Exploring the Smart Data Layout of Parallel Bitonic Sort on a EMU/Lucata Migratory Thread Architecture

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

* 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

***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

Eric An initial characterization of the Emu Chick

tom Optimizing Data Layouts for Irregular Applications on a Migratory Thread Architecture

2018 IEEE High Performance Extreme Computing Conference, Exploring Parallel Bitonic Sort on a Migratory Thread Architecture

III. The Emu Chick Architecture

The Emu architecture is a scalable near-memory processing system that employs migrating threads. The program context migrates to the data that is to be accessed, rather than fetching data from memory. 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 [6]. 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 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 [7]. 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. The Emu Chick system supports both a single-node and multi-node execution configuration. Figure 1 shows the general layout of a single node in the Emu Chick system,

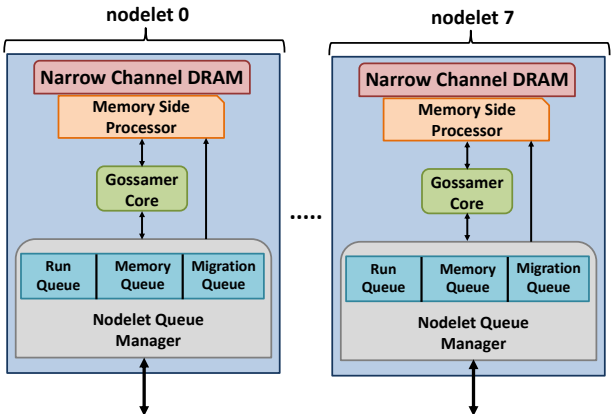

Fig. 1. A single node in the current Emu architecture. "GC" refers to a Gossamer core. The migration engine provides communication between other nodelets. There are 8 nodelets within a single node.

which is identical to the system used by Hein et al. in their study [6]. Note that in the current system, there is only 1 GC per nodelet and the clock rate of each GC is 175MHz. 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 [6]–[8]. In this work, we focus our experiments within a multi-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 [9]. 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 `mw_malloc1d` and `mw_malloc2d` routines. The `mw_malloc1d` routine allocates an array where elements are striped across the nodelets in a round-robin fashion. On the other hand, `mw_malloc2d` allows programmers to place contiguous elements of an array on the same nodelet, providing a blocked data layout.

IV. Bitonic Sort

The bitonic sorting network is not dependent on the input data and has a very low algorithmic overhead. The sequence of comparison is fixed in this sorting algorithm. The parallel nature of this algorithm is best suited for the parallel machines like EMU.

In our previous publication, we have discussed in detailed on the bitonic sorting network implementation and the iterative comparison-based approach. Our bitonic sorting implementation is based largely on the description provided Batcher [10] as well as Ionescu and Schauser [11].

In this section, we describe our C programming based implementation of the bitonic sort algorithm for the Emu Chick system and used Cilk for parallelization.

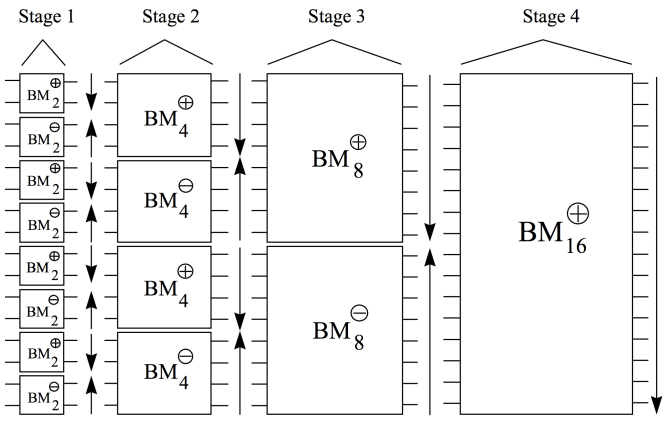

Fig. 2. Bitonic sorting network for an input of 16 elements taken from [11]. $BM_k^{\oplus}$ and $BM_k^{\ominus}$ refer to an increasing and decreasing bitonic merge of size k, respectively.

V. Implementing smart remapping layout on the Emu Chick

In this work, we implemented the smart remapping layout of the parallel bitonic sort on a migratory thread architecture, namely the Emu Chick system.

we implemented the efficient data layout for bitonic sort, as described by Ionescu and Schauser [11], where the data is remapped after different stages of the algorithm to ensure that all compare-and-exchange operations are performed locally. Also, we expand our evaluation to a multi-node environment on the Emu Chick system to understand the effects of the inter-node communication topology and hardware. Furthermore, we investigated larger data sets to show the reliability and the scalability of the hardware.

