Toward FPGA-Based HPC: Advancing Interconnect Technologies

DOI:
10.1109/MM.2019.2950655

Document Version
Accepted author manuscript

Published in:
IEEE Micro

Making the case for FPGA based HPC

Joshua Lant, Javier Navaridas, Mikel Lujan, and John Goodacre

***Abstract*—HPC architects are currently facing myriad challenges from ever tighter power constraints and changing workload characteristics. In this article we discuss the current state of FPGAs within HPC systems. Recent technological advances show that they are well placed for penetration into the HPC market. However, there are still a number of research problems to overcome; we address the requirements for system architectures and interconnects to enable their proper exploitation, highlighting the necessity of allowing FPGAs to act as full-fledged peers within a distributed system rather than attached to the CPU. We argue that this model requires a reliable, connectionless, hardware-offloaded transport supporting a global memory space. Our results show how our fully-fledged hardware implementation gives latency improvements of up to 25% versus a software-based transport, and demonstrates that our solution can outperform the state of the art in HPC workloads such as matrix-matrix multiplication achieving a 10% higher computing throughput.**

***Index Terms*—HPC, FPGA, Interconnect, Transport Layer**

I. Current Trends in HPC

In recent years there has been two great changes which have affected the way systems architects must think about future technology within HPC. The first and most obvious of these is the breakdown of Dennard scaling around 2004. Since this time there has been an explosion in the scale-out of HPC systems architectures, and consequently a dramatic rise in their power consumption.

The second great change is that we have now entered the *Fourth Paradigm* of scientific discovery. Modern applications are moving from computational science into big-data analytics. The rapid growth of data-intensive workloads has instigated a convergence between data centre and HPC technology, and techniques such as hyper-converged storage are increasingly used to bring data closer to compute resources. The price of computation has been falling dramatically for decades, to the point that the energy required for on-chip data movement is now significantly higher than floating point operations [1]. As such, reducing data movement is essential for reducing power consumption.

In order to deal with these changes, accelerated computing has become commonplace within HPC, such that over 25% of the TOP500 list now employ GPU acceleration. GPU manufacturers have catered to the HPC market by providing increasingly powerful architectures, with ever higher floating point performance and memory bandwidth. Unfortunately these massive developments in performance have come at the cost of much higher power consumption. While high-end GPUs are demonstrated to be more efficient than CPUs, this gap is narrowed by appropriate optimizations [2] and thus their performance scalability must be questioned in relation to their current configuration within HPC architectures.

The main issue with these more traditional forms of architecture is the fact that many still view the GPU accelerator simply as a PCIe bus-attached coprocessor. While they may allow limited inter-GPU communication between a subset of the accelerators using protocols such as NVLink, such limitations can cause additional data copying and decrease locality.

A. FPGAs for HPC

Given that reducing data movement is *the* key factor for improving energy efficiency [1], FPGAs have become a promising candidate for increasing energy efficiency of heterogeneous HPC systems. The main rationale for FPGA technology is its ability to use novel algorithms and custom memory layouts to optimize the number of memory accesses. Furthermore, since FPGAs do not require a stored program they alleviate the effects of the von Neumann bottleneck with reduced memory bandwidth requirements. Utilizing the increasing capacity of on-chip memory and using multiple FPGAs is a key method to enable larger datasets to be stored closer to the compute, reducing data movement and increasing memory bandwidth significantly. Since memory bandwidth is the limiting factor for a number of HPC workloads, reducing off-chip DRAM accesses and storing data closer to compute is desirable for increased performance as well as reduced power consumption.

FPGA vendors are also now pushing for penetration into the HPC/data centre arena and are slowly addressing the performance gap between GPUs and FPGAs. We are now starting to see advanced memory systems (e.g. HBM) integrated on the same package. Furthermore the floating point performance of FPGAs is increasing; with hardened floating point blocks and increased DSP capability within the architecture. Such advances are key to enabling the FPGA to be fully exploited in the HPC domain.

