Exploring Parallel Bitonic Sort on a Migratory Thread Architecture

Kaushik Velusamy*, Thomas B. Rolinger†§, Janice McMahon‡, and Tyler A. Simon*

* University of Maryland, Baltimore County, MD USA
† University of Maryland, College Park, MD USA
‡ Emu Solutions, Inc., South Bend, IN, USA
§ Laboratory for Physical Sciences, College Park, MD USA

kaushikvelusamy@umbc.edu, tbrolin@cs.umd.edu, jmcmahon@emutechnology.com, tsimo1@umbc.edu

***Abstract*—Large scale, data-intensive applications pose challenges to systems with a traditional memory hierarchy due to their unstructured data sources and irregular memory access patterns. In response, systems that employ migratory threads have been proposed to mitigate memory access bottlenecks as well as reduce energy consumption. One such system is the Emu Chick, which migrates a small program context to the data being referenced in a memory access. Sorting an unordered list of elements is a critical kernel for countless applications, such as graph processing and tensor decomposition. As such applications can be considered highly suitable for a migratory thread architecture, it is imperative to understand the performance of sorting algorithms on these systems. In this paper, we implement parallel bitonic sort and target the Emu Chick system. We investigate the performance of an explicit comparison-based approach as well as a sorting network implementation. Furthermore, we explore two different data layouts for the parallel bitonic sorting network, namely cyclic and blocked. From the results of our performance study, we find that while thread migrations can dictate the overall performance of an application, the cost of thread creation and management can out-grow the cost of thread migration.**

***Keywords*-bitonic sort, performance evaluation, Emu, migratory threads, near-memory processing**

I. Introduction

Due to the growing popularity of large scale data analytics, the need to efficiently process large, unstructured data is becoming increasingly common. In applications such as graph processing and tensor decomposition, performance is largely dictated by how effectively the memory hierarchy is utilized [1]. However, due to the sparse nature of these applications, the memory access patterns are highly irregular and exhibit little to no locality. Such characteristics pose performance challenges for systems with a traditional memory hierarchy. In response, migratory thread architectures have been proposed in order to adapt to the above mentioned challenges. In such a system, when a remote memory access is requested, the program context is migrated to the location of the data being referenced, rather than moving the data to the program.

Unlike irregular applications, sorting algorithms have much more structure and are easier to optimize for traditional systems. However, many of the irregular applications mentioned above rely heavily on having a fast and efficient sorting implementation. For example, in some tensor decomposition implementations, it is necessary to sort the non-zero indices of a tensor as a pre-processing step to performing the decomposition [2], [3]. Therefore, to effectively utilize migratory thread systems for these applications, it is important to understand the performance of sorting algorithms on such systems.

Regardless of whether the algorithm is irregular or structured, there are key factors to consider when implementing codes on a migratory thread architecture. For one, it is imperative to mitigate thread migration overhead. This has been shown to be true in other migration-based systems such as Charm++ [4], where thread migrations are considered a critical factor for performance. Another factor is proper data layout on the system. As on other architectures, such as graphics processing units (GPUs), the layout of the data has profound impacts on the overall performance of an application [5]. Data structures should be organized in such a way as to reduce the number of thread migrations and prolong the number of instructions a given thread will execute before performing a migration.

In this paper, we investigate the performance of the parallel bitonic sort algorithm on the Emu Chick system, which is a migratory thread architecture with near-memory processing capabilities. Our purpose is to understand the performance characteristics of sorting on the Emu architecture as well as the influence of data layout.

This paper makes the following contributions:

1) Provides the first implementation of a parallel sorting algorithm on the Emu Chick hardware. We developed two different implementations of bitonic sort: a traditional comparison-based implementation and a sorting network implementation.
2) Explores two different data layouts for a bitonic sorting network on the Emu Chick: cyclic and blocked.
3) Presents a performance evaluation of the different bitonic sort implementations and the different data layouts. Through this evaluation, we find that while thread migrations can dictate performance, the overhead of creating and managing threads has a much stronger

impact on the overall performance.

The rest of this paper is organized as follows: Section II presents work as it relates to migratory threads and near-memory processing as well as bitonic sort. We describe the Emu architecture in Section III. Section IV provides a description of the bitonic sort algorithm and its sorting network formulation. We describe our implementation of bitonic sort on the Emu Chick system in Section V. Section VI presents the performance evaluation results of our bitonic sort implementation. Finally, we present concluding remarks and future work in Section VII.

