Zedwulf: Power-Performance Tradeoffs of a 32-node Zynq SoC cluster

Pradeep Moorthy
School of Computer Engineering
Nanyang Technological University
Singapore, 639798
pradeep003@e.ntu.edu.sg

Nachiket Kapre
School of Computer Engineering
Nanyang Technological University
Singapore, 639798
nachiket@ieee.org

***Abstract*—**
Commodity SoCs with hybrid architectures that combine CPUs with programmable FPGA fabric such as the Xilinx Zynq SoC have become a competitive energy-efficient platform for addressing irregular parallelism in graph problems. In this paper, we prototype a 32-node cluster composed from these Zynq SoC chips to accelerate communication-bound sparse graph-oriented applications such as neural network simulations. We develop specialized MPI routines specifically developed for irregular accelerator-to-accelerator communication of small message traffic. We use the ARM processor for handling the MPI stack while offloading compute-intensive calculations to the FPGA. For graphs with 32M nodes and 32M edges, Zedwulf delivers the highest 94 MTEPS (Million Traversed Edges Per Second) throughput over other x86 multi-threaded platforms in our study by 1.2–1.4×. For this experiment, Zedwulf operates at an efficiency of 0.49 MTEPS/W when using ARM+FPGA which is 1.2× better than using ARMv7 CPUs alone, and within 8% of the Intel Core i7-4770k platform.

I. Introduction

Commodity FPGA-based platforms offer the promise of improving application performance along with power consumption for parallel problems when compared to conventional x86 processors. However, these platforms are expensive, tedious to program, and require low-level design of system connectivity functions to become useful. Barring a few exceptions, they have largely remained out of the reach of the majority of parallel programmers. Over the past decade, there have been many attempts such as the SRC-6 [17], Cray XD1 [1], BEE2 [2], Microsoft Bing [15] which show tremendous promise and opportunity for FPGA-based accelerators. However, their impact and proliferation has been restricted to niche markets and specific domains. While modern FPGA platforms have fairly impressive hardware capabilities, they lack broader acceptance due to cost, closed and non-standard interface drivers, unfamiliar design flows and a high-barrier to developing and porting high-performance computing (HPC) problems to the platform. This is unfortunate as the underlying FPGA technology allows low-power, high-performance implementation of computations by exploiting circuit-style spatial parallelism and customization supported with TB/s of on-chip memory bandwidth and low-latency, high-bandwidth on-chip data transfers.

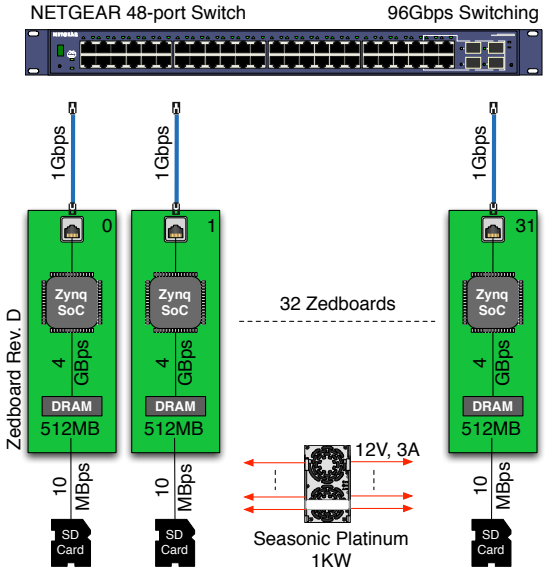

Fig. 1: High-Level Diagram of a 32-node Zynq SoC Compute Cluster with link throughputs and capacities

With the emergence of small, cheap, hybrid SoC chips that combine the FPGA fabric with an ARM CPU such as the Zynq SoC, the barrier to entry for traditional software developers is potentially lower due to variety of reasons – (1) The hybrid SoC combines a 32-bit ARMv7 CPU that can run a complete operating-system which offers tools and environments familiar to software developers. When directly exposed to bare-metal FPGA substrates without OS layers or drivers and a different conceptual model of computation, software developers are often lost and unprepared. (2) The SoC also provides high levels of system connectivity and integration of interfaces to the ARM CPU such as Ethernet, USB, DRAM, and others thereby simplifying I/O management. This allows system developers to concentrate on accelerating the core functions of their problem instead of worrying about low-level bootstrapping details that typically dominate the FPGA development cycle. (3) The emergence of high-level synthesis toolflows from C/C++ [14] offers a familiar path to design and deployment of application

kernels and a simpler model for offloading critical kernels to the FPGA from applications running on the CPU. (4) The hybrid CPU-FPGA integration also eases the mechanisms of configuring and managing the FPGA logic resources from the CPU. FPGA bitstream compilation and configuration times for Zynq Z7020 SoC is quite small (<15–20 minutes from RTL→bitstream).