The next step towards FPGA based HPC computing– and the one we focus on within this paper– is the ability to scale-out FPGA resources by enabling distributed FPGA computing. We, and many others within the community, argue that for this to be an efficient process, the FPGAs need to break free from CPU control and become full-fledged, network-capable computing units; an objective that we address with a custom interconnect.

B. Programming Models

Given the complete ubiquity of MPI, any new HPC system produced in at least the medium term is certain to use MPI as one of its communication paradigms. While MPI is highly efficient for the CSP (Communicating Sequential Processes) model for parallelism, modern architectural features are not easily represented or exploited in MPI, and shared memory cannot be utilized within MPI ranks on the same node.

The hybrid use of MPI+PGAS (Partitioned Global Address Space) languages is gaining traction as it overcomes some of the main limitations with MPI; providing naturally simpler one-sided operation semantics, reduced memory footprint, and solving issues with overlapping communication and computation. The main hurdle towards a wider adoption of PGAS is the lack of direct hardware support for its communication primitives. Our custom interconnect supports these primitives by enabling to write operations directly into a global memory space.

With respect to the programming of the FPGA hardware itself, the use of traditional frameworks and tools such as OpenCL poses problems as it offers only a host/device model of programming. The code is split into two portions; the *host* code which runs on the CPU, and the *device/kernel* code which runs on the accelerator. An API allows the host code to use the hardware kernel, loading data into the hardware for execution and then transferring the data back out to the host once computation has been completed. We argue that this model entrenches the main limitation for exploiting distributed FPGAs; dependency on the CPU to orchestrate data movement. The communication primitives and interfacing of our NI (see Section V) support data transfer and synchronization directly between accelerator blocks on different FPGAs. This enables more suitable programming models to be explored which facilitate dataflow processing not only at the intra-node level but at a much larger scale.

II. Changing Workloads

While the memory bandwidth of GPUs is incredibly high and is important for many HPC applications, there exist numerous applications which are more sensitive to memory latency, messaging rate and overlap. These have been overlooked in the development of GPU architectures. The Berkeley taxonomy of HPC applications [3] shows that over half of the application types are limited by memory-latency, instead of memory-throughput. Such workloads are well suited for FPGAs as custom memory layouts can be exploited.

Workloads exhibiting irregular memory access patterns combined with high levels of arithmetic computation already provide for more efficient FPGA based implementations over CPU or GPU solutions [4]. Workloads such as stencil codes are suited to the FPGA due to the high volume of on-chip memory, reducing DRAM accesses. Other computations such as sparse matrix-matrix/matrix-vector multiplication are highly suitable for FPGAs, as they feature a relatively low FLOP count per memory access, meaning that on-chip storage for values is preferable in this situation.

The main limitation with FPGAs is currently in their floating point capabilities. Typical number crunching algorithms (dense matrix-matrix/matrix-vector multiplication, FFTs, N-body simulations etc.) obtain the best performance on GPUs. However, if we consider energy-to-solution rather than raw performance, FPGAs offer a clear advantage with much higher FLOPs/watt.

III. Evolving FPGA Architectures

Recent works [4], [5] argue that FPGAs should evolve beyond a simple coprocessor solution towards integrating hard-core CPUs with coherent access to the FPGA fabric. Such architectures enable the effective acceleration of workloads with more complex memory handling, e.g. graph traversal and branch-and-bound problems. Indeed, [5] examines pointer-chasing on modern FPGA substrates with direct, shared-memory access between tightly-coupled accelerator and CPU, concluding that memory latency is the main bottleneck for these workloads, and that interleaving memory access over several concurrent traversals can alleviate this problem.

