Near-Data FPGA-Accelerated Processing of Collective and Inference Operations in Disaggregated Memory Systems

Carsten Heinz, Andreas Koch
Embedded Systems and Applications Group
TU Darmstadt
Darmstadt, Germany
{heinz, koch}@esa.tu-darmstadt.de

***Abstract*—With growing data set sizes, many scientific and data center HPC workloads observe an increasing scaling imbalance, e.g., between compute and memory capacities. As a solution, disaggregated system architectures employ spatial distribution of the different resources. They aim for independent scaling of the different resource kinds (e.g., compute, non-volatile storage, memory), and use fast communication fabrics for their interconnection.**

However, for some bulk operations, such as reductions and collections, it is still beneficial to perform them close to the memories, avoiding the need to move large volumes of data over the fabric.

This work realizes a disaggregated system capable of performing such near-data processing (NDP) operations by extending the distributed memory controllers with hardware-accelerated compute capabilities. The actual computations execute on FPGAs and can be abstractly described using C/C++ as compilable by high-level hardware synthesis (HLS) tools.

We have aimed for high usability of our technology also by HPC experts unfamiliar with hardware design. An automated toolflow encapsulates the creation and deployment of the actual accelerators in the disaggregated system. The NDP operations execute distributed across all memory nodes, and are easily accessed using a simple MPI-based programming interface that requires only minimal effort to use in existing applications.

Our solution is demonstrated using a prototype disaggregated system based on the low-latency EXTOLL fabric for communication. We evaluate both conventional reductions/collectives as well as complete machine-learning inference tasks.

I. Introduction

HPC performance is constantly increasing, with the first system reaching over one Exaflop on specialized benchmarks already being available [1]. Workloads harnessing that much compute power typically operate on massive amounts of data. However, a scaling imbalance can be observed between the available compute power and memory capacities. As a result, disaggregated system architectures have emerged, employing spatial distribution of compute and memory subsystems to address scalability issues.

Compared to local memory, accesses to a memory location on a distributed memory node have additional latency imposed by the interconnect network. This downside can be addressed by moving parts of the computation closer to the memory, following the Near-Data Processing (NDP) paradigm. However, as the memory nodes do not have general-purpose processors, and instead just contain SDRAM-level memory controllers and network interfaces, a different processing paradigm will be needed.

A suitable paradigm for such an architecture is reconfigurable computing based on Field Programmable Gate Arrays (FPGAs). Not only can FPGAs realize the low-level memory controllers and network interfaces very efficiently, they also provide the capability to add dedicated compute accelerators close to both of these resources. However, using such accelerators generally does require specialized knowledge, e.g., in digital circuit design and computer architecture, that will be unfamiliar to most HPC software developers. Thus, to make the technology widely usable, powerful tool support is required to bridge the gap.

The tool support must address two challenges: First, providing a familiar developer-level view of the capabilities of the NDP memory nodes. Second, a robust tool flow for describing, generating, and deploying the actual hardware accelerators in the disaggregated system.

This paper addresses both of these issues by making the following contributions. First, we provide a software library to utilize an NDP-capable disaggregated memory system with a familiar MPI-based programming interface. Secondly, we supply an automated toolflow for creating hardware designs for the NDP accelerators. Our current prototype uses NDP-capable memory nodes integrating an FPGA and HMC memory. In combination, these two contributions offer a simple way of utilizing a disaggregated system in combination with FPGA-based NDP processing. We demonstrate the capabilities and performance of our solution using two sample applications.

II. System Architecture

We begin by discussing the design considerations for our implementation. In Section II-A, we provide an introduction to MPI, followed by the required additions for memory nodes in Section II-B. Section II-C describes the utilized low-latency

This research was funded by the German Federal Ministry for Education and Research (BMBF) with the funding ID 01 IS 17091 B.

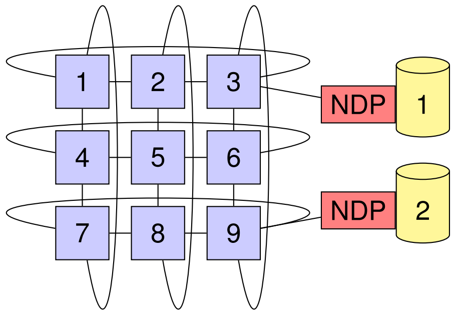

Fig. 1. MPI cluster with disaggregated compute (blue) memory nodes (yellow), extended with NDP accelerators (red).

EXTOLL interconnect fabric. Finally, the complete system topology for our prototype is described in Section II-D.

A. Message Passing Interface

A computing cluster typically consists of hundreds or thousands of compute nodes. In order to fully harness its compute power, applications need to compute on multiple nodes simultaneously. To distribute computations across nodes, the message passing interface (MPI) [2] is used for communication between the nodes. Communication can occur in two ways: *two-sided* communication involves sender and receiver, whereas with *one-sided* communication, where a node can access parts of another node's memory without interrupting that node's computations.

In addition to this node-to-node communication, MPI provides communication across multiple nodes, called *collectives*: Synchronization functionality is provided by MPI_Barrier. Data can be distributed to all nodes by MPI_Bcast or MPI_Scatter. Collecting data on the root node is realized by MPI_Gather. MPI_Reduce performs a reduction operation over data distributed over all nodes. The reduction function to execute is often selected from a catalog of predefined operations. However, in MPI it is also possible to define custom operations for the reduction.

