2018 IEEE International Parallel and Distributed Processing Symposium Workshops

An Initial Characterization of the Emu Chick

Eric Hein, Tom Conte
School of Electrical and Computer Engineering
Georgia Institute of Technology
{ehein6,conte}@gatech.edu

Jeffrey S. Young
School of Computer Science
Georgia Institute of Technology
jyoung9@gatech.edu

Srinivas Eswar, Jiajia Li, Patrick Lavin, Richard Vuduc, Jason Riedy
School of Computational Science and Engineering
Georgia Institute of Technology
{seswar3,jiajiali,plavin3,richie,jason.riedy}@gatech.edu

***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 doing computational work and a stationary core to run basic operating system functions and migrate threads between nodes. In this initial characterization of the Emu Chick, we study the memory bandwidth characteristics of the system through benchmarks like STREAM, pointer chasing, and sparse matrix vector multiply. We compare the Emu Chick hardware to architectural simulation and Intel Xeon-based platforms. While it is difficult to accurately compare prototype hardware with existing systems, our initial evaluation demonstrates that the Emu Chick uses available memory bandwidth more efficiently than a more traditional, cache-based architecture. Moreover, the Emu Chick provides stable, predictable performance with 80% bandwidth utilization on a random-access pointer chasing benchmark with weak locality.**

I. 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, timestamps for streaming graph edges), analysis algorithms tend to access data in a more random fashion. Limited spatial locality in traditional analysis kernels means that cache lines are underutilized, prefetch engines are confounded, and thus overall memory bandwidth is underutilized. Furthermore, common analysis kernels may exhibit dynamic parallelism and create many data-dependent memory references, which can stall architectures that cannot otherwise keep as many contexts and requests in flight. Consequently, today's "big data" platforms are frequently outperformed by a single thread accessing a large SSD [1].

This state of affairs motivates novel architectures like the Emu migratory thread system [2], the subject of this study. The Emu is a cache-less system built around "nodelets" that each execute lightweight threads and migrate threads to data rather than moving data through a traditional cache hierarchy.

This paper is the first, independent characterization of the Emu Chick prototype. 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 [3], [4]. 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 and ParTI [5] 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 the *locations* of data critical to reducing migrations.

This study's specific demonstrations include

- the first 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;
- a discussion of memory allocation, data layout, and "smart" thread migration on the Emu architecture with respect to SpMV kernels;
- and an initial 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 affect on thread migration.

II. 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 Figure 1:

- A common *stationary* processor runs the operating system (*e.g.* Linux) and manages storage and network devices.
- *Nodelets* combine narrowly banked memory 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

978-1-5386-5555-9/18/$31.00 ©2018 IEEE
DOI 10.1109/IPDPSW.2018.00097

IEEE computer society

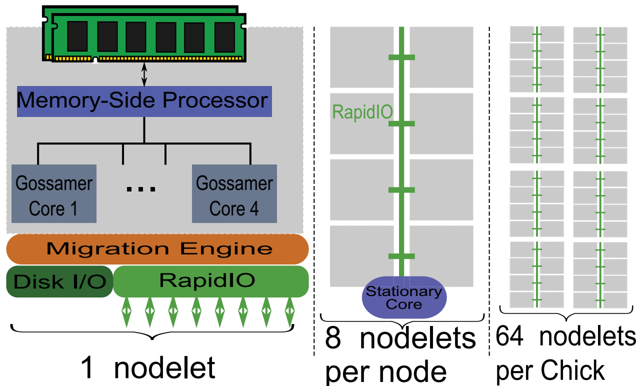


Fig. 1: Emu architecture: The system consists of *stationary* processors for running the operating system and up to four *Gossamer* processors per nodelet tightly coupled to memory. The cache-less Gossamer processing cores are multi-threaded to both source sufficient memory references and also provide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

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 III. A more detailed description of the Emu architecture is available elsewhere [2].

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 [6], 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.

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 [7]. 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.

