eBSP: Managing NoC traffic for BSP workloads on the 16-core Adapteva Epiphany-III Processor

Siddhartha
Nanyang Technological University
50 Nanyang Avenue, Singapore
siddhart005@e.ntu.edu.sg

Nachiket Kapre
University of Waterloo
200 University Ave W, Waterloo, Canada
nachiket@uwaterloo.ca

***Abstract*—We can deliver high performance and energy efficient operation on the multi-core NoC-based Adapteva Epiphany-III SoC for bulk-synchronous workloads using our proposed eBSP communication API. We characterize and automate performance tuning of spatial parallelism for supporting (1) random access load-store style traffic suitable for irregular sparse computations, as well as (2) variable, data-dependent traffic patterns in neural networks or PageRank-style workloads in a manner tailored for the Epiphany NoC. We aggressively optimize traffic by exposing spatial communication structure to the fabric through offline pre-computation of destination addresses, unrolling of message-passing loops, selective squelching of messages, and careful ordering of communication and compute. Using our approach, across a range of applications and datasets such as Sparse Matrix-Vector multiplication (Matrix Market datasets), PageRank (BerkStan SNAP dataset), and Izhikevich spiking neural evaluation, we deliver speedups of 6.5–10× while lowering power use by 2× over optimized ARM-based mappings. When compared to optimized OpenMP x86 mappings, we observe a 11–31× improvement in energy efficiency (GFLOP/s/W) for the Epiphany SoC. Epiphany is also able to beat state-of-the-art spatial FPGA (ZC706) and embedded GPU (Jetson TK1) mappings due to our communication optimizations. Our library is open-source and available at github.com/sidmontu/ebsp.git.**

I. Introduction

Modern embedded SoCs (systems-on-chip) are increasingly composed of host CPUs supported by custom accelerators capable of accelerating certain classes of problems with high efficiency. It is vital that such SoCs are augmented by a fast and capable communication fabric (NoC, network-on-chip) that makes it possible to accelerate a range of communication-intensive problems while lowering power requirements by avoiding complex shared-memory controllers. In this paper, we investigate the raw potential and tuneability of the Epiphany III SoC, a bespoke computing platform optimized for processing of communication-rich floating-point applications. A high-level diagram of this Parallella development board we use in this study is shown in Figure 1. The Epiphany-III SoC consists of 16 RISC `eCores` operating at 667 MHz, equipped with single-precision floating-point ALUs supported by a 32 KB scratchpad RAM per eCore. The centerpiece of this parallel organization is the communication fabric that is composed of three separate NoCs for on-chip writes, off-chip writes and reads. The chip provides simple bare-metal programming control over the RISC eCores with software control over local

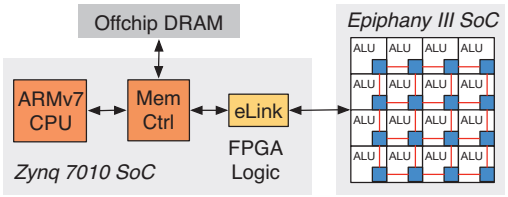


Fig. 1: Parallella Board with Zynq 7010 SoC and 16-core single-precision floating-point Epiphany-III SoC.

32 KB scratchpads in each eCore, and dedicated communication APIs that allow the RISC eCores to directly inject appropriate load/store or DMA-style traffic into the NoC. In this paper, we develop the eBSP API for exposing and optimizing communication for NoC-based SoCs such as the Epiphany-III through a graph-centric bulk-synchronous model. We expect our ideas to be broadly applicable to other NoC-based accelerators such as Kalray [6] and Tilera [4] SoCs.

The key contributions in our paper include:

1) Development of a communication library for expressing bulk-synchronous communication patterns with underlying optimizations targeting the Epiphany SoC.
2) Performance analysis and microbenchmarking of the Epiphany-III NoC to build an understanding of communication capabilities in the system and contrast it against FPGA overlay NoCs.
3) Characterization of Epiphany performance and power across a variety of workloads including: (a) Sparse Matrix-Vector multiplication (SpMV), (b) Google PageRank, and (c) the Izhikevich spiking neural network evaluation. Also, performance and energy-efficiency comparisons of Epiphany against state-of-the-art FPGA (ZC706), GPU (Jetson TK1) and x86 (Xeon) implementations.

