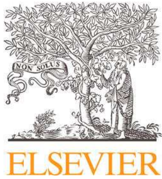
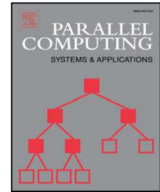
Parallel Computing 87 (2019) 60–69



Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco



A microbenchmark characterization of the Emu chick☆

Jeffrey S. Young[b,*], Eric Hein[c], Srinivas Eswar[a], Patrick Lavin[a], Jiajia Li[d], Jason Riedy[a], Richard Vuduc[a], Tom Conte[b]

[a] School of Computational Science and Engineering, Georgia Institute of Technology, North Avenue, Atlanta, GA 30332, United States
[b] School of Computer Science, Georgia Institute of Technology, North Avenue, Atlanta, GA 30332, United States
[c] Emu Technology, 270 West 39th Street, 13th Floor, New York, NY 10018, United States
[d] Pacific Northwest National Laboratory, 902 Battelle Blvd, Richland, WA 99354, United States

ARTICLE INFO

Article history:
Received 25 August 2018
Revised 2 April 2019
Accepted 28 April 2019
Available online 24 May 2019

Keywords:
Emerging architectures
Benchmark characterization
Distributed computing
Sparse algorithms

ABSTRACT

The Emu Chick is a prototype system designed around the concept of migratory memory-side processing. Rather than transferring large amounts of data across power-hungry, high-latency interconnects, the Emu Chick moves lightweight thread contexts to near-memory cores before the beginning of each memory read. The current prototype hardware uses FPGAs to implement cache-less "Gossamer" cores for computational work and rely on a typical stationary core (PowerPC) to run basic operating system functions and migrate threads between nodes. In this multi-node characterization of the Emu Chick, we extend an earlier single-node investigation [1] of the memory bandwidth characteristics of the system through benchmarks like STREAM, pointer chasing, and sparse matrix-vector multiplication. We compare the Emu Chick hardware to architectural simulation and an Intel Xeon-based platform. Our results demonstrate that for many basic operations the Emu Chick can use available memory bandwidth more efficiently than a more traditional, cache-based architecture although bandwidth usage suffers for computationally intensive workloads like SpMV. Moreover, the Emu Chick provides stable, predictable performance with up to 65% of the peak bandwidth utilization on a random-access pointer chasing benchmark with weak locality.

© 2019 Elsevier B.V. All rights reserved.

1. Introduction

Analysis of data represented as graphs, sparse tensors, and other non-regular structures poses many challenges for traditional computer architectures because the data locality of these applications typically occurs in small bursts. While individual data elements may have multiple associated attributes nearby (e.g. neighbors, weights, semantic attributes, and timestamps for streaming graph edges), analysis algorithms tend to access these small chunks in a more random fashion. Limited spatial locality in traditional analysis kernels leads to underutilizing cache lines, confounding prefetch engines, and thus reducing overall effective memory bandwidth. Furthermore, common analysis kernels may exhibit dynamic parallelism and create many data-dependent memory references. These references can stall architectures that cannot maintain enough contexts and requests in flight. Consequently, today's "big data" platforms frequently are outperformed by a single thread accessing a large SSD [2].

This state of affairs motivates novel architectures like the Emu migratory thread system [3], the subject of this study. The Emu is a cache-less system built around "nodelets" that each execute lightweight threads. These threads migrate to data rather than moving data through a traditional cache hierarchy.

This paper expands on the first independent characterization of the Emu Chick prototype [1] by exploring multiple distributed nodes that consist of those nodelets (see Section 2). Our study uses microbenchmarks and small kernels—namely, STREAM, pointer chasing, and sparse matrix-vector multiplication (SpMV)—as proxies that reflect some of the key characteristics of our motivating computations, which come from sparse and irregular applications [4,5]. Indeed, one larger goal of our work beyond this paper is to develop a performance-portable, Emu-compatible API for Georgia Tech's STINGER open-source streaming graph framework [4] and ParTI [6] tensor decomposition algorithms (e.g. CP and Tucker decomposition). Mapping such applications to the Emu architecture is difficult because the thread migration makes programming around *data's location* critical to reducing migrations.

☆ Declaration of Competing Interest: The authors declare that they do not have any financial or nonfinancial conflict of interests.

* Corresponding author.
E-mail addresses: jyoung9@gatech.edu (J.S. Young), ehein6@gatech.edu (E. Hein), seswar3@gatech.edu (S. Eswar), plavin3@gatech.edu (P. Lavin), jiajiali@gatech.edu (J. Li), jason.riedy@gatech.edu (J. Riedy), richie@gatech.edu (R. Vuduc), conte@gatech.edu (T. Conte).

https://doi.org/10.1016/j.parco.2019.04.012
0167-8191/© 2019 Elsevier B.V. All rights reserved.

This study's specific demonstrations include

- a detailed characterization of the Emu Chick hardware using custom Cilk kernels derived from optimized OpenMP kernels;
- an analysis of memory bandwidth on the Chick system and comparison to a more traditional cache-based architecture with Emu results tested on 64 nodelets across eight nodes
- a discussion of memory allocation, data layout, and "smart" thread migration on the Emu architecture with respect to SpMV kernels;
- an investigation and validation of the Emu architectural simulator for projecting larger configurations' performance.

The main high-level finding is that an Emu-style architecture can more efficiently utilize available memory bandwidth while reducing the variability of that bandwidth to the memory access pattern. However, achieving such results still requires careful consideration of the interplay between data layout and its effect on thread migration. Additionally, our current Chick prototype is still compute-bound for some algorithms like SpMV which hurts its usage of available memory bandwidth when compared to a traditional CPU-based system.

2. The Emu architecture

The Emu architecture focuses on improved random-access bandwidth scalability by migrating lightweight, *Gossamer* threads or "threadlets" to data and emphasizing fine-grained memory access. A general Emu system consists of the following processing elements, as illustrated in Fig. 1:

- A common *stationary* processor runs the operating system (e.g. Linux) and manages storage and network devices.
- *Nodelets* combine narrowly banked memory (NCDIMMs) with several highly multi-threaded, cache-less *Gossamer* cores to provide a memory-centric environment for migrating threads.

These elements are combined into nodes that are connected by a RapidIO fabric. The current generation of Emu systems include one stationary processor for each of the eight nodelets contained within a node. System-level storage is provided by SSDs. We talk more specifically about some of the prototype limitations of our Emu Chick in Section 3. A more detailed description of the Emu architecture is available elsewhere [3].

For programmers, the Gossamer cores are transparent accelerators. The compiler infrastructure compiles the parallelized code for the Gossamer ISA, and the runtime infrastructure launches threads on the nodelets. Currently, one programs the Emu platform using Cilk [7], providing a path to running on the Emu for simple OpenMP programs whose translations to Cilk are straightforward. The current compiler supports the expression of task or fork-join parallelism through Cilk's `cilk_spawn` and `cilk_sync` constructs, with a future Cilk Plus software release in progress that would include `cilk_for` (the nearly direct analogue of OpenMP's `parallel for`). Many existing C and C++ OpenMP codes can translate almost directly to Cilk Plus.