In smart data layout, one can alternate between blocked and cyclic layouts to ensure that all compare-exchange operations are performed locally [11].

We obtained the number of thread migrations by running our codes in the Emu simulator, as the hardware can only measure clock cycles. We explored the efficacy of a smart layout when compared to our previous blocked and cyclic data layout , and quantified their performance on the Emu architecture. Our findings can be summarized as follows:

A. Spawn Strategy

Explaining the linear, recursive and the recursive linear spawns

B. Smart Vs Dumb

Explaining the with and without remapping comparison to show the performance improvement.

C. Remap Threads and Comparison Threads

Explaining the working of Remap Threads and Comparison Threads.

D. Inner Loop Parallelism

Explaining the working of Inner Loop Parallelism.

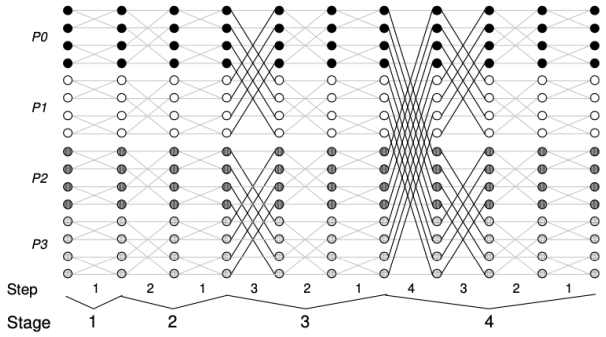

Fig. 3. A. Blocked data layout for 16 elements on 4 processors / nodelets. The shading of the elements reflect the allocation onto the nodelets [12]

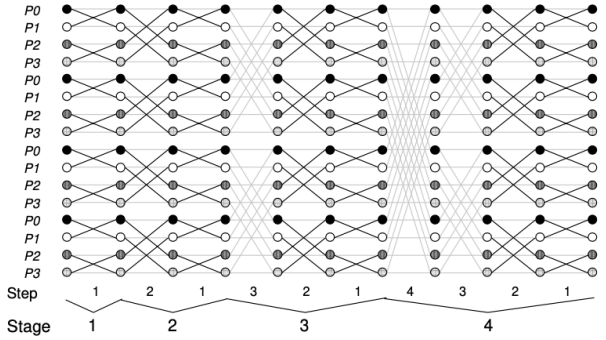

Fig. 4. B. Cyclic data layout for 16 elements on 4 processors / nodelets. The shading of the elements reflect the allocation onto the nodelets [12]

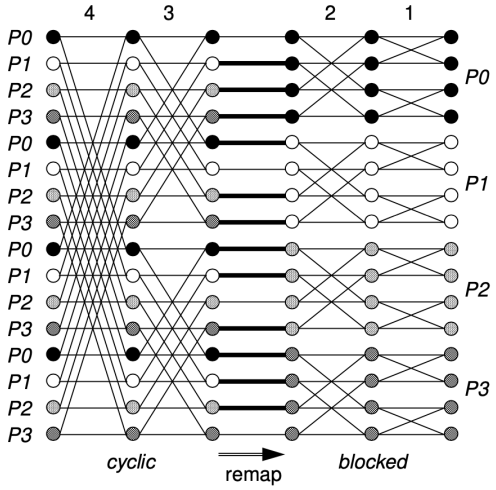

Fig. 5. C. Remap from a cyclic to a blocked data layout with 4 processors / nodelets and 16 elements [12].

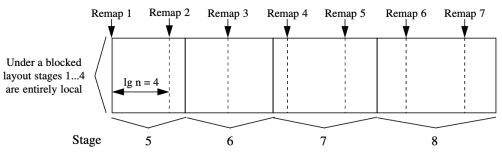

Fig. 6. D. Example of the remapping strategy for 256 elements on 16 nodelets starting with a blocked layout [12].

Fig. 7. A. B. C. D. Bitonic sorting algorithm as proposed by Mihai Florin Ionescu [12]. Grey arcs indicate local access while black arcs indicate remote communication.