II. Related Work

There have been a number of processing-in-memory (PIM) and near-memory processing systems in the past, including EXECUBE [6], DIVA [7], [8], and Intelligent RAM [9]. In these systems, the memory and logic was combined on the same chip to improve memory latency and bandwidth, as well as reduce overall energy consumption. The Emu architecture, as described by Dysart et al. [10], provides highly scalable near-memory processing using migrating threads. In this work, we use the Emu architecture as our hardware platform to investigate the performance of bitonic sort.

Other researchers have evaluated the performance of sorting algorithms on Emu systems. Minutoli et al. [11] implemented a radix sort and discussed the similarities and differences of the implementation with respect to other systems. That work was only simulated, not evaluated on a real system. Hein et al. [12] provided an initial characterization of the Emu Chick system by evaluating popular benchmarks such as STREAM [13], pointer chasing and sparse matrix vector multiply. We consider our work to be complimentary to Hein et al. in that we further characterize the Emu Chick system by evaluating sorting, which was not investigated in their study.

The bitonic sort algorithm and sorting network was described by Batcher [14]. Since then, there have been many architectures and systems targeted by bitonic sort, including mesh-connected processor arrays [15], GPUs [16], [17], and distributed memory systems [18]. Ionescu and Schauser [19] presented optimizations for parallel bitonic sort, focusing on a smart data layout that alternates between cyclic and blocked. Our work leverages Ionescu and Schauser's description of the bitonic sort network and the different data layouts.

III. The Emu Chick Architecture

The Emu architecture is a scalable near-memory processing system that employs migrating threads. The total size of the program context that is migrated is less than 200 bytes, as it consists of 16 general-purpose registers, a program counter, a stack counter and status information [12]. Therefore, a migratory payload is generally many times smaller than the total size of the data that would be fetched from memory in a traditional system. A system under the Emu architecture consists of some number of nodes and an I/O system between the nodes. Within each node are some number of *nodelets*, where each nodelet

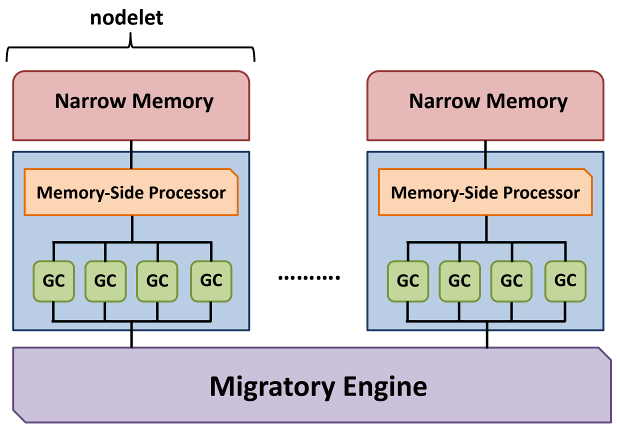

Fig. 1. A single node in the current Emu architecture. "GC" refers to a Gossamer core. The migration engine provides communication between other nodelets. There are 8 nodelets within a single node.

consist of an 8-bit wide bank DDR4 DRAM and multiple multi-threaded, cache-less *Gossamer* cores (GCs).

Nodelets are the building blocks of the Emu architecture and enable the near-memory processing aspect of the system [11]. Threads migrate between nodelets and nodes when performing memory accesses, ensuring that each read is performed locally. If a thread initiates a read request for data that is not local to its current location, the thread context will be packaged up and migrated to the node/nodelet which is connected to the memory containing the requested data. A memory-side processor exists on each nodelet, where atomic update operations can be performed on small amounts of data without requiring a migration.

For this work, we utilize the Emu Chick system, which contains 8 nodes and 8 nodelets per node implemented on FPGAs. Figure 1 shows the general layout of a single node in the Emu Chick system, which is identical to the system used by Hein et al. in their study [12]. Note that in the current system, there is only 1 GC per nodelet and the clock rate of each GC is 150MHz. Each nodelet can support up to 64 concurrent threads, for a maximum of 512 threads per node. The migration engine acts as a communication interface between the nodelets and facilitates the migration of threads throughout a given node. We refer the reader to more extensive studies on the Emu architecture for further information [10]–[12]. In this work, we focus our experiments within a single-node execution environment.