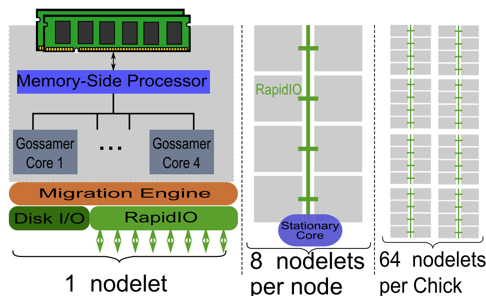


Fig. 1. Emu architecture: The system consists of *stationary* processors for running the operating system and up to four *Gossamer* processors per nodelet tightly coupled to memory. The cache-less Gossamer processing cores are multi-threaded to both source sufficient memory references and also provide sufficient work with many outstanding references. The coupled memory's narrow interface ensures high utilization for accesses smaller than typical cache lines.

A launched Gossamer thread only performs local reads. Any remote read triggers a migration, which will transfer the context of the reading thread to a processor local to the memory channel containing the data. Experience on high-latency thread migration systems like Charm++ identifies migration overhead as a critical factor even in highly regular scientific codes [8]. The Emu system keeps thread migration overhead to a minimum by limiting the size of a thread context, implementing the transfer efficiently in hardware, and integrating migration throughout the architecture. In particular, a Gossamer thread consists of 16 general-purpose registers, a program counter, a stack counter, and status information, for a total size of less than 200 bytes. The compiled executable is replicated across the cores to ensure that instruction access always is local. Limiting thread context size also reduces the cost of spawning new threads for dynamic data analysis workloads. Any operating system requests are forwarded to the stationary control processors through the service queue.

The highly multi-threaded Gossamer cores, which are reading only local memory, do not need caches nor, therefore, cache coherency traffic. Additionally, "memory-side processors" provide atomic read or write operations that can be used to access small amounts of data without triggering unnecessary thread migrations. A node's memory size is relatively large (64 GiB) but with multiple, narrow memory channels (8 channels with 8 bit interfaces), in order to extract weak spatial locality from data analysis kernels while maintaining low-latency read and write operations. The high degree of multi-threading also helps to cover the migration latency of the many threadlets. The Emu architecture is designed from the ground up to support high bandwidth utilization and efficiency for demanding data analysis workloads.

3. Experimental setup

3.1. Emu chick prototype

The Emu Chick prototype is still in active development. The current hardware iteration uses an Arria 10 FPGA on each node card to implement the Gossamer cores, the migration engine, and the stationary cores. Several aspects of the system are scaled down in the current prototype with respect to the next-generation system which will use larger and faster FPGAs to implement computation and thread migration. The current Emu Chick prototype has the following features and limitations:

- Our system has one Gossamer Core (GC) per nodelet with a concurrent max of 64 threadlets. The next-generation system will have four GC's per nodelet, supporting 256 threadlets per nodelet.
- A full Chick system has 64 nodelets across eight nodes, implementing a distributed Partitioned Global Address System (PGAS) architecture that is connected by the RapidIO network.
- Our GC's are clocked at 175 MHz currently (up from 150 MHz in [1]) rather than the planned 300 MHz for later-generation Emu development systems.
- The Emu's DDR4 DRAM modules are clocked at 1600MHz rather than the full 2133 MHz.
- The current Emu software version provides support for C++ but does not yet include functionality to translate Cilk Plus

features like `cilk_for` or Cilk `reducers` [9]. All benchmarks currently use `cilk_spawn` directly, which also allows more control over spawning strategies.

All experiments are run using Emu's 18.09 compiler and simulator toolchain, and the Emu Chick system is running NCDIMM firmware version 2.5.1, system controller software version 3.1.0, and each stationary core is running the 2.2.3 version of software.

3.2. Emu simulator

Emu provides a timing simulator implemented using SystemC, and this simulator can be used to test and evaluate software before running on the hardware. Previous characterization experiments in [1] employed a configuration of the simulator to match the characteristics of a single node (8 nodelets) of the Chick hardware for validation and to do basic projections to a stable, 64-node Chick system. We have not repeated these experiments as the projections in earlier work have been superseded by real-time results on the 64-node Chick system and many of the most interesting characterizations cannot be run on the timing simulator.

3.3. Common CPU-focused comparison platform

In order to make an initial comparison of the Emu's memory bandwidth characteristics with commodity hardware, each benchmark is also run on an four-socket Intel Xeon E7-4850 v3 (Haswell) machine with 2 TiB of DDR4 (referred to as `Haswell Xeon` in associated results). The CPUs on the Haswell server are each clocked at 2.20 GHz and each have a 35 MiB L3 cache, while the memory is clocked at 1333 MHz (although it is rated for 2133 MHz, the risers used to increase density decrease the frequency). Each socket has a peak theoretical bandwidth of 42.7 GB/s because of the underclocked memory.

For each benchmark, Emu-specific intrinsics (e.g. localized mallocs) are swapped out for their x86 equivalents, and the benchmarks are compiled with GCC 5.5.0. The Cilk keywords are left unchanged, allowing GCC's Cilk runtime to implement the parallel functionality. Intel's MKL library (version 20180001) is used for some of the SpMV comparisons made in Section 4.3. STREAM is run with default OpenMP settings including OMP_PROC_BIND=false and OMP_SCHEDULE=static.

3.4. Metrics for comparing the Emu prototype with cache-based hardware

The architectural design choices that enable the Emu computational model (migrate threads instead of data, narrow memory channels, limited thread context) and the base platforms for the prototype (FPGAs with lower clock frequencies) make it difficult to accurately compare the Emu and CPU- or GPU-focused systems in terms of execution or runtime.

Additionally, the Emu platform uses Narrow-Channel DRAM (NCDRAM) which reduces the width of the DRAM bus to 8 bits. Otherwise, the memory uses standard DDR4 chips. An 8-byte word can be transferred in a single burst. The smaller bus means that each channel of NCDRAM has only 2 GB/s of bandwidth, but the system makes up for this by having many more independent channels. Because of this, it can sustain more simultaneous fine-grained accesses than a traditional system with fewer channels and the same peak memory bandwidth specification.

Due to difficulties in comparing differently clocked architectures with different memory controller configurations, we focus our initial characterization not on runtime but on memory bandwidth (MB/s) and effective memory bandwidth utilization (% of measured peak memory bandwidth). In a CPU-focused system, this might be analogous to effective cache line utilization while in the Emu it correlates more closely to how much bandwidth can be achieved with respective to other system overheads, such as thread migration and queuing delays.

3.5. Benchmarks

As discussed in Section 3.1, the Emu Chick toolchain currently lacks support for `cilk_for` and Cilk reducers. However, we present several benchmarks that use Cilk semantics to characterize the performance of the system, specifically focusing on kernels that expose the memory bandwidth characteristics of the system and test important kernels like SpMV that are key for applications like sparse tensor decomposition. For each benchmark result, we present the average memory bandwidth (usually expressed as megabytes per second) over ten trials.

STREAM The STREAM benchmark [10] has been ported and tuned for the Emu hardware to measure raw memory bandwidth. The ADD kernel computes the vector sum of two large arrays of 8-byte integers, storing the result in a third array. On the Emu Chick these arrays are striped across all the nodelets in the system.