III. Experimental Setup

A. 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 prototype Emu system versus the next-generation Emu system which will use larger and faster FPGAs to implement computation and thread migration. Here we describe the current status and limitations of the current Emu Chick prototype:

- Our system has only one Gossamer Core (GC) per nodelet with a concurrent max of 64 threadlets. The final system will have four GC's per nodelet, supporting 256 threadlets per nodelet.
- Our GC's are clocked at 150MHz rather than the planned 300MHz in the next-generation Emu system.
- The DDR4 DRAM modules are clocked at 1600MHz rather than the full 2133MHz allowed by the specification.
- Firmware bugs in the inter-node routing engine limit us to using one node (8 nodelets, single-node) at a time, rather than the full 8 nodes (64 nodelets, multi-node) in the Emu Chick.
- 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` to Emu threads. For this reason, all benchmarks are currently implemented using `cilk_spawn`. However, the use of `cilk_spawn` does allow for more control over spawning strategies.

All experiments are run using Emu's 17.11 compiler and simulator toolchain, and the Emu Chick system is running the 1.0 firmware.

B. Emu Simulator

Emu provides a simulator along with the compiler toolchain to aid in testing and evaluating software before running on

the hardware. The simulator counts key performance events such as the number of thread spawns, migrations, and memory operations per nodelet. This work employs two configurations of the simulator: one to simulate the performance of the Emu Chick as it was designed to be at full speed (Figure 11 in Section IV-D), and one configuration that aims to match the characteristics of our current hardware for validation. We also compare simulation results with the actual hardware in Section IV-D.

C. CPU-based 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 one of two Intel multi-socket server systems. STREAM and pointer chasing benchmarks are run on a system that has a dual-socket Intel Xeon E5-2670 with 64GiB of DDR3 memory (referred to as `Sandy Bridge Xeon`). This processor has a 20MiB shared L3 cache and is clocked at 2.6GHz. Four memory channels clocked at 1600MHz lead to a peak theoretical bandwidth of 51.2 GB/s. SpMV is tested on a four-socket Xeon E7-4850 v3 (Haswell) machine with 2 TiB of DDR4 (referred to as `Haswell Xeon`). The CPUs on the Haswell server are each clocked at 2.20GHz and each have a 35 MiB L3 cache, while the memory is clocked at 1333 MHz (although it is rated for 2133 MHz). Each socket has a peak theoretical bandwidth of 85 GB/s.

For each benchmark, Emu-specific intrinsics (*e.g.* mallocs) are swapped out for their x86 equivalents, and the benchmarks are compiled with GCC 5.4.0. The Cilk keywords are left unchanged, allowing GCC's Cilk runtime to implement the parallel functionality.

D. Metrics for Comparing the Emu Prototype with Common 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-based 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 2GB/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-based system, this might be analogous to effective cacheline 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.

E. Benchmarks

As discussed in Section III-A, the Emu Chick toolchain currently does not support all of Cilk Plus. 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 [8] benchmark is ported and tuned for the Emu hardware in order 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, these arrays are striped across all the nodelets in the system.

Several variants of this benchmark are developed to determine the most efficient way to spawn threads on this architecture. Since `cilk_for` is not yet supported, we use hand-written loops to implement various kinds of spawn trees to saturate the system. These spawn trees 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 sums 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. Figure 2 explains the list initialization further.

The pointer chasing benchmark was designed to have three key properties.

- 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

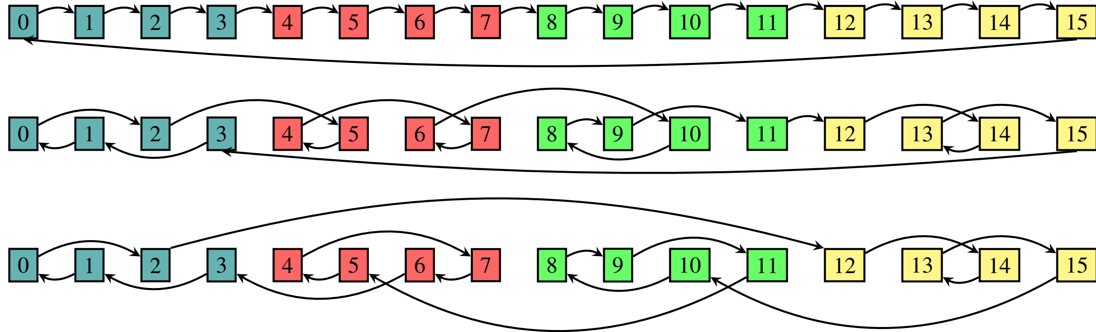


Fig. 2: (top) An ordered linked list, in which consecutive elements have sequential memory addresses, (middle) A linked list with a intra-block shuffle permutation applied to randomize the ordering of elements within a block. Note that all elements within a block are accessed before jumping to the next 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.

