Fixed-Function Hardware Sorting Accelerators for Near Data MapReduce Execution

Seth H. Pugsley
Intel Labs, OR, USA
seth.h.pugsley@intel.com

Arjun Deb
University of Utah, UT, USA
arjundeb@cs.utah.edu

Rajeev Balasubramonian
University of Utah, UT, USA
rajeev@cs.utah.edu

Feifei Li
University of Utah, UT, USA
lifeifei@cs.utah.edu

***Abstract*—**

A large fraction of MapReduce execution time is spent processing the Map phase, and a large fraction of Map phase execution time is spent sorting the intermediate key-value pairs generated by the Map function. Sorting accelerators can achieve high performance and low power because they lack the overheads of sorting implementations on general purpose hardware, such as instruction fetch and decode. We find that sorting accelerators are a good match for 3D-stacked Near Data Processing (NDP) because their sorting throughput is so high that it saturates the memory bandwidth available in other memory organizations. The increased sorting performance and low power requirement of fixed-function hardware lead to very high Map phase performance and energy efficiency, reducing Map phase execution time by up to 92%, and reducing energy consumption by up to 91%. We further find that sorting accelerators in a less exotic form of NDP outperform more expensive forms of 3D-stacked NDP without accelerators. We also implement the accelerator on an FPGA to validate our claims.

I. Introduction

Many commercial workloads rely on efficient processing of very large datasets. Several hardware innovations are required to grapple with the challenges of big data workloads. For example, Wu et al. [1] propose a processing unit (Q100) that is customized for database workloads, Lim et al. [2] propose the use of memory blades to expand server memory capacities, and Pugsley et al. [3] design a near data processing architecture for in-memory MapReduce workloads.

These examples demonstrate that while we do need efficient processing units for big data workloads, we also need efficient memory systems. In this work, we propose an architecture that targets both of these aspects. We start with a near-data processing architecture that offers very high memory bandwidth. We then identify a function that is frequently exercised and design an accelerator for it. The benefit from this accelerator is amplified by its high-bandwidth connection to memory.

In this paper, we focus on in-memory MapReduce workloads. MapReduce [4] is a very popular framework used to analyze large datasets. While MapReduce has traditionally been employed for disk-resident datasets, it is increasingly being used to process in-memory datasets. The Map and Reduce functions in MapReduce workloads can take arbitrary forms. Therefore, they are poor candidates for custom accelerators and are best handled by general-purpose processors. However, we show that most MapReduce workloads have a few common functions, such as the sorting of intermediate outputs after the Map phase, that can greatly benefit from acceleration. In a near data processing architecture with highly optimized Map and Reduce phases, the sort function emerges as a significant bottleneck.

This paper explores the design space of sorting accelerators, and we show that an accelerator that performs merge sort on eight input streams has very low overheads and offers performance that is very close to more sophisticated, idealized sort algorithms. The merge sort-based accelerators reduce overall Map phase execution time from 78% to 92%, and reduce Map phase compute and memory energy consumption from 77% to 91%, compared to performing the sort function in software on a near-data core.

This work was supported in part by NSF grants CNS-1302663 and CNS-1423583.

II. Background

A. Near Data Processing

Near Data Processing (NDP) refers to the tight coupling of memory and compute resources in order to reduce the overheads of data movement and remove performance bottlenecks [5]. In this work, we consider two implementations of NDP, one where compute and memory resources are vertically integrated with 3D stacking technology using through-silicon vias (TSVs), and one using conventional compute and memory chips that are paired with one another in a 1:1 ratio, and arranged on DIMM-like memory module boards. Both of these implementations have previously been proposed as platforms for running MapReduce workloads [6], and can be seen in Figure 1.