This benchmark demonstrates that thread spawning is important for performance. Different thread spawning mechanisms achieve different memory bandwidths. Our tests control thread spawning and do not rely on `cilk_for`. The spawning methods follow trees that are briefly described as follows:

- **serial_spawn**: threads spawn locally on a single nodelet using a for loop,
- **recursive_spawn**: threads are spawned locally using recursive calls,
- **serial_remote_spawn**: threads are spawned on each nodelet, which in turn uses a for loop to spawn threads locally, and
- **recursive_remote_spawn**: threads are spawned recursively across all nodelets, and then each nodelet recursively spawns new threads locally.

Pointer Chasing. In this benchmark, each thread adds up all the elements in a linked list. Each element consists of an 8-byte payload and an 8-byte pointer to the next element. After the elements of this linked list are grouped into blocks, their ordering is randomized. This permutation may be applied to the ordering of the elements within each block (`intra_block_shuffle`), or the ordering of the blocks themselves (`block_shuffle`), or both (`full_block_shuffle`). The block size is also varied to emulate different levels of spatial locality that may arise in a workload. Fig. 2 explains the list initialization further.

The pointer chasing benchmark has three key properties by design.

- Data-dependent loads: Memory-level parallelism is severely limited since each thread must wait for one pointer dereference to complete before accessing the next pointer.
- Fine-grained accesses: Spatial locality is restricted since all accesses are at a 16B granularity. This is smaller than a 64B cache line on x86 platforms, and much smaller than a typical DRAM page size.
- Random access pattern: Since each block of memory is read exactly once in random order, caching and prefetching are mostly ineffective.

The pointer chasing benchmark simulates a worst-case memory fragmentation scenario that can arise in memory intensive workloads such as streaming graph analytics. When small list elements are dynamically allocated and deallocated from a shared

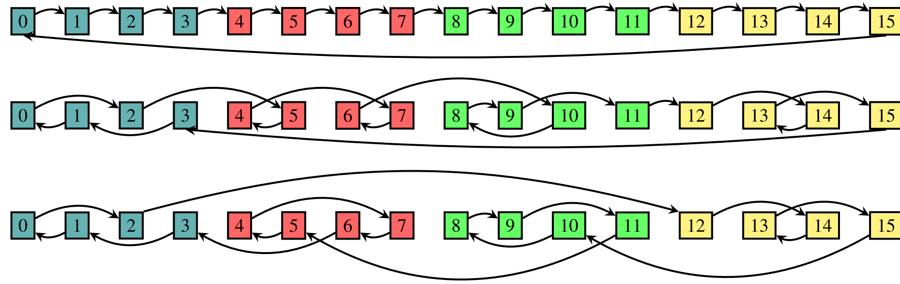


Fig. 2. (top) An ordered linked list, in which consecutive elements have sequential memory addresses, (middle) A linked list with a intra-block shuffle permutation applied to randomize the ordering of elements within a block. Note that all elements within a block are accessed before jumping to the next block. (bottom) A linked list with a full block shuffle permutation applied. Not only are the elements within a block shuffled, but the traversal order of the blocks themselves has also been randomized. Not shown is the full shuffle which is equivalent to a block shuffle followed by an intra-block shuffle.

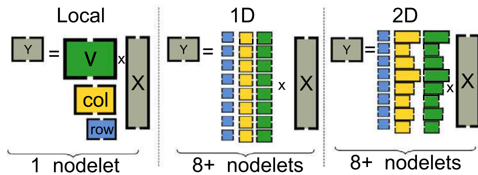


Fig. 3. Emu-specific layout for CSR-SpMV.

memory pool, the resulting data structure will exhibit all of these characteristics when it is traversed. The pointer chase benchmark otherwise is quite similar to the GUPS/RandomAccess benchmark [11], however GUPS lacks data-dependent loads, and pointer chase does not modify the list.

Sparse Matrix-Vector Multiplication (SpMV). In addition to being a fundamental kernel for graph analytics and sparse tensor decomposition applications, SpMV provides an opportunity to investigate data layout strategies on the Emu's global physical address space. Emu provides a "local" malloc (`mw_localmalloc`) similar to a traditional contiguous malloc as well as a "striped" malloc (`mw_malloc1dlong`, called 1D) that places data in a round-robin fashion across nodelets and a two-dimensional malloc (`mw_malloc2d`, called 2D), that distributes entire data structures across nodelets.

Fig. 3 demonstrates the three layouts that are tested with inputs in the popular Compressed Sparse Row (CSR) format. For an $I \times J$ matrix A with M non-zeros, a size-$(I+1)$, a size-M, and another size-M arrays are stored for row pointers, column indices, and nonzero values respectively. We use $y = Ax$ to illustrate an SpMV operation and use one node case as an example which can be extended to the multi-node case. In the local case, contiguous mallocs are used to place all data on a single nodelet, which include the output vector Y, the input vector X, and the input CSR matrix: `row`, `col`, and V. Only one nodelet's computing power is employed. 1D layout stripes the `row`, `col`, and V arrays of the input matrix across the nodelets using `mw_malloc1dlong` while X is replicated across all nodelets and Y is on nodelet 0. Due to the diverse sizes of `row`, `col`, and V and the round-robin pattern of `mw_malloc1dlong`, the nonzeros and column indices in the same row are distributed to different nodelets, which introduces frequent thread migration. For the 2D allocation, we use a two-stage allocation rather than Emu's 2D malloc function to partition V and `col` across multiple nodelets. First, the length of each row that is assigned to a nodelet is computed and then V and `col` are allocated in units of rows on each nodelet. We still employ the `mw_malloc1dlong` function but use it to allocate a variable length array. X is also replicated across nodelets as is the output, Y. Using this modified 2D layout, we can wipe out most thread migrations with the exception for some initial spawn migrations and reduction at the end of the computation. This is an ideal case to test the highest performance Emu can achieve in the SpMV benchmark.

4. Results

The updated characterization of the Emu Chick repeats the STREAM, pointer chasing, and SpMV experiments that were initially investigated in [1], but these experiments focus on further characterizing the entire Chick "multi-node" system that uses all 64 nodelets (across 8 nodes) within the Chick. Single node results are presented for specific experiments including SpMV layout and simulation, primarily due to limitations in the current firmware (i.e., certain configurations encounter hardware faults) and slowness of the Emu architectural simulator.

4.1. STREAM

Fig. 4 shows the results from running the STREAM benchmark on a single Emu nodelet. Performance scales up with thread count through 32 threads and then plateaus. Two methods of thread creation are tested here. In the `serial_spawn` strategy, a single thread uses a for loop to create each worker thread, while `recursive_spawn` uses a recursive spawn tree. There is not much difference between the two approaches, indicating that thread creation is not terribly expensive on the Emu platform.