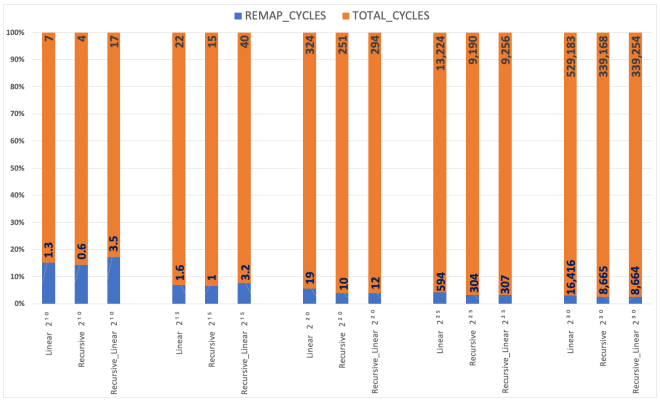

Fig. 8. Comparing the performance of linear Vs recursive Vs recursive linear spawns. Values represent total cycles (in millions) with 64 remap threads and 64 comparison threads when using the smart layout with remapping.

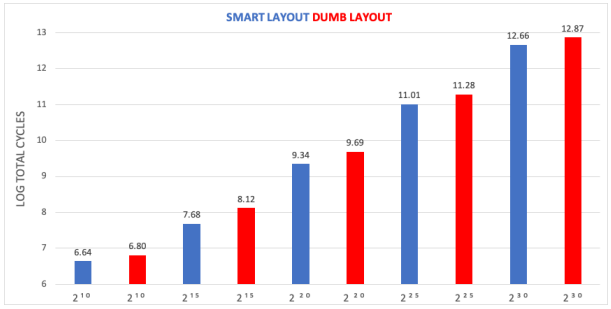

Fig. 9. using 1 remap thread and 1 comparison thread

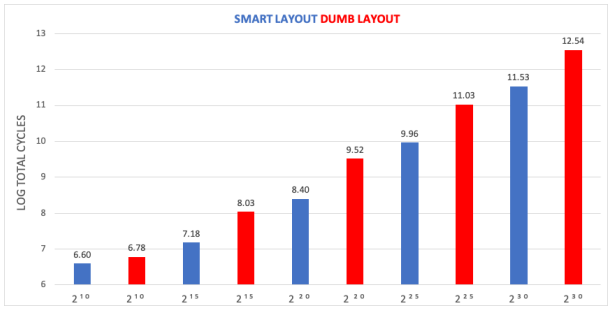

Fig. 10. using 64 remap threads and 64 comparison threads.

Fig. 11. A. B. Comparing the performance of with (smart) and without (dumb) remapping using recursive spawns. Note that the vertical axis represent the total clock cycles on a log - scale.

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.

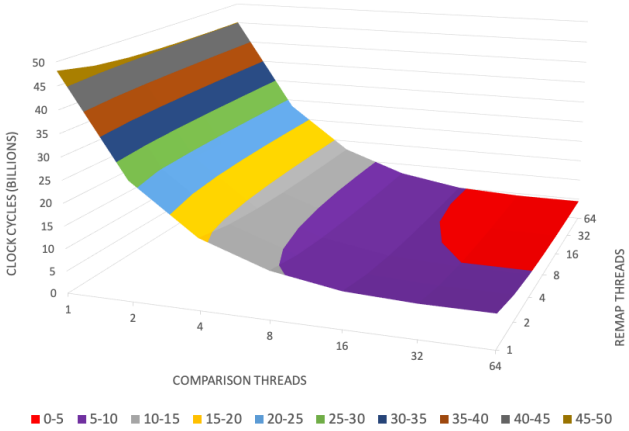

Fig. 12. Comparing the performance of remap threads and comparison threads using smart layout and recursive spawns for sorting 2^{24} elements. Vertical axis represent total clock cycles in billions.

A. Experimental Setup

Our codes were built using version emu-20.01-hotfix of the Emu toolchain. The Emu hardware has the following configuration. chick-helpers version is v6.0.0. sc-driver-tests version is v4.1.1 and ncdimm version is v3.2.0. 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^{30}. The metric used to measure the performance is clock cycles. As the current Emu Chick system is running at a reduced clock speed of 175Mhz, measuring clock cycles rather than absolute runtime provides a more suitable metric for comparisons. All our cycle counts were taken with respect to the sorting routine, which excludes any file I/O and pre-processing.