II. Background

A. Epiphany-III SoC

We use the Parallella SoC board [17], [12] for our experiments. The board consists of a Xilinx Zynq 7010 SoC with an ARM CPU, an on-die FPGA co-processor and a separate Epiphany-III SoC [11]. The Zynq SoC provides the frontend to the Epiphany chip for programming and debugging. The Epiphany chip interfaces with the Zynq SoC over an eLink interface implemented on the FPGA that allows loading of

978-3-9815370-8-6/17/$31.00 ©2017 IEEE

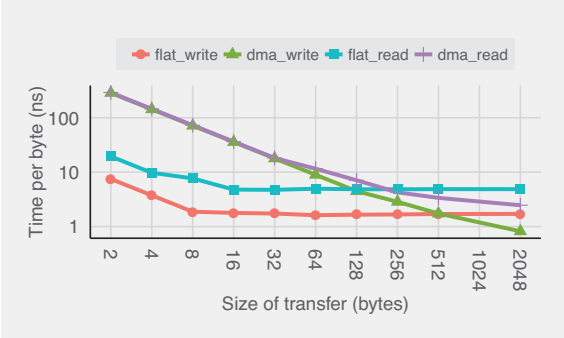


Fig. 2: Understanding `e_dma_copy` and `flat`-addressing.

program binary as well as DMA transfers of data between the host (ARM) and the accelerator (Epiphany).

Hardware: The Epiphany SoC is a custom architecture with a RISC-style ISA optimized for floating-point processing and NoC communication. Each eCore supports a single-precision floating-point ALU capable of achieving up to 1.2 GFLOPs/s per eCore (667 MHz fused multiply-add). The minimal RISC implementation of the eCore enables a lightweight implementation optimized for fast, energy-efficient operation *i.e.* the 16-core chip only draws 1–2 W power. The program and data are stored in the 4-bank 32 KB scratchpad that must be *explicitly managed by the programmer*.

Software: The Epiphany cores are programmed bare-metal directly in C and are supported by the Epiphany SDK that is hosted on the ARMv7 Zynq CPU. Each Epiphany eCore can be loaded with a unique binary in MIMD style. Thus, the programmer is able to run host code on the ARMv7 CPU with full Linux OS support for management of acceleration while the accelerated code runs on the Epiphany eCore as a bare metal executable. Special low-level functions are provided for NoC communication and synchronization.

B. Microbenchmarking the Epiphany NoC

Instead of relying on coherent caches for sharing state between parallel processing units, the Epiphany eCores coordinate and move data between each other explicitly using message-passing on the NoC. The `eMesh` NoC on the Epiphany consists of three parallel physical channels: one channel for on-chip write traffic (cMesh), one channel for off-chip write requests (xMesh) and the last channel for read-request style traffic (rMesh). The on-chip write channel handles 8-byte transfer per cycle, the off-chip write channel only supports 1-byte transfer per cycle, while the read request channel supports a single request every 8 cycles. This represents a wide variance in capabilities and also represents peak behavior rather than real-world performance. Thus, to get a better understanding of what the NoC can do for us, we run a series of microbenchmark tests to stress-test the design and identify opportunities and find bottlenecks.

Communication between each eCore is supported with two types of memory transfers: an explicit `flat`-addressing memory transfer or an `e_dma_copy`[1] (streaming read/write) that uses the DMA channel. Since the address space of the 16 eCore scratchpads is global, the `flat`-addressing style memory transfer allows any eCore to write or read any value from any eCore by simply instantiating global pointers to the local memory banks.

In Fig. 2, we show performance results for both the supported strategies with varying transfer sizes. Note that each write request is sent as a packet with an 8-byte payload on the cMesh, and hence, any <8-byte payloads are zero-padded appropriately. The DMA operation is managed by dedicated DMA engines that can orchestrate memory transfers over two DMA channels alongside regular instruction execution by the eCores' CPUs. However, there is a startup cost to launch DMA transfers ($\approx$ 500 cycles), which is amortized only for long transfers. Fig. 2 shows the performance of `flat` addressing transfer strategy vs the performance of the `e_dma_copy`, with varying total transfer sizes. The plot suggests crossover threshold of 256–512 bytes for using DMA-based communication. Furthermore, we observe that `read` understandably has lower bandwidth than `write` transfers as the wider 8-byte NoC channels are available for writes while only 1-byte NoC channel is available for reads.