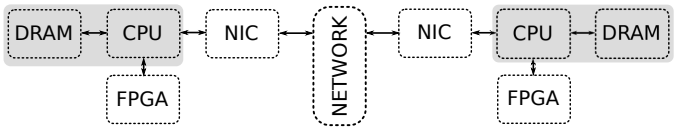
Figure 1 illustrates different levels of memory coupling and addressability of FPGA-based systems. Figure 1a shows the traditional bus-based coprocessor model, in which the FPGA acts as a slave attached to the CPU, which must direct memory copying between main memory and accelerator. Figure 1b depicts a tighter memory coupling between CPU and FPGA, where the FPGA can directly access main memory, but relies on the CPU for communications.

Extending beyond this tight memory coupling is the idea, as shown in Figure 1c, that the FPGA should become an independent, standalone compute element within the system, accessing the network as a full peer and having access to the main memory system. This disaggregates the FPGA resources from the CPU, so they can be scaled independently and multiple FPGAs can communicate with one another over the network, enabling the pooling of FPGA resources.

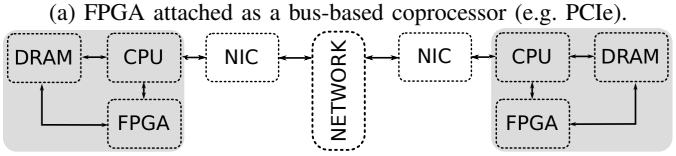
Having the FPGA tightly coupled with the network is vital to allow simple and efficient algorithm mapping onto distributed FPGAs. We argue that a custom Network Interface (NI) is required to allow for this without CPU involvement or heavy limitations on the network (as is seen in TCP offload for example). With suitable work distribution in this system configuration FPGAs may eventually be able to compete with GPUs even on dense linear-algebra problems where the GPU excels [4].

There has been previous effort towards this goal; upgrading the FPGA to a more central role, eliminating the CPU entirely from computation (or at least forcing all network traffic through the FPGA). Forcing all traffic bound for the network through the accelerator is known as a *bump-in-the-wire* architecture, see Figure 1d. Implementations are effective for enabling pools of FPGA resources to be used for scaled acceleration, and for Near Data Processing (NDP) where the FPGA may take control of the network, storage capability, local memory or caches. The most sophisticated of these *bump-in-the-wire* architectures is Microsoft Catapult2. While it offers a powerful platform it does not support NUMA-like operations which is integral for enhancing the performance of workloads with irregular memory access patterns.

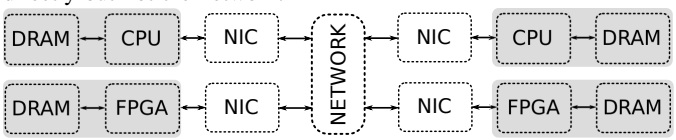
Consequently, we propose the solution in Figure 1e where FPGAs can operate independently within a global shared memory space. Our NI permits scaling-out FPGA resources, by allowing the FPGA to read and write directly into the remote memory of other FPGAs or CPU resources. We encapsulate the system bus protocol for use over the network,



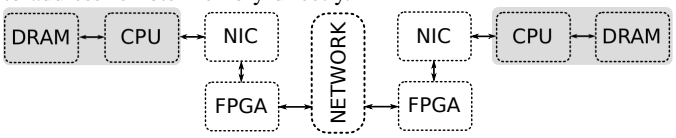
(a) FPGA attached as a bus-based coprocessor (e.g. PCIe).



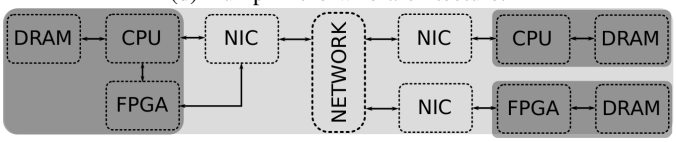
(b) FPGA attached via system-bus, able to access local memory directly but not the network.



(c) Disaggregated FPGA, able to act as a full network peer but unable to address remote memory directly.



(d) Bump-in-the-wire architecture.



(e) Global address space in which the FPGA can act as a full network peer and access remote and local memory directly.