In Fig. 5a, we extend the STREAM benchmark to run on eight nodelets (one node card) of the Emu Chick. Two new thread creation strategies are introduced here, `serial_remote _spawn` and `recursive_remote_spawn`. A remote spawn on Emu means that the thread is created on a remote nodelet, rather than

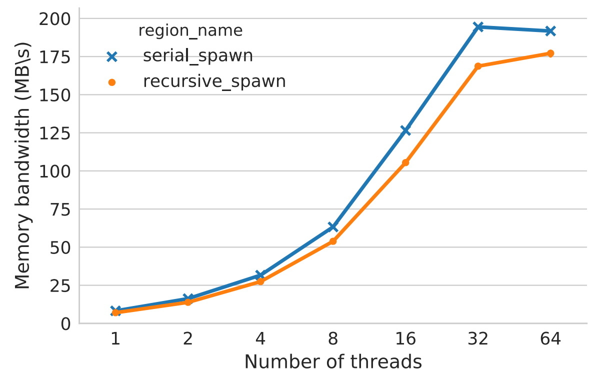
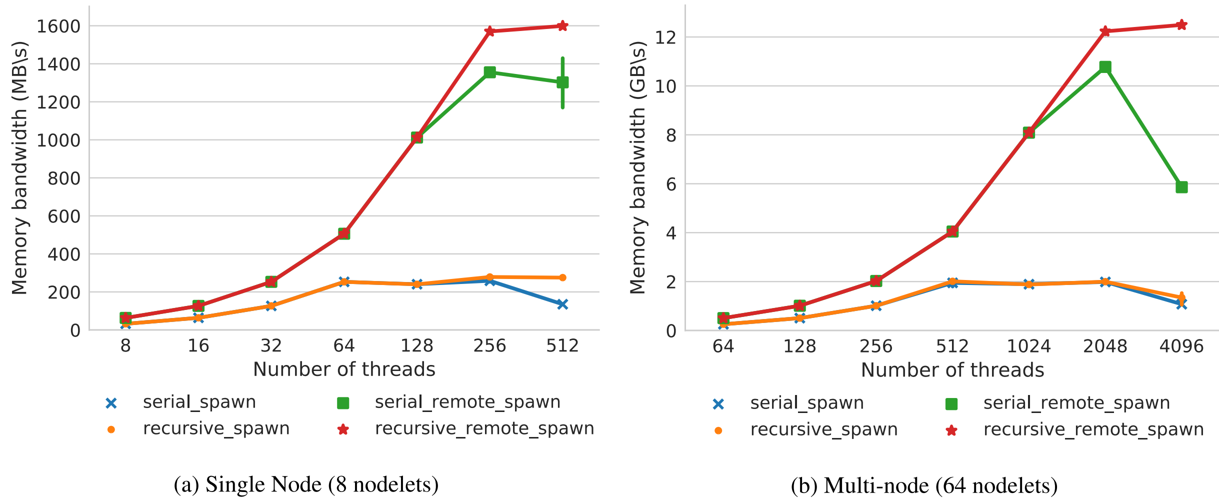


Fig. 4. Memory bandwidth achieved on a single nodelet of the Emu Chick. Threads are created using a serial loop or a recursive spawn tree.



(a) Single Node (8 nodelets)

(b) Multi-node (64 nodelets)

Fig. 5. Emu STREAM performance with different spawn strategies.

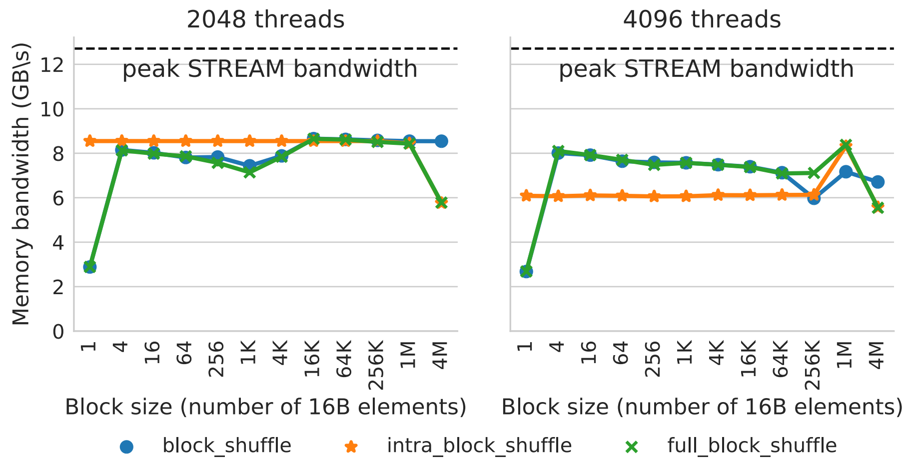


Fig. 6. Pointer chasing performance on the Emu Chick (8 nodes, 64 nodelets).

being created locally and allowed to migrate to the remote data. The "remote" thread creation strategies first create a thread on each nodelet (either one at a time or with a recursive spawn tree), and then perform a second level of spawning on the local nodelet, as in the single nodelet case. Fig. 5b extends the analysis to 64 nodelets and up to 4096 threads showing that recursive remote spawn continues to scale for large numbers of threads up to 12 GB/s across 8 nodes. Both sets of results show that remote spawns are essential to achieving maximum bandwidth on Emu.

In comparison to the Emu, the Xeon system (`Haswell`) achieves 100 GB/s on the STREAM benchmark (with an interleaved NUMA layout across four sockets) while the Emu Chick has a maximum STREAM bandwidth of 12 GB/s. The Emu bandwidth is currently limited by CPU speed and thread count rather than DDR bus transfer rates. However even with this prototype system we can observe improvements in other benchmarks where the memory access pattern is not as linear and predictable as it is with STREAM.

4.2. Pointer chasing

Figs. 6 and 7 compare the performance of the Emu Chick against our Haswell Xeon server system for the pointer chasing benchmark. These results reveal important characteristics of both systems and highlight the unique advantages of the Emu Chick.

Pointer chasing on the Xeon architecture performs poorly for several reasons. For small block sizes, the memory system bandwidth is used inefficiently. An entire 64-byte cache line must be transferred from memory, but only 16 bytes will be used. The best performance is achieved with a block size between 256 and 4096 elements. This corresponds to a memory chunk of about 8KiB, the size of one DRAM page. Regardless of the size of the access, an entire DRAM row must be activated for each element traversed. Adding more threads at this point increases the number of simultaneous row activations. As the block size grows beyond the size of a DRAM page, performance declines again.

Performance on Emu remains mostly flat regardless of block size. Emu's memory access granularity is 8 bytes, so it never transfers unused data in this benchmark. As long as a block fits within a single nodelet's local memory channel, there is no penalty for random access within the block. However, block size of 1 provides an interesting case; here Emu threads are likely to migrate on every access, and so performance is greatly reduced. But performance recovers when even as few as four elements are accessed between each migration.