Pure ARM-based SoCs have been studied and promoted as alternatives to x86-based server-class systems for quite some time. These are particularly appealing for applications that are constrained by communication bottlenecks rather than compute intensity. However, attempts by HP, Dell, Cavium, Calxeda (defunct) and other vendors have enjoyed limited success. The most exciting aspect of these new hybrid SoCs, is the integration of FPGA logic for low-power acceleration. This makes it an intriguing choice for development of HPC-like cluster configurations where the ARM CPUs can handle MPI APIs while the FPGAs provide compute acceleration while requiring lower power. Our Zedwulf design, shown in Figure 1, helps evaluate the energy efficiency potential of Zynq SoCs are for fast parallel evaluation of sparse graph problems like a neural network simulation? We prototype and characterize performance and power of an MPI Beowulf cluster built out of 32 Zedboards each having a Xilinx Zynq Z7020 SoC running Xillinux 1.3 and communicating over a 1G Ethernet network connection. We implement neural network simulations of networks in the range 1M–32M nodes, 1M–32M edges and measure performance under various configurations and workloads.

II. Zynq SoC Specifications and Benchmarking

A. Zynq SoC Architecture

The Xilinx Zynq SoC is a heterogeneous computing architecture that combines 2-core 32b ARMv7 CPUs with a Programmable Logic (PL) fabric coupled using an high-performance AXI/ACP interface as shown in Figure 2. This is a radical departure from traditional FPGA computing platforms that are typically integrated over PCIe interfaces with independent configuration and control from the host CPU. The integration of Ethernet port directly to the ARM CPU and MPI library support makes it particularly attractive for cluster design. In our proposed cluster, we leverage the ARM CPU for data marshaling the MPI structures and protocols while relying on the FPGA fabric for compute acceleration.

B. Benchmarking the SoC

Before we delve into the details of our cluster design, it is worthwhile to compare the architecture capabilities of the embedded SoC against that of an x86 processor as shown in Table I just to contrast the raw capacity of these devices. We use the `lmbench-3.0-a9` benchmark suite [10] to measure the memory bandwidth and latency for various memory operations on L1 cache, L2 cache, and off-chip DRAM. We configure serial, strided and random address generation (self-written μbenchmark) to mimic patterns from communication-bound sparse graph workloads. `netperf-2.6.0` allows us

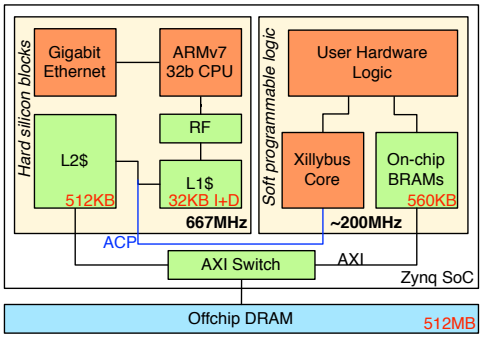

Fig. 2: Zynq Z7020 SoC Block Diagram with key metrics highlighted

to measure TCP and UDP bandwidths from the SoC node under various CPU loads, connection counts and transfer sizes. We employ `coremark` [6] and `dhrystone` [19] to estimate instruction throughput of the CPUs.

As expected, Table I show us that the x86 beats the ARM on almost all metrics except power consumption which is the basis for this design. For x86-based systems, the Ethernet NIC and DRAM interfaces are faster and better integrated. We observed network performance was able to saturate 111 MB/s throughput quite comfortably while random access performance on the DRAM was also faster than the Zynq. On the bright side, the significantly lower power consumption per chip provides an opportunity for a cluster-oriented design where multiple Zynq SoCs can be configured in parallel for energy-efficient operation.

To address this potential, we focus on the ARM SoC capabilities in Figure 3 and show the various system throughputs for the on-chip memories, offchip memories, CPU-FPGA links and the network interfaces. Except the on-chip FPGA BRAM bandwidth which depends on logic configuration, all other data are sustained real-world measurements. We note that FPGA BRAM interfaces support the highest bandwidths across all interfaces. The DRAM throughput of the ARM CPU is also quite high for regular access patterns at ≈600 MB/s but drops by over an order of magnitude to ≈30 MB/s for random access when access distance greater than L2 capacity of 256kB. The AXI-ACP throughputs for communication between the CPU and FPGA fabric are high ≈400 MB/s but not as fast as the sequential memory performance. The MPI throughput for large transfers is around 60-70 MB/s but is comparable to random-access memory performance.

Random access bandwidth to local DRAMs matches the MPI communication bandwidth to other Zynq nodes. This means that, from a throughput perspective, MPI access to data residing on a neighboring node is just as fast as random, irregular access to local DRAM for sufficiently large transfers.

III. Cluster Setup and Performance Analysis

In this section, we describe the engineering of our experimental Zynq SoC cluster and characterize key network behavior patterns that are relevant for our design. The setup

TABLE I: Comparing architecture specifications and microbenhmark results of the Zynq SoC and x86 systems