Fig. 1: Possible system architectures and FPGA configurations. The shaded regions represent the limits of the addressability from a given node.

so that any traffic arriving from the NI is dealt with as a local memory access, no matter the origin of the transaction.

IV. Interconnecting independent FPGAs

As we have now established the benefits of advancing towards standalone FPGA computing units, we move to discuss the interconnection substrate required for such architectures. At first glance TCP/IP-based solutions seem like the ideal candidate given their ubiquitous presence in commodity cluster computing. Unfortunately, TCP based reliability suffers severe performance degradation when implemented in software, and hardware offloaded techniques are non-scalable due to the complexity of the TCP protocol and the dedicated per-connection send/receive buffering requirements.

Infiniband is another obvious alternative, but there are a number of issues which make Infiniband unsuitable for FPGA-based HPC. The most significant of these issues is the sheer complexity of the Verbs API specification, which makes hardware offloading incredibly difficult. A clear example can be seen in previous attempts to implement the Verbs API on a GPU [6]. They found that the large overheads involved in work request generation are not compensated by the savings in context switching; concluding that CPUs are better suited for this task. They suggest that GPU architectures will need to be adapted for future systems; including an on-board processor to handle Infiniband interactions. The alternative being to seek a different RDMA capable networking hardware. The same holds true for FPGAs. The only FPGA implementations of Infiniband cores are typically very limited in their feature set, reliability, and number of concurrent connections.

If we look at custom interconnect solutions for distributed FPGA computing, there exist many proposals, but they are typically limited in (at least) one of the following ways which our solution overcomes:

- They provide only simple point-to-point connections between FPGAs; limiting available topologies and scalability of solutions to those typically situated within a single rack or chassis [7].
- They require CPU intervention to issue transactions to the network. This causes additional latency/bandwidth overheads, with additional buffering, system calls or control information required.
- They do not provide tight coupling to system memory, reducing the ability of the system to take advantage of fine grained parallelism over distributed FPGAs or proper coordination between CPU and FPGA, inhibiting certain workloads [4].
- They do not guarantee reliable transfer, so are not amenable for production-level systems.
- Their hardware offloaded transport layers require per-connection state information/buffering to be stored within the hardware, limiting scalability or degrading performance by excessive connection setup/teardown.

V. Our Solution

Taking into consideration all of these limitations, we have developed a custom interconnect solution that supports reliable communication among independent and distributed FPGAs. The first step was to design a custom NI with hardware primitives to support a hybrid MPI+PGAS programming model. We provide support for MPI via an RDMA mechanism and PGAS-like communications via a separate interface to the NI to read/write into remote memory using transparent load/store operations from the CPU/FPGA. We encapsulate and extend the system-bus protocol to work over the network. In doing this there is no change from the user perspective in accessing remote memory from local memory (other than the NUMA effects on latency).

Another important aspect of our implementation is that we created a novel transport layer which lays within the FPGA fabric. This way, the CPU is not involved in network communication among FPGA resources. Our solution provides two entirely separate methods for reliable transfer: one for RDMA transfers and another one for shared-memory operations. This is justified by the drastic reduction of latency for small memory operations.

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 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.

Yet another benefit of our implementation is that our transport layer allows for both reliable *and* connectionless communications to guarantee message delivery. Typically reliable transport layers are connection-based, requiring large amounts of persistent state information when scaled. Our solution only maintains transient information regarding outstanding operations which have left the sender, making it far more amenable to a hardware implementation. We are able to do this because we separate the transport layers, having one for shared-memory operations in which retransmission information can safely be held within the NI. We then provide an entirely separate transport for reliable RDMA transfer. One which is more scalable as the data is held in its source memory location, rather than being copied into retransmission buffers within the NI.

