version could have an effect on the outcome of the test or benchmark. In the image transfer example, this could be seen when an FPGA binary compiled from an older version of the source code and with an older compiler would run a single time, but a newer version of the code would not run even a single time when compiled with a newer compiler. It is unclear if this is a compiler issue, a source code issue, or an issue with a mismatch between compile and run-time environments.

However, the ACCL *pre-release* code also encountered similar issues – unexpected stalls in the network stack which appeared as if bytes were not being sent or received correctly. From this we can observe that vendor support to build more stable toolchains is crucial in advancing the use of FPGAs in HPC as performance portability would require some leeway in range of compilers, run-times and execution environments that would produce good performance results. This is especially true when one considers that compilation of a simple example can take approximately 2 hours on a high-performance workstation.

3 GASPI on FPGAs

As it was shown in D2.3, the GASPI memory management and communication mechanisms can be used for distributed computation in heterogeneous systems with FPGA accelerators. For devices connected to a CPU host over PCI-e, which is the typical hardware setup found in different clusters, the communication between a pair of FPGAs is done through the hosts, using the one-sided asynchronous mechanisms and the configurable RDMA memory segments of GPI-2. The high-level OpenCL API is used in turn to manage the communication between parent host and device, through buffer objects which point to the GPI-2 segments. For a GASPI write-notify operation, the sender host starts the communication when the data-transmission from its PCI-e attached device has finished. On the receiver side, on the other hand, the device can read the data once the GASPI notification has arrived to its corresponding host. This lightweight approach does not require particular extensions of the GASPI specification and allows fast and easy prototyping of GPI-2 applications on FPGAs, however, at least two fundamental improvements can be considered in order to build high performance applications. First, the data transfers between host and device can be overlapped with kernel execution; second, the host CPUs participation and the buffer copies in the system memory can be eliminated by communicating the devices directly over the fabric. The former change only requires the OpenCL API to set up properly operation dependencies and to synchronize host threads and device operations (although one can consider to encapsulate into GPI-2 the OpenCL events and synchronization API calls). The second is more elaborate and, in general, can require low level programming of the DMA capabilities of the accelerators, if any are provided. In contrast to GPU accelerators, where intranode or internode DMA features are nowadays offered out of the box by the vendors (naturally, depending on the device), for FPGAs this demands a more detailed knowledge of the hardware specifics. Apart from some specific IP software (e.g., Xilinx QDMA), the usage of DMA for intranode FPGA to FPGA (or even FPGA to GPU [22]) communication, as well as, a FPGADirect-RDMA API [23] is still on-going research. The forthcoming technologies, e.g., smart-NICs, can also play an important role in the problem. This, and upcoming advances in hardware, will demand a suitable software development in order to increase the throughput, and reduce the latency, in heterogeneous applications using not only FPGAs, but also other types of accelerators with low power processors.

4 Conclusion

In this document we report experiences of using the networking technologies MPI and GASPI on FPGAs. While it's clear that FPGAs have potential to be powerful accelerators in HPC machines at exascale and beyond, using networked FPGAs for distributed computation is still extremely challenging. This challenge is exacerbated by the general difficulty of programming FPGAs. Higher level tools exist, such as Xilinx High-Level Synthesis (HLS) which allow the user to generate designs from C/C++ code, and as noted in this deliverable, python interfaces exist, but programming for performance often involves significant redesign of compute kernels to use pipelines which are suited to the FPGAs' dataflow architecture. It is clear that more work needs to be done to support application developers in making use of FPGAs, but also to ensure that they can be used robustly. We noted that even simple applications could be extremely fragile to both the build and runtime environments, even when working exclusively with one vendor's tools and hardware, which means portability is problematic. In the work done on GASPI, OpenCL was deployed to get around this issue, but communication was done exclusively between hosts rather than direct between FPGAs.

A key finding overall was that vendor support is crucial, given the requirement for very specific versions of all the tools necessary. This is in part also because building more complex programs for an FPGA is typically done by composing vendor-licensed 'IP blocks', rather than implementing network drivers yourself for example. More success was had with the higher-level interfaces which were developed by vendor experts – the obvious drawback however is that it becomes extremely difficult to debug issues which are at the closed source lower level. Good relationships with and support from vendors are therefore necessary to ensure timely resolution of such issues.

Given the above, it's hard to recommend FPGAs for large-scale distributed computation currently. With that said it is hoped that the full release of the ACCL library, and FPGADirect-RDMA APIs will go some way to alleviating this. With those in place, it will be easier to make specific recommendations for extensions to the MPI standard, as one can better examine gaps in functionality. At this early stage we can however recommend that in order to better support FPGAs, MPI should extend its persistent interfaces, and implementations should make use of one-sided communication where possible. Persistent communications using one-sided communications are conceptually similar to streaming interfaces, which are well supported on FPGAs, and a more natural fit to the dataflow architecture. Furthermore, the functionality required for polling for messages would take up valuable space in the FPGAs reprogrammable fabric. This is the focus of some of the work done by UEDIN in WP2. The situation is better for GASPI, where the standard already enables incorporation of FPGAs, but improvements allowing more advanced usage are an active area of research.