	ARM-FPGA SoC		x86	Ratio	x86	Ratio	x86	Ratio
Datasheet Specifications								
Platform	Xilinx Zynq Z7020		Intel E5-1650		Intel E5-2609		Intel i7-4770	
Technology	28nm (TSMC HPL)		32nm		32nm		22nm	
μarch	In-order	FPGA	OoO		OoO		OoO	
	(2-core)	-	(6-core)		(4-core)		(4-core)	
Clock Freq.	667 MHz	200 MHz	3.2 GHz	5×	2.4 GHz	3×	3.5 GHz	5×
Memory	32KB L1 x2	560KB	32KB L1 x6	3×	32KB L1 x4	2×	32KB L1 x4	2×
	512KB L2	-	256KB L2 x6	3×	256KB L2 x4	2×	256KB L2 x4	2×
	-	-	12MB L3	-	10MB L3	-	8MB L3	-
Microbenchmark Measurements								
Coremark	1591		14850	9×	10756	7×	20204	12×
DMIPS	1138		17250	15×	13236	12×	22766	20×
Power	4–6 Watts		160 Watts	26×	80 Watts	13×	130 Watts	21×
TCP B/W	70 MB/s		111 MB/s	1.6×	111 MB/s	1.6×	111 MB/s	1.6×
L1 B/W	7.7 GB/s		80 GB/s	10×	59 GB/s	8×	107 GB/s	14×
L2 B/W	1.4 GB/s		57 GB/s	40×	42 GB/s	30×	75 GB/s	54×
DRAM B/W								
(Seq.)	0.6 GB/s		14 GB/s	23×	9 GB/s	15×	20 GB/s	33×
(Rand.)	32 MB/s		330 MB/s	10×	221 MB/s	7×	520 MB/s	16×

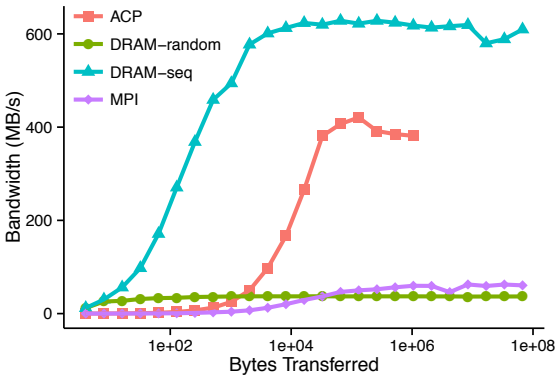

Fig. 3: Zynq SoC bandwidth measurements on the ACP, DRAM and Ethernet interfaces. MPI remote access bandwidth is closely tracking local DRAM random access bandwidth. ACP and sequential DRAM throughputs are 10–20× higher than MPI and random DRAM accesses.

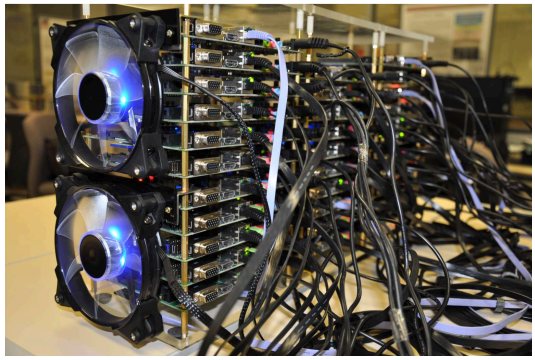

Fig. 4: Zedwulf 32-node Zynq SoC cluster in operation

of the cluster required consideration for embedded boards interfaces and associated software environment configuration.

A. Physical Setup

We use the Xilinx Zedboard Rev. D with the Xilinx Z7020 SoC for our experiments and provide the OS on the Sandisk Ultra Class 10 sdcards. We interconnect the 32 Zedboards using the NETGEAR GS748T 48-port Gigabit Smart Switch with a rated switching capacity of 96 Gb/s with 2 Gb/s throughput per port. We setup the Power Distribution Unit based on the COTS Seasonic Platinum 1KW PSU and leached power from the single PCIe EPS12 rail to power all 32 boards at 12V, 3A with appropriate fuse protection. We record power consumption at the input to this PSU as well as the network switch separately. We provide cooling fans to regulate temperatures in the cluster.

B. Software Environment

We setup the Zedboard with the customized Xillinux 1.3 [20] operating system which ships with low-level FPGA-specific Xillybus drivers. These drivers allow us to interface with the ARM CPU to the FPGA logic fabric using high-speed AXI and ACP interfaces through canonical `/dev` node mapping. We also use the `xdevcfg` driver to reconfigure the FPGA logic in-system during debugging and system testing. We compile FPGA bitstreams using Xilinx Vivado 2013.4 tools to convert high-level C++ descriptions of our parallel code into FPGA logic. We evaluate both OpenMPI 1.8 and

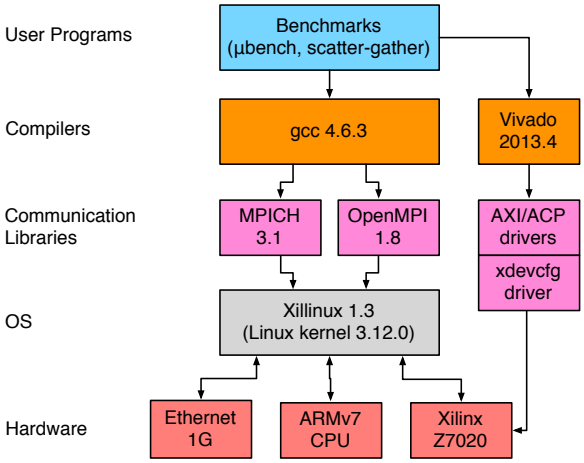

Fig. 5: System Stack for Zynq SoC Experiments

MPICH 3.1 libraries, built from source for ARMv7 using `gcc-4.6.3` with -O3 optimization, for message-passing support across the Zedboard fabric.