For a finer-grained control over the nodes to be involved in a computing task, the concept of *Communicators* exists. A communicator can be MPI_COMM_WORLD, which represents all available nodes in the cluster, but it can also just refer to a subset of nodes.

Nowadays, nodes in high-performance computing clusters are often extended by accelerators (e.g., GPUs). This increases the processing power, but as they are usually connected to and controlled by general purpose CPUs, their presence does not change the fundamental programming model of MPI.

B. Heterogeneous Setup

Figure 1 shows a simplified sketch of a disaggregated cluster. On the left are the general-purpose nodes, which are able to execute MPI applications. Memory nodes capable of accelerated near-data processing are attached to the cluster-wide interconnect. The use of heterogeneous processing elements (general-purpose processors, NDP accelerators) in a single MPI network could be enabled in multiple ways:

1) Re-create the *full* MPI application on the NDP accelerator. On an FPGA, this could be achieved by using High-Level Synthesis (HLS) to translate a C/C++ application to a hardware description language (HDL), or by directly writing custom HDL. The available FPGA size may pose a constraint for larger codes. In a heterogeneous system, the maximum MPI performance is limited by the performance of the slowest node, as faster nodes have to wait at the next synchronization point, or for the next data transfer. In such a heterogeneous system, the performance advantages of the FPGA acceleration cannot be fully realized.
2) Implement just *part* of the computing task on the NDP nodes, and include the NDP accelerators in an additional communicator disjunct from MPI_COMM_WORLD.
3) Use one-sided communication to access the memory of the NDP node. The entire workload runs on the general-purpose node, the memory node is used as pure data storage. No additional computations are performed at the remote memory node itself.

Our approach is a hybrid of option 2) and 3). One-sided communication is used for data transfers between the nodes, for example an MPI_Bcast uses an Remote Memory Access (RMA) *PUT* operation to distribute the data from the root node to all other nodes. In addition, we move suitable computing tasks from general purpose compute nodes to the NDP memory nodes, and interact with them from the general-purpose compute nodes using an MPI-based API. With this approach, we can provide the desired NDP capabilities. At the same time, the approach does not require the implementation of the full MPI application on the accelerator. This allows us to focus on the parts of an application which actually benefit from FPGA-based NDP.

C. EXTOLL Fabric

Our work is based on the EXTOLL interconnection fabric [3]. In contrast to Infiniband, the EXTOLL design integrates the switching functionality within the network interface controller (NIC). This allows the creation of switchless network topologies without scaling issues. At the same time, the NIC has a competitive latency between just 600 and 800 ns [4].

The network interface is realized by the Tourmalet ASIC. This chip provides six network links with 12 lanes each, and thus provides a maximum raw bandwidth per link of 100.8 Gbps. To the host, the NIC provides a PCIe 3.0 connection with 16 lanes.

In addition to the ASIC implementation, an FPGA implementation for an EXTOLL interface exists. It was initially used for prototyping the ASIC, but now forms a suitable base for adding the NDP functionality. Due to the logic capacity constraints on the FPGA, it does not offer the full functionality of the ASIC (e.g., inter-link switching is not supported). But these limitations do not affect our NDP prototype.

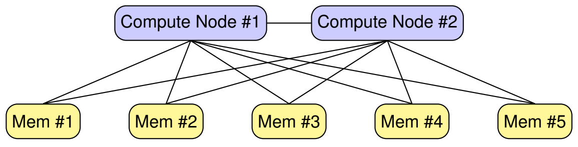

Fig. 2. System setup consisting of two general purpose compute nodes (blue) directly connected to five NDP memory nodes (yellow)

D. Topology of the System Prototype

Each of the Aspin v2 boards [5] at the heart of our prototype implements a network-attached memory for the EXTOLL fabric and provides two EXTOLL links. It utilizes a $2\,\mathrm{GB}$ Hybrid-Memory-Cube (HMC) as memory, and reaches over $8\,\mathrm{GBps}$ peak write and read bandwidth for a single network link. We use this board to realize the NDP memory nodes of our disaggregated system. In a real system, the memory nodes would of course provide larger memory capacities (at least $1\,\mathrm{TB}$ each). For our proof-of-concept implementation, though, the available memory size on the prototype board does not influence the performance.

With the switchless design of the EXTOLL fabric, it is possible to create a variety of network topologies. A common deployment is a 3D torus. This is achievable using the six available network links on every NIC, combined with the ASIC-integrated switching fabric.

The Aspin v2 board we use does not have this routing fabric, and can thus only be connected directly to up to two other EXTOLL nodes. As the focus of this work is on large memory sizes, our proposed topology maximizes the number of directly attached memory nodes. Such a system, with two general purpose nodes having full-scale ASIC-based EXTOLL NICs, and five memory nodes supporting FPGA-based NDP processing, is shown in Figure 2. One link connects the two general purpose compute nodes, leaving the remaining five network ports available for connections to the NDP memory nodes.