Fig. 8 shows the normalized bandwidth usage (*i.e.*, effective bandwidth usage) for the Haswell and Emu systems. The performance of each system has been normalized to the peak measured bandwidth of the system (*i.e.*, the best result on the STREAM benchmark). In the pointer chasing benchmark, the Emu system is much better at using the available system bandwidth, using 65% of available system bandwidth in most cases and 25% in the worst case. The Haswell Xeon uses less than 50% of peak bandwidth

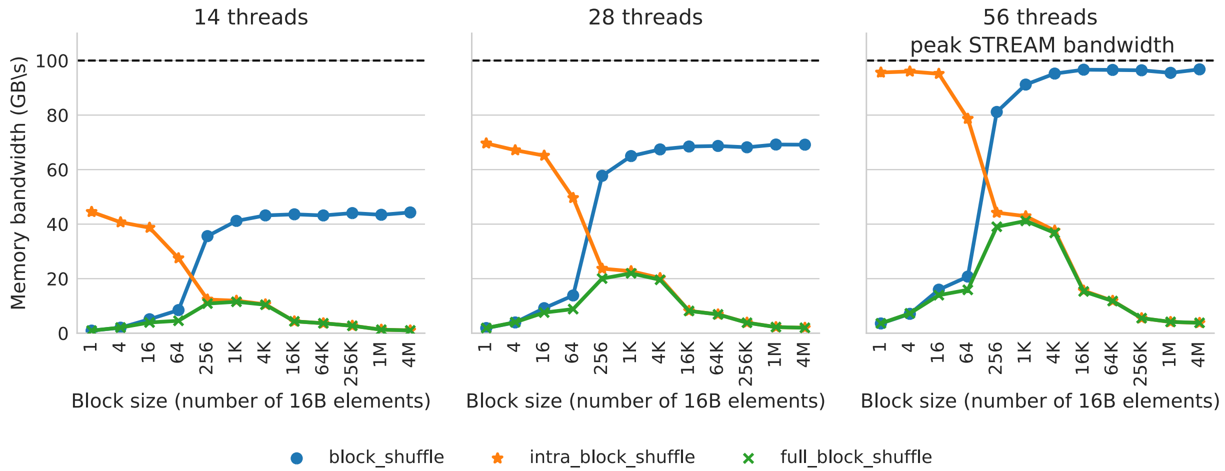


Fig. 7. Pointer chasing performance on Haswell Xeon.

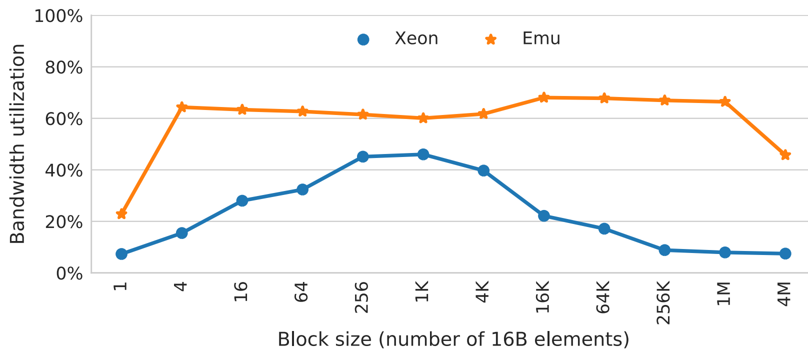


Fig. 8. Bandwidth utilization of pointer chasing, compared between Sandy Bridge Xeon and Emu (64 nodelets).

in most cases and less than 10% in the worst case, relying on multi-kilobyte levels of locality to efficiently transfer the data. These results bode well both for the targeted streaming graph and tensor decomposition applications which have pointer chasing behavior and rely on random accesses to compute SpMV and SpMM (sparse matrix-matrix product) operations, respectively.

4.3. Sparse matrix-vector multiplication

We use a synthetic Laplacian matrix as an input corresponding to a d-dimensional k-point stencil on a grid of length n in each dimension. $d = 2$ and $k = 5$ specify a sparse matrix generated from a 5-point, 2-D, $n \times n$ stencil. With the Laplacian size n, the matrix is $n^2 \times n^2$ with 5 diagonals. CPU tests are run on the Haswell Xeon-based system described in Section 3.3, using SpMV from Intel's Math Kernel Library (MKL) with `MKL_MAX_THREADS` set at 56 (the number of physical cores in the system as opposed to total threads). We include two Cilk SpMV kernels for comparison, labeled `cilk_for` and `cilk_spawn`, which are written with the respective Cilk primitives, compiled using GCC 5.5.0, and run with `CILK_NWORKERS` set to 56. Data is distributed across NUMA regions using `numactl --interleave=0-3`.

Fig. 9 shows the memory bandwidth achieved by SpMV on a single node (8 nodelets) of the Chick using each of the three data layout strategies: local, 1D, and 2D layouts. The local layout on the Emu suffers from a limited amount of thread parallelism while the 1D layout suffers from a large number of thread migrations, resulting in max bandwidths of close to 96.13 MB/s and 148.66 MB/s, respectively. Overall, the 2D memory layout provides the only scalable solution for SpMV, scaling up to 846.39 MB/s for n = 2300.

Fig. 10a shows results for experiments run on the Haswell Xeon machine with four different code configurations and two NUMA layouts, the default "current" policy (place all data in current node) and "interleaved" which interleaves data across four sockets at a page granularity. 1) *MKL* refers to an implementation of SpMV using Intel's MKL library, 2) *cilk_for* is a native Cilk implementation using cilk_for, 3) *cilk_spawn* is a native Cilk implementation using cilk_spawn, and 4) *emu* is an implementation of SpMV that is backported from the Emu-optimized version of the code. This last version of the code includes the 2D layout optimization described in Section 3.5 and evaluated in Fig. 9.

The Haswell Xeon results in Fig. 10a show that the MKL implementation of SpMV with the "current" NUMA layout achieves the highest bandwidth, getting close to 175 GB/s across four nodes. Meanwhile, *cilk_for* and *cilk_spawn* show similar scaling up to n = 200 with the interleaved versions of both implementations closely mirroring each other's performance from n = 500 to n = 8000. Finally, the Emu "backported" code only shows scalable performance with the NUMA interleaved setting, peaking in performance at n = 8000 and around 50 GB/s. While it is unclear at the moment why the MKL version scales so well with a non-interleaved data layout (we suspect it may have to do with first-touch layouts being amenable to relevant MKL data structures and computation), the performance of the Emu backported code seems to mesh well with our understanding of the Emu system as a distributed PGAS machine. That is, the optimal 2D layout of SpMV for the Emu keeps data accesses "local" on the Emu, but a normal x86 NUMA allocator does not stripe data like Emu's allocator does, meaning that most data accesses on the x86 emu "current" setup are remote NUMA accesses. Furthermore, x86 NUMA interleaved layouts have much larger granularity for striping (pages *versus*

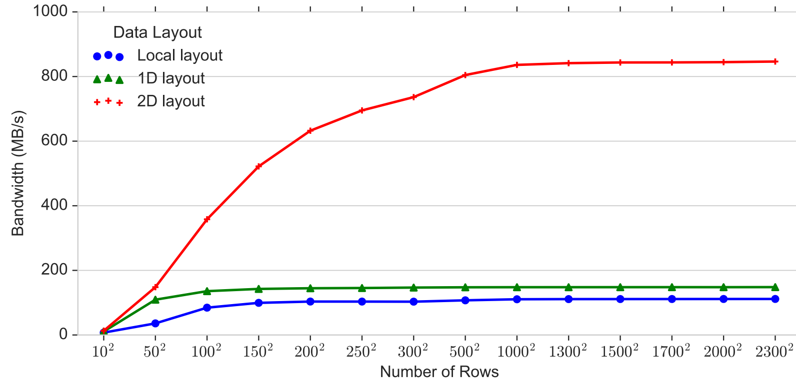
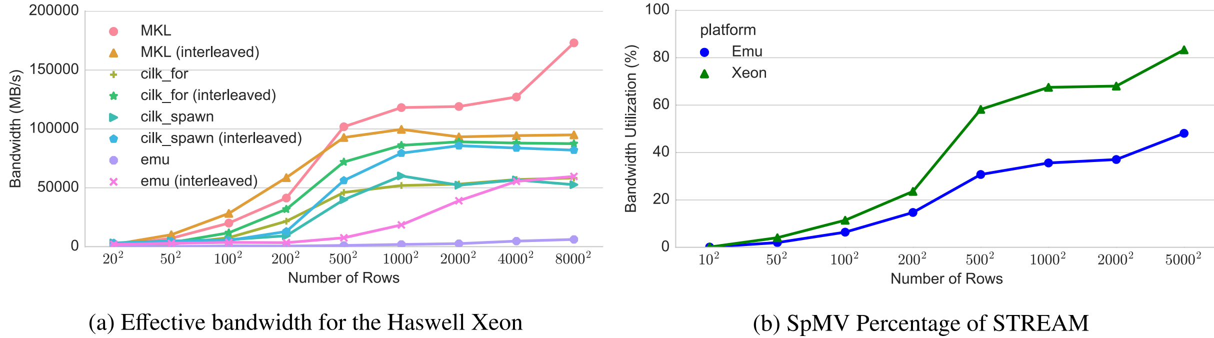