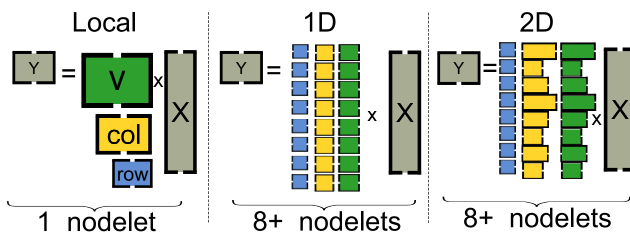


Fig. 3: Emu-specific layout for CSR SpMV

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 memory pool, the resulting data structure will exhibit all three of these characteristics when it is traversed. The pointer chase benchmark is quite similar to the GUPS/RandomAccess benchmark[9], however GUPS lacks data-dependent loads, and pointer chase does not modify the list.

Sparse Matrix Vector Multiply (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 similar to a traditional contiguous malloc (`mw_localmalloc`) as well as a "striped" malloc (`mw_malloc1dlong`) that places data in a round-robin fashion across nodelets and a 2D malloc (`mw_malloc2d`), that stripes entire data structures across nodelets.

Figure 3 demonstrates the three layouts that are tested with inputs in Compressed Sparse Row (CSR) format. In the local case, contiguous mallocs are used to place the output matrix, `Y`, the input CSR matrix, `V`, the row pointer and column index arrays and the vector, `X`, all on a single node. For the 1D layout, `mw_malloc1dlong` is used to stripe the input matrix and row and column arrays across the nodelets (and across nodes in the multi-node case) while the output matrix is on nodelet 0 and `X` is replicated across all nodelets. For the 2D allocation, we use a two-stage allocation rather than Emu's 2D malloc to partition `V` across multiple nodelets. First, the lengths of each row that is assigned to a nodelet are computed and then data for `V` and the column index array is allocated on each nodelet using `mw_malloc1dlong`. `X` is replicated across each nodelet and the output is placed on nodelet 0 in both the 1D and 2D cases.

This benchmark uses these different layout strategies to test performance for placing all data within a nodelet and striping it in a 1D and 2D fashion across multiple nodelets. In the 2D case no thread migrations occur when accessing elements in the same row as opposed to a migration for every element within a row in the 1D layout. Synthetic Laplacian matrix inputs are created corresponding to a d-dimensional k-point stencil on a grid of length n in each dimension. For the tested synthetic matrices, $d = 2$ and $k = 4$, resulting in a $n^2 * n^2$ Laplacian with 5 diagonals. CPU tests are run on `Haswell Xeon`, 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 5.4.0, and run with `CILK_NWORKERS` set to 56. Data is distributed across NUMA regions using `numactl --interleave=0-3`.

Future work with SpMV will investigate new state-of-the-art SpMV formats and algorithms such as SparseX, which uses the Compressed Sparse eXtended (CSX) format for storing matrices[10].

Ping Pong: Simulation validation results (Section IV-D) demonstrate a need for a more fine-grained microbenchmark to illustrate potential differences between hardware and simulated hardware Emu platforms. To explore the cause of this discrepancy, we present another small benchmark called ping pong migration. This micro-benchmark measures the bandwidth of thread migrations on the Emu Chick. In each trial, N threads simply migrate back and forth between two nodelets several thousand times.

IV. RESULTS

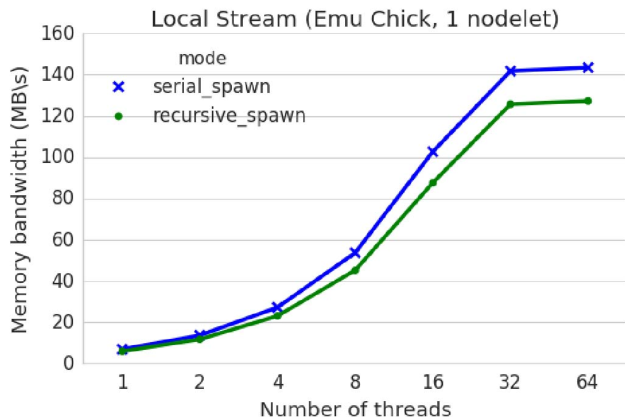


Fig. 4: Memory bandwidth achieved on a single node of the Emu Chick. Threads are created using a serial loop or a recursive spawn tree.