Figure 2 shows the full network stack implemented within the FPGA, with segregated datapaths for shared-memory and RDMA operations. Our solution allows the accelerator logic to issue commands to the NI 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 a 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 NI. This means that both the accelerator and CPU can access the network completely independently from each other, and simple reading/writing to remote memory lends itself to established FPGA programming models. In the following section we show how the ability of the accelerator to write directly to the network reduces the control and datapath complexity of network transfers between distributed FPGA resources significantly.

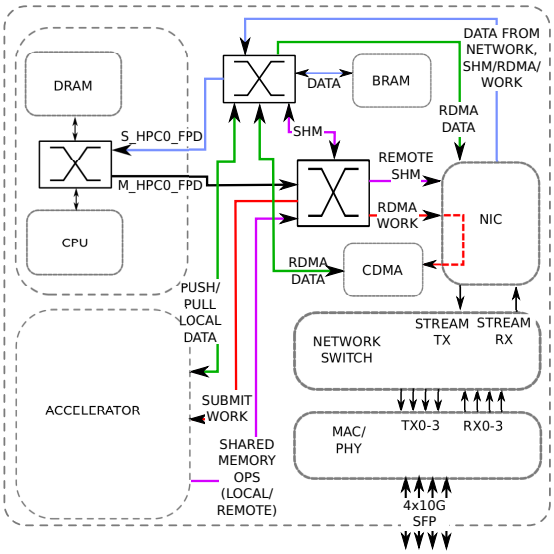
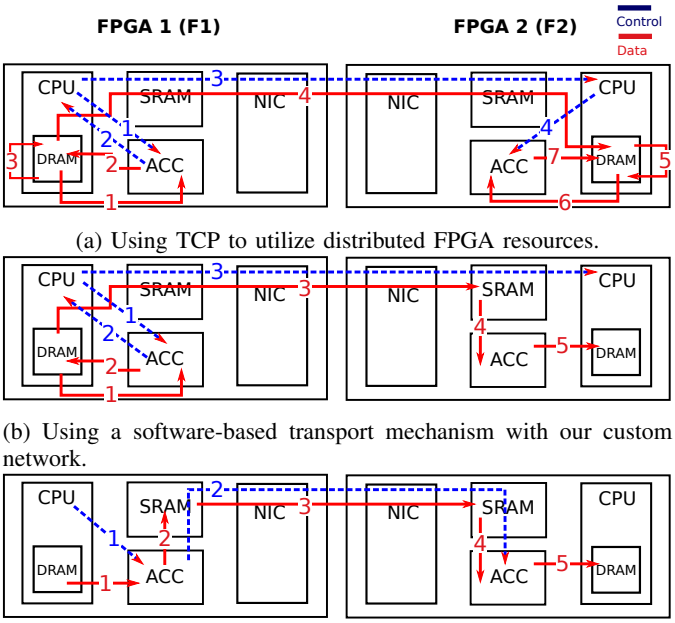


Fig. 2: Full IO and on-chip network stack within FPGA fabric.



(a) Using TCP to utilize distributed FPGA resources.

(b) Using a software-based transport mechanism with our custom network.

(c) Using our hardware offloaded transport layer.

Fig. 3: Flow of data and control when using distributed FPGA resources using different transport layers.

VI. Reduced Control and Datapath Complexity

In order to illustrate the benefits of our approach, Figure 3 shows the critical path for data and control communications for dataflow style processing initiated through various styles of transport layer by a CPU, for processing in a local (in F1) and then remote (in F2) FPGA. Figure 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.

Figure 3b shows how a software based implementation of our transport layer would work. In this instance 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 would still be required to coordinate between the two accelerators.

Finally, Figure 3c shows control and dataflow when using our fully implemented hardware-offloaded transport solution. In this instance, once the accelerator at F1 has completed its work it issues an RDMA operation directly to the NI, 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.