C. Network Experiments

The performance and scalability of the MPI networking stack on the ARM SoC is crucial to building a platform that is competitive. The goal is to understand system-level trends and quantify the gap with commodity x86 systems. In our preliminary experiments, we wrote a few new tests and mostly used existing micro-benchmarks to characterize the key system-level performance metrics of the 32-node cluster such as achievable bandwidths and latencies. Again, we use `netperf-2.6.0` and `ping` to measure TCP and UDP bandwidths and end-to-end latencies between various nodes of our cluster under different communication patterns. We ran the Intel MPI benchmark suite [8] to measure latencies and bandwidths of various MPI functions. We measure an `MPI_Barrier` latency of roughly a millisecond due to limitations of the ARM MPI implementation. This is in contrast with the network ping latency of $\approx 150\mu s$ and an x86 barrier latency that was $2\times$ better.

In Figure 6, we show the bandwidth scaling trends for `MPI_Put` and `MPI_SendRecv` API calls to identify the performance gaps between one-sided and two-sided communication primitives as function of transfer size. We also report performance for the MPICH and OpenMPI libraries for `MPI_Put`. This is the *mean* observed bandwidth across multiple trials. From the graph, we can see how bandwidth improves with transfer size but it saturates at 60 MB/s (full duplex) in the best case for large transfers across all three scenarios. For small transfers, we note MPICH offer better behavior while for larger transfers OpenMPI catches up.

Generally speaking, we fail to fully utilize the link bandwidth for the 1G Ethernet port. Equivalent implementations on x86 systems with 1G NICs can often reach 80–90% quite easily. In Figure 7 we plot the MPI function latencies on the x86 and SoC and identify a gap of almost a factor of two in

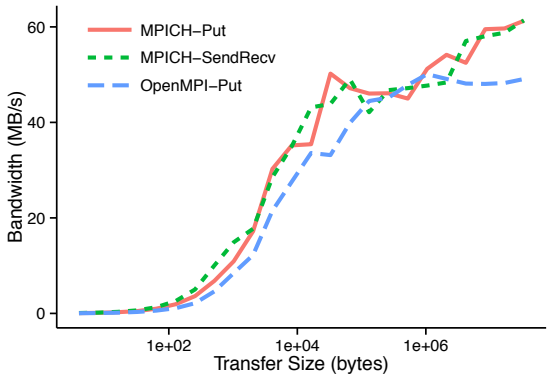

Fig. 6: `MPI_Put` and `MPI_SendRecv` Latency as a function of transfer-size, OpenMPI generally offered poorer throughputs vs. MPICH library

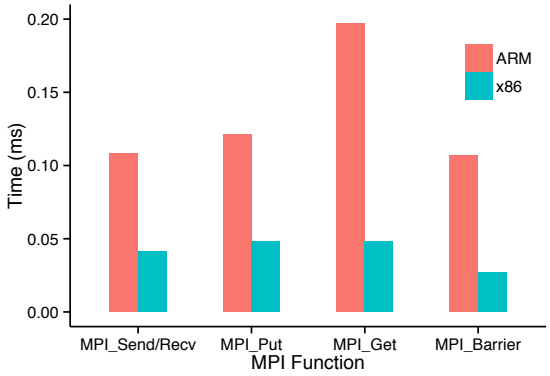

Fig. 7: MPI Function Latency (Intel MPI Benchmark Suite). x86-based MPI stacks run 2–4× faster than equivalent ARM implementations.

all cases. We attribute this gap to the faster CPU and better NIC interfacing available on the x86 platform when compared to the 32b ARMv7 CPU and its 1G network interface. Even without MPI, when measuring raw TCP and UDP (lossless) throughputs, we only managed ≈70–80 MB/s throughputs on the ARM SoC. The MPI throughput is calculated on the raw data and does not include MPI metadata transfers. We also performed a switch saturation test that setup multiple parallel MPI transfers between various pairs of MPI nodes. In this test, we noted no slowdown in the individual link bandwidths; each link ran at the full 60 MB/s that was already possible.

In a cluster environment, MPI throughputs of the Zynq SoCs running Xillinux OS saturates at ≈50% of the theoretical 1G peak capacity of the Ethernet port. This suggests opportunity for careful optimization for sparse, small-message irregular MPI traffic

IV. Sparse Graph Processing

In this paper, we are interested in mapping parallel graph algorithms that fit the Bulk Synchronous Parallel (BSP) [18],

[3], [4] model to the cluster. The BSP model is applicable to a wide variety of parallel computations such as sparse matrix-vector multiply, contextual reasoning, belief propagation, all-pairs shortest path search, betweeness centrality, and many others. In this discussion, we refer to two kinds of nodes (1) graph node in the data-structure, and (2) MPI cluster node that performs the computation. The BSP graph algorithm executes as a sequence of steps where the steps are separated by a global synchronization barrier. In each step, we perform parallel operations on graph nodes where all graph nodes send and receive messages from their corresponding neighbors. When these neighbors are local to a cluster node, the messages are managed internally in the memory space, whereas those that are on remote cluster nodes use a specialized MPI message-passing operation (details in Section IV-B) . Once all messages reach their destinations, each graph node computes on data received along incoming edges.