In 3D-stacked NDP, several memory dies are stacked on top of a single logic die, similar to Micron's Hybrid Memory Cube (HMC) [7], but with the addition of 16 low power cores, one for each of the HMC's 16 independent vaults of memory. In module-based NDP, quad-core SoCs are paired with individual x32 LPDDR2 chips to approximate the advantages of 3D-stacked NDP without using expensive 3D stacking. We refer to these styles of NDP as NDP-HMC, and NDP-Module respectively. Because NDP-HMC has more bandwidth per core, its performance will be higher than NDP-Module.

B. MapReduce Execution

MapReduce is a programming framework designed to efficiently process large amounts of data with a high degree of parallelism. An input data set is typically a set of independent database records. The programmer writes a Map function which consumes these records and produces key-value pairs as an intermediate output. The programmer also writes a Reduce function, which consumes these intermediate key-value pairs and produces the final output.

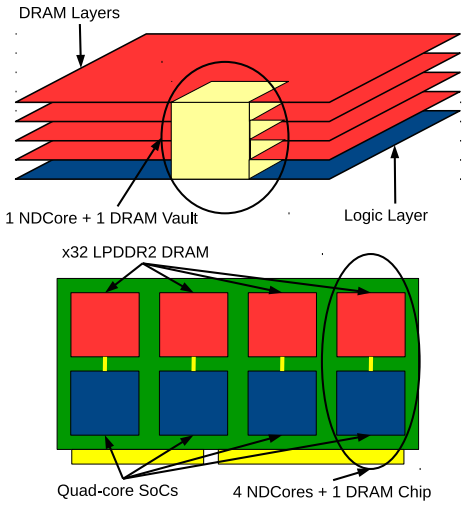

Fig. 1. Two previously proposed implementations of Near Data Processing (NDP): HMC-based NDP (above), and memory module-based NDP (below).

The Map and Reduce functions can be anything, and vary by application domain, so they therefore require general programmability. Each Map task produces a wide range of intermediate output, but each Reduce task consumes only a narrow range of the Map phase's output. In between the Map and Reduce phases, the intermediate output is Shuffled between Map tasks and Reduce tasks. To reduce communication during this Shuffle phase, each Map task performs a Combine function, which is like a localized Reduce function. However, before the Combine can be done, the intermediate data must first be Sorted. Because Sorting is a part of most MapReduce workloads, it is a good candidate for hardware acceleration.

C. Map Phase Execution Breakdown

Among workloads that include Sort and Combine phases, Sort takes the largest fraction of Map task execution time, by a large margin, as seen in Figure 2. Because RangeAgg does not have Sort or Combine phases, and can therefore not benefit from Sort acceleration, we will not consider it in our forthcoming results. In the other workloads, the Sort phase dominates execution time, taking 84-94% of the total Map time. Earlier work evaluating MapReduce workloads on NDP architectures [3][6] shows that the Map phase typically accounts for 80-90% of MapReduce execution, further reinforcing the importance of accelerating Sort.

III. Related Work

Several recent works have focused on developing efficient architectures for big data workloads. As we have already referenced, Near Data Processing, by Pugsley et al. [3][6], evaluates an optimized, throughput-oriented baseline with a high bandwidth HMC-based memory system, and then improve upon it by moving some of the computation to the memory devices themselves. The Mercury and Iridium architectures, proposed by Gutierrez et al. [8], use 3D-stacked memory and logic devices to create an efficient and high performing key-value store, with a primary goal of increasing memory density. Meet the Walkers, by Kocberber et al. [9] uses specialized hardware to accelerate the data access itself in order to improve the process of building database indices. The HARP accelerator

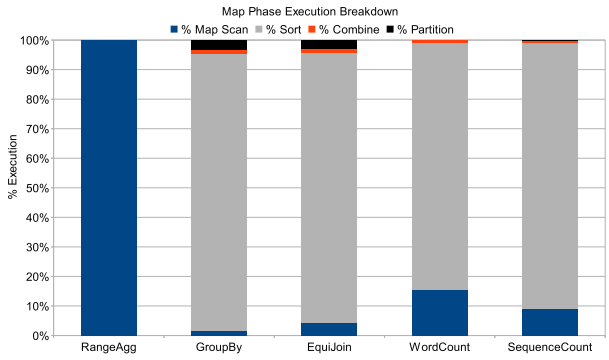

Fig. 2. Execution breakdown of a Map task. Other than in RangeAgg, sorting dominates execution time.