A. STREAM

Figure 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 Figure 5, 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 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. The results show that remote spawns are essential to achieving maximum bandwidth on Emu.

In comparison to the Emu, our reference Xeon system (`Sandy Bridge`) achieves close to the nominal bandwidth of

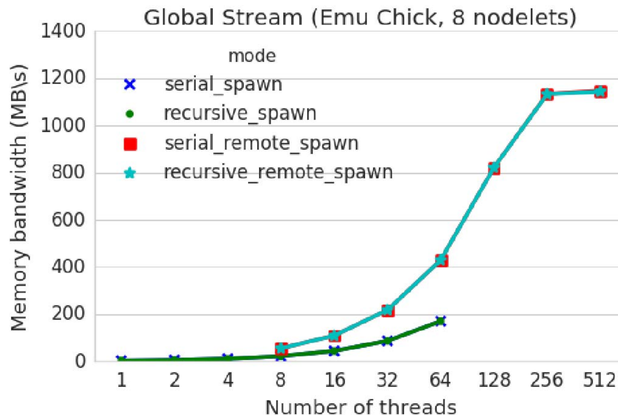


Fig. 5: Memory bandwidth achieved on eight nodelets of the Emu Chick. The remote spawn variants create a thread on each nodelet which subsequently creates the local worker threads.

51.2 GB/s on the STREAM benchmark while the Emu Chick has a maximum STREAM bandwidth of 1.2 GB/s on a single node card. An initial test of the full 8-node configuration of the Emu Chick yielded 6.5 GB/s, but this configuration was not stable enough to collect further results at the time of writing. Future Emu systems are planned to have much higher bandwidth than the initial prototypes, with up to 160 GB/s of bandwidth. 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.

B. Pointer Chasing

Figures 6 and 7 compare the performance of the Emu Chick against our Sandy Bridge 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 threadlets

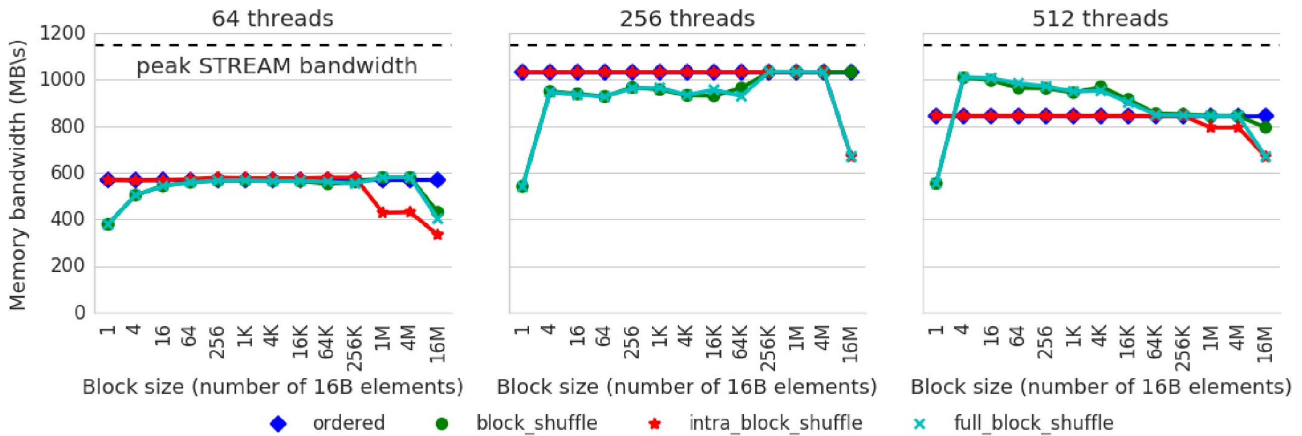


Fig. 6: Pointer chasing performance on a single node of the Emu Chick.