Function izhikevich(V,I,u,weight,spike)

```
   /* loop over timesteps                                   */
1  foreach t = time do
       /* loop over all neurons                             */
2      foreach i = neuron in neurons do
           /* loop over all synaptic inputs                 */
3          foreach j = synaptic_inputs to neuron i do
4              I[i] = synapse(I[i],weight[j],spike[j]);
       /* loop over all neurons                             */
5      foreach i = neuron in neurons do
6          V[i],u[i],spike[i] = neuron(I[i],V[i],u[i]);
```

A. Sparse Neural Network Simulation

As an example, consider the pseudocode shown in Function izhikevich based on the Izhikevich model [9] of a neuron. This operates on a neural network, or sparse graph, where we can represent the neurons as nodes while synaptic connections between neurons are the edges of the network. They communicate using spikes and perform local computations that recompute node membrane potentials V, recovery variables u and input currents I in the neural network. The outer loop in Function izhikevich (over t) steps through time to simulate different stages of evolution in the network. The first inner loop in Function izhikevich (over i and j) requires irregular graph-oriented communication between neurons along the synaptic connections. In our cluster, we partition and distribute a subset of the neural network to each cluster node. Spike communication is managed using local memory operations, or MPI traffic as appropriate. The second inner loop in Function izhikevich (over i) is a simple data-parallel operation on neurons that performs local updates. This is implemented using configurable logic on the FPGA.

Spinnaker [5] aims to build a large-scale platform for simulation of billions of neurons over a custom ARM-based processing and networking fabric. Our aim is somewhat orthogonal to Spinnaker in that we focus on sparse graph processing rather than brain-scale simulations. Furthermore, our hardware design strategy considers using hybrid low-cost COTS ARM-FPGA SoCs rather than developing a custom ASIC. Other neural simulation accelerators based on VLSI systems [11], GPUs [12], and FPGAs [13] have also explored parallelism using alternative high-performance computing fabrics that are either harder to manufacture, more expensive and some of them are power-hungry. Our goal is to minimize cost and power, while delivering a programmable and scalable computing platform.

Using Vivado HLS 2013.4, we compile high-level C++ descriptions of the neural network evaluations to FPGA logic. Each processor with neural and synaptic computations in IEEE single-precision arithmetic took 4418 LUTs (8%), 4081 FFs (4%) and 24 DSP blocks (9%) of the chip operating at 200 MHz. While these utilization figures look low, system performance is currently saturated by the ACP transfer bandwidth. We still see value in distributing the processing across 32 parallel, independent ACP communication channels, each operating at 400 MB/s throughputs.

B. Communication Engineering

As identified earlier, we need to carefully design our MPI routines for small-message sparse graph traffic to exploit the constrained MPI bandwidths possible using the ARMv7 CPUs. To support this communication pattern between the FPGA accelerators distributed across 32 boards, we use a multi-level mechanism for data movement:

- **Local ACP Traffic**: For neurons connected to synapses *within* the same chip, we simply use the ACP links to communicate streaming data. ACP links operate at roughly 400 MB/s and we are careful to ensure the streaming operation happens in a giant coalesced transfer. The shared arrays are reordered in each BSP step on the host ARMv7 CPU to facilitate local sharing of data.
- **MPI Traffic**: Across-node communication that traverse chip boundaries must use Ethernet links. We use the MPI library to support these message transfers in a manner that supports the sparse irregular nature of the transfers. Efficiency of MPI transfers is critical as our bisection bandwidth for the NETGEAR switch is 96 Gb/s with a 32-node MPI barrier latency of $1ms$ (and $\approx 150\mu s$ ping latency). The FPGA links to the ARM CPU over the ACP channel as shown earlier in Figure 2, resulting in 400 MB/s (sustained) egress and ingress capacity for external traffic into the FPGA. Additionally the Ethernet link only supports $\approx$480 Mb/s (full-duplex) operation further curtailing network throughput.

C. Supporting MPI Communication

For sparse graph edges that cross SoC chip boundaries, we use MPI to support communication in a manner that is cognizant of (1) the irregularity of the traffic pattern, (2) the short transfer sizes of each unique message, and (2) the limited processing capacity of the host ARM CPU.

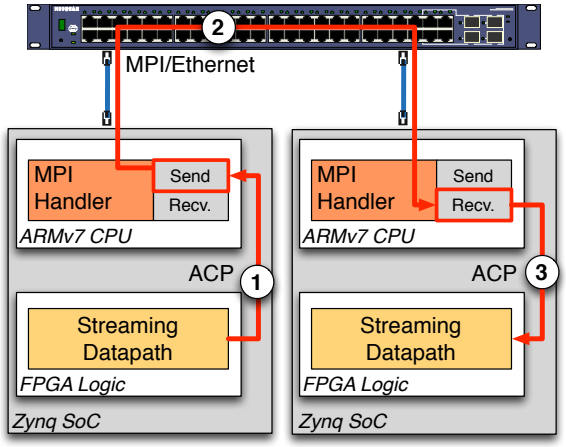

Fig. 8: Sequence of steps for synaptic communication along edge of sparse graph. Step ① and Step ③ operate over the ACP links using Xillybus FIFOs, while Step ② is managed by the MPI library over Ethernet.

