Enabling Standalone FPGA Computing

Joshua Lant, Javier Navaridas, Andrew Attwood, Mikel Lujan, John Goodacre
School of Computer Science, University of Manchester, UK, M13 9PL. Email: {firstname.lastname}@manchester.ac.uk

***Abstract*—One of the key obstacles in the advancement of large-scale distributed FPGA platforms is the ability of the accelerator to act autonomously from the CPU, whilst maintaining tight coupling to system memory. This work details our efforts in decoupling the networking capabilities of the FPGA from CPU resources using a custom transport layer and network protocol. We highlight the reasons that previous solutions are insufficient for the requirements of HPC, and we show the performance benefits of offloading our transport into the FPGA fabric. Our results show promising throughput and latency benefits, and show competitive Flops being achievable for network dependent computing in a distributed environment.**

***Index Terms*—Interconnects, FPGA, Transport Layer, HPC.**

1. Introduction and Motivation

FPGAs have shown great promise for next generation HPC systems, given their strong performance and energy characteristics when faced with data-intensive and stream-like workloads. Traditionally shunned due to their difficult programmability and low off-chip memory bandwidth (vs. GPU), maturing ecosystems, larger on-chip memories and integrating advanced memory systems (e.g. HBM) mean architects are now focusing on the potential of FPGAs in HPC/data-center systems. For instance, Microsoft [1] are turning towards the FPGA in their data-centers to improve power-efficiency, in a domain where power consumption is becoming an ever growing issue. For example, in [2] it is shown that the FPGA can achieve similar performance to GPUs on a number of the BLAS routines, but achieving much higher energy efficiency. We anticipate that as tighter power constraints are placed upon systems, we are likely to see the FPGA feature as a standard component of large heterogeneous HPC clusters. Another advantage of FPGAs over GPUs is the flexibility for custom optimizations and fine grained parallelism to achieve greater performance. As an example, the feed-forward nature of computation involved in Deep Neural Networks (DNNs) can benefit from very deep pipelining. We can also take advantage of custom data-types and reduced precision [3], [4].

However, a *key barrier* to the exploitation of FPGAs within the context of HPC systems is they are traditionally used as a mere co-processor [5], loosely coupled to the CPU and network resources—attached via PCIe or other equivalent bus-based interconnect (see Fig. 1a). This architectural model exacerbates the limited off-chip memory bandwidth of the FPGA by distancing the accelerator from the memory hierarchy of the system. In addition it severely limits the feasibility of data-flow processing among distributed FPGAs because they depend on the CPUs for communicating.

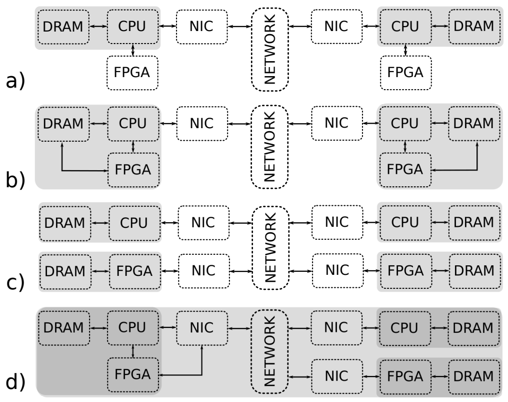

Figure 1. Possible distributed FPGA System Configurations, showing the bounds of memory addressing. a) Loosely coupled through PCIe. b) Tightly coupled to the CPU. c) FPGA as an independent network peer. d) Independent network peers within a global memory space.

Modern SoCs such as Xilinx Zynq Ultrascale+, and Intel Stratix 10, feature tighter integration of FPGAs and hard-core processors by supporting cache-coherent shared memory between the CPUs and FPGA, see Fig. 1b. While this allows for lower latency local transfers, it does nothing to alleviate the overhead of cumbersome SW network stacks such as TCP/IP.

The community agrees that the remedy to the issue is to promote the FPGA resources to the status of a full peer within the network [6], capable of issuing and receiving its own reliable transactions. Enabling the FPGA to perform RDMA operations directly and offloading network stacks into HW, i.e. TCP offloading, allow the FPGA to become a peer within the network, fully disaggregated from the CPU, see Fig. 1c. Such an architecture enables FPGA resources to be scaled out without increasing the number of CPUs. However, in this setup the FPGA is unable to exploit a lower latency, shared memory model with other distributed memory spaces; a property which is vital for workloads with irregular memory accesses, and which has been shown in a recent study to be a limiting factor for many of the identified workload types to be performed efficiently on FPGAs [7].