Fig. 9. Bandwidth utilization of Emu Single Node (8 nodelets) with different SpMV Data Layouts.



(a) Effective bandwidth for the Haswell Xeon

(b) SpMV Percentage of STREAM

Fig. 10. SpMV Emu Comparisons with Haswell Xeon.

elements in an array on the Emu), which likely also penalizes the "emu (interleaved)" implementation.

Following the Haswell Xeon results, we compare the total percentage of STREAM bandwidth that is achieved for SpMV on the Emu *versus* on the Haswell machine in Fig. 10b. The Emu results use the peak multi-node STREAM bandwidth of 12 GB/s and are compared to the Haswell STREAM peak *without NUMA interleaving*, which peaks at 175 GB/s. In the case of the Haswell results, the best case SpMV (MKL non-interleaved from Fig. 10)a is used as the comparison point. As opposed to the pointer chasing results in Fig. 8, we see that both systems scale in terms of bandwidth utilization as the amount of synthetic data is increased with the Emu peaking at about 50% of peak STREAM bandwidth *versus* the Haswell system's 80% of peak STREAM bandwidth. SpMV can achieve between 50% and 60% of the peak STREAM bandwidth, but with its address calculations and the multiply-and-accumulate, the 175 MHz Gossamer Cores cannot generate loads quickly enough to saturate the available memory bandwidth. As we discuss in the following section, a primary limitation of the current Chick prototype is that even simple benchmarks are compute-bound.

5. Discussion

This characterization raises important topics for programming memory-centric architectures like the Emu Chick and also for building realistic comparisons between prototype novel architectures and existing architectures.

5.1. Achievable bandwidth for the current Emu chick prototype

Using STREAM, pointer chasing, and SpMV, we provide an initial look at the performance of the Emu Chick for these fundamental operations. However, these results also point to a fundamental issue with the initial Emu Chick prototype. The design of the Chick

```
noinline void
recursive_remote_spawn_level2_worker(long begin,
long end, long * a, long * b, long * c)
{
    for (long i = begin; i < end; ++i) {
        c[i] = a[i] + b[i];
    }
}
```

Listing 1. STREAM ADD worker function.

with 1 Gossamer Core per nodelet and 8 nodelets per node leads to a situation where most codes are currently compute bound due to limited GCs and low frequencies of the GCs on the FPGA prototype.

To demonstrate, we look at the inner loop of a STREAM ADD operation in Listing 1 and analyze the relevant assembly code generated by Emu's gossamer64-objdump tool. As Listing 2 shows, each inner loop of the STREAM ADD kernel performs two loads, one add operation, and one store operation for a total of 3 memory operations out of a total of 21 instructions.

Using this information from the assembly code, we can determine the peak achievable bandwidth by one GC running at the current frequency of 175 MHz. As Eq. (1) shows, one GC can achieve up to 200 MB/s for the STREAM ADD kernel, which we have used as our "peak" achievable bandwidth for comparison with other microbenchmarks.

$$175MHz \Rightarrow \frac{\text{175M cycles}}{second} \times \frac{\text{1 instruction}}{cycle} \times \frac{\text{3 mem ops}}{\text{21 instructions}} \times \frac{\text{8 Bytes}}{\text{1 mem op}} = 200MB/s \quad (1)$$

Investigating further in Table 1, we see that the measured results for STREAM ADD for single-node and multi-node execution are very close to the 200 MB/s peak value for STREAM ADD. However, if we look at the ideal case where all instructions are mem-

```
etd     2   // D = E2
sllc    3   // D <<= 3
dpeta   6   // A = D + E6
lde     7   // LOAD: E7 = *A
etd     2   // D = E2
sllc    3   // D <<= 3
dpeta   5   // A = D + E5
lde     8   // LOAD: E8 = *A
etd     2   // D = E2
sllc    3   // D <<= 3
dpeta   4   // A = D + E4
etd     8   // D = E8
adde    7   // ADD: D += E7
wrd         // STORE: *A = D
eta     2   // A = E2
aaimb   1   // A += 1
ate     2   // E2 = A
etd     3   // D = E3
xore    2   // D ^= E2
bdz
jmp
```

Listing 2. STREAM ADD Assembly.

Table 1
STREAM and Memory Bandwidths (BW in MB/s).

Operation	Nodelets	Scale	Threads	BW
ADD (Measured)	8	30	512	1,600
ADD (Measured)	64	31	4096	12,790
Ideal - all *ld* ops	1			1400
NCDIMM	8			12,800
NCDIMM	64			102,400

ory operations, the peak value for a single GC would be closer to 1400 MB/s. Moreover, looking at the Chick's memory system design, we see that the measured results for STREAM ADD are 8x slower than the NCDIMM's theoretical peak achievable bandwidth.

This analysis points to one conclusion - the current Emu Chick prototype is compute-bound for all microbenchmarks due to a low number of GCs and by low frequencies of the GCs, both of which are restricted by limited FPGA area and speed grades in the Arria 10 host device. We estimate that 8 GCs at the same speed of 175 MHz per nodelet or NCDIMM channel would be needed to move from a regime where the Chick is compute-bound to one where applications are memory-bound.

5.2. Caveats for programming the Emu chick

While Cilk provides an easy entry point for programming microbenchmarks for the Chick, our initial characterization has demonstrated some pitfalls for obtaining good performance on the Chick prototype. Primarily, the programmer must consciously design algorithms that optimize data layouts across nodes and that limit load imbalance by limiting thread migration. While CPU-based systems typically are optimized using techniques like cache-blocking, the distributed Partitioned Global Address Space (PGAS) nature of the Chick system requires that the programmer explicitly think about data placement in a different fashion. The Emu Chick has an existing profiler to inspect postmortem where threads end have migrated to, but detailed profiling and inspection of a program's execution requires the use of a simulation-based profiling tool.

The results from SpMV demonstrate that data layout can have an impact on performance on the Emu, application performance also depends on where threads are spawned and how many migrations occur between nodes and nodelets. In the initial development of our benchmarks, we debated explicitly minimizing thread movement and keeping computation local to a specific node to limit load imbalance on the existing compute resources for each nodelet. However, this strategy both goes against the "lightweight, migrating threadlets" model of computation with the Emu, and it is hard to implement in practice.