Currently, the programming interface for the Emu architecture is based in the C language and uses Cilk semantics to provide mechanisms for parallelism [20]. Task parallelism is achieved via the `cilk_spawn` and `cilk_sync` constructs. The Emu programming interface provides several system calls to control how and where memory is allocated across nodes/nodelets. Most relevant to our study are the `malloc1d` and `malloc2d` routines. The `malloc1d` routine allocates an array where elements are striped across the nodelets in a

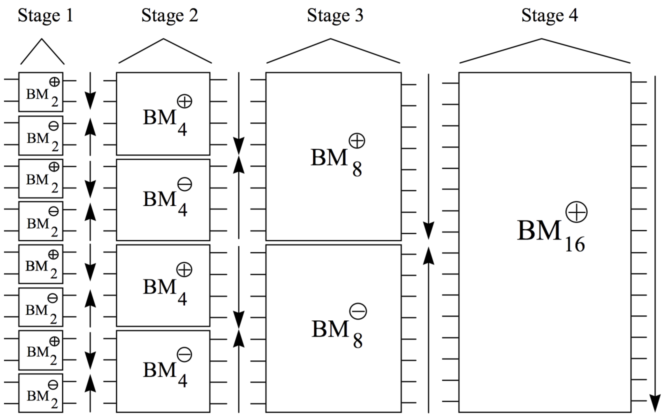

Fig. 2. Bitonic sorting network for an input of 16 elements taken from [19]. $BM_k^{\oplus}$ and $BM_k^{\ominus}$ refer to an increasing and decreasing bitonic merge of size k, respectively.

round-robin fashion. On the other hand, `malloc2d` allows programmers to place contiguous elements of an array on the same nodelet, providing a blocked data layout.

IV. Bitonic Sort

The bitonic sort algorithm works by taking as input an unsorted list of elements and constructing a *bitonic sequence*. A bitonic sequence is a sequence of values $a_0, ..., a_{n-1}$ such that there exists an index $0 \leq i \leq n-1$ where $a_0, ..., a_i$ is monotonically increasing and $a_i, ..., a_{n-1}$ is monotonically decreasing, or there exists some cyclic shift of indices that satisfy this condition. Once a bitonic sequence is constructed from the unordered input data, the sequence is split into two smaller bitonic sequences, such that all elements in one sequence are smaller than all elements in the other. This process is repeated until sequences of size 1 are obtained, which indicates that the input data is sorted. This splitting until sorted procedure is known as *bitonic merging*.

The bitonic sort algorithm can be implemented in a traditional comparison-based manner, where elements in the input list are swapped based on the result of a explicit comparison. However, it is well known that bitonic sort can be modeled as a *sorting network* [14], [19]. In a sorting network, elements from the input list are carried on wires, which can be actual hardware components or an abstraction in software. Comparators connect two wires together and the elements on these wires are switched to be in either increasing or decreasing order. A sorting network only performs comparisons and the sequence of comparisons performed in the sorting network are set in advance and are independent of the outcome of any past comparison results. In other words, the sorting network is not dependent on the input data, and therefore, highly suitable for parallel execution. An illustration of a bitonic sorting network for an input size of 16, taken from [19], is presented in Figure 2. To sort N elements, the bitonic sorting network requires $\log N$ sorting stages, where the i^{th} stage consists of $N/2^i$ bitonic merges of size 2^i.

Implementing a compare-and-exchange operation between two values in a sorting network can be achieved by performing a min and max on the two wires carrying those values. If the values are not in the desired order, then the min/max operations will put them into the correct order. If the values were already in the desired order, then the result of the min/max operations would leave the values unchanged. Hence, the sorting network does not need to be aware of the relationship between values to perform the appropriate compare-and-exchange.

V. Implementing Bitonic Sort on the Emu Chick

In this section, we describe our implementation of the bitonic sort algorithm for the Emu Chick system. We present the details of a comparison-based approach as well as a sorting network implementation. As data layout on a migratory thread system is one of the focal points of this work, we describe how we adapted the sorting network implementation to use a cyclic or blocked data layout.

A. Comparison-based Implementation

We started with a C implementation of an iterative comparison-based approach to bitonic sort and adapted it to use Cilk for parallelization. In the bitonic merge stages, threads are created via `cilk_spawn` to perform a series of compare-and-exchange operations. As this implementation is data dependent, the amount of work that each thread performs can vary. This can pose challenges from a loading-balancing perspective, as well as present difficulties in amortizing the cost of thread creation and management with computation.

The input data is mapped across the 8 nodelets via `malloc1d`, so the layout is cyclic. This provides trivial indexing into the distributed array, as the original index domain is preserved. However, adjacent elements in the input list are not on the same nodelet, so performance issues in terms of non-local comparisons can arise. Furthermore, the number of nodelets actually used and the number of elements per nodelet is dependent on the number of elements to sort.