Our solution is the first (to our knowledge) to maintain both tight-coupling with system memory, *and* fully decoupled networking capability. We analyze our lightweight NIC which leverages a custom network protocol based on a simple geographic addressing scheme [8]; supporting HW prim-

TABLE 1. AREA UTILIZATION FOR NETWORK INTERFACE.

Xilinx ZCU102	LUTs	FFs	BRAM	DSPs
Available Resources	274,080	548,160	912	2,520
IO/On-Chip network stack total (See Fig. 2.)	46,915 (17.1%)	58,734 (10.7%)	126.5 (13.9%)	0 (0%)
NIC utilization	22,670 (8.2%)	18,986 (3.4%)	65.5 (7.1%)	0 (0%)

itives for both shared memory and RDMA operations, and including a custom reliable transport layer. Our system-level transport layer enables modern, low-diameter topologies to be built, rather than using simple point-to-point connections as most of the literature uses (limiting scalability). Our solution enables the configuration shown in Fig. 1d, where the FPGA acts as a fully disaggregated peer on the network, but can also write directly into a shared memory space between the CPU and other resources (local or remote). This opens up the possibility for fine grained acceleration across distributed FPGAs, as well as near-data processing in the network. Earlier work [9] details a preliminary version of this NIC; lacking capability for direct communication between distributed accelerator resources, enabling communication between CPU and remote resources only.

2. Our Solution

We proceed to detail our solution, including our transport protocol. First, we show the resource utilization of our design in Table 1. Note that our NIC burden is not beyond normal expected bounds [10]. We devised a connectionless (datagram-based) transport, which maintains a small and, more importantly, transient amount of information within the NIC about in-flight transactions to provide reliability, rather than keeping static connection state information about each existing src-dest pair.

Our NIC contains two segregated data and control paths; one for shared memory transactions, and another for RDMA operations. In this manner the FPGA fabric is capable of writing shared memory operations directly to the NIC, enabling it to submit work directly to remote accelerators. Data can be written to a remote RAM (on-chip or off-chip) using the DMA engine, and shared memory operations can be used to inform the remote accelerator that there is new data to be worked on. This decision was made because these two methods have very different requirements and properties:

Shared-memory operations are formed of small messages, so we decided to store data in the NIC to minimize latency while still offering end-to-end reliability.

RDMA operations normally consist of much larger data transfers. This makes storing data in the FPGA prohibitive, so our NIC keeps data in their original location and tracks outstanding RDMA operations to retransmit partial transfers if needed, again, providing end-to-end reliability.

As an example, a user-space application sending a 16B transfer to the BRAM of a remote FPGA (1-hop distance) will take 1.1μs using our shared memory engine, but 1.49μs using our RDMA engine. The benefits of performing smaller operations using our dedicated shared-memory path are

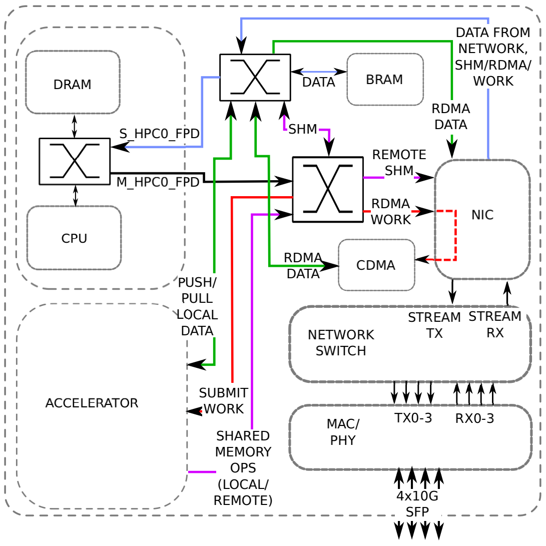

Figure 2. Full IO and on-chip network stack within FPGA fabric.

clear: reducing the latency by over 25%. These savings come from the shared-memory path using a transparent write instruction into remote memory whereas the RDMA engine requires an extra memory copy.