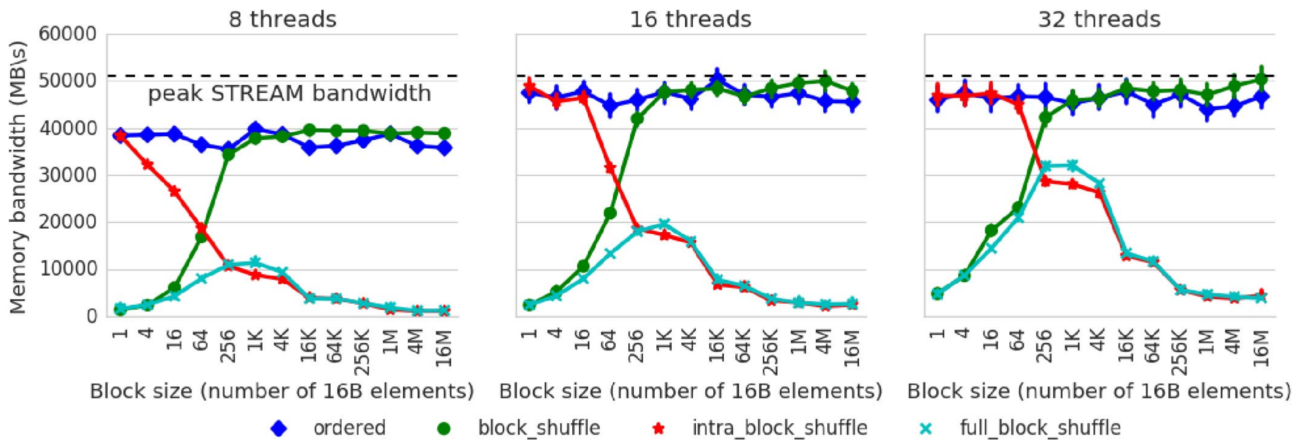


Fig. 7: Pointer chasing performance on Sandy Bridge Xeon

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.

Figure 8 shows the normalized bandwidth usage (i.e., effective bandwidth usage) for the Sandy Bridge 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 80% of available system bandwidth in most cases and 50% in the worst cases. The Sandy Bridge Xeon uses less than 25% of peak bandwidth in most cases, 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

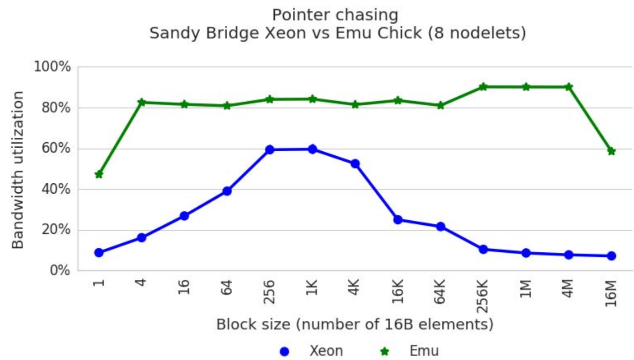


Fig. 8: Bandwidth utilization of pointer chasing, compared between Sandy Bridge Xeon and Emu

to compute SpMV and SpMM (sparse matrix-matrix product) operations, respectively.

C. Sparse Matrix-Vector Multiply

Figure 9 shows the memory bandwidth achieved by SpMV using each of the three data layout strategies on the Emu versus three different CPU implementations. A Laplacian size of n specifies a sparse matrix corresponding to a 5-point, 2-D, $n \times n$ stencil, which is a matrix of $n^2 * n^2$ with 5 diagonals. 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 50 MB/s and 100 MB/s, respectively. As `cilk_for` is not yet supported on the Emu hardware, the `cilk_spawn` CPU version mostly closely resembles the Emu SpMV implementation. While Figure 9b demonstrates a higher maximum bandwidth, it is interesting to note that the Emu system provides good scaling of bandwidth at lower values of n, especially for the 2D memory layout, which scales up to 250 MB/s for n=100.

On the Haswell system (Figure 9b), both MKL and `cilk_for` show good scaling with matrix size, while `cilk_spawn` performance depends largely on grain size, or number of elements per spawn. A large grain size of 16,384 for `cilk_spawn` works best for CPU-based SpMV while a much smaller grain size of 16 elements per spawn is most effective for the Emu implementation. A smaller grain size on the Emu results in more active threads but also better work balance while the larger grain size on the CPU-based system results in lower overhead and fewer overall thread spawns.

D. Emu Simulation Validation and Prediction

We wish to predict the performance of an Emu Chick system operating at full speed as well as larger configurations by using the provided Emu simulator. First we validate the simulated measurements by configuring it to match the specifications of our current hardware system. The results of this evaluation are displayed in Figure 10. While the STREAM benchmark results match well for both single nodelet and multi-nodelet operation, the pointer chase benchmark results do not. Despite the error in magnitude, the shape of the results matches well.