B. Sorting Network Implementation

Our bitonic sorting network implementation is based largely on the description provided Batcher [14] as well as Ionescu and Schauser [19]. As the sorting network is not data dependent, we can specify the number of nodelets to use as well as the number of threads per nodelet. For each bitonic merge, we spawn a thread on each of the nodelets in use. Each of these nodelet threads then spawn the specified number of threads to perform the compare-and-exchange operations, which are performed via min and max functions as described in Section IV. In the following two sections, we describe how the cyclic and blocked data layouts were implemented for the sorting network.

1) Cyclic Data Layout: At a high level, the cyclic data layout for the sorting network is identical to that of the one described for the comparison-based approach in Section V-A. However, since the sorting network implementation requires that the number of nodelets and threads per nodelet be specified, our data allocation scheme is more complicated. To this end, we make use of the `malloc2d` routine to allocate blocks of elements on each of the nodelets specified. Given N elements and P nodelets labeled 0 to $P-1$, we will assign N/P elements per nodelet (EPN). To achieve the cyclic layout, the i^{th} element of the input list is placed on nodelet i/EPN and the offset of the i^{th} element on this nodelet is $i \pmod{EPN}$. The effect of the cyclic layout in terms of remote accesses is that the first $\log(EPN)$ stages of the algorithm require non-local accesses. Then for the subsequent $\log(EPN)+k$ stages, where $1 \leq k \leq \log P$, the first k steps require only local accesses while the last $\log(EPN)$ steps require remote accesses.

2) Blocked Data Layout: As opposed to the cyclic data layout, a blocked data layout assigns contiguous elements in the input list to the same nodelet. We achieve this by again using the `malloc2d` allocation routine, but we modify the mapping of indices to nodelets as follows: the i^{th} element of the input list is placed on nodelet $i \pmod{P}$ and the offset of the i^{th} element on this nodelet is i/P, where P is the number of nodelets used. The programming model for the Emu architecture allows for an easy transition from the cyclic to blocked layout by simply redefining the offset formulas described, avoiding the need for a different memory allocation routine. As far as the effect of the blocked layout on remote accesses, the first $\log(EPN)$ stages of the algorithm require only local accesses. Then for the subsequent $\log(EPN)+k$ stages, the first k steps require remote accesses while the last $\log(EPN)$ steps require local accesses. This is the opposite effect of the cyclic layout.

VI. Performance Evaluation

Upon implementing the comparison-based approach to the bitonic sort algorithm as well as the sorting network, we then focused on evaluating their respective performance. We also explored the performance of the sorting network with a cyclic and blocked data layout. Our goal was to understand and quantify the differences between the comparison and network based approaches, as well as the data layouts, with the hope of being able to draw conclusions about the efficacy of mapping an algorithm such as bitonic sort onto the Emu architecture.

A. Experimental Setup

Our codes were built using version 18.04.1 of the Emu toolchain. The Emu Chick system we used for the experiments is as described in Section III.

For our data sets, we generated list of numbers that were chosen uniformly at random. We varied the number of elements to sort from 2^1 to 2^{21}. In the following section, we focus our discussion on sorting 2^{15} to 2^{21} elements, as those data set sizes provide the most meaningful results.

Our performance metrics include clock cycles and the number of thread migrations. As the current Emu Chick system is running at a reduced clock speed of 150Mhz, measuring clock cycles rather than absolute runtime provides a more suitable metric for comparisons. We obtained the number of thread migrations by running our codes in the Emu simulator, as the hardware can only measure clock cycles. All our cycle counts and thread migration measurements were taken with respect to the sorting routine, which excludes any file I/O and pre-processing.

For this work, we utilized a single node on the Emu Chick system, which has 8 nodelets. In our implementation of the bitonic sorting network, the number of nodelets and the number of threads per nodelet are tunable parameters. To this end, we performed runs with 1, 2, 4, and 8 nodelets as well as varied the number of threads per nodelet from 1 to 64. Therefore, the maximum number of threads that we utilized was 512. For the comparison based implementation, the number of nodelets and threads utilized is dynamic and depends on the number of elements to be sorted, as described in Section V.

B. Results

In this section, we present and discuss our results as they pertain to the following questions: (1) how does a comparison-based implementation of bitonic sort compare to a sorting network implementation and (2) how do different data layouts affect performance for the sorting network approach. The first question aims to gain insight into the appropriate structure for sorting algorithms on a migratory thread architecture, while the second question attempts to quantify the importance of data layout on such an architecture.