For this reason, we have settled on a strategy of "smart thread migration" for future benchmarking and application development with the Emu system. In short, this means 1) using "smart" thread spawn techniques like the two-level recursive remote spawn as in Section 4.1, 2) using replicated allocations for commonly used inputs like the vector X in the SpMV benchmark, and 3) picking the appropriate layout strategy for the application. In this last case, it is likely that good application performance will be most easily achieved through proper data layouts like with CSR SpMV's striped allocation across nodelets and per-nodelet secondary allocation for different-length rows. In this sense, we have created our own custom 2D allocator for SpMV, but we expect that higher-level memory allocation constructs will eventually be supported to help use the Emu's novel global address space layout.

5.3. Performance models and comparisons to existing architectures

One of the challenges in evaluating a drastically different architecture like the Emu is performing a realistic comparison between a prototype architecture and existing platforms using CPUs or other mainstream accelerators. Many aspects of the prototype Emu Chick present challenges. The Chick is a cacheless architecture and uses thread migration and atomic operations to avoid buffering large chunks of data. Even when compared with accelerators like GPUs, the low-latency access of the Chick, different memory clock speeds and data widths, and the lack of shared memory or caches provide a challenge for modeling how much more "efficient" the Chick is in terms of memory bandwidth. As shown in Section 4, different STREAM numbers for x86 systems based on NUMA interleaving settings also complicates this comparison. Additionally, the Chick is a full-scale prototype built using FPGA devices, which are useful for their flexibility and customization capabilities but naturally are slower than a traditional, hardwired ASIC. Firmware upgrades to the Chick prototype can also affect application performance dramatically by changing the gossamer cores' maximum frequency and by adding new functionality.

These comparison challenges are common not only to the Emu Chick but also to other new, experimental hardware like neuromorphic and quantum computing platforms. We may need to define additional metrics to supplement traditional characterization metrics like performance (FLOPS), memory bandwidth balance (FLOPS/B), and power efficiency (FLOPS/W). While we do not yet have enough application experience with the Emu Chick to fully define new metrics, we propose that there may be promise in focusing on comparison metrics that highlight the differences listed above. For example, a cache-less system like the Emu Chick may not actually move data physically across the system, but a comparable metric to a traditional CPU-based system might be some combination of network traffic (*i.e.*, threads migrated measured using context size and time, or B/s) and cache misses avoided (B/s). We plan to investigate how to better model and define these types of differences in future work to effectively quantify not just the high-level application benefits of novel architectures like the Chick but also the fundamental qualities that help define which applications are the best fit for these new architectures.

6. Related work

Advances in memory and integration technologies provide opportunities for profitably moving computation closer to data [12]. Some proposed architectures return to the older processor-in-memory (PIM) and "intelligent RAM" [13] ideas. Simulations of architectures focusing on near-data processing [14] including in-memory [15] and near-memory [16] show great promise for

increasing performance while also drastically reducing energy usage. Other than our previous study [1], and related work on characterizing the Emu by other research groups [17,18] few of these architectures have been implemented in hardware, even FPGAs, limiting the data scales on which applications can be evaluated.

Other hardware architectures have tackled massive-scale data analysis to differing degrees of success. The Tera MTA/Cray XMT [19,20] could provide high bandwidth utilization by tolerating long memory latencies in applications that could produce enough threads. In the XMT all memory interactions were remote incurred the full network latency. The Chick instead moves threads to memory on reads, assuming there will be a cluster of reads for nearby data. The Chick processor needs to tolerate less latency and need not keep as many threads in flight. Also, unlike the XMT, the Chick runs the operating system on the stationary processors, currently PowerPC, so the Chick processors need not deal with I/O interrupts and highly sequential OS code. Similarly to the XMT, programming the Chick requires language and library extensions. Future work with performance portability frameworks like Kokkos [21] will explore how much must be exposed to programmers. Another approach is to push memory-centric aspects to an accelerator like Sparc M7's data analytics accelerator [22] for database operations or Graphicionado [23] for graph analysis.

Moving computation to data via software has had a successful history in supercomputers and clusters via Charm++ [8], which manages dynamic load balancing on distributed memory systems by migrating the computational objects. Previously data analysis systems like Hadoop had moved computation to data when the network was a data bottleneck, but that no longer appears to be useful [24].

Finally, algorithms research related to SpMV could prove beneficial to future implementations for Emu-like architectures. New state-of-the-art SpMV formats and algorithms such as SparseX, which uses the Compressed Sparse eXtended (CSX) format for storing matrices [25] provide an alternative data structure and data layout that can be used to improve the performance of SpMV-based operations on the Emu. Related to our characterization, other researchers have investigated techniques [26] to implement SpMV with a focus on creating a load-balanced implementation using BFS and the METIS graph partitioner to place pieces of a graph on different nodelets. Note that this load balancing reduces thread migrations and hotspots but may require a large amount of initial preprocessing.

Other recent work has also looked to extend from low-level characterizations like those presented here by providing initial Emu-focused implementations of Breadth-First Search [17], Jaccard index computation [27], bitonic sort, [28] and compiler optimizations like loop fusion, edge flipping, and remote updates to reduce migrations [29].

7. Conclusion

Our microbenchmark evaluation of the Emu Chick demonstrates some of the limitations of the existing prototype system as well as some potential benefits for massive data analytics applications like streaming graph analytics and sparse tensor decomposition. We demonstrate multi-nodelet (64 nodelets across 8 nodes) performance for a variety of benchmarks including STREAM, pointer chasing, and SpMV. Initial results demonstrate relatively low overall bandwidth for the Emu system with a peak of 12 GB/s STREAM bandwidth for the current Chick prototype (compared to 80+ GB/s on a Haswell CPU server socket). However, we also show that algorithms implemented on the Emu can achieve a high percentage of effective memory bandwidth even in a worst-case access scenario like pointer chasing. The pointer chasing benchmark in Section 4.2 achieves a stable 60–65% bandwidth utilization across a wide range of locality parameters. These pointer chasing results and data layout studies show how random accesses with SpMV can be improved and while performance of SpMV does not quite match a well-optimized x86 implementation, these optimizations can provide a template for future benchmarking and application development and show how application memory layouts and "smart" thread migration can be used to maximize performance on the Emu system.

Acknowledgments

This work was supported in parts by NSF Grant ACI-1339745 (XScala), NSF Grant OAC-1710371 (SuperSTARLU), an IARPA contract, and the Defense Advanced Research Projects Agency (DARPA) under agreement #HR0011-13-2-0001. Any opinions, findings, conclusions, or recommendations in this paper are solely those of the authors and do not necessarily reflect the position or the policy of the sponsors.

The authors also gratefully acknowledge support by the Laboratory Directed Research and Development program at Sandia National Laboratories, a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Finally, thanks to the Emu Technology team for their continued support and debugging assistance with the Emu Chick prototype and to the many reviewers with their helpful suggestions.

References

[1] E. Hein, T. Conte, J.S. Young, S. Eswar, J. Li, P. Lavin, R. Vuduc, J. Riedy, An initial characterization of the Emu Chick, in: The Eighth International Workshop on Accelerators and Hybrid Exascale Systems (AsHES), 2018, pp. 579–588, doi:10.1109/IPDPSW.2018.00097.