As NoC performance for short packets is slower than larger bulk transfers, this is a significant concern for irregular, communication-bound problems. Computations such as sparse graph-oriented processing that exchanges a larger number of short messages are affected. Hence, we need to exploit opportunities for NoC optimizations for these workloads.

III. Communication Optimization

In this paper, we model our communication workload as a graph G where computation happens at the graph node n while communication is along graph edge e. The edges represent communication within the application. The implementation model for the applications is an instance of Bulk Synchronous Parallelism [21] (BSP). Under this model, in each application iteration (or epoch), all graph nodes first perform computations based on state stored in the node followed by a communication along the edges. For our application scenarios, the nodes may be matrix rows (SpMV), webpages (PageRank), or neurons (Izhikevich model). Computation performed at each node is supplied by the programmer separately while the eBSP optimization APIs are generic. The iterations are separated from each other with a global barrier implemented on the Epiphany using the fast `e_barrier()` routine (≈300 cycles/call). As the write bandwidth on the Epiphany is 8× higher, we describe all our BSP message transfers as write-oriented operations. The basic communication template used in our optimization is shown in Listing 1. Here, we have a for loop over all message injections into the NoC. Each message transfer requires the destination address of the remote eCore, the memory address within the remote, and the actual data being sent. We use a CSR-inspired [20] storage format optimized for compressed storage of the sparse communication graph structure on the 32 KB/eCore scratchpad. As a result, we need to unpack and process relevant fields to generate destination address.

[1]All routines prefixed with "e_" only run on Epiphany eCores.

```
for(int m=0; m<MESSAGES; m++) {
  float data = state[m];
  int addr = node_id[m]*offset + edge_id[m];
  int dest = remote[m];
  send(dest,addr,data);
}
```

Listing 1: Original BSP Message-Passing Code Template.

- **Address Precomputation**: For typical BSP workloads, the graph connectivity information is statically known and does not change structure at runtime. This means that for each NoC message, we can store the physical destination address in suitably encoded data structures directly on the eCores. Hence, instead of calculating destination memory addresses for the data movement at runtime, we can simply look them up. This is particularly effective on the Epiphany eCore because (1) it has poor 32b integer support which is critical for memory address calculation, and (2) the BSP model requires repeated evaluation of the complete communication graph across multiple epochs. However, this comes at the expense of extra storage cost to hold the calculated address information within the 32 KB scratchpad, typically doubling memory requirements for storing each edge.
```
send(dest[m],addr[m],data[m]);
```
- **Communication Squelching**: For certain BSP applications (see PageRank and Neural Network descriptions later in Section IV), it is frequently observed that certain message values sent along an edge in a given BSP iteration are not different from those in the previous iteration. This happens when a PageRank score for a node has stabilized, or a neuron has not fired in a given step. We exploit this observation to significantly reduce NoC message activity by squelching a message that has not changed state. The destination node may simply reuse an older value cached in the message buffer for that edge.
```
if(cond) {send(dest[m],addr[m],data[m]);}
```
- **Message Loop Unrolling**: As highlighted earlier, poor support for 32b integer operations results in non-trivial runtime overhead in Epiphany code for managing loop iterators, data loads, and function calls to the communication API within the loop body. This results in very low message injection rate particularly for short transfers. For these cases, we manually unrolled the loops that generate traffic for the NoC to help push more data into the NoC and encourage higher utilization.
```
send(dest[2*m],addr[2*m],data[2*m]);
send(dest[2*m+1],addr[2*m+1],data[2*m+1]);
```

Finally, all optimizations discussed in this section are part of our eBSP library's API, full documentation for which can be found on our open-source github repository at github.com/sidmontu/ebsp.git.