1) Comparison Sorting Versus Sorting Network: Figure 3 presents the total number of cycles (in billions) required to sort 2^{15} to 2^{21} elements for the comparison-based implementation of bitonic sort and the sorting network implementation. We observe that the comparison based approach requires 19.2x more cycles on average than the sorting network. To understand this large gap in performance between the two implementations, we investigated the total number of thread migrations as well as the total number of threads created during each code's execution, which are presented in Figures 4 and 5, respectively. It is interesting to note that while the sorting network is clearly superior in terms of clock cycles, it exhibits an average of 3.3x more thread migrations than the comparison based approach. However, when we measured the total number of threads created, as shown in Figure 5, we observed that the comparison based approach creates up to 688x more threads than the sorting network. This underscores the importance of having efficient thread creation strategies, which can be more beneficial to performance than reducing migrations.

From these results, it is evident that the parallel design of a sorting network is much better suited to Emu's migratory thread architecture than that of a traditional comparison-based implementation. But what is more interesting is that these experiments highlight the efficiency of thread migration on the Emu Chick system and the importance of minimizing the overhead of thread creation.

2) Cyclic Versus Blocked Data Layout: In this section, we present and discuss the performance of the cyclic and blocked

TABLE I

BLOCKED DATA LAYOUT: HARDWARE CLOCK CYCLES (BILLIONS) WHEN SORTING 2^{21} ELEMENTS

	1 thread/nodelet	2 threads/nodelet	4 threads/nodelet	8 threads/nodelet	16 threads/nodelet	32 threads/nodelet	64 threads/nodelet
1 nodelet	612.2	306.8	155.6	81.3	46.2	**41.2**	41.4
2 nodelets	307.1	153.8	78.7	40.7	23.1	**20.7**	20.8
4 nodelets	153.6	77.1	39.4	20.4	11.6	**10.4**	10.5
8 nodelets	77	38.7	19.8	10.2	5.8	**5.2**	5.3

TABLE II

CYCLIC DATA LAYOUT: HARDWARE CLOCK CYCLES (BILLIONS) WHEN SORTING 2^{21} ELEMENTS

	1 thread/nodelet	2 threads/nodelet	4 threads/nodelet	8 threads/nodelet	16 threads/nodelet	32 threads/nodelet	64 threads/nodelet
1 nodelet	818.9	412.1	206.8	108.1	76.2	**75.2**	79.6
2 nodelets	446.2	223.4	113.1	58.8	38.5	**37.7**	37.9
4 nodelets	232.9	117.5	60.6	31.3	19.5	**19**	19.2
8 nodelets	119.8	60.5	31.4	16.2	**10**	10.2	16.3

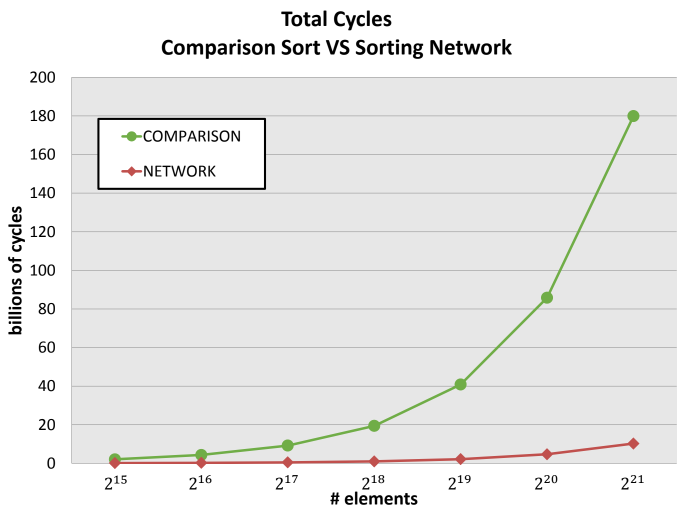

Fig. 3. Total cycles (in billions) for the comparison sort implementation and the sorting network implementation. Both implementations use a cyclic data layout. The sorting network utilized 8 nodelets with 32 threads per nodelet.