If a cluster consists of more than one compute node, the total number of available network ports increases. Depending on the exact network topology, NDP memory nodes can be attached anywhere in the network. The switchless EXTOLL interconnection provides routing within the ASIC-based NICs, so that communication is possible between all nodes. Of course, when a node is accessed via another node, the multi-hop communication has a higher latency than the direct communication.

In our actual prototype setup, we connect two Aspin v2 cards acting as NDP memory nodes to a single ASIC-based NIC in a general-purpose compute node. As the root node has six available network ports, it would be possible to connect up to six memory nodes to this root node, but we do not have that many Aspin v2 cards available in our lab.

III. Implementation

Our system has two main components: the software on the general purpose compute node (Section III-A), and the FPGA toolflow for the NDP memory node (Section III-B).

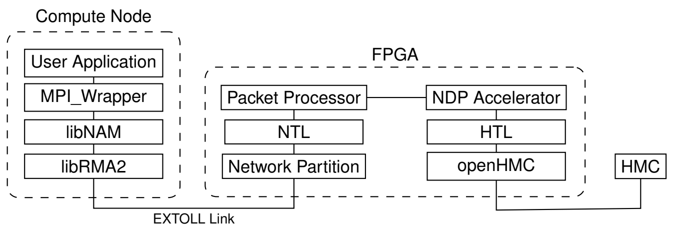

Fig. 3. Software (left) and hardware stack (right) on a system with one compute node and one NDP memory node.

A. Host-Side Software

For interacting with the NDP memory nodes, we use a separate communicator and exclude the NDP nodes from MPI_COMM_WORLD.

If a call to an MPI collective operation is issued on the NDP communicator, we use wrapper code inside our MPI library to call a specialized implementation for accessing the NDP memory node communicator. Memory management is handled in software, and the actual network transfers use RMA primitives provided by the EXTOLL NIC. All existing MPI collective operations are supported by our implementation.

For the added functionality of *custom* NDP operations, we introduce new functions to the MPI library, which are required for the integration of the accelerators into the application. We try to stay close to the original MPI terminology, the key semantical difference being that these calls now specify operations to occur *locally* on the memory nodes in an NDP fashion. Each of these calls will execute their operations on *all* NDP memory nodes in the NDP communicator in parallel.

- `MPI_Local_Alloc(bytes)` is used for allocation of local memory on the NDP memory node.
- `MPI_Local_Dealloc(void*)` is used for deallocation of node-local memory
- `MPI_Local_Map(function, input, output, length)` executes a custom operation on the NDP memory node, which *maps* the elements in an array of input data to output data of the same length (n:n).
- `MPI_Local_Reduce(function, input, length)` locally executes a custom operation on the memory nodes, which *reduces* an array of input data to a single result value (n:1).

The software stack is shown on the left side in Figure 3. The user application interacts with our wrapped MPI library. Internally, the wrapper utilizes the EXTOLL-provided libNAM for managing the Aspin-based *Network-attached Memory* (NAM), which in turn calls the libRMA2 to use low-level EXTOLL-link specific communication primitives.

Figure 4 considers three different cases: It shows the operation of a non-disaggregated MPI cluster in (a), a disaggregated, but non-NDP-capable cluster using RMA mechanisms in (b), and finally a disaggregated NDP-capable cluster in (c). Each of these setups executes the same workload, in this case the

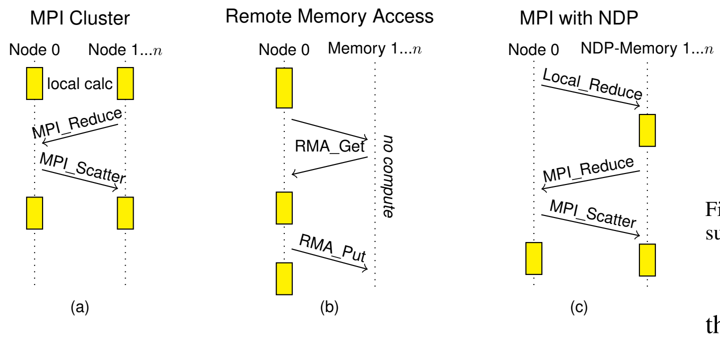

Fig. 4. Network sequence diagrams for different approaches

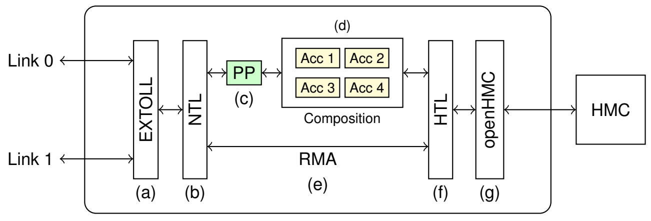

Fig. 5. Hardware Architecture for NDP accelerators on FPGA

accumulation of larger arrays of data distributed across all nodes, with the result to be stored on all nodes (compute or memory).

In a conventional MPI cluster (4.a), this can be realized by first computing a local sum on all nodes, then performing a MPI_Reduce followed by a MPI_Scatter. On a memory node without NDP capabilities (4.b), all data has to be transferred to the root node and accumulated there, with the result then being distributed back to the memory nodes. The third diagram (4.c) represents the approach chosen in this work on a system with NDP-capable memory nodes. Parallel local accumulation operations on all NDP memory nodes are invoked by MPI_Local_Reduce, and the local partial sums can then be collected using a MPI_Reduce from the memory nodes back to the root node, accumulated there, with the final sum then distributed again using a MPI_Scatter. In this manner, it is possible to distribute workloads across NDP memory nodes without having to implement the complete MPI application on the NDP memory nodes.