IV. Case Studies

In this section, we describe the various bulk synchronous applications we evaluate, and associated code sketches. We report the resulting performance achieved when mapped to the Epiphany-III SoC as well a performance breakdown explaining the trends. We also compare results against state-of-the-art FPGA (ZC706) and embedded GPU (Jetson TK1) mappings reported in literature. These platforms consume less than 10 W power and have approximately similar capabilities as the Epiphany-III. We develop reference C implementations of each of the applications on the ARMv7 and x86 CPUs for timing evaluation and functional correctness testing. We measure runtime using the lightweight PAPI library for C code compiled using `-O3` optimization that exploits SIMD vectorization where possible. We develop a lightweight memory manager to help perform memory layout (for both **code** and **data**) in the tight 32 KB scratchpad space (*e.g. e_malloc* to assign memory on each eCore). We measure accelerated eCore runtime using the lightweight Epiphany event timer API.

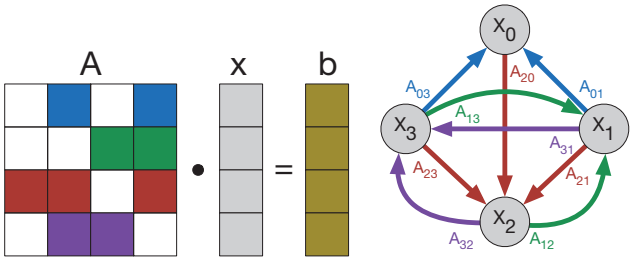
Since we manage the local data memories ourselves, we partition the graph into sub-graphs and load them via overlapped DMA transfers. In state-of-the-art parallel CPU and GPU implementations, this partitioning (or load-balancing) of the workload is performed as row permutations for SpMV kernels, k-means clustering for PageRank workloads and pseudo-random partitioning for Neural networks. We use identical one-time heuristics to decompose the graph and reuse the partitions across iterative BSP steps. We use real-world datasets from the Matrix Market benchmark suite [3], and BerkStan web graphs from the Stanford SNAP [15] datasets.

A. Limitations of epiphany-bsp library (git hash 437c01b, github.com/codeuin/epiphany-bsp)

The *epiphany-bsp* library provides an easy-to-program BSP framework for the Parallella SoC, and offers several high-level software routines for moving and organizing data. Our experiments show that the performance of the *epiphany-bsp* library suffers at the expense of keeping the abstraction compatible with BSPLib. We attribute this poor performance to complex design strategies that incorporate interrupts, mutexes and off-chip buffering techniques to implement the software routines in the library. While the library does offer powerful software abstractions (*e.g.* tagged message-passing between eCores using buffered queues), they are unsuitable for practical use on the bulk synchronous designs we evaluate in this study. We observe that the *epiphany-bsp* is 16× slower than even ARMv7 implementations in the best case scenario. We expect the optimizations proposed in our work can be adapted to help improve and enhance the software abstractions provided by *epiphany-bsp* library.

B. Sparse-Matrix Vector Multiply (SpMV)

Irregular floating-point computations such as sparse matrix-vector multiplication $A \cdot x = b$ (commonly used in embedded scenarios such as software defined radio, computer vision) are constrained by the cost of irregular memory accesses and subsequent poor utilization of sequential processing engines. It is often repeatedly called inside an iterative algorithm (e.g. conjugate-gradient) and it quickly becomes a compute bottleneck as problem sizes exceed cache limits.



(a) Sparse Matrix Form of SpMV Input (b) Graph Form of the Sparse Matrix

Fig. 3: Graph representation of sparse matrix **A** where each A_{ij} is an edge and each row is a vertex.

To parallelize SpMV [7], [1], we represent the computation as a graph, which is partitioned and evaluated on multiple processors using the BSP abstraction. We consider datasets derived from the Matrix Market benchmark set and evaluate those for multiple BSP iterations. On the Epiphany, we can partition the graph across 16 eCores, where each eCore processes fanins and fanouts of all nodes local to the eCore. We show high-level code sketches of SpMV on the Epiphany in Listing 2. The outer loop iterates over all nodes in the eCore, the first loop over fanins performs the row-vector product while the last loop sends the resulting vector values to other eCores in preparation for the next SpMV iteration. Here, the `send` function sends the data along the graph edge to the appropriate eCore and is a dominant component of parallel runtime when left unoptimized. The communication pattern seen in this application is an instance of **Irregular, Fine-Grained Store**.