This is a unique requirement and there are no out-of-the-box MPI library routines that support this. In [7], the authors introduce a two-sided solution that can outperform oblivious `MPI_alltoallv` implementation for such irregular transfers but aimed at commodity x86 clusters. In [16], the authors develop a strategy for supporting inter-GPU communication for irregular workloads over a high-performance Infiniband cluster. In our approach, we develop a different strategy that needs neither a heavy-duty x86 processor nor Infiniband networks. The key idea is to buffer a sufficient amount of remote traffic into local memory buffers on the CPU supplied from the FPGA datapaths before initiating MPI transfers. This avoids the need to launch a heavyweight MPI message for each fine-grained transfer. We show the path taken by a packet from FPGA in MPI node 0 to FPGA in MPI node 1 in Figure 8. We allocate send and receive buffers that allow an aggregation of messages into larger MPI packets prior to transfer. Data in the send buffer is directly updated by messages from the FPGA over the ACP interface as shown in Step ① and Step ③ in Figure 8. Once the send buffer is filled, we initiate an MPI operation on the data to distribute the data to its intended destination receive buffers as indicated by Step ② in Figure 8. After a global barrier, we then distribute the receive buffer data into the local network. We discuss the specific details of our MPI optimizations briefly below:

- **Building MPI types**: We first translate the adjacency lists into MPI-compatible communication types that encode the graph structure as a series of addresses and counts for send and receive between all-possible pairs of MPI nodes. To construct this type, we loop over all graph nodes and record the destination MPI node for the output edges from the node. Recall that, this is unlike the fine-grained shared-memory access where we have a direct memory lookup for each synaptic edge. Adapting that fine-grained method to MPI access would impose an excessive per-message overhead on the ARM CPU slowing down performance. These counts and address arrays are helpful in aggregating multiple transfers into a single fused MPI message.

```
1  // prepare MPI graph data-structures
2  // recv/send_count/addr are derived from
3  // adjancency-lists
4  for(j=0:total_proc-1) {
5      send_type=MPI_Datatype();
6      recv_type=MPI_Datatype();
7      MPI_Type_indexed(send_count,
8              send_addr,send_type);
9      MPI_Type_indexed(recv_count,
10             recv_addr,recv_type);
11     MPI_Type_commit(send_t);
12     MPI_Type_commit(recv_t);
13 }
14
15 for(BSP steps) {
16     // Local
17     for(i=local nodes)
18         // copy array
19
20     // Send MPI data
21     for(j=0:total_proc-1) {
22         // scheduling to avoid conflicts
23         int target = (rank+j)%total_proc
24         int source = (total_proc+rank-j)%total_proc;
25         MPI_Sendrecv (send_buf, recv_buf, ...);
26     }
27     MPI_Barrier();
28
29     // Do compute stuff
30 }
```

Fig. 9: Basic MPI Communication Skeleton that shows how the `MPI_Datatype` is built and the mechanism of using `MPI_SendRecv` for sparse communication

- **Optimizing MPI types**: We perform a coalesced transfer to one MPI target in a single function call to avoid MPI overheads of finer-grained messages. To achieve this coalesced transfer, we setup the `MPI_Datatype` using `MPI_Type_indexed` to encode a custom sequence of blocks with source and destination positions as shown in Line 4–13 of the code sketch in Figure 9. Each process receives the necessary displacement vectors from a common master process at the start of execution, and builds a pair of `MPI_Datatype` for every target process. When using `MPI_SendRecv`, we do not need to send the `MPI_Datatype` for the receive in each transmission step; it is sent only once at the start. When using `MPI_Put` to allow one-sided distribution of data items to their targets, it is required to send this metadata each time resulting in unnecessary overheads.
- **Protocol Optimization**: We first use Passive Target Communication paradigm, using `MPI_Win_lock` and `MPI_Win_unlock` functions, for executing Remote Memory Access (RMA) calls. In a naive implementation, each process will start an RMA access epoch with win_lock, call a put operation, and close the epoch with win_unlock sequentially for every process, including itself. Profiling this code allowed us to uncover opportunities for overlapping communication in the system by (1) having simultaneous

epoch sessions in progress and ensuring load-balanced scheduling of message transfers in a cyclic fashion, and (2) replacing local `MPI_Put` with simple array-indirection. However, for RMA-based distribution of data, there is an inherent need for the MPI protocols to encode the receive metadata each time resulting in unnecessary network communication overheads. By resorting to MPI two-sided communication using `MPI_SendRecv`, this drawback is rectified, by having both the origin and target process specify the send and receive buffers in every matching MPI call. Similar to the RMA approach, this approach schedules data transfer in a periodic fashion to avoid contention across nodes. We show this in Lines 21–26 in Figure 9.