B. Hardware on Memory Node

In our proof-of-concept design, the disaggregated NDP-capable memory is realized on Aspin v2 boards. Each board contains a Xilinx Virtex-7 FPGA, which has connectors for two EXTOLL links and utilizes a HMC memory with a capacity of 2 GB for the actual storage. The FPGA system-on-chip (SoC) design (shown in Figure 5) is based on the Network-Attached-Memory architecture [5], which provides the EXTOLL network stack (a) and an openHMC memory controller (g) to interface with the discrete HMC memory chip. A common streaming-based protocol is used inside of the SoC, with translation blocks *Network Transaction Layer* (NTL) (b) and *HMC Transaction Layer* (HTL) (f) performing

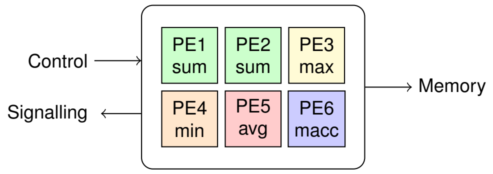

Fig. 6. Sample TaPaSCo composition with FPGA PEs for the NDP operations sum, maximum, minimum, average and multiply-accumulate.

the network-side and HMC-side conversions. This base design already provides the RMA operations required in disaggregated systems.

The new NDP functionality that is the focus of this work is placed on a processing path parallel to that of the original RMA operations. NDP requires two additions: A *packet parser* (PP), shown as (c), which interprets incoming NDP requests, and creates reply packets from the locally computed results, as well as the actual NDP accelerators, which can perform their own local memory accesses autonomously fom host-induced RMA operations.

Actually creating the accelerators for the custom map and reduce operations, though, does require expertise in hardware design for FPGAs. To lower this key barrier to productive use of the system by non-experts, we apply our experience in automated toolflows for reconfigurable computing.

Thus, we integrated the Aspin v2 FPGA base-design as a board target into the open-source framework TaPaSCo [6]. The TaPaSCo framework provides a software runtime, an SoC base architecture, and a toolflow for the automated generation of task-based FPGA-based computing systems. In the TaPaSCo framework, workloads are partitioned into tasks, that are then launched on hardware *processing elements* (PEs). A PE can be provided as an IP core (existing hardware block), or as C/C++ code. In the latter case, TaPaSCo automatically uses High-Level-Synthesis (HLS) to generate IP cores for the codes. PEs for the same task are grouped into a *hardware thread pool*, multiple thread pools are grouped in a *composition*. Note that the actual mix of PEs can be matched to the needs of the workload (e.g., more GEMM units than FFT units), allowing more parallel task launches on the larger thread pools. For more complex scenarios involving a larger number of task types, TaPaSCo does provide tool support for automated Design Space Exploration to discover an optimal mix. Figure 6 shows an example composition with accelerators for five different kinds of tasks, where two sum tasks can be computed in parallel.

Prior TaPaSCo versions assume that there is a host which controls the PEs on the FPGA, either in the form of a PCIe expansion board, or with host CPU and FPGA integrated on the same chip. For the scenario described here, this is no longer the case, and TaPaSCo has to be modified accordingly. Here, EXTOLL network packets are used to send control commands to the PEs and receive-back results. As the base for this modification, we use a recently introduced optional TaPaSCo feature called Cascabel [7]. Cascabel moves part of the TaPaSCo task

```c
int mpi_sum(int arr[MAX_SIZE], int length) {
        #pragma HLS ARRAY_RESHAPE variable=arr \
            factor=16 dim=1
        int i = 0, res = 0;
        for (; i < length; ++i)
                res += arr[i];
        return res;
}
```

Listing 1: Sample HLS kernel implementing the MPI_SUM operation for the integer data type.

scheduler/dispatcher into dedicated hardware. In the original TaPaSCo use-case, this allowed greatly increased task launch rates of up to 6 million tasks per second in [7].

For the network-centric scenario targeted here, we replace the front-end of the Cascabel dispatcher, which was initially realized using a memory-mapped I/O (MMIO) region, with a custom EXTOLL network packet parser (Figure 5.c), which moves the MMIO functionality across the network. Thus, EXTOLL RMA PUT messages can also launch tasks on the accelerators, while RMA GET can also retrieve the status and results of a task. Note that on the software side, this mechanism is encapsulated in the MPI wrappers discussed earlier, and the GET/PUT messages are also used to perform their expected RMA operations.

As described above, the PEs implementing the actual NDP operations can be described in a number of ways. For most traditional HPC developers, the most accessible one will be the use of C/C++ and HLS tools. Listing 1 shows an example C HLS kernel for the MPI_SUM operation, demonstrating how a standard C/C++ function could be extracted from an existing MPI application, and inserted into our toolflow targeting the NDP memory nodes. The data type (integer in the example) can easily be changed to more complex scalar or record types. Since MPI operations can have variable length, the example accepts the length as a parameter. When using a constant length, or specifying an upper bound at compile time, HLS tools are able to perform loop-unrolling and pipelining for increased performance. The example in Listing 1 has an additional HLS pragma for increasing the memory data width. Instead of fetching a single 32 bit integer for every loop iteration, the generated kernel can request *sixteen* integers (64 bytes) from the array arr in parallel. In this way, a partial loop-unrolling is possible, and at the same time, the kernel makes better use of the available memory bandwidth by employing a 512 bit memory interface.