by Wu et al. [10], and its follow-up, the Q100 database processor [1] by Wu et al., use hardware accelerators to improve the execution of common database operations. A recent paper by Farmahini-Farahani [11] considers a general-purpose, coarse-grained reconfiguratable accelerator (CGRA) framework that delivers higher performance by placement within a 3D-stacked memory device. The Tesseract architecture by Ahn et al. [12] executes graph algorithms in-memory across networks of 3D-stacked memory+compute devices.

IV. Hardware Sorting Strategies

A. Sorting on Keys

The Sort phase in a MapReduce workload deals in arrays of key-value pairs, not just integers. The key in a key-value pair is often an integer, but it may be something else, according to the domain of the currently executing workload. Even in instances where the key represents a floating point number, or even a non-number entity, it can still make sense to sort the keys as if they were integers. The purpose of the Sort phase is not to create a lexicographically "correct" ordering, but instead the purpose is to create any ordering where all instances of each key appear consecutively. Treating all keys as integers, and sorting on those "integers" accomplishes this goal.

B. Hardware Merge Sort

Merge sort works by interleaving the items from multiple sorted input lists into a larger sorted output list. At each step of the algorithm, the least of the items at the head of all the input lists is moved into the output list. In this work, we consider hardware merge sorters that merge just two input lists of key-value pairs, and eight input lists of key-value pairs.

1) Two-Input Merge Sorter: In this implementation, the input stream buffers must keep track of two sorted input lists. On each cycle, the sorter compares its two input keys, then moves the smaller of the two key-value pairs onto its output. This is more efficient than the software implementation of merge sort running on a general purpose core, because an accelerator can sort one item every cycle.

It takes a number of cycles equal to the number of input key-value pairs to complete one iteration of merge sort, and a number of iterations equal to log base two of the number of input key-value pairs to completely sort the entire data set.

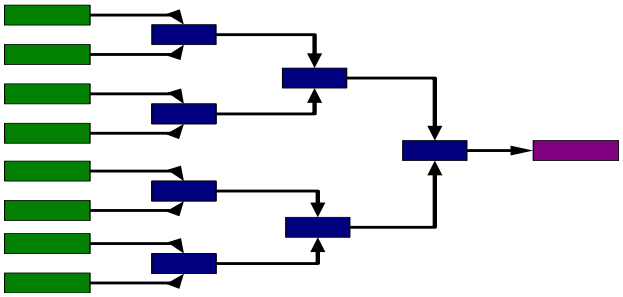

Fig. 3. A Merge Sort unit that sorts eight input lists.

	Area	Power	Frequency
MergeSort2	0.005 mm^2	1.2 mW	1.0 GHz
MergeSort8	0.035 mm^2	8.4 mW	1.0 GHz
BitonicSort+RP	1.130 mm^2	81.3 mW	400 MHz (BS), 300 MHz (RP)
BitonicSort+MS8	0.223 mm^2	60.9 mW	400 MHz (BS), 1.0 GHz (MS)

TABLE I. HARDWARE SORT COMPONENT PARAMETERS.

2) Eight-Input Merge Sorter: In this implementation, seen in Figure 3, the input stream buffers must keep track of eight sorted input lists. This sorter is internally organized like a heap, and each cycle the smallest of two input key-value pairs will be chosen to advance to the next level. This means that each cycle, the smallest of all eight input lists will appear at the output, and be written to memory. Like the two-input merge sorter, it takes a number of cycles equal to the number of input key-value pairs to complete one iteration of merge sort. However, the eight-input merge sorter takes only log base eight iterations to completely sort the entire data set.