In Figure 4, we show the performance comparison of our fully-optimized BSP implementation against optimized ARM mapping of the SpMV application across various datasets (matrices). We observe speedups of 2–8× across our benchmark set while consuming ≈50% power for the Epiphany chip over the Zynq baseline. We provide a performance breakdown in Figure 5 for the `jpwh_991` benchmark. Without NoC optimizations, the parallel Epiphany code runs 1–2× faster than the ARMv7 reference implementation. We deliver a further 2–4× speedup by adopting the address precomputation optimization. Loop unrolling of the fanin and fanout computations yield a marginal further improvement of ≈10–20%. We observe that speedups increase and saturate around 2–8× when we run multiple BSP iterations as the cost of launching the Epiphany accelerator gets amortized out with more launches. Each iteration is separated by an on-chip `e_barrier()` call that avoids a round-trip to the host ARMv7 CPU.

```
foreach node n in eCore {
  sum = 0;
  foreach fanin f of n {
    // x(f) is the received value on edge
    sum += A(n,f) * x(f); // multiply-accum
  }
  x(n) = sum; //row result
  e_barrier(); // global barrier
  foreach fanout f of n {
    val = x(n); //x(n) is the value to send
    c = get_coreid(f);
    o = get_local_offset(f);
    address = e_get_global_address(c,o);
    send(val,address); // NoC
  }
}
```

Listing 2: SpMV Code Sketch. Each message needs destination eCore, destination address and value being sent.

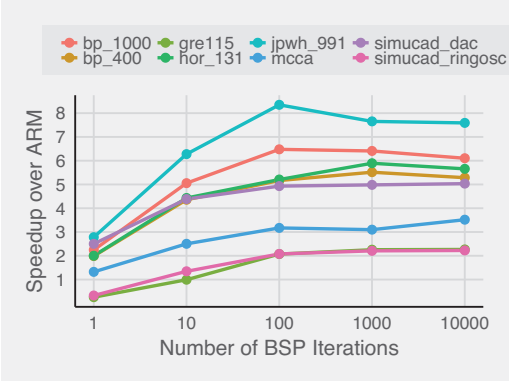


Fig. 4: SpMV speedups on representative sparse matrices over optimized ARM implementation.

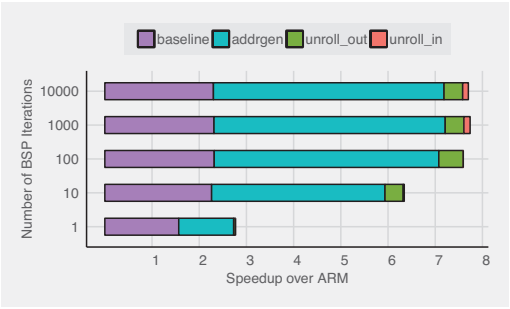


Fig. 5: SpMV Performance Breakdown by NoC optimizations for the `jpwh_991` dataset.

C. Google PageRank

PageRank [18] is a well-known link-analysis algorithm used by Google to rank webpages for its search engine. An accelerated energy-efficient implementation of this computation is highly desirable due to the dynamic nature of web content and power and cost constraints of existing implementations. Each webpage in the Google database is given a PageRank score based on its connectivity to other webpages. The input to the PageRank algorithm is a web connectivity graph, where each vertex represents a webpage, and each directed edge represents a hyperlink from edge-source to edge-sink webpage. The PageRank algorithm is highly parallel and we directly formulate it as a BSP graph problem instead of the SpMV-based Power Iteration method. The algorithm runs in multiple bulk-synchronous timesteps. In each step, the nodes send updates along fanout edges. The algorithm terminates when all PageRank scores have stabilized. Here the transmission of messages from a node is conditional to whether the node has stabilized already. This application is an instance of **Data-Dependent, Fine-Grained Store** as messages are selectively *squelched* (suppressed) if a certain threshold condition is met.