Despite the advances in HLS tools, HLS does not always exploit the hardware to its fullest potential. In these cases, our system can use the TaPaSCo capability of also including manually-designed hardware blocks (similar to assembly-level programming, e.g., in HPC maths libraries). Thus, very highly optimized blocks using micro-architectural features such as custom pipelines, special numeric formats and their associated arithmetic functions, as well as custom caches/prefetchers can be employed. This feature will be used in Section IV-C.

Figure 3 shows the entire interaction between the software stack on the compute node, and hardware units on the NDP memory node. A user application employing the proposed software stack communicates over the EXTOLL link with the FPGA. After passing through the network stack, the packet processor dispatches the NDP operation to the appropriate accelerator. During execution, the accelerator can access the HMC memory through the memory stack. Results or the completion of operations are obtained through polling.

Memory management is performed by libNAM and stored inside the communicator structure. Memory handles allocated by the provided function (MPI_Local_Alloc) are automatically utilized in MPI calls of our library.

IV. Experimental Evaluation

For our evaluation, we employ the setup shown in Figure 2. We use a single host server, which is equipped with an 3.7 GHz Intel i7-4820K. The EXTOLL NIC resides in a PCIe 3.0 slot with 16 lanes. Two Aspin v2 boards are attached to the host server. Both are connected by 12 lanes, resulting in a bandwidth of 100.8 Gbps for each board. For synthesizing the bitstreams, the Vivado toolsuite (version 2020.1) by the FPGA vendor Xilinx was used. The accelerator runs at a frequency of 180 MHz, the network and memory I/O subsystems at 160 MHz.

A. Latency Overhead

The first performance metric we evaluate is NDP latency. We measure the time of dispatching an NDP operation by running a NOP operation, which immediately completes. The average after 1,000,000 runs is 3.7 µs of launch latency, compared to the base latency of 2.5 µs to just execute a single EXTOLL *GET* RMA primitive for fetching data from the NAM to the host.

The difference in latency is mainly due to the communication mechanism used to launch the NDP operation. The launch itself is realized by an EXTOLL *PUT*, while the result is fetched by polling using EXTOLL *GET*. That communication overhead will not change, of course, when processing larger data sets using NDP. Without NDP, the communication time will scale linearly with the growth of the data set size, as more EXTOLL RMA operations have to performed to transfer the data to the host handling the actual processing.

B. Standard MPI Operations

The MPI specification lists a catalog of predefined functions for reduce operations. These functions can easily be formulated in the HLS toolflow, as shown in Listing 1. For evaluating our NDP proof-of-concept system, we have implemented and measured the execution time for the MPI_SUM operation. We benchmark over array sizes from 8 up to 2^{21} integers.

As a baseline, a non-NDP memory node is connected to a general purpose compute node, resembling the system as sketched in Figure 4.b. The application first fetches the data from the memory node over the EXTOLL network, and then calculates the sum in local memory. The data fetches are realized by *EXTOLL GET* RMA primitives. As expected, the execution time increases with growing array sizes.

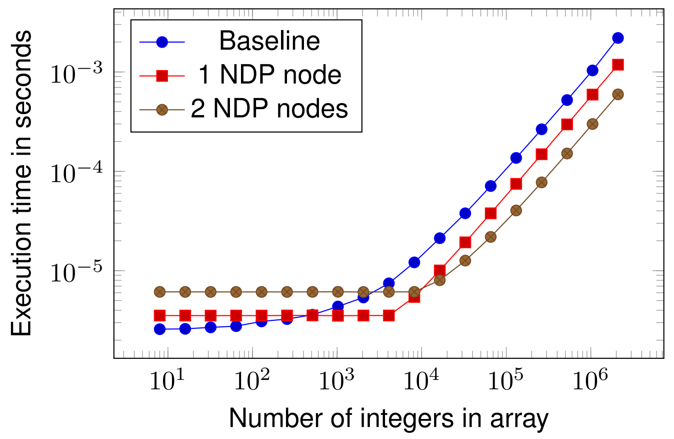

Fig. 7. Execution time over increasing array size for non-NDP baseline vs. one or two NDP memory nodes

The NDP implementation is measured for a system having (a) one, or (b) two NDP-capable memory nodes. The NDP accelerator is generated from the HLS code shown in Listing 1, which utilizes a 512 bit wide memory interface to the HMC memory controller. In the setup with two nodes, it is assumed that the data is distributed (partitioned) across both nodes. Thus, a single accelerator only has to process half of the total array size, and both accelerators will run in parallel. The measured execution time covers both the dispatching of the accelerator, NDP reduce operation itself, and the return of the partial sums to the host for the final accumulation.

The graph in Figure 7 shows the measured execution times for different array sizes. It can be seen that for small arrays up to 256 integers, the baseline is the fastest implementation. In these cases, the dispatch overhead is not amortized over the very short actual NDP processing time.