To help explain this difference, Figure 10 also shows results from the hardware and simulated ping pong benchmark. While the simulator can perform 16 million migrations per second, the hardware is currently limited to only 9 million migrations per second. Since pointer chasing is a migration-heavy benchmark, the performance of the thread migration engine affects its performance to a much greater degree than STREAM. Our experiments indicate that the latency for a single thread migration on the current system is approximately 1-2 μs.

Finally, Figure 11 plots simulation results for the full-speed configuration of a 64 node Emu system running the pointer chasing benchmark. Despite the increase in scale, the system performance is still not sensitive to the granularity of spatial locality, and bandwidth scales well even up to thousands of threads.

V. DISCUSSION

This initial 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.

A. Impacts of Data Location and Thread Migration

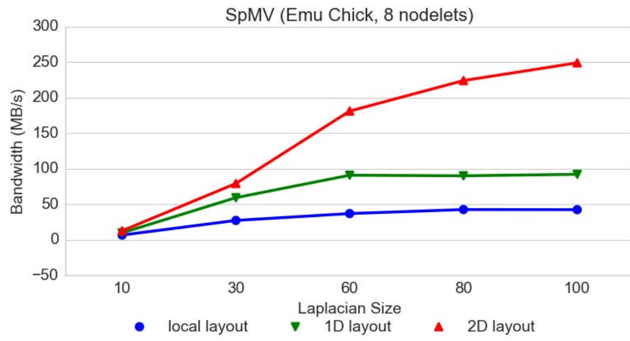
While the results from SpMV demonstrate that data layout can have an impact on performance on the Emu, application performance also depends on where threads are spawned and how many migrations occur between nodes and nodelets. In the initial development of our benchmarks, we debated explicitly minimizing thread movement and keeping computation local to a specific node. 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 IV-A, 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.

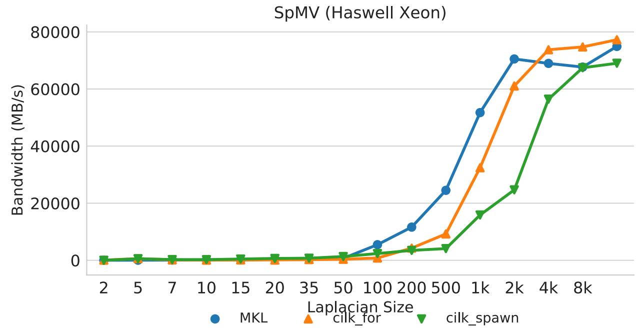
B. Performance Models and Comparisons to Existing Architectures

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. 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