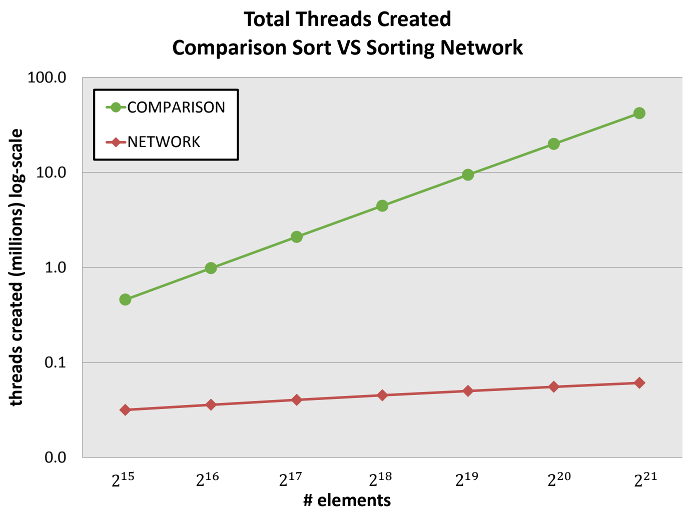

Fig. 5. Total number of threads created (in millions) for the comparison sort implementation and the sorting network implementation. Both implementations use a cyclic data layout. The sorting network utilized 8 nodelets with 32 threads per nodelet. The vertical axis is on a log-scale.

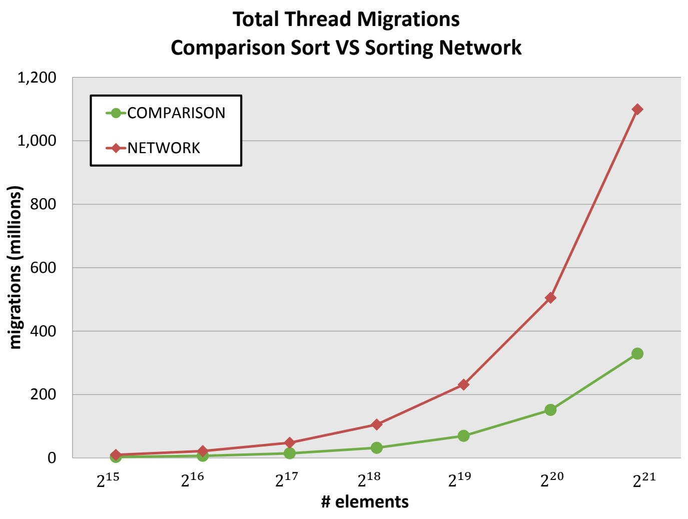

Fig. 4. Total number of thread migrations (in millions) for the comparison sort implementation and the sorting network implementation. Both implementations use a cyclic data layout. The sorting network utilized 8 nodelets with 32 threads per nodelet.

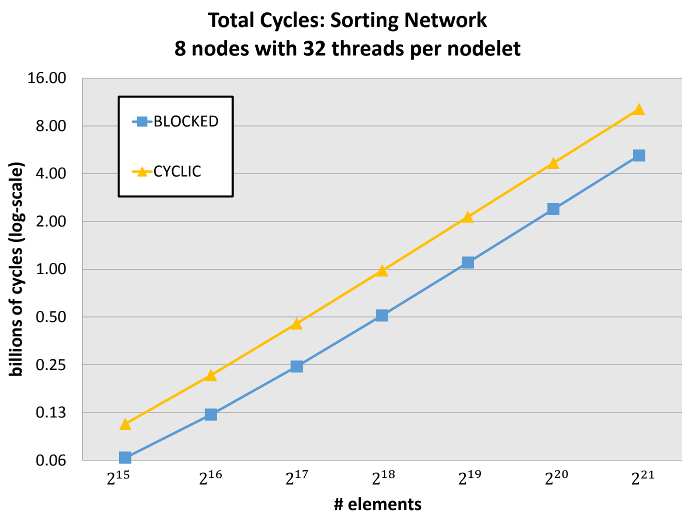

Fig. 6. Total cycles (in billions) for the blocked and cyclic data layouts when using 8 nodelets with 32 threads per nodelet. Note that the vertical axis is on a log-scale.

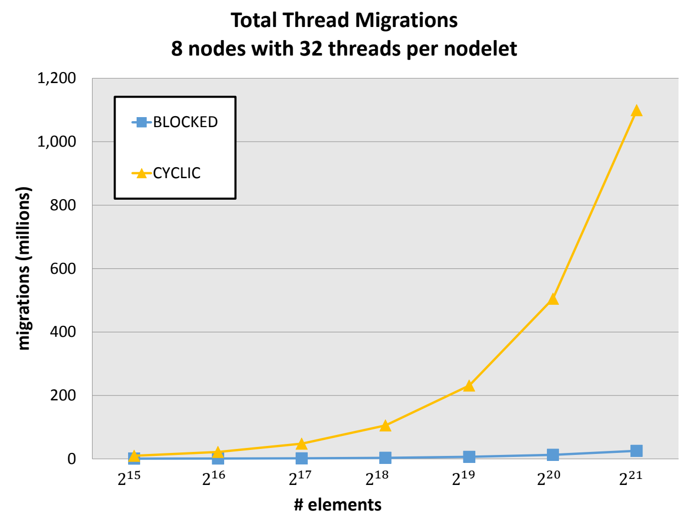

Fig. 7. Total thread migrations (in millions) for the blocked and cyclic data layouts when using 8 nodelets with 32 threads per nodelet.