For larger data sizes, the single NDP accelerator in (a) benefits from the available local memory bandwidth, and speedups of over 2x are observed. The actual performance of the accelerator is limited by the memory controller, specifically by the HTL logic and protocol conversions between the AXI4 interface used on the chip and the HMC packets required for communicating with the discrete memory chip. According to on-chip measurements, the accelerators can exploit between 80 to 90% of the theoretical HMC memory bandwidth in this manner.

In the system with two NDP memory nodes (b), the execution time for smaller arrays is even longer than with one node. The reason for this is, that in the prototype, the two partial results are currently fetched *sequentially*. When scaling to a system with more nodes, these fetches should be performed in parallel as well. The hardware is capable of this, but the EXTOLL-provided libNAM would need to be extended for such interrupt-driven non-blocking *GET* operations. On the FPGA side, TaPaSCo already includes the necessary interrupts. For larger datasets, the system with two NDP memory nodes benefits from two factors: (1) the computation load is now distributed over two nodes, only half of the data has to be processed per node, (2) the total available memory bandwidth is doubled for this setup. This results in a speedup close to 2x compared to the single node system and over 3.5x compared to the baseline.

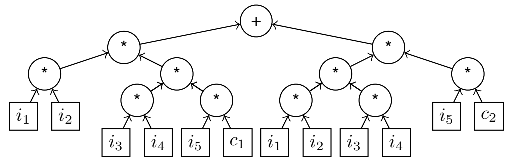

Fig. 8. Inference using the NIPS5 SPN.

C. Case Study: Inference in Sum-Product-Networks

The ability of our approach to NDP-accelerate functions much more complex than those in the MPI standard opens up new implementation options. In this section, we employ the technology to move an entire machine-learning inference computation to the NDP memory nodes.

To this end, we implement the inference in a Sum-Product Network (SPN) as an NDP operation. In contrast to traditional neural networks, SPNs are *Probabilistic Graphical Models* (PGM), which handle probabilistic queries. The calculation in an SPN is conducted by evaluating a directed, acyclic graph consisting of products, sums, and leaf nodes (often histograms of probability distributions) [8].

Inference in SPNs has been successfully accelerated using FPGAs in prior work [9]. In contrast to that work, our NDP approach does not incur the data transfer times of using a PCIe-based FPGA board in a host system, as the data is already resident in each of the NDP-capable memory nodes.

For this evaluation, we realize an accelerator for a probabilistic query on the NIPS corpus [10], which in our case is one of the smaller examples with just five input variables. Figure 8 shows the corresponding acyclic graph, which has five inputs i_n, two normalization factors c_n, and computes a probability as the result. In total, the calculation requires 10 floating-point multiplications and one addition. Note that this kind of converging computation does not map well to GPUs [9]. For the measurement of the execution time, we run 10,000 iterations of the calculation.

When implemented in C using double precision numbers and submitted to Xilinx Vivado HLS for compilation, the generated FPGA implementation requires 107 clock cycles for a single loop iteration (=initiation interval, II) without pipelining. Even when disregarding any memory latency, this results in over one million clock cycles for computing all iterations. However, this is actually a *slow-down* of roughly *50x* over the non-NDP baseline, which uses EXTOLL RMA primitives to move the data to the general-purpose host for the computation, and then RMAs the results back. The reason for that slow-down is the highly inefficient microarchitecture created by Vivado HLS, which was not able to generate a pipelined accelerator for this code. However, this deficiency can be worked around by employing a different way to more efficiently map the SPN to hardware.

TABLE I
PERFORMANCE EVALUATION OF NIPS5 INFERENCE

	Pipeline depth	t_{exec} (μs)	SPN rows/μs	Speedup
Baseline	-	127.94	78.16	1.00
Generic HLS	107	5942.00	1.68	0.02
Custom HLS	24	57.49	173.94	2.23

As an alternative to the generic C/C++-based HLS of Vivado, HLS from *domain-specific* languages and descriptions has the potential to achieve much higher performance. E.g., when compiling image processing kernels from the Halide DSL using a custom HLS system, a performance 2…4x better than using generic HLS has been reported [11].

We can follow a similar approach here, and use a custom HLS compiler for generating high-performance SPN inference accelerators [12]. When applied to the NIPS5 benchmark, the custom SPN compiler generated a highly optimized hardware block, having a 24-stage hardware processing pipeline capable of accepting a new data item every cycle (II=1).

The results in Table I show that the generic Vivado HLS-created kernel is the slowest implementation. With the fast EXTOLL network, even being close to the memory cannot compensate for this inefficient "accelerator" implementation. On the other hand, the SPN unit created using the custom SPN-specific HLS tool, performs as expected and achieves a speed-up of roughly 2x over the baseline when operating in NDP fashion.

V. RELATED WORK

Some aspects of this work have been addressed in related work. These publications can generally be classified into three groups: optimizations on the network, node acceleration, and network-attached memories.

A. Optimizations on the Network

Improving the communication between nodes in a cluster can greatly benefit MPI applications. Collective operations are particularly communication-heavy, and have been the focus of much research. Mellanox SHArP [13] presents an ASIC implementation, where aggregation trees are handled in hardware, specifically in the switches. However, the switch ASICs only provide the standard MPI collectives/reductions, and cannot deal with the custom operations, such as the SPN inference, that are the focus of our work.