Fig. 2 shows the full network stack within the FPGA, which allows the accelerator logic to issue commands to the NIC in exactly the same way as the CPU. This is enabled by the use of memory-mapped AXI interfaces through the whole stack. We effectively translate AXI transactions into our custom packet format, whilst providing additional information so that it can be used effectively over a wider, full-system scale network; serialized and transmitted over high speed transceivers. The AXI transactions are then rebuilt remotely at the receiving NIC. This means that both the accelerator and CPU can access the network completely independently from each other, and the standard interfacing lends itself very well to HLS programming models.

The RDMA engine consists of a Xilinx AXI CDMA, configured in scatter-gather mode. RDMA commands are issued as simple AXI writes (shared memory operations) through the NIC, which passes commands to the CDMA, while building and maintaining meta-data regarding the operations in order to implement the reliability mechanism. The accelerator communicates with the NIC using shared memory operations. If addressed to the command queues in the NIC for the local RDMA engine this will issue data transfers to remote nodes. If addressed to remote nodes the same mechanism can be used to issue direct read and writes to remote memory locations. By doing this direct translation between the wider network protocol and simple memory-mapped read and write operations, two setup scenarios are enabled. We enable both a distributed shared-memory type setup, with FPGA resources tightly coupled to the system bus of the CPU, affording fine grained parallelism. In addition, we enable a setup where the FPGA is completely disaggregated from the CPU resources, allowing for non-linear scaling between FPGA and CPU resources in a full system architecture. This decoupled scaling is targeted by other works [11], [12], [13], and may be very desirable for systems which use the FPGA as the main compute element, or in creating more flexible low-diameter topologies to interconnect acceleration resources.

In order to illustrate the benefits of our approach, Fig. 3 shows the critical path for data and control communications for a CPU (in F1) to use a local FPGA and then send the data to be processed at a remote FPGA (in F2). Fig. 3a represents a traditional TCP stack, which requires multiple extra copies of the data between buffers *and* intervention from the CPU, creating significant additional latency. Fig. 3b shows a SW implementation of our transport layer. Now, we are able to submit work directly to the remote accelerator, reducing the latency induced at F2. However, additional control and off-chip DRAM access is needed in F1 because the CPU is needed to coordinate between the two accelerators. Finally, Fig. 3c shows our HW-offloaded solution. In this instance, once the accelerator at F1 has completed its work it issues an RDMA operation directly to the NIC, and then writes shared memory operations to the F2 accelerator's work buffer, informing it that there is new work to be performed. Once this is completed then the F2 accelerator notifies its local CPU that the work has been done and it has new data to process. As is shown, this solution is far more amenable to data-flow type processing, allowing for simpler pipelining through the distributed FPGA resources.

While several other solutions [14], [15], [16], allow for this sort of dataflow processing, they typically only support point-to-point links between the FPGAs, severely limiting the topologies which can be created [16], which in turn severely limits scalability. Combined with our switch design [8], our solution can exploit modern HPC topologies such as Jellyfish, Dragonfly and Fat-Trees.

3. Experimental Work

Our experiment is performed on the Xilinx Ultrascale+ ZCU102 development board, with all logic and transceivers within the FPGA fabric run at 156.25MHz in order to enable saturation of the 10G SFP links with a 64-bit wide data-path. Since there is no distributed runtime environment available for our HW, we are restricted to the use of a single FPGA for our experiments. In addition this ensures clock consistency and accurate timing measurements. To emulate the distributed setup with complete accuracy, the entire HW acceleration and network stack is implemented twice within the same FPGA, using a partitioned memory space to maintain complete independence between the two portions. Not only are the memory spaces segregated, but the data path to the memory hierarchy of the CPU is also completely segregated. Every component of the network stack and the accelerator block in Fig. 2 is implemented on the board twice (with the exception of the MAC/PHY), with the second instantiation interfacing with the hard-core Processing System via a separate interface port, and the two subsystems being connected to each other via separate TX/RX ports of the transceivers. This eliminates any

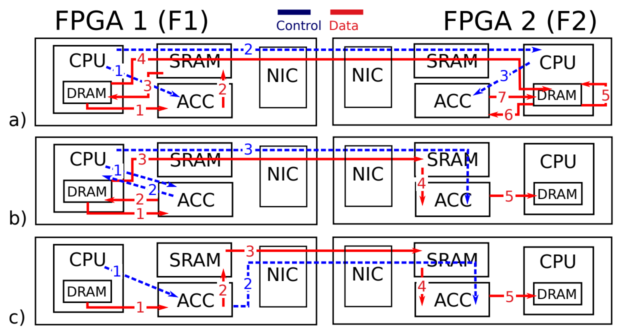

Figure 3. Control and data paths for a) SW based TCP, b) custom SW transport, c) our HW offloaded solution.

contention for resources, and emulates accurately a fully distributed environment.