A. Experimental Set-Up

In order to demonstrate the benefits of our implementation we move now to recreate on the FPGA the control and dataflow examples described in Figure 3b and 3c, transmitting

different sizes of data through the data paths. We ignore the TCP setup as the performance is known to be degraded, and setup enabling distributed inter-FPGA communication would be difficult. The hardware transport layer is fully implemented within the FPGA fabric, but the software reliability is not implemented. In this instance we simply pass control information and data around the system and over the network as would be required with a complete implementation. The main aim here is to show the effects of the reducing control and datapath complexity, as opposed to the actual effects of the specific implementation.

Our experimental hardware platform consists of a ZCU102 development board where two full instances of the networking stack (two of everything shown in Figure 2 other than the CPU and MAC/PHY) are implemented within the board and connected with each other via a 10G SFP cable. All hardware within the FPGA fabric is clocked at 156.25MHz, and the ARM-based CPU is clocked at 1GHz. One instance of the entire networking stack requires 17.1% (46,915) of the avilable LUTs on the FPGA, giving reasonable remaining area to dedicate to accelerator processing elements.

The set-up using a single CPU solution as opposed to two separate boards is done in order to obtain more accurate timing measurements at the application level. A user-space application takes a system time-stamp before submitting blocks of data for processing in the accelerator block in the F1 stack (Figure 3), which performs some work and sends the result forward (through the network) to the accelerator in the F2 stack. Upon reception of the data F2 performs some work and then writes the result in local memory and notifies the CPU, which will take another system time-stamp to determine the overall run time at the application-level.

For simplicity, we keep the data block size constant through the whole data flow of the experiment and use dummy accelerator blocks which do not perform any genuine function, but merely add a *computation latency*. In this way the results can be generalised and applied to many acceleration functions. Blocks of data are transferred using DMA engines to the accelerator, as is the typical case in FPGA based implementations of HPC operations such as matrix-vector or matrix-matrix multiplication. We adjust the block size of the data, and adjust the *computation latency* of the accelerator block 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's *computation latency* will be the same time as the whole system spends communicating.

B. Results

Figure 4 shows the latency and throughput for the control and data flow through our fully implemented hardware based transport and the emulated software transport as shown in

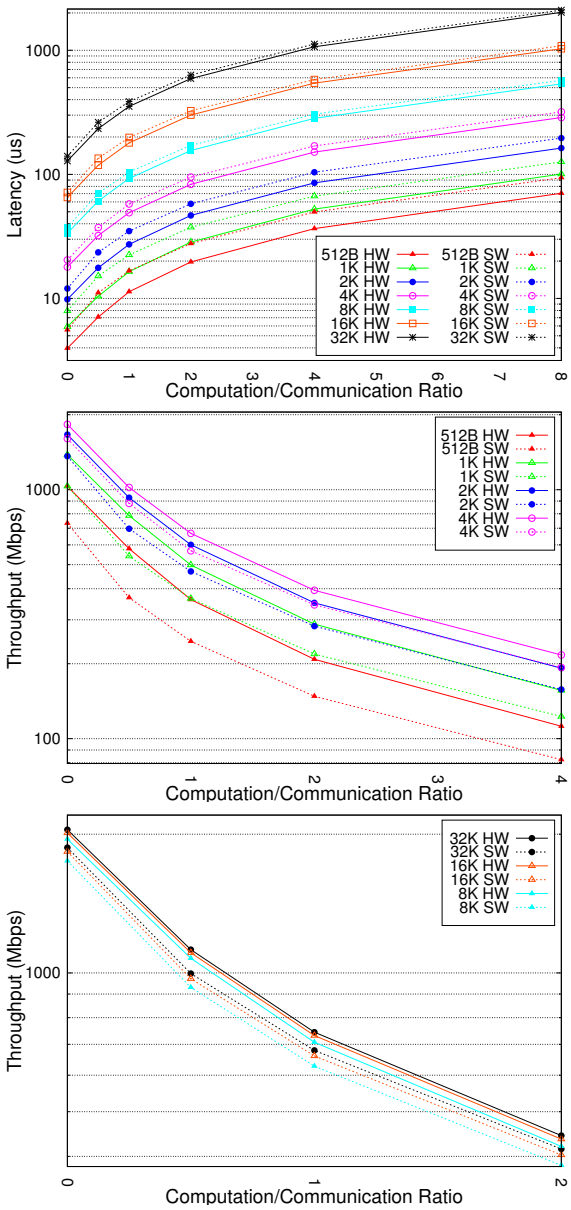


Fig. 4: Latency to perform a single operation (top). Achievable throughput for small block sizes (middle) and larger block sizes (bottom).