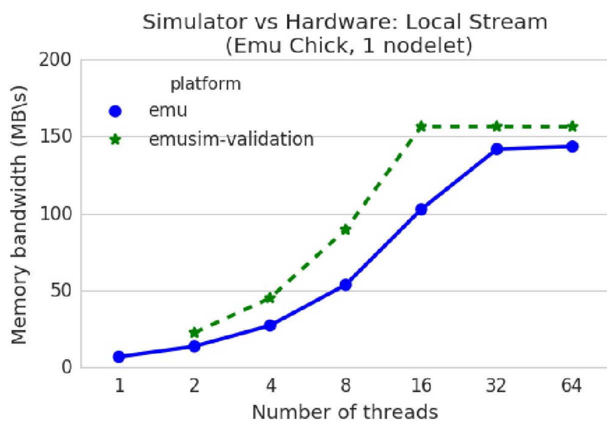


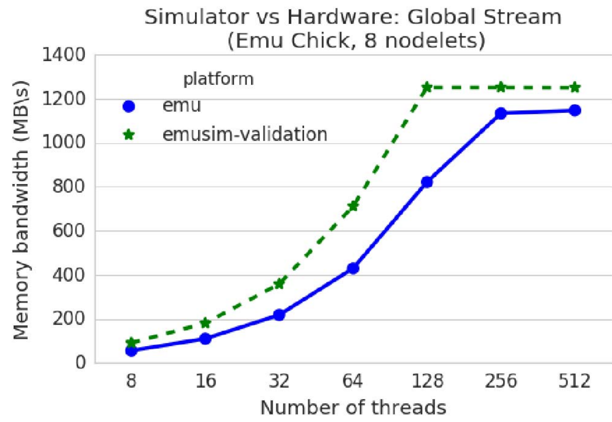
(a) Emu SpMV BW

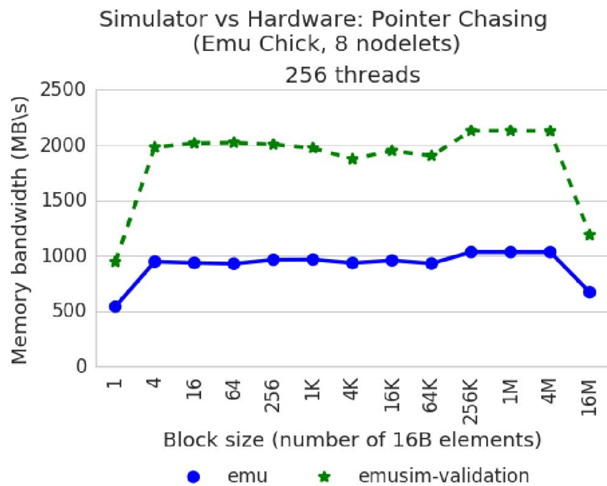


(b) CPU SpMV BW

Fig. 9: Effective bandwidth for the Emu and Xeon E7-4850 platform running with synthetic inputs with 512 and 56 threads, respectively.







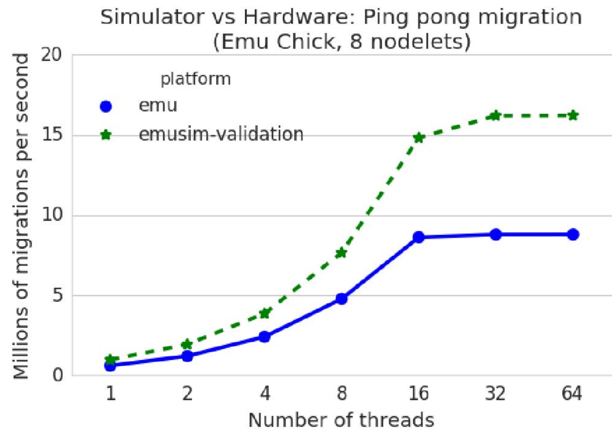


Fig. 10: Emu hardware performance compared with simulator results

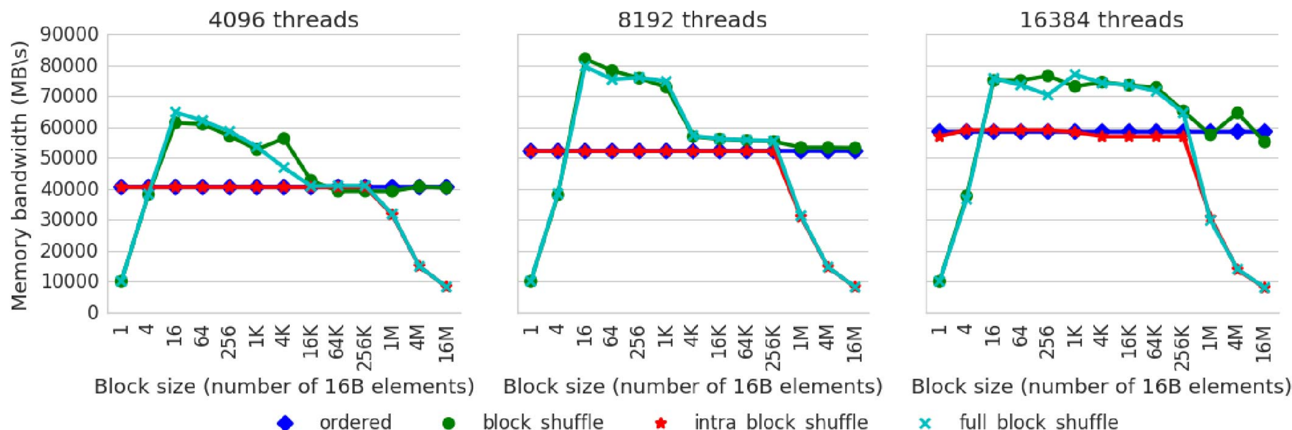


Fig. 11: Simulated results for the pointer chasing benchmark running on an Emu Chick at full speed.