For this work, we utilized the multi node configuration on the Emu Chick system (hardware), which has 8 nodes and each node has 1 gossamer cores. In our implementation of the bitonic sorting network, the number of comparison threads per nodelet and the number of remap threads per nodelet are tunable parameters varying from 1 to 64.

B. Results

In this section, we present and discuss our results as they pertain to the following questions: (1) how does the smart layout implementation of bitonic sort compare to a dumb layout. Our previous work [13] showed that the blocked data layout outperformed the cyclic data layout by 2x, hence the dumb layout here represent the blocked data layout without any remapping to other layouts and smart layout represents remapping from block to cyclic data layouts. (2) how do different data layouts affect performance for the sorting network approach. (3) Which thread spawning strategy provides the best performance for the hardware. Linear Vs Recursive Vs Recursive Linear (4). what is optimal number for comparison threads and remap threads. (5). What percentage of the work

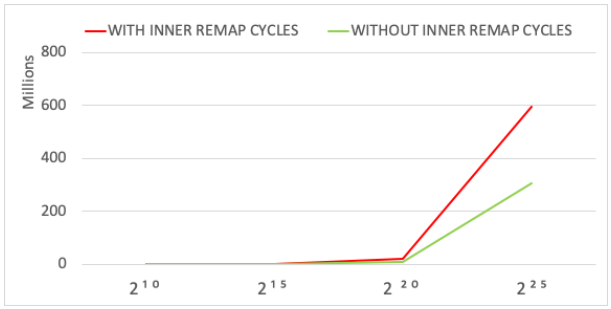

Fig. 13. Remap clock cycle performance

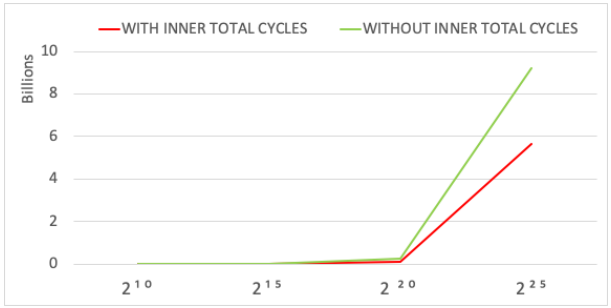

Fig. 14. Total clock cycle performance

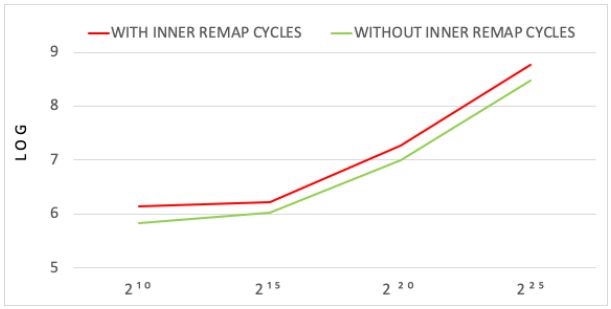

Fig. 15. Remap clock cycle performance in Log Scale

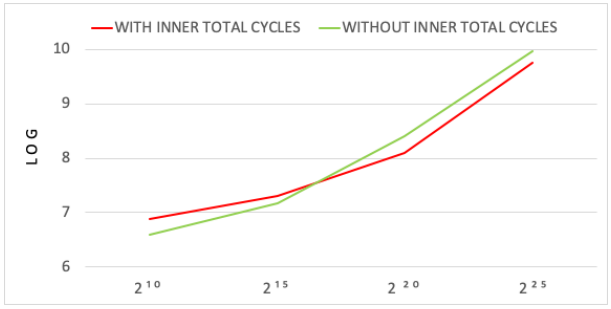

Fig. 16. Total clock cycle performance in Log Scale

Fig. 17. Comparing the performance of with and without inner loop parallelism using recursive spawns using 64 remap threads and 64 comparison threads.

is spent on remapping for the above combinations. (6) How much benefit can we get using inner loop parallelism. These questions aims to gain insight into the appropriate structure for sorting algorithms on a migratory thread architecture and quantify the importance of data layout on such an architecture for many other applications.