References

[1] Stephen M. Trimberger. Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology. *Proceedings of the IEEE*, 103(3):318–331, 2015.

[2] XACC ETH Zurich. ETH Zurich XACC Cluster. `https://xilinx.github.io/xacc/ethz.html`. Accessed: 2021-10-22.

[3] Xilinx. Alveo U280 Data Center Accelerator Card Data Sheet. `https://www.xilinx.com/content/dam/xilinx/support/documentation/data_sheets/ds963-u280.pdf`, September 2021. Accessed: 2021-10-27.

[4] Nick Brown, Mark Klaisoongnoen, and Oliver Thomson Brown. Optimisation of an FPGA credit default swap engine by embracing dataflow techniques. *CoRR*, abs/2108.03982, 2021.

[5] DEEP Projects. DEEP-EST prototype system. `https://www.deep-projects.eu/hardware/prototypes.html`. Accessed: 2021-10-27.

[6] X. Guo, M. Bull, W. Klijn, L. Kusch, M. van der Vlag, K. Sontheimer, S. Durr, I. Kabadshow, L. Morgenstern, O. Brown, M. Bareford, S. Krieg, E. Gregory, C. Alexandrou, and L. Breitweiser. DEEP-EST D7.5 Case Study Booklet. `https://cordis.europa.eu/project/id/754304/results`, 2021. Accessed: 2021-10-29.

[7] ExCALIBUR. FPGA testbed. `https://excalibur.ac.uk/projects/fpga-testbed/`. Accessed: 2021-10-27.

[8] ARM Limited. big.LITTLE resources. https://www.arm.com/why-arm/technologies/big-little. Accessed: 2021-20-28.

[9] Adrian Jackson, Andrew Turner, Michèle Weiland, Nick Johnson, Olly Perks, and Mark Parsons. Evaluating the arm ecosystem for high performance computing. *CoRR*, abs/1904.04250, 2019.

[10] GW4. Isambard. `https://gw4.ac.uk/isambard/`. Accessed: 2021-10-28.

[11] Simon McIntosh-Smith, James Price, Tom Deakin, and Andrei Poenaru. A performance analysis of the first generation of hpc-optimized arm processors. *Concurrency and Computation: Practice and Experience*, 31(16):e5110, 2019. e5110 cpe.5110.

[12] Xilinx. Xilinx adaptive compute cluster (xacc) program. `https://www.xilinx.com/support/university/XUP-XACC.html`. Accessed: 2021-10-22.

[13] Xilinx. Alveo U280 Data Center Accelerator Card. `https://www.xilinx.com/products/boards-and-kits/alveo/u280.html`. Accessed: 2021-10-22.

[14] Dask Development Team. *Dask: Library for dynamic task scheduling*, 2016.

[15] Xilinx. Xup vitis network example (vnx). `https://github.com/Xilinx/xup_vitis_network_example.git`, 2021.

[16] Xilinx. Accelerated collective communication library (accl). `https://github.com/Xilinx/ACCL.git`, 2021.

[17] Xilinx. Vnx basic image transfer. `https://github.com/Xilinx/xup_vitis_network_example/blob/master/Notebooks/vnx-basic-image-transfer.ipynb`, 2021.

[18] Xilinx. Pynq: Python productivity. Accessed: 2021-10-28.

[19] Xilinx. Vnx benchmark throughput switch. `https://github.com/Xilinx/xup_vitis_network_example/blob/master/Notebooks/vnx-benchmark-throughput-switch.ipynb`, 2021.

[20] Michael R. Bareford. Vnx pipeline benchmark python script. `https://github.com/mbareford/Xilinx-VNx-pipeline-benchmark.git`, 2021.

[21] IEEE. Ieee standard for systemverilog–unified hardware design, specification, and verification language. *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, pages 1–1315, 2018.

[22] Direct communication between FPGA and GPU using Frame Based DMA (FDMA). `https://blog.zhaw.ch/high-performance/2019/10/03/direct-communication-between-fpga-and-gpu-using-frame-based-dma-fdma/`.

[23] Ryohei Kobayashi, Norihisa Fujita, Yoshiki Yamaguchi, and Taisuke Boku. OpenCL-enabled High Performance Direct Memory Access for GPU-FPGA Cooperative Computation. In *Proceedings of the HPC Asia 2019 Workshops on ZZZ - HPCAsia'19 Workshops*, pages 6–9, Guangzhou, China, 2019. ACM Press.