V. Experimental Results

In this section we present system-level performance and power trends for the Zynq cluster as well as the x86 platforms we use in this study. For our graph problems, we use the MTEPS (Million Traversed Edges/second) metric which is used to rank graph-processing platforms. We consider various graph sizes 1–32M nodes/edges that fit the 512 MB/board DRAM limits of 32 Zedboard platforms. We compare our performance against a single x86 node running fully-optimized neural evaluations directly across multiple (4–6) OpenMP threads. We compare total system power including the power of the network switch for the Zedwulf measurements using the Energenie power meter. We use Boost graph library to capture the graph structure and partition the neural networks (or graphs) across multiple nodes at the start when building the MPI data types for concurrent evaluation.

A. Comparing Zedwulf against x86

We are interested in comparing the performance-power characteristics of a high-performance x86 platform against the low-power Zynq SoCs. We plot the power utilization (W) vs. overall performance (MTEPS) in Figure 10. As we can see, the 32-node Zynq system can match and exceed the performance of the multi-threaded 1-node x86 platforms for graphs with 32M nodes and 32M edges. When we disable FPGA acceleration and exclusively use the ARMv7 CPUs for both compute and communication, we note a drop in performance by around 20–25% and a slight saving in power ≈10 W due to fewer ACP/DRAM IO activations. All platforms (except Zedwulf with ARM-only processing) lie approximately along the iso-energy-efficiency line suggesting that FPGA co-processing allows the ARM SoCs to match the efficiency of the x86 machines for irregular problems.

We directly compare the energy efficiency (MTEPS/Watt) as shown in Figure 11. This confirms our observation that the Zynq SoCs offer near identical energy efficiency compared to x86 implementations. They are within 8% of the best x86 efficiency reported by Intel Core i7-4770k system.

B. Scaling Trends on the Zedwulf cluster

In Figure 12, we show performance scaling trends for the computation and communication phases of the sparse

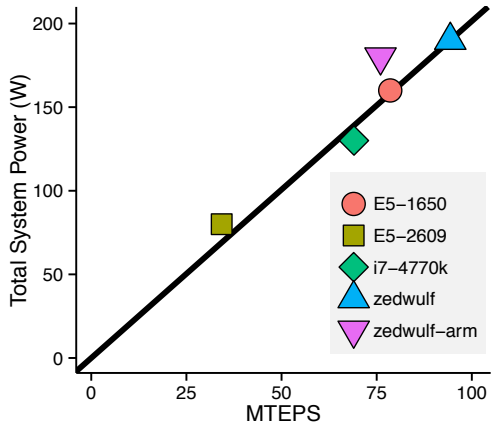

Fig. 10: Performance-Power tradeoffs for x86 systems against the Zedwulf cluster (MTEPS – Million Traversed Edges per Second)

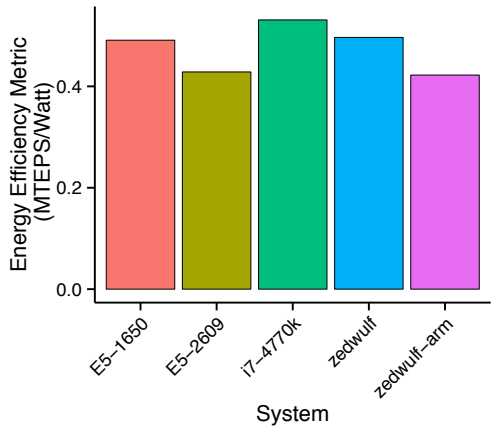

Fig. 11: Energy Efficiency Comparison between x86 and 32-node Zedwulf cluster (Higher MTEPS/Watt is better)

graph processing with and without FPGA co-processing across varying number of Zedboards. As expected, we are able to smoothly scale down the compute time with more MPI nodes when using ARM processors alone but spend as much time in MPI communication. When we use FPGA accelerators, compute time drops significantly and only approaches communication time at larger system sizes. It is interesting to note that MPI overheads are disproportionately higher at smaller system sizes even though the number of remote messages grow with system size.

With 32 Zynq-nodes, we consider the impact of the size of the graph on speedup over x86. Not surprisingly, we observe higher speedups when the number of edges are low suggesting less time spent communicating over MPI. As we increase the number of edges, speedup starts to drop for most graphs with 8M or more nodes due to MPI overheads and good caching behavior on the x86 system. When the number of nodes is

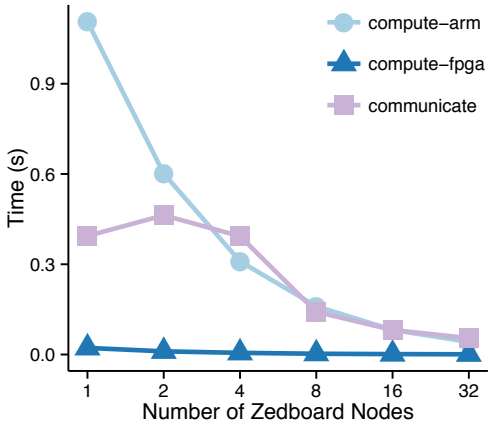

Fig. 12: Performance scaling of compute and communication time for graph with 4M nodes and 4M edges

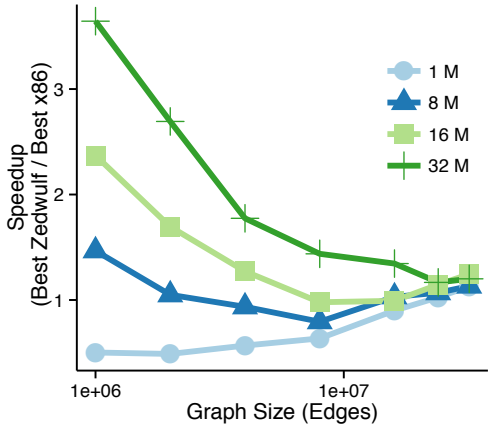

Fig. 13: Impact of graph sizes on performance on 32-node cluster (colored lines are Nodes, x-axis is Edges)

small 1–4M, there are practically no speedups as the x86 caches can fit the complete graphs quite easily. Above 10M edges, across all graph node counts, speedups start to increase again, as the rising x86 cache misses (see Table II) match Zedwulf MPI overheads.