1) Recursive Spawns Versus Linear Spawns: Figure 8 presents the total number of cycles (in millions) required to sort 2^{10} to 2^{30} elements for the smart layout based implementation of bitonic sort. Here, the remap threads and comparison threads are set at constant 64 throughout. We observe that the linear spawns approach requires approximately 2x more total clock cycles on average than the recursive spawns. The same holds true for the clock cycles spent on the remapping part as well. This shows the importance of having an efficient thread creation strategies, which can be more beneficial to performance. Just as an experiment, we tried a combination of recursive and linear spawns which did not perform any better than recursive spawns. Also, the percentage of clock cycles spent on remapping the data layout is very small and decreases further as we increase the size of the elements to sort. For the recursive case, the remap percentage for sorting 2^{10} elements is 18%, 2^{15} elements is 15%, 2^{20} elements is 8%, 2^{25} elements is 6% and 2^{30} elements is 3%.

Therefore, for the remainder of our discussion, we will focus only on the recursive spawns approach.

2) Smart Layout Versus Dumb Layout: Figure 11 presents the total number of cycles (in log scale) required to sort 2^{10} to 2^{30} elements using the smart layout and the dumb layout implementation of bitonic sort. In figure 9 the remap threads and comparison threads are set to constant 1 and in figure 10 the remap threads and comparison threads are set to constant 64. When the threads were set to 1, the dumb layout requires roughly 2x more total clock cycles than the smart layout. When the threads were set to 64, the dumb layout requires roughly 10x more total clock cycles than the smart layout. Clearly, the smart layout has a better performance than the dumb layout. Note that in dumb layout there are no remappings involved and the smart layout includes a percentage of cycles spent on the remapping. Therefore, for the remainder of our discussion, we will focus only on the smart data layout with recursive spawns approach.

3) Optimal threads for remapping and comparison: Figure 12 presents the total number of cycles (in billions) required to sort 2^{24} elements using the smart layout and recursive spawns. Here the comparison threads and the remapping threads are varied from 1 to 64. Here the performance plateaus after 32 comparison threads and 32 remap threads. Table I shows the exact values on this region. To be precise, 64 remap threads and 64 comparison threads provided the lowest clock cycles. With 64 threads per nodelet, we begin to reach the limits of the current Emu Chick hardware as it can only support 64 concurrent threads on a single nodelet due to only having one Gossamer Core per nodelet. It is interesting to note that for the smart data layout, the difference in clock cycles between 32 and 64 threads per nodelet when using 8 nodelets is negligible.

TABLE I
TOTAL CLOCK CYCLES (BILLIONS) WHEN SORTING 2^{24} ELEMENTS FOR DIFFERENT REMAP THREADS AND COMPARISON THREADS

Remap Threads	Comparison Threads		
	16	**32**	**64**
8	5.32	4.69	4.68
16	5.13	4.49	4.49
32	5.07	4.44	4.43
64	5.08	4.44	4.43

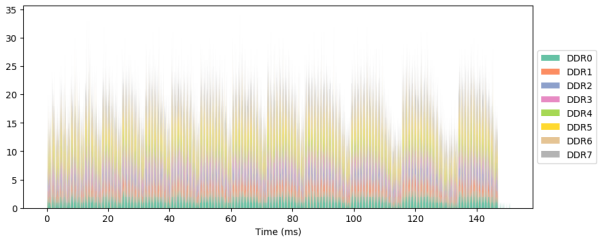

Fig. 18. With inner loop parallelism

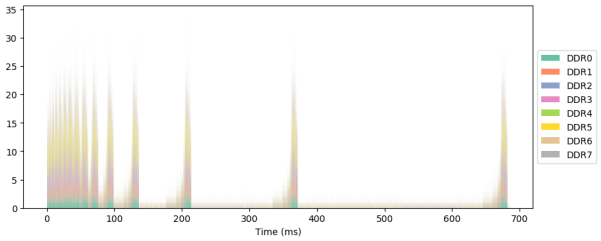

Fig. 19. Without inner loop parallelism

Fig. 20. Comparing the performance of DDR utilization.

Based off of these results, it is clear that the smart data layout with recursive spawns approach using 64 comparison and remap threads, is efficient for a bitonic sorting network with remapping data layout strategy in terms of clock cycles.

4) Optimal threads for remapping and comparison: Figure 17 presents the performance of remap clock cycles and total clock cycles with using inner loop parallelism and without using inner loop parallelism. Even though in figure 13 the clock cycles spent just on remapping the data layout increased after 2^{15} elements with inner loop parallelism compared to without using inner loop parallelism, the total clock cycles was significantly lower when using inner loop parallelism 14. Figure 15 shows the clock cycles spent just on remapping the data layout in log scale. Figure 16 shows the total clock cycles in log scale. Here, we can clearly see the cross over point beyond 2^{15} elements after which using inner loop parallelism takes lesser clock cycles than without using inner loop parallelism.