[2] F. McSherry, M. Isard, D.G. Murray, Scalability! but at what COST? 15th Workshop on Hot Topics in Operating Systems (HotOS XV), USENIX Association, Kartause Ittingen, Switzerland, 2015.

[3] T. Dysart, P. Kogge, M. Deneroff, E. Bovell, P. Briggs, J. Brockman, K. Jacobsen, Y. Juan, S. Kuntz, R. Lethin, Highly scalable near memory processing with migrating threads on the Emu system architecture, in: Irregular Applications: Architecture and Algorithms (IA3), Workshop on, IEEE, 2016, pp. 2–9.

[4] D. Ediger, R. McColl, J. Riedy, D.A. Bader, STINGER: high performance data structure for streaming graphs, in: The IEEE High Performance Extreme Computing Conference (HPEC), Waltham, MA, 2012, doi:10.1109/HPEC.2012.6408680.

[5] J. Li, Y. Ma, C. Yan, R. Vuduc, Optimizing sparse tensor times matrix on multi-core and many-core architectures, in: 2016 6th Workshop on Irregular Applications: Architecture and Algorithms (IA3), 2016, pp. 26–33, doi:10.1109/IA3.2016.010.

[6] ParTI, ParTI Github, 2018, (online). https://github.com/hpcgarage/ParTI.

[7] C.E. Leiserson, Programming irregular parallel applications in Cilk, in: International Symposium on Solving Irregularly Structured Problems in Parallel, Springer, 1997, pp. 61–71.

[8] B. Acun, A. Gupta, N. Jain, A. Langer, H. Menon, E. Mikida, X. Ni, M. Robson, Y. Sun, E. Totoni, L. Wesolowski, L. Kale, Parallel programming with migratable objects: charm++ in practice, in: SC14: International Conference for High Performance Computing, Networking, Storage and Analysis, 2014, pp. 647–658, doi:10.1109/SC.2014.58.

[9] M. Frigo, P. Halpern, C.E. Leiserson, S. Lewin-Berlin, Reducers and other Cilk++ hyperobjects, in: Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures, in: SPAA '09, ACM, New York, NY, USA, 2009, pp. 79–90, doi:10.1145/1583991.1584017.

[10] J.D. McCalpin, Memory bandwidth and machine balance in current high performance computers, in: IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter, 1995, pp. 19–25.

[11] P.R. Luszczek, D.H. Bailey, J.J. Dongarra, J. Kepner, R.F. Lucas, R. Rabenseifner, D. Takahashi, The HPC Challenge (HPCC) benchmark suite, in: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing - SC, 2006, doi:10.1145/1188455.1188677.

[12] P. Siegl, R. Buchty, M. Berekovic, Data-centric computing frontiers: a survey on processing-in-memory, in: Proceedings of the Second International Symposium on Memory Systems, MEMSYS '16, ACM, New York, NY, USA, 2016, pp. 295–308, doi:10.1145/2989081.2989087.

[13] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, K. Yelick, A case for intelligent RAM, IEEE Micro 17 (2) (1997) 34–44, doi:10.1109/40.592312.

[14] M. Gao, G. Ayers, C. Kozyrakis, Practical near-data processing for in-memory analytics frameworks, in: 2015 International Conference on Parallel Architecture and Compilation (PACT), 2015, pp. 113–124, doi:10.1109/PACT.2015.22.

[15] T. Finkbeiner, G. Hush, T. Larsen, P. Lea, J. Leidel, T. Manning, In-memory intelligence, IEEE Micro 37 (4) (2017) 30–38, doi:10.1109/MM.2017.3211117.

[16] A. Farmahini-Farahani, J.H. Ahn, K. Morrow, N.S. Kim, Nda: Near-DRAM acceleration architecture leveraging commodity DRAM devices and standard memory modules, in: 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA), 2015, pp. 283–295, doi:10.1109/HPCA.2015.7056040.

[17] M. Belviranli, S. Lee, J.S. Vetter, Designing algorithms for the EMU migrating-threads-based architecture, in: High Performance Extreme Computing Conference 2018, 2018.

[18] M. Minutoli, S. Kuntz, A. Tumeo, P. Kogge, Implementing radix sort on Emu 1, 3rd Workshop on Near-Data Processing (WoNDP), 2015.

[19] D. Mizell, K. Maschhoff, Early experiences with large-scale Cray XMT systems, in: 2009 IEEE International Symposium on Parallel Distributed Processing, 2009, pp. 1–9, doi:10.1109/IPDPS.2009.5161108.

[20] D. Ediger, J. Riedy, D.A. Bader, H. Meyerhenke, Computational graph analytics for massive streaming data, in: H. Sarbazi-azad, A. Zomaya (Eds.), Large Scale Network-Centric Computing Systems, Wiley, 2013, doi:10.1002/9781118640708.ch25.

[21] H.C. Edwards, C.R. Trott, D. Sunderland, Kokkos: enabling manycore performance portability through polymorphic memory access patterns, J. Parallel Distrib. Comput. 74 (12) (2014) 3202–3216. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing. doi: 10.1016/j.jpdc.2014.07.003.

[22] K. Aingaran, S. Jairath, G. Konstadinidis, S. Leung, P. Loewenstein, C. McAllister, S. Phillips, Z. Radovic, R. Sivaramakrishnan, D. Smentek, T. Wicki, M7: Oracle's next-generation SPARC processor, IEEE Micro 35 (2) (2015) 36–45, doi:10.1109/MM.2015.35.

[23] T.J. Ham, L. Wu, N. Sundaram, N. Satish, M. Martonosi, Graphicionado: a high-performance and energy-efficient accelerator for graph analytics, in: 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2016, pp. 1–13, doi:10.1109/MICRO.2016.7783759.

[24] G. Ananthanarayanan, A. Ghodsi, S. Shenker, I. Stoica, Disk-locality in datacenter computing considered irrelevant, in: Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems, HotOS'13, USENIX Association, Berkeley, CA, USA, 2011. 12–12.

[25] A. Elafrou, V. Karakasis, T. Gkountouvas, K. Kourtis, G. Goumas, N. Koziris, SparseX: a library for high-performance sparse matrix-vector multiplication on multicore platforms, ACM Trans. Math. Softw. 44 (3) (2018) 26:1–26:32, doi:10.1145/3134442.

[26] T. Rolinger, C.D. Krieger, Impact of traditional sparse optimizations on a migratory thread architecture, in: Proceedings of the 8th Workshop on Irregular Applications: Architectures and Algorithms, IA3'18, 2018.

[27] G.P. Krawezik, P.M. Kogge, T.J. Dysart, S.K. Kuntz, J.O. McMahon, Implementing the Jaccard index on the migratory memory-side processing Emu architecture, in: High Performance Extreme Computing Conference 2018, 2018.

[28] K. Velusamy, T.B. Rolinger, J. McMahon, T.A. Simon, Exploring parallel bitonic sort on a migratory thread architecture, in: High Performance Extreme Computing Conference 2018, 2018.

[29] P. Chatarasi, V. Sarkar, A preliminary study of compiler transformations for graph applications on the Emu system, in: Proceedings of the Workshop on Memory Centric High Performance Computing, MCHPC'18, ACM, New York, NY, USA, 2018, pp. 37–44, doi:10.1145/3286475.3286481.