Figure 3b and 3c *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 [5].

The latency difference in the hardware and software based transport comes from the additional control path complexity seen between Figures 3b and 3c, with the former requiring

CPU orchestration for inter-FPGA data transfer. It is also worth noting that in a full implementation of the software transport additional overheads would be incurred, further increasing the latency we see here for the software solution.

If we focus on throughput, we see a 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 not translate into higher performance. Note however, that we support multiple processing elements within the same FPGA to exploit spatial parallelism.

Using the raw throughput results we can estimate the achievable memory-bound Flops of such a solution, using methods similar to [8], [9]. Lets assume a 1KB block size for transfer, feeding an accelerator block 128 double precision floats to perform an 88 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. ($1.1 \times 10^9/8,192$ 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.

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 [7] where a theoretical peak of 8.9 GFlops over 8 FPGAs (1.11 GFlops per FPGA) was reported for network bound processing, we can claim our communication solution is more effective, particularly since they are limited to a basic ring topology, severely limiting scalability.

VII. Concluding Remarks

There are many in the HPC community who doubt the viability of reconfigurable computing within this domain. It is certainly true that there are many impediments to the uptake of FPGA technology and many research questions still to be answered; such as standard HPC library support, portability, and reliability at scale. However, we have demonstrated that there is much scope for the use of FPGAs within HPC.

While we have only dealt with a small piece of the puzzle within our work—the requirements of interconnect technologies to enable the efficient exploitation of FPGAs—we believe an inflexion point is being reached. We see that many other important areas are reaching sufficient maturity to push the use of FPGAs into mainstream HPC. HLS tools have progressed enormously, as has the architecture of FPGAs themselves (enhanced floating-point capability, hardened tranceiver technology etc.). With tighter power constraints and burgeoning data-intensive workloads we see that demand is also growing. It seems that as the pressures on Moore's Law become greater, so will the pressure to seek alternative technologies.

ACKNOWLEDGEMENTS

This work was funded by the European Union's Horizon 2020 research and innovation programme under grant agreements No 671553 (ExaNeSt) and 754337 (EuroEXA).

References

[1] A. Tate *et al.*, "Programming abstractions for data locality." PADAL Workshop 2014, April 28–29, Swiss National Supercomputing Center , 2014.

[2] V. W. Lee *et al.*, "Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu," *ACM SIGARCH computer architecture news*, vol. 38, no. 3, pp. 451–460, 2010.

[3] K. Asanovic *et al.*, "The landscape of parallel computing research: A view from berkeley," Technical Report UCB/EECS-2006-183, Tech. Rep., 2006.

[4] 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, 2016.

[5] G. Weisz, J. Melber, Y. Wang, K. Fleming, E. Nurvitadhi, and J. C. Hoe, "A study of pointer-chasing performance on shared-memory processor-fpga systems," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2016, pp. 264–273.

[6] L. Oden, H. Fröning, and F.-J. Pfreundt, "Infiniband-verbs on gpu: A case study of controlling an infiniband network device from the gpu," in *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*. IEEE, 2014, pp. 976–983.

[7] R. S. Correa and J. P. David, "Ultra-low latency communication channels for fpga-based hpc cluster," *Integration*, vol. 63, pp. 41–55, 2018.

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

[9] J. Williams, C. Massie, A. D. George, J. Richardson, K. Gosrani, and H. Lam, "Characterization of fixed and reconfigurable multi-core devices for application acceleration," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 3, no. 4, p. 19, 2010.