Figure 20, 23, 26, 29, 32, 35 shows the advantage of using inner loop parallelism from the profiler statistics.

VII. CONCLUSIONS

In this work, we implemented parallel bitonic sort on a migratory thread architecture, namely the Emu Chick system.

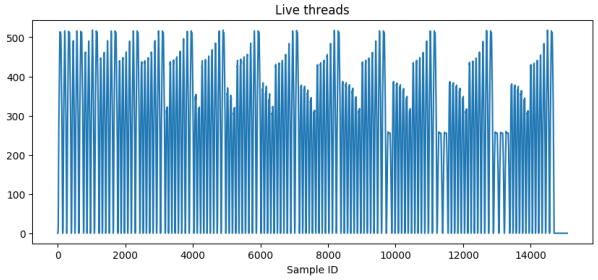

Fig. 21. With inner loop parallelism

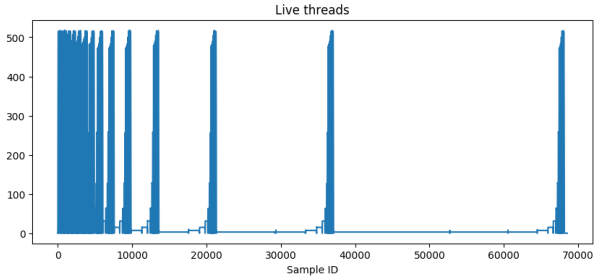

Fig. 22. Without inner loop parallelism

Fig. 23. Comparing the performance of live threads utilization.

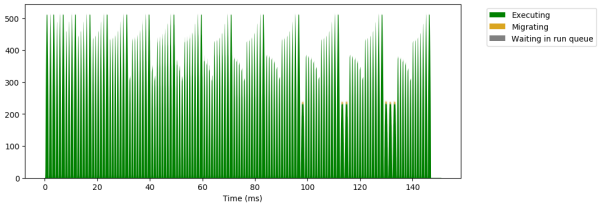

Fig. 24. With inner loop parallelism

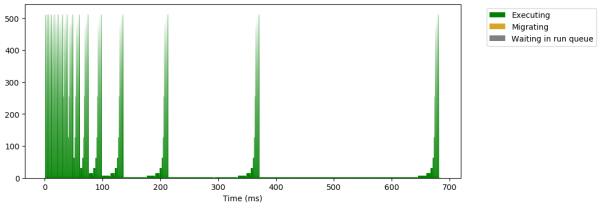

Fig. 25. Without inner loop parallelism

Fig. 26. Comparing the performance of threads activity.

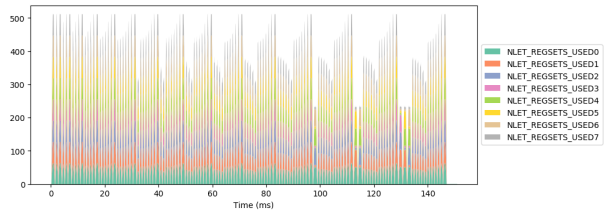

Fig. 27. With inner loop parallelism

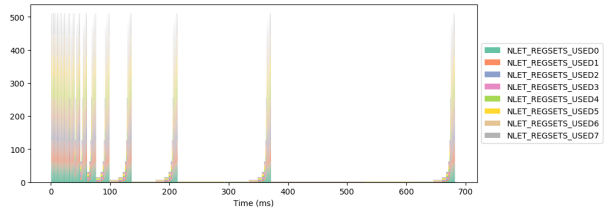

Fig. 28. Without inner loop parallelism

Fig. 29. Comparing the performance of NLET REGSETS USED.

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.

ACKNOWLEDGEMENTS

The authors would like to thank Laboratory of Physical Sciences and EMU/Lucata 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.