Edges	**1M**	**2M**	**4M**	**8M**	**16M**	**24M**	**32M**
Miss Counts	10K	14K	18K	27K	91K	174K	260K
Miss Rate	15%	16%	16%	16%	29%	34%	37%

TABLE II: Last Level Cache (LLC) Misses (E5-1650 x86, 32M nodes, varying edges)

VI. Conclusions

The Zedwulf 32-node Zynq cluster is able to deliver high sparse graph processing throughput for neural network simulations of 94 MTEPS at an efficiency of 0.49 MTEPS/W. This is competitive with optimized multi-threaded x86 implementations because we use ARMv7 host CPUs to handle MPI communication while relying on FPGA co-processing for compute acceleration of irregular problems. While the current generation Zynq SoCs show promise, we can widen their energy efficiency margins over x86 by moving to modern ARMv8 microarchitectures and providing support for faster Ethernet and networking support. With process technology advances, these improvements will likely accelerate and bridge the gap between SoCs and conventional processors.

References

[1] U. Bondhugula, A. Devulapalli, J. Fernando, P. Wyckoff, and P. Sadayappan. Parallel FPGA-based all-pairs shortest-paths in a directed graph. *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, 2006.

[2] C. Chang, J. Wawrzynek, and R. Broderson. BEE2: a high-end reconfigurable computing system. *Design & Test of Computers, IEEE*, 22(2):114–125, 2005.

[3] M. deLorimier, N. Kapre, N. Mehta, and A. DeHon. Spatial hardware implementation for sparse graph algorithms in GraphStep. *ACM Transactions on Autonomous and Adaptive Systems*, 6(3):1–20, Sept. 2011.

[4] M. deLorimier, N. Kapre, N. Mehta, D. Rizzo, I. Eslick, R. Rubin, T. E. Uribe, T. F. J. Knight, and A. DeHon. GraphStep: A system architecture for sparse-graph algorithms. In *Field-Programmable Custom Computing Machines, 2006. FCCM'06. 14th Annual IEEE Symposium on*. IEEE, IEEE Computer Society, 2006.

[5] S. Furber. Low-power chips to model a billion neurons. *IEEE Spectrum August. http://spectrum. ieee. org/computing/hardware/lowpower-chips-to-model-a-billion-neurons*, 2012.

[6] S. Gal-On and M. Levy. Exploring CoreMarkTM – A Benchmark Maximizing Simplicity and Efficacy. Technical report, EEMBC, Apr. 2012.

[7] T. Hoefler and J. Traff. Sparse Collective Operations for MPI. In *IPDPS '09 Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–8, Aug. 2014.

[8] Intel. Intel MPI Benchmarks. Oct. 2013.

[9] E. M. Izhikevich. Simple model of spiking neurons. *Neural Networks, IEEE Transactions on*, 14(6):1569–1572, 2003.

[10] L. W. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. *USENIX annual technical conference*, 1996.

[11] C. Mead and M. Ismail. *Analog VLSI implementation of neural systems*. Springer, 1989.

[12] J. M. Nageswaran, N. Dutt, J. L. Krichmar, A. Nicolau, and A. V. Veidenbaum. A configurable simulation environment for the efficient simulation of large-scale spiking neural networks on graphics processors . *Neural Networks*, 22(5-6):791–800, July 2009.

[13] M. Naylor and S. W. Moore. Rapid codesign ofa soft vector processor and its compiler. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–4, June 2014.

[14] S. Neuendorffer and F. Martinez-Vallina. Building Zynq accelerators with Vivado high level synthesis. Technical report, Xiilnx, 2013.

[15] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J. Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 13–24, 2014.

[16] R. Shi, S. Potluri, K. Hamidouche, M. Li, D. Rossetti, and D. K. Panda. Designing Efficient Small Message Transfer Mechanism for Inter-node MPI Communication on InfiniBand GPU Clusters, Conference on High Performance Computing . In *International Conference on High Performance Computing*, Dec. 2014.

[17] M. C. Smith, J. S. Vetter, and X. Liang. Accelerating Scientific Applications with the SRC-6 Reconfigurable Computer: Methodologies and Analysis. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, page 157, 2005.

[18] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8), Aug. 1990.

[19] R. Weicker. Dhrystone: a Synthetic Systems Programming Benchmark. *Communications of the ACM*, 27:1–18, Jan. 1984.

[20] Xillybus. Xillinux: A Linux distribution for Zedboard. Oct. 2014.