In the experiment we recreate the setups described in Fig. 3b and c. A user-space application takes a system time-stamp before submitting work to the accelerator at F1, which performs some work and sends the result forward to the accelerator at F2. Upon reception F2 performs some work and then writes the result in its local memory and notifies the CPU, which will take another system time-stamp to determine the overall time for the application to run at the application-level. For simplicity, we keep the data size constant and use a dummy accelerator block which does not perform any genuine function, but merely adds a *computation latency*. Data must be transferred to the accelerator and worked on in blocks, as would be the case in typical implementations for HPC operations such as matrix-vector or matrix-matrix multiplication. We adjust the block granularity for the data, and adjust the *computation latency* with relation to the pure communication time for a given solution (block size and data path). A computation/communication ratio of zero denotes instantaneous processing time: Data (at block level granularity) is simply written into the accelerator block and back out. A communication/computation ratio of R denotes that the system spends R times as long computing inside the accelerator as the communication path on moving data and control information around the system. For instance, with $R = 1$, each accelerator will spend the same time computing as the whole system does communicating.

Fig. 4 shows latency, and throughput *in terms of purely remote-memory bound data processing per processing element* (not to be confused with the communication throughput over the links or the raw computing throughput in Flops). We see that latency is improved (up to ≈29% reduction) for small and medium block sizes, which could have dramatic effects on tightly-coupled applications with many small messages, especially if they have irregular access patterns, such as workloads involving pointer-chasing, with list, tree or graph traversal for example; workloads which are of increasing interest within the FPGA community [17].

If we focus on throughput, we see a throughput gain of 8.6% over the SW transport solution. However, there appears to be a saturation point towards the upper limit of the block sizes, suggesting that further increasing the maximum block size for a single accelerator module will

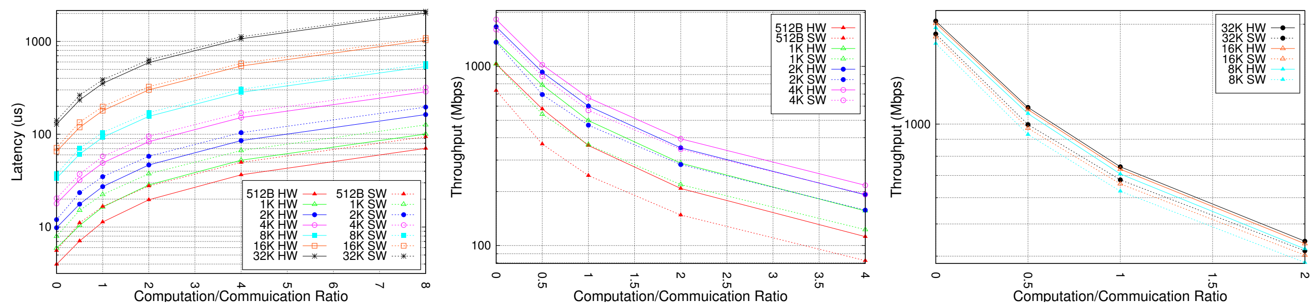

Figure 4. Latency to perform a single operation (left). Achievable throughput for small block sizes (middle) and larger block sizes (right).

not translate into higher performance. Note however, that we support multiple accelerator blocks within the same FPGA to exploit spatial parallelism [16]. We proceed to estimate the achievable memory-bound Flops of such a solution, using methods similar to [10], [18]. Let's assume a 1KB block size for transfer, feeding an accelerator block 128 double precision floats to perform an 8×8 matrix-matrix multiplication. This gives us 1,024 Flops per block (512 multiply-adds). According to Vivado HLS tools a simple matrix-matrix multiply will have a latency of 288 cycles (or $\approx$0.20 computation/communication ratio in our experiments above and a throughput of 1.1Gb/s). Thus we can make approximately 134,277 block transfers per-second. (1.1×10^9/8,192 bits per block transfer.) Using this we see that we can extract approximately 137 MFlops per IP block (1,024 Flops/block transfer). According to the HLS synthesis output each block requires 11,722 LUTs, with the implementation being LUT bound on the ZCU102 device. Even with the resources used by our network stack, we could theoretically fit 19 IP blocks on a single FPGA. However, if we allow for 9 blocks, which are enough to saturate a single 10G link, we could obtain 1.233 GFlops (double precision) per FPGA (9 Blocks $\times$ 137 MFlops). These results are completely bound by the network, rather than the off-chip memory bandwidth. Compared with [16] where a theoretical peak of 8.9 GFlops over 8 FPGAs (1.11 GFlops per FPGA) was reported, we can claim our communication solution is more effective, particularly since they are limited to a basic ring topology, creating a completely non-scalable solution.