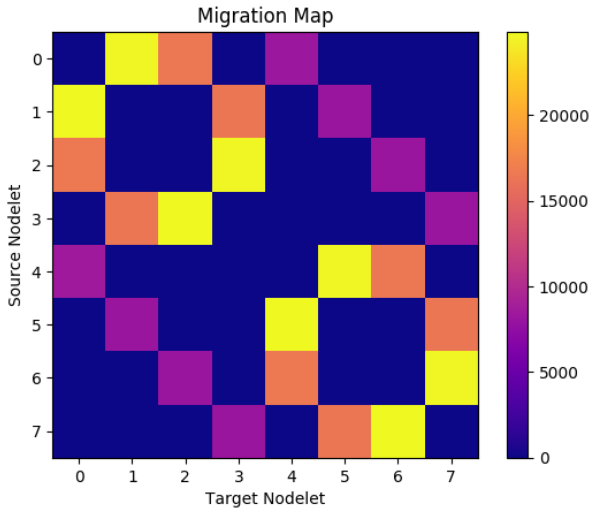

Fig. 30. With inner loop parallelism

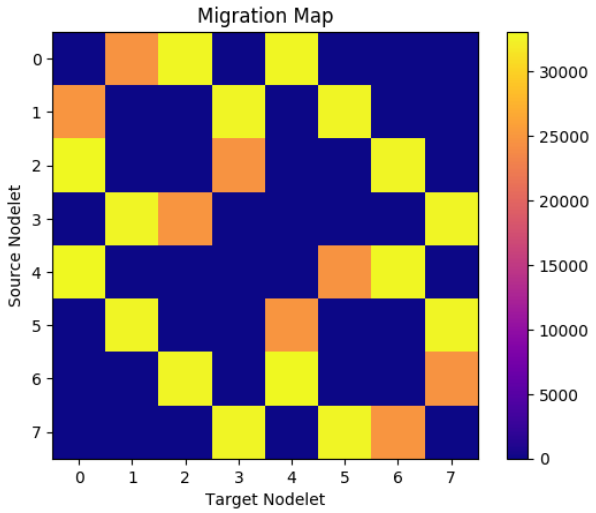

Fig. 31. Without inner loop parallelism

Fig. 32. Comparing the performance of migration map.

[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] 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.

[7] 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.

[8] 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

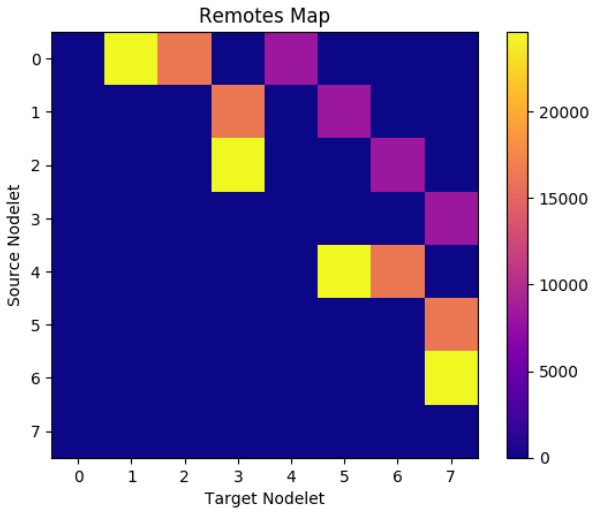

Fig. 33. With inner loop parallelism

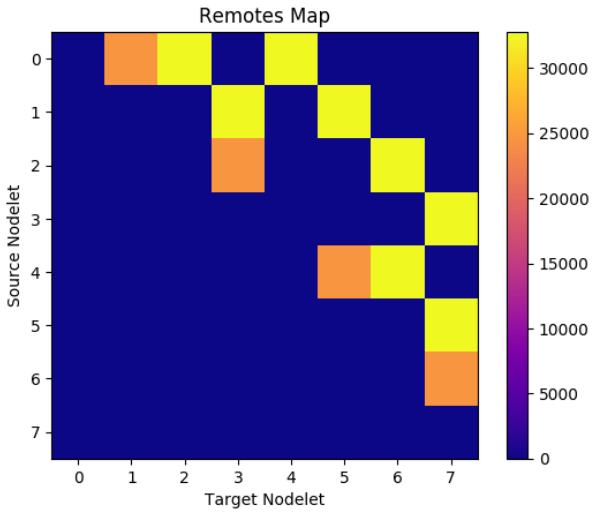

Fig. 34. Without inner loop parallelism

Fig. 35. Comparing the performance of remote map.

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

[9] 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.

[10] 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

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

[12] M. F. Ionescu, "Thesis on optimizing parallel bitonic sort," in *Master's thesis UCSB*, Aug 1996.

[13] J. M. T. S. Kaushik Velusamy, Thomas Rollinger, "Exploring parallel bitonic sort on a migratory thread architecture," in *2018 IEEE High Performance Extreme Computing Conference*, Aug 2018.