Another approach to reduce the communication overhead is the deployment of Smart NICs. The management queues in Mellanox ConnectX-2 NICs [14] enable hardware offloading of collective operations. Another Smart NIC implementation uses the NetFPGA plattform for offloading MPI collectives [15], resulting in lower latencies for the collective operations. Our work takes a different approach, as we offload computation to NDP accelerators instead of offloading communication to the NIC. Also, we can support much higher network speeds of 2x 100G EXTOLL instead of the 4x 10G Ethernet ports on the NetFPGA board.

Matteis et al. [16] propose a Streaming Message Interface, a communication architecture with streams instead of MPI messages. A streaming-based model is very suitable for pipelined and dataflow oriented execution, as it avoids large buffers. However, that approach gives up the compatibility with the MPI API that we have aimed for.

B. Accelerating Compute Nodes

Instead of offloading the network communication onto an FPGA, it is also possible to use the FPGA as a standalone compute node in an MPI cluster. A sample FPGA-based system implementing TMD-MPI [17] utilizes soft-cores as generic processing elements and only employs specialized hardware accelerators for the Jacobi algorithm. The more general FPGA implementation of Ringlein et al. [18] uses HLS to generate hardware for a complete MPI node. The big advantage of the FPGA-based nodes is the low network latency. The presented proof-of-concept shows only a single kernel, which might limit the usability for complex applications. Our system is capable of handling multiple accelerators, and allows to expends FPGA hardware area only for those kernels that actually yield the highest performance gains, instead of spending area on providing full general MPI functionality.

Near-data processing is a well-known option for improving full-system performance, even outside of MPI clusters. JAFAR [19] is an NDP accelerator for column-stores in database systems. While the original idea targeted DRAM, an actual hardware implementation was realized for processing close to flash storage. It demonstrates a speedup of up to a 2.7x for these NDP operations [20].

C. Disaggregated or Network-Attached Memories

Performance benefits of memory disaggregation have been demonstrated for various applications. A system with dedicated memory blade-servers [21] has shown performance improvements and cost-savings when employing memory disaggregation. The work by Kwon et al. [22] presents advantages of high-bandwidth disaggregated memory for deep learning. The outcome is similar to our implementation. However, instead of increasing the memory bandwidth at the compute node, we move the workload close to the memory.

For accessing block devices over a network, protocols such as NVMe-over-fabrics [23] are available. The focus here is on high-bandwidth, low-latency access, NDP computations are not yet covered. Initial discussions on their inclusion in the underlying NVMe standard have already started, though.

Major computer vendors have founded the Gen-Z Consortium [24] to develop a "memory-semantic" protocol for the interconnection of processors, memories and accelerators. The consortium proposes a common fabric to connect various devices such as memory, GPUs, and accelerators to hosts. However, this new standard is not widely adopted yet and, in contrast to our approach of requiring only minor modifications to existing codes, is likely to require more significant changes in existing applications. To our knowledge, no datacenter-scale deployments of Gen-Z exist as of now.

VI. Conclusion and Future Work

This work presented a new proof-of-concept system to utilize near-data processing in a disaggregated memory system. By extending the system-on-chip providing the EXTOLL 100G networking and memory controller functions with compute capabilities, and wrapping these in a simple MPI-based programming interface, the system is easily usable from existing applications.

On the hardware side, we presented an automated toolflow that considerably simplifies creating the NDP processing accelerators. When fed with codes amenable for HLS, such as the standard MPI collective/reduction operators, our flow can perform all of the complex and often error-prone steps of creating FPGA accelerators automatically. It can also easily encapsulate the even higher-performance accelerators generated by custom domain-specific HLS systems, as we demonstrated for the ML SPN inference use-case.

When using a single of our NDP-capable memory nodes, we achieve speed-ups of around 2x over the disaggregated, but non-NDP baseline. The performance of our approach scales up with more NDP memory nodes, as more NDP units will then execute in parallel and more local memory bandwith is available.

The key barriers to even higher performance gains here were the practical limitations of the base hardware of our current Aspin v2 NDP platform. A successor board having newer, larger and faster FPGAs, as well as higher memory capacity and bandwidth, is already in fabrication.

Two other aspects of our approach could benefit from further investigation as well: First, so far, we have focused only on the computation parts of the problem. With the coming multi-link hardware, we can also use better inter-board communication strategies (e.g., tree-structured reductions). Second, our current work did not address the efficient deployment of the FPGA bitstreams for the NDP accelerators over the network. We intend to use Dynamic Partial Reconfiguration for this purpose in the new system.

References

[1] RIKEN, "Japan's fugaku gains title as world's fastest supercomputer." [Online]. Available: https://www.riken.jp/en/news_pubs/news/2020/20200623_1/

[2] W. Gropp, R. Thakur, and E. Lusk, *Using MPI-2: advanced features of the message passing interface*. MIT press, 1999.

[3] H. Fröning, M. Nüssle, H. Litz, C. Leber, and U. Brüning, "On achieving high message rates," in *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*. IEEE, 2013, pp. 498–505.