C. Hardware Bitonic Sort

In this work, we also consider a Bitonic Sort unit that can sort 1024 inputs, similar to Q100 [1]. Sorting inputs larger than 1024 items will require additional effort and other techniques. We consider two such complementary techniques. In the first technique, Bitonic Sort is used as a finishing step to finalize the sorting that has been partially accomplished by a hardware range partitioning unit. In the second technique, Bitonic Sort is used as a pre-processing step to prepare an input to be operated on by an eight-input hardware merge sorter.

D. Summary of Hardware Sorting Unit Characteristics

Table I shows area, power, clock speed, and sorting throughput figures for all hardware sorters considered in this work. The Merge Sort units are trivially small, even compared to our 0.51 mm^2 general purpose NDCores. The Range Partitioner, however, is approximately 1.85× the size of an NDCore, and the Bitonic Sorter is approximately 37% the size of an NDCore.

V. EVALUATION

A. Methodology

In this work we evaluate the performance and energy characteristics of only the Map phase of MapReduce workloads. This has previously been identified as the largest phase in MapReduce execution on NDP architectures [3]. We evaluate four different workloads:

GroupBy Aggregation scans the website log for the 1998 Word Cup website [13] and counts the instances of each unique IP address. **Self Equi-Join Aggregation** uses the same website log dataset as GroupBy to perform a self equi-join operation. **WordCount** scans HTML files from Wikipedia [14] and counts the number of instances of each unique word. **SequenceCount** uses the same input as WordCount, but counts the number of instances of each unique set of three consecutive words.

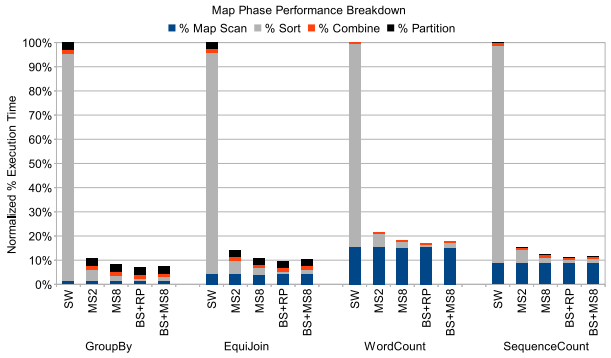

Fig. 4. The amount of time spent in each phase of a Map task, normalized to the software case running on NDP.

We model a low-power, in-order core with performance and energy characteristics similar to the ARM Cortex-A5 operating at 1 GHz and consuming 80 mW of power. We use WindRiver Simics [15] to model the processor core, and USIMM [16] to model DRAM access latency and bandwidth. We augment the core with the various hardware sorting units described in Table I.

Each NDNode in the system has access to a private vault of memory, and does not share resources with any other NDNode when using it. This means that all of the bandwidth of each vault can be dedicated to a single hardware sorting unit. We assume each vault has a 32-bit internal data bus operating at 1600 Mbps per bus wire, for a maximum bandwidth of 6400 MB/s.

B. Performance Evaluation

1) Map Task Latency: Figure 4 shows the breakdown of the Map phase as executed first with software running on NDP, and then with the various accelerators. Although the highest performing hardware sorter was able to offer two orders of magnitude performance over the software case handling the Sort phase of a Map task, that does not directly translate into two orders of magnitude of overall Map task performance.

After sort has been accelerated so much, it eventually ceases to be a substantial bottleneck in the system, and improving it further offers negligible additional gains. For example, for WordCount, when considering the Map phase as a whole, MergeSort2 reduces execution latency by 78%, MergeSort8 reduces execution latency by 81%, and the BitonicSort combination is able to reduce execution latency by 83%. Given the area and power advantages of MergeSort8, we believe it is the most compelling design point.