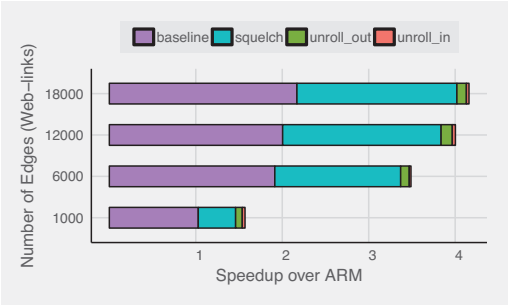


Fig. 6: Google PageRank binned-average speedup contributions by optimization type.

In the context of this Epiphany implementation, we expect a cluster of low-power, floating-point, message-passing processors to provide a customized energy-efficient implementation of PageRank for a cloud-based accelerator. Hence, the amount of network traffic varies depending on the specific conditions of the network and distribution of activity across the network. This implies we have NoC traffic patterns that vary over time and stress the network with lightly loaded conditions in one phase and heavy load in another phase. We use the BerkStan web graph from the SNAP [15] library as input. We threshold the minimum PageRank to 0.15 and set the damping factor, d, to 0.85. The PageRank results are verified with reference CPU implementations once all nodes on all eCores have stabilized.

We show a breakdown of speedups due to various NoC optimizations in Figure 6. Here, the various datasets are binned into groups based on the number of edges and then each bin is **averaged**. For these workloads, the inherent parallelism in the PageRank algorithm delivers the first 1.5–2× improvements when mapped to the 16-eCore Epiphany. Additional speedups are delivered by a combination of squelching (as much as 9× for certain large graphs), and loop unrolling optimizations (10–20%). Thus, squelching delivers bulk of the speedup improvements by eliminating needless NoC traffic. This also has an effect on the `addrgen` optimization we use in the SpMV case study. Due to the lower activation rate of the edges with squelching enabled, the performance gains from `addrgen` optimization are significantly reduced, and therefore, the memory-performance tradeoff is no longer favorable here.

For smaller graphs <500 edges, the baseline Epiphany performance is *slower* than the ARM (0.1–0.9× slower) as the problem size fits within the ARM caches. However, as the graph sizes increase, the ARM can no longer avoid cache misses, and with the added advantage of multicore parallelism from the Epiphany, we observe speedups of up to 10× for larger graphs. We see a spread in speedups due to variations in fanin and fanout sizes of certain bottleneck nodes. These nodes are popular webpages with multiple links.

D. Spiking Neural Networks

In computing, a neural network is an information processing architecture inspired by the biological ways in which the human nervous system and human brain neurons process information. Energy-efficient, configurable implementations of a neural network is useful to help build large-scale simulators of neural and biological phenomena. In this paper, we represent the neural network as a graph of neurons interconnected by synaptic edges. Each node in the graph is a neuron, while all directed edges between nodes are synapses that relay information between connected neurons. In each timestep, based on modeled heuristics, a neuron (node) can "fire". A "fire" trigger event is then relayed downstream to any other neurons (nodes) that are connected to the "firing" neuron. This trigger event is modeled as a bulk synchronous communication step in the evaluation. Each neuron then processes any incoming trigger events and does a compute and local update, which can vary depending on the neuron model adopted. We map the simple model of spiking neurons by E.M. Izhikevich [13] to the Epiphany using synthetically generated neural graphs. Here, the local compute at each neuron is composed of simple addition, subtraction and multiplication operations that are suitable for parallel execution on simple RISC eCores.

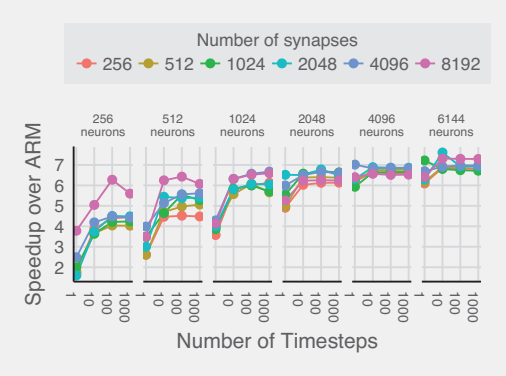


Fig. 7: Overall speedups achieved when simulating Izhikevich model neural networks.