data layouts for the sorting network implementation. As the data layouts only differ in how they store and access the input array in memory, the same number of threads are created in both approaches. Tables I and II present the total hardware clock cycles (in billions) required to sort 2^{21} elements when using the blocked and cyclic data layouts, respectively. The bold face values in each table indicate the optimal number of threads per nodelet. For the blocked data layout, the optimal number of threads per nodelet is 32, irrespective of the number of nodelets used. The best performing configuration is 8 nodelets with 32 threads per nodelet (256 threads total). With 64 threads per nodelet, we begin to reach the limits of the current Emu Chick system as it can only support 64 concurrent threads on a single nodelet due to only having one GC per nodelet. The same is true for the cyclic data layout results shown in Table II, except when using 8 nodelets, where the optimal number of threads per nodelet is 16. We also observed this trend across other data set sizes. However, the difference between those results and those when using 32 threads per nodelet was negligible (less than 5%). Therefore, for the remainder of our discussion when we compare the two data layouts, we will focus on the 8 nodelets with 32 threads per nodelet configuration for both. It is interesting to note that for the blocked data layout, the difference in clock cycles between 32 and 64 threads per nodelet when using 8 nodelets is negligible, while there is a 1.6x difference between these results for the cyclic data layout.

To further compare the cyclic and blocked data layouts for the sorting network, we measured their clock cycles and thread migrations across a range of data set sizes when using 8 nodelets with 32 threads per nodelet. Figure 6 presents the number of hardware clock cycles (in billions) required to sort 2^{15} to 2^{21} elements for the two different data layouts. We observe a consistent trend of the cyclic data layout requiring roughly 2x more cycles than the blocked layout. To further quantify the differences between these two data layouts, we measured the total number of thread migrations, which we present in Figure 7. It is evident that the blocked data layout is significantly more efficient at limiting the number of thread migrations than the cyclic layout, requiring up to 43.7x fewer migrations. Similar to what we observed in the previous section, we can see that while there is a significant difference in thread migrations between the layouts, the overall difference in clock cycles is much less severe. This further suggests that while unnecessary thread migrations on the Emu architecture should be avoided, the penalty of a thread migration is not as high as one might expect. We believe that this is largely due to the fact that the program context that is migrated is very small, as mentioned in Section III.

Based off of these results, it is clear that the blocked data layout is more efficient for a bitonic sorting network than a cyclic layout, both in terms of clock cycles and thread migrations. But again, these results underscore the efficiency of thread migrations on the Emu architecture.

VII. Conclusions and Future Work

In this work, we implemented parallel bitonic sort on a migratory thread architecture, namely the Emu Chick system. We explored the efficacy of a sorting network implementation when compared to an explicit comparison-based approach. Furthermore, we investigated two different data layouts for the sorting network, cyclic and blocked, and quantified their performance on the Emu architecture. Our findings can be summarized as follows:

- We found that while the sorting network exhibits an average of 19.2x fewer clock cycles than the comparison based approach, it required an average of 3.3x more migrations. However, the sorting network created up to 688x fewer threads, highlighting the importance of efficient thread creation strategies, which can be more beneficial to performance rather than reducing migrations.
- A blocked data layout for the sorting network out performed a cyclic layout by roughly 2x. However, the difference in thread migrations between the two layouts varied by as much as 43.7x, underscoring that while a proper data layout should serve to reduce thread migrations, the effect of migrations on overall performance is not as critical as previously thought.

In our future work, we plan to implement the efficient data layout for bitonic sort, as described by Ionescu and Schauser [19], where the data is remapped after different stages of the algorithm to ensure that all compare-and-exchange operations are performed locally. Also, we intend to expand our evaluation from a single-node environment on the Emu Chick system to a multi-node environment to understand the effects of the inter-node communication topology and hardware. Furthermore, investigating larger data sets is also of interest to us, as well as exploring other sorting algorithms besides bitonic sort.

Acknowledgements

The authors would like to thank Dr. John Dorband, Dr. Milton Halem and Dr. TC Tuan for their continued support for this research effort. In addition, the authors thank the Center for Hybrid Multicore Productivity Research (NFS Award Number 1439633), Laboratory of Physical Sciences and Emu Technology for their support and access to their computational resources.