4. Conclusions

In this paper we discussed the need for a HW-offloaded, connectionless transport layer for reconfigurable HPC computing, and present a novel solution to facilitate this. Our solution performs better in terms of latency (up to 29% reduction) and throughput (up to 9% increase) over a similar SW transport, by reducing the complexity of the control and data path through the network. We show that our results are competitive with those in the literature regarding theoretical limits on network-bound compute power. By maintaining the ability for the FPGA to be used simply within a distributed shared-memory setup, the reduced latency could have great impact on parallel applications with irregular parallelism and access patterns.

References

[1] A. M. Caulfield *et al.*, "A cloud-scale acceleration architecture," in *IEEE/ACM Intl. Symposium on Microarchitecture*, IEEE Press, 2016.

[2] S. Kestur *et al.*, "Blas comparison on fpga, cpu and gpu," in *2010 IEEE CS Annual Symposium on VLSI*, pp. 288–293, IEEE, 2010.

[3] E. Nurvitadhi *et al.*, "Can fpgas beat gpus in accelerating next-generation deep neural networks?," in *ACM/SIGDA Intl. Symposium on Field-Programmable Gate Arrays*, pp. 5–14, ACM, 2017.

[4] E. Nurvitadhi *et al.*, "Accelerating binarized neural networks: Comparison of fpga, cpu, gpu, and asic," in *2016 Intl. Conf. on Field-Programmable Technology*, pp. 77–84, IEEE, 2016.

[5] C. Kachris and D. Soudris, "A survey on reconfigurable accelerators for cloud computing," in *2016 26th International conference on field programmable logic and applications (FPL)*, pp. 1–10, IEEE, 2016.

[6] K. D. Underwood *et al.*, "From silicon to science: The long road to production reconfigurable supercomputing," *ACM Trans. on Reconfigurable Technology and Systems*, vol. 2, no. 4, p. 26, 2009.

[7] F. A. Escobar, X. Chang, and C. Valderrama, "Suitability analysis of fpgas for heterogeneous platforms in hpc," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 2, pp. 600–612, 2015.

[8] C. Concatto *et al.*, "A cam-free exascalable hpc router for low-energy communications," in *Intl. Conf. on Architecture of Computing Systems*, pp. 99–111, Springer, 2018.

[9] J. Lant *et al.*, "Enabling shared memory communication in networks of mpsocs," *Concurrency and Computation: Practice and Experience*.

[10] J. Williams *et al.*, "Characterization of fixed and reconfigurable multi-core devices for application acceleration," *ACM Trans. on Reconfigurable Technology and Systems*, vol. 3, no. 4, p. 19, 2010.

[11] F. Abel *et al.*, "An fpga platform for hyperscalers," in *IEEE High-Performance Interconnects (HOTI)*, pp. 29–32, IEEE, 2017.

[12] J. Weerasinghe *et al.*, "Network-attached fpgas for data center applications," in *2016 Intl Conference on Field-Programmable Technology (FPT)*, pp. 36–43, IEEE, 2016.

[13] K. Katrinis *et al.*, "Rack-scale disaggregated cloud data centers: The dredbox project vision," in *Conference on Design, Automation & Test in Europe*, pp. 690–695, EDA Consortium, 2016.

[14] A. Putnam *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 13–24, 2014.

[15] R. Baxter *et al.*, "Maxwell-a 64 fpga supercomputer," in *NASA/ESA Conf. on Adaptive Hardware and Systems*, pp. 287–294, IEEE, 2007.

[16] R. S. Correa and J. P. David, "Ultra-low latency communication channels for fpga-based hpc cluster," *Integration*, vol. 63, 2018.

[17] G. Weisz *et al.*, "A study of pointer-chasing performance on shared-memory processor-fpga systems," in *ACM/SIGDA Intl Symposium on Field-Programmable Gate Arrays*, pp. 264–273, ACM, 2016.

[18] D. Strenski, "Fpga floating point performance–a pencil and paper evaluation," *HPC Wire*, 2007.