[4] J. Schmidt, "Accelerating checkpoint/restart application performance in large-scale systems with network attached memory," Ph.D. dissertation, 2017.

[5] ——, "NAM-Network Attached Memory," in *Doctoral Showcase poster at International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, 2016.

[6] J. Korinth, J. Hofmann, C. Heinz, and A. Koch, "The TaPaSCo open-source toolflow for the automated composition of task-based parallel reconfigurable computing systems," in *International Symposium on Applied Reconfigurable Computing (ARC)*, 2019.

[7] C. Heinz, J. Hofmann, L. Sommer, and A. Koch, "Improving job launch rates in the TaPaSCo FPGA middleware by hardware/software-co-design," in *Proceedings of the 10th International Workshop on Runtime and Operating Systems for Supercomputers (ROSS)*, 2020.

[8] H. Poon and P. Domingos, "Sum-product networks: A new deep architecture," in *2011 IEEE International Conference on Computer Vision Workshops (ICCV Workshops)*, 2011, pp. 689–690.

[9] L. Sommer, J. Oppermann, A. Molina, C. Binnig, K. Kersting, and A. Koch, "Automatic Mapping of the Sum-Product Network Inference Problem to FPGA-based Accelerators," in *IEEE International Conference on Computer Design (ICCD)*. IEEE, 2018.

[10] D. Dua and C. Graff, "UCI machine learning repository," 2017. [Online]. Available: http://archive.ics.uci.edu/ml

[11] J. Li, Y. Chi, and J. Cong, "HeteroHalide: From Image Processing DSL to Efficient FPGA Acceleration," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 51–57.

[12] L. Sommer, L. Weber, M. Kumm, and A. Koch, "Comparison of arithmetic number formats for inference in sum-product networks on fpgas," in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2020, pp. 1–10.

[13] R. L. Graham, D. Bureddy, P. Lui, H. Rosenstock, G. Shainer, G. Bloch, D. Goldenerg, M. Dubman, S. Kotchubievsky, V. Koushnir, L. Levi, A. Margolin, T. Ronen, A. Shpiner, O. Wertheim, and E. Zahavi, "Scalable Hierarchical Aggregation Protocol (SHArP): A Hardware Architecture for Efficient Data Reduction," in *2016 First International Workshop on Communication Optimizations in HPC (COMHPC)*, 2016, pp. 1–10.

[14] R. L. Graham, S. Poole, P. Shamis, G. Bloch, N. Bloch, H. Chapman, M. Kagan, A. Shahar, I. Rabinovitz, and G. Shainer, "ConnectX-2 InfiniBand Management Queues: First investigation of the new support for network offloaded collective operations," in *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, 2010, pp. 53–62.

[15] O. Arap, G. Brown, B. Himebaugh, and M. Swany, "Software Defined Multicasting for MPI Collective Operation Offloading with the NetFPGA," in *Euro-Par 2014 Parallel Processing*, F. Silva, I. Dutra, and V. Santos Costa, Eds. Cham: Springer International Publishing, 2014, pp. 632–643.

[16] T. De Matteis, J. de Fine Licht, J. Beránek, and T. Hoefler, "Streaming Message Interface: High-Performance Distributed Memory Programming on Reconfigurable Hardware," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019.

[17] M. Saldana, D. Nunes, E. Ramalho, and P. Chow, "Configuration and Programming of Heterogeneous Multiprocessors on a Multi-FPGA System Using TMD-MPI," in *2006 IEEE International Conference on Reconfigurable Computing and FPGA's (ReConFig 2006)*, 2006, pp. 1–10.

[18] B. Ringlein, F. Abel, A. Ditter, B. Weiss, C. Hagleitner, and D. Fey, "Programming Reconfigurable Heterogeneous Computing Clusters Using MPI With Transpilation," in *Sixth International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*, 2020.

[19] S. L. Xi, O. Babarinsa, M. Athanassoulis, and S. Idreos, "Beyond the wall: Near-data processing for databases," in *Proceedings of the 11th International Workshop on Data Management on New Hardware*, ser. DaMoN'15. New York, NY, USA: Association for Computing Machinery, 2015.

[20] T. Vinçon, L. Weber, A. Bernhardt, C. Riegger, S. Hardock, C. Knoedler, F. Stock, L. Solis-Vasquez, S. Tamimi, and A. Koch, "nKV in Action: Accelerating KV-Stores on Native Computation Storage with Near-Data Processing," in *Proceedings of the VLDB Endowment, Volume 13*, 2020.

[21] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch, "Disaggregated memory for expansion and sharing in blade servers," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 267–278.

[22] Y. Kwon and M. Rhu, "A disaggregated memory system for deep learning," *IEEE Micro*, vol. 39, no. 5, pp. 82–90, 2019.

[23] Z. Guz, H. H. Li, A. Shayesteh, and V. Balakrishnan, "NVMe-over-Fabrics performance characterization and the path to low-overhead flash disaggregation," in *Proceedings of the 10th ACM International Systems and Storage Conference*, ser. SYSTOR '17. New York, NY, USA: Association for Computing Machinery, 2017.

[24] Gen-Z Consortium, "Computer industry alliance revolutionizing data access." [Online]. Available: https://genzconsortium.org/