References

[1] T. B. Rolinger, T. A. Simon, and C. D. Krieger, "Performance considerations for scalable parallel tensor decomposition," *Journal of Parallel and Distributed Computing*, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731517302897

[2] S. Smith, J. Park, and G. Karypis, "Sparse tensor factorization on many-core processors with high-bandwidth memory," *31st IEEE International Parallel & Distributed Processing Symposium (IPDPS'17)*, 2017.

[3] T. Rolinger, T. Simon, and C. Krieger, "Parallel sparse tensor decomposition in chapel," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2018.

[4] 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.

[5] N. Bell and J. Hoberock, "Thrust: A productivity-oriented library for cuda," in *GPU computing gems Jade edition*. Elsevier, 2011, pp. 359–371.

[6] P. M. Kogge, "Execube-a new architecture for scaleable mpps," in *1994 International Conference on Parallel Processing Vol. 1*, vol. 1, Aug 1994, pp. 77–84.

[7] J. Draper, J. Chame, M. Hall, C. Steele, T. Barrett, J. LaCoss, J. Granacki, J. Shin, C. Chen, C. W. Kang, I. Kim, and G. Daglikoca, "The architecture of the diva processing-in-memory chip," in *Proceedings of the 16th International Conference on Supercomputing*, ser. ICS '02. New York, NY, USA: ACM, 2002, pp. 14–25. [Online]. Available: http://doi.acm.org/10.1145/514191.514197

[8] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, J. Brockman, A. Srivastava, W. Athas, V. Freeh, J. Shin, and J. Park, "Mapping irregular applications to diva, a pim-based data-intensive architecture," in *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, ser. SC '99. New York, NY, USA: ACM, 1999. [Online]. Available: http://doi.acm.org/10.1145/331532.331589

[9] 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.

[10] T. Dysart, P. Kogge, M. Deneroff, E. Bovell, P. Briggs, J. Brockman, K. Jacobsen, Y. Juan, S. Kuntz, R. Lethin, J. McMahon, C. Pawar, M. Perrigo, S. Rucker, J. Ruttenberg, M. Ruttenberg, and S. Stein, "Highly scalable near memory processing with migrating threads on the emu system architecture," in *Proceedings of the Sixth Workshop on Irregular Applications: Architectures and Algorithms*. Piscataway, NJ, USA: IEEE Press, 2016, pp. 2–9. [Online]. Available: https://doi.org/10.1109/IA3.2016.7

[11] M. Minutoli, S. K. Kuntz, A. Tumeo, and P. M. Kogge, "Implementating radix sort on emu 1," in *The 3rd Workshop on Near-Data Processing (WoNDP)*, 2015.

[12] E. Hein, T. Conte, J. Young, S. Eswar, J. Li, P. Lavain, R. Vuduc, and J. Riedy, "An initial characterization of the emu chick," in *The 8th International Workshop on Accelerators and Hybrid Exascale Systems (AsHES)*, 2018.

[13] 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.

[14] K. E. Batcher, "Sorting networks and their applications," in *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, ser. AFIPS '68 (Spring). New York, NY, USA: ACM, 1968, pp. 307–314. [Online]. Available: http://doi.acm.org/10.1145/1468075.1468121

[15] D. Nassimi and S. Sahni, "Bitonic sort on a mesh-connected parallel computer," *IEEE Transactions on Computers*, vol. C-28, no. 1, pp. 2–7, Jan 1979.

[16] H. Peters, O. Schulz-Hildebrandt, and N. Luttenberger, "Fast in-place sorting with cuda based on bitonic sort," in *Parallel Processing and Applied Mathematics*, R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 403–410.

[17] ——, "A novel sorting algorithm for many-core architectures based on adaptive bitonic sort," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, May 2012, pp. 227–237.

[18] Y. C. Kim, M. Jeon, D. Kim, and A. Sohn, "Communication-efficient bitonic sort on a distributed memory parallel computer," in *Proceedings. Eighth International Conference on Parallel and Distributed Systems. ICPADS 2001*, 2001, pp. 165–170.

[19] M. F. Ionescu and K. E. Schauser, "Optimizing parallel bitonic sort," in *Proceedings 11th International Parallel Processing Symposium*, Apr 1997, pp. 303–309.

[20] C. E. Leiserson, "Programming irregular parallel applications in cilk," in *Solving Irregularly Structured Problems in Parallel*, G. Bilardi, A. Ferreira, R. Lüling, and J. Rolim, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 61–71.