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 (ie, 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.

VI. Related Work

Advances in memory and integration technologies provide opportunities for profitably moving computation closer to data[11]. Some proposed architectures return to the older processor-in-memory (PIM) and "intelligent RAM"[12] ideas. Simulations of architectures focusing on near-data processing[13] including in-memory[14] and near-memory[15] show great promise for increasing performance while also drastically reducing energy usage. 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 Cray XMT[16] could provide high bandwidth utilization by tolerating long memory latencies in applications that could produce enough threads. Another approach is to push memory-centric aspects to an accelerator like Sparc M7's data analytics accelerator[17] for database operations or Graphicionado[18] for graph analysis.

Moving computation to data via software has had a successful history in supercomputering via Charm++[7], 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[19].

VII. Conclusion

The initial 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 performance for a variety of benchmarks including STREAM, pointer chasing, and SpMV. Initial results demonstrate low overall bandwidth for the Emu system but illustrate that it 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 IV-B achieves a stable 80% bandwidth utilization across a wide range of locality parameters. These results and initial results on how data layouts can improve random access with SpMV provide a template for future benchmarking and application developmen and show how application memory layouts and "smart" thread migration can be used to maximize performance on the Emu system.

VIII. Acknowledgments

This work partially was supported by NSF Grant ACI-1339745 (XScala), 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 does not necessarily reflect the position or the policy of the sponsors. Thanks also to the Emu Technology team for support and debugging assistance with the Emu Chick prototype.

REFERENCES

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

[2] T. Dysart, P. Kogge, M. Deneroff, E. Bovell, P. Briggs, J. Brockman, K. Jacobsen, Y. Juan, S. Kuntz, and 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.

[3] D. Ediger, R. McColl, J. Riedy, and D. A. Bader, "STINGER: High performance data structure for streaming graphs," in *The IEEE High Performance Extreme Computing Conference (HPEC)*, Waltham, MA, Sep. 2012.

[4] J. Li, Y. Ma, C. Yan, and 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)*, Nov 2016, pp. 26–33.

[5] "ParTI Github," online, 2018. [Online]. Available: https://github.com/hpcgarage/ParTI

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

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

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

[9] P. R. Luszczek, D. H. Bailey, J. J. Dongarra, J. Kepner, R. F. Lucas, R. Rabenseifner, and D. Takahashi, "The HPC Challenge (HPCC) benchmark suite," *Proceedings of the 2006 ACM/IEEE conference on Supercomputing - SC 06*, 2006.

[10] A. Elafrou, V. Karakasis, T. Gkountouvas, K. Kourtis, G. Goumas, and N. Koziris, "SparseX: A library for high-performance sparse matrix-vector multiplication on multicore platforms," *ACM Trans. Math. Softw.*, vol. 44, no. 3, pp. 26:1–26:32, Jan. 2018.

[11] P. Siegl, R. Buchty, and M. Berekovic, "Data-centric computing frontiers: A survey on processing-in-memory," in *Proceedings of the Second International Symposium on Memory Systems*, ser. MEMSYS '16. New York, NY, USA: ACM, 2016, pp. 295–308.

[12] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, "A case for intelligent RAM," *IEEE Micro*, vol. 17, no. 2, pp. 34–44, Mar. 1997.

[13] M. Gao, G. Ayers, and C. Kozyrakis, "Practical near-data processing for in-memory analytics frameworks," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, Oct. 2015, pp. 113–124.

[14] T. Finkbeiner, G. Hush, T. Larsen, P. Lea, J. Leidel, and T. Manning, "In-memory intelligence," *IEEE Micro*, vol. 37, no. 4, pp. 30–38, Aug. 2017.

[15] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, and 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)*, Feb. 2015, pp. 283–295.

[16] D. Mizell and K. Maschhoff, "Early experiences with large-scale cray xmt systems," in *2009 IEEE International Symposium on Parallel Distributed Processing*, May 2009, pp. 1–9.

[17] K. Aingaran, S. Jairath, G. Konstadinidis, S. Leung, P. Loewenstein, C. McAllister, S. Phillips, Z. Radovic, R. Sivaramakrishnan, D. Smentek, and T. Wicki, "M7: Oracle's next-generation sparc processor," *IEEE Micro*, vol. 35, no. 2, pp. 36–45, Mar. 2015.

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

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