In Figure 7, we show overall speedups for various synthetic neural network graphs generated from the Izhikevich spiking model for various combinations of neuron and synapse counts. We are able to deliver high speedups 5–7× when running 100 or more iterations of the spiking evaluation. The speedups are consistently higher at larger neuron counts. As we increase number of neurons in the network, we increase the amount of overall work that needs to be done to evaluate local computations at each neuron.

V. Related Work and Discussion

For the SpMV kernel, the authors in [9] report a best-case efficiency of 0.11 GFLOPs/W on the Jetson TK1 GPU and as opposed to the 0.24 GFLOPs/W we are able to achieve on the Epiphany. A recent effort on a Stratix V custom FPGA board [8] reports a best-case 0.221 GFLOPs/W (25 W total power). Considering a 5× improvement possible from CoRAM++ [22], an SpMV mapping could potentially improve from 50 MFLOPs/s for CoRAM to 250 MFLOP/s for CoRAM++ on an 8 W ZC706 system while still delivering a

TABLE I: Energy-efficiency comparison of Epiphany SoC.

Application	Our mapping			State-of-the-Art		
	Epiphany	ARM	x86	FPGA	GPU	x86
SpMV (MFLOP/s/W)	276	45	26	221 [8]	110 [9]	75 [5]
PageRank (MEPS/W)	55	13	5	-	3.6 [10]	1.9 [10]
NeuralSim[a] (μJ/synap. event)	1.9	13	59	6.3 [2]	2.2 [19]	9.8 [19]

[a]Lower is better for Neural Network, for other rows, higher is better

paltry 0.03 GFLOPs/W. For PageRank, in [10], the authors report peak edges per second (EPS) of 0.18 billion EPS (BEPS) on a single Intel Nehalem Xeon X5650 CPU. In contrast, we achieve a peak EPS of 0.11 BEPS on the Epiphany, at almost 50× less power. In [19], the authors simulate a Izhikevich spiking network with 10K neurons and 18M synapses for 3s on a cluster composed of Jetson TK1 boards. An FPGA implementation on a ZC706 board [2] with 746 neurons and 174K synapses is evaluated for a 1s simulation of an oscillatory grid. The Epiphany NoC delivers an energy efficiency of 1.9μJ/synaptic event (2W power, 544ms, 581k synaptic events), compared to 2.2μJ/synaptic event for the Jetson TK1 and 6.3μJ/synaptic event on the ZC706[2].

In Table I, we compare the performance of our optimized Epiphany implementations with ARM (and x86 for context). While the x86 outperforms the alternatives on absolute performance, it loses to the ARM by 2–5× and the Epiphany by 11–31× when considering energy efficiency. This suggests a clear advantage for simpler RISC-like cores for optimized processing of floating-point workloads while also supporting spatial parallelism using a fine-grained packet-switched NoC. For the neural network simulations, communication intensity is sporadic and a less dominant component of total runtime, hence the optimizations deliver lower improvements.

For FPGA-based overlays, NoC traffic compilation strategies were previously explored in [14]. NoC-optimizations have also been previously explored in [16] (Computational Biology), [6] (Timing Critical systems), [4] (community detection), among others. Our work attempts to develop a reusable API that is generalizable to these application domains, and other NoC-centric architectures.

VI. Conclusions

We show how to expose and optimize communication on the floating-point Epiphany-III embedded SoC to deliver 4–7× energy-efficiency improvements over ARMv7, and 11–31× improvements over OpenMP-optimized x86 mappoings (GFLOP/s/W) with our eBSP API. Epiphany is also able to deliver superior results to other platforms in its class such as CUDA-optimized implementations on the embedded GPUs (Jetson TK1) as well as spatial FPGA mappings (ZC706). The use of offline calculation of destination addresses, unrolling of message-passing loops, selective squelching of NoC traffic, and careful ordering of compute and communicate phases are the key optimizations provided by our library. Our eBSP library is open-source and available at github.com/sidmontu/ebsp.git.

[2]Calculated based on ZC706 platform power of ≈8 W with 120 K spiking events and 94 ms of compute time reported in [2].