2) Map Task Energy: Next, we consider total energy consumption of the memory and compute systems during Map phase execution, including both the Sort phase, and non-Sort

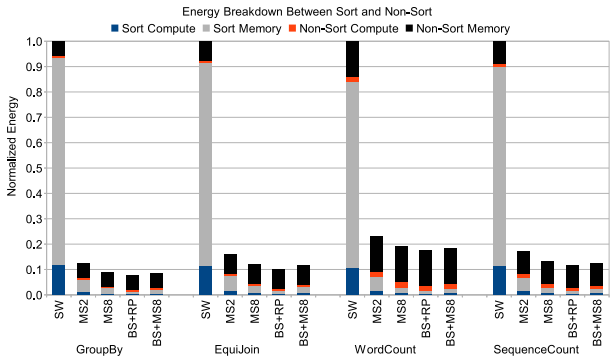

Fig. 5. Breakdown of where both compute and memory energy is spent during Map phase execution, normalized to the NDCore Software case.

phases. Figure 5 shows the breakdown for each workload and hardware sort accelerator, normalized to the case of the Software sort running on an NDCore.

Most of the energy is spent in the memory subsystem, and not in compute. All of the compute is being performed by low-EPI cores or fixed-function hardware accelerators, which offer large energy savings compared to conventional out-of-order server CPUs. Most of the memory energy consumption is due to high background power of HMC-based devices.

3) Sorting Accelerators with NDP-Module: We also evaluated our sorting accelerators in the context of an NDP-Module system that avoids 3D-stacked memory devices and instead tightly couples simple in-order cores to LPDDR2 memory chips. For space reasons, we provide only a brief summary here. Sorting acceleration allows our workloads to maximally utilize the available memory bandwidth. The accelerators reduce Map phase execution time from 64.8% in WordCount up to 88.1% in GroupBy. In all cases, NDP-Module *with* sorting acceleration is able to improve upon the performance of the more expensive NDP-HMC *without* sorting acceleration by at least a factor of 2×. Further, the NDP-Module system *with* sorting acceleration is able to achieve greater energy efficiency than the more expensive NDP-HMC system *without* sorting acceleration by at least a factor of 5×.

4) FPGA Emulation: We also implemented our sorting accelerator on a Virtex-7 FPGA. The 100 MHz accelerator is controlled by a MicroBlaze soft processor core (emulating our NDCore) and the dataset is stored on a 1GB DDR3 SODIMM, controlled by a 200 MHz memory controller (MIG). Without the accelerator, the software sort function on the MicroBlaze does not exhaust memory bandwidth. With the accelerator, performance is entirely dictated by the modest memory bandwidth. The FPGA implementation serves as a proof of concept, confirming that the sort accelerator has a simple design that can be effectively orchestrated by an NDCore.

VI. Conclusions

In this paper, we have explored using fixed-function hardware accelerators as a way to speed up the Map phase of MapReduce workloads. We considered hardware merge sorters, bitonic sorters, and range partitioners. Ultimately we found the performance difference between the various hardware sorters to be small, as they all offer very large speedups compared to the case where sort is running in software on an NDCore. We therefore recommend the use of the MergeSort8 accelerator, because of its very low area and power requirements. Compared to an NDP-HMC system without accelerators, incorporating fixed-function sorting accelerators offered Map phase execution time reductions ranging from 78% in WordCount using the MergeSort2 accelerator, to 92% reduction in GroupBy using the MergeSort8 accelerator. Similarly, energy consumption by the compute and memory subsystems during Map phase execution was reduced from 77% in WordCount by MergeSort2, to 91% in GroupBy by MergeSort8. Also compared to an NDP-HMC system without accelerators, an NDP-Module system with accelerators has greatly increased performance and energy efficiency, offering a cheaper alternative to expensive 3D-stacking to achieve very high MapReduce performance.

References

[1] L. Wu, A. Lottarini, T. Paine, M. Kim, and K. Ross, "