References

[1] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. In *NVIDIA Tech. Report NVR-2008-004*, pages 1–32, 2008.

[2] H. Blair, J. Cong, and D. Wu. Fpga simulation engine for customized construction of neural microcircuits. In *Computer-Aided Design (ICCAD), 2013 IEEE/ACM Int. Conf. on*, pages 607–614, Nov 2013.

[3] R. F. Boisvert, R. Pozo, K. A. Remington, R. F. Barrett, and J. Dongarra. Matrix market: a web resource for test matrix collections. 1996.

[4] D. Chavarria-Miranda, M. Halappanavar, and A. Kalyanaraman. Scaling graph community detection on the tilera many-core architecture. In *2014 21st International Conference on High Performance Computing (HiPC)*, pages 1–11, Dec 2014.

[5] J. D. Davis and E. S. Chung. Spmv: A memory-bound application on the gpu stuck between a rock and a hard place. Technical Report MSR-TR-2012-95, September 2012.

[6] B. D. de Dinechin, D. van Amstel, M. Poulhiès, and G. Lager. Time-critical computing on a single-chip massively parallel processor. In *Proceedings of the Conference on Design, Automation & Test in Europe*, DATE '14, pages 97:1–97:6, 3001 Leuven, Belgium, Belgium, 2014. European Design and Automation Association.

[7] M. Delorimier and A. DeHon. Floating-point sparse matrix-vector multiply for FPGAs. *2005 ACM/SIGDA 13th*, page 75, 2005.

[8] J. Fowers, K. Ovtcharov, K. Strauss, E. S. Chung, and G. Stitt. A high memory bandwidth fpga accelerator for sparse matrix-vector multiplication. In *Proceedings of the 2014 IEEE 22Nd International Symposium on Field-Programmable Custom Computing Machines*, FCCM '14, pages 36–43, Washington, DC, USA, 2014. IEEE Computer Society.

[9] M. Geveler, T. Stefan, and R. Dirk. Realization of a low energy HPC platform powered by renewables - a case study: Technical, numerical and implementation aspects. 2015.

[10] A. Gharaibeh, L. Beltrao Costa, E. Santos-Neto, and M. Ripeanu. On graphs, gpus, and blind dating: A workload to processor matchmaking quest. In *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 851–862, May 2013.

[11] L. Gwennap. Adapteva: More flops, less watts. *Microprocessor Report*, 6(13):11–02, 2011.

[12] S. J. Hollis and S. Kerrison. Swallow: Building an energy-transparent many-core embedded real-time system. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 73–78, March 2016.

[13] E. M. Izhikevich. Which model to use for cortical spiking neurons? *IEEE transactions on neural networks*, 15(5):1063–1070, 2004.

[14] N. Kapre and A. Dehon. An noc traffic compiler for efficient fpga implementation of sparse graph-oriented workloads. *International Journal of Reconfigurable Computing*, 2011, 2011.

[15] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[16] T. Majumder, P. P. Pande, and A. Kalyanaraman. On-chip network-enabled many-core architectures for computational biology applications. In *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 259–264, March 2015.

[17] A. Olofsson, T. Nordström, and Zain-ul-Abdin. Kickstarting high-performance energy-efficient manycore architectures with epiphany. *CoRR*, abs/1412.5538, 2014.

[18] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: bringing order to the web. 1999.

[19] P. S. Paolucci, R. Ammendola, A. Biagioni, O. Frezza, F. L. Cicero, A. Lonardo, M. Martinelli, E. Pastorelli, F. Simula, and P. Vicini. Power, energy and speed of embedded and server multi-cores applied to distributed simulation of spiking neural networks: ARM in NVIDIA tegra vs intel xeon quad-cores. *CoRR*, abs/1505.03015, 2015.

[20] Y. Saad. Sparskit: a basic tool kit for sparse matrix computations, 1994.

[21] L. G. Valiant. Why BSP computers? [bulk-synchronous parallel computers]. In *Parallel Processing Symposium, 1993., Proceedings of Seventh International*, pages 2–5, 1993.

[22] G. Weisz and J. C. Hoe. Coram++: Supporting data-structure-specific memory interfaces for fpga computing. In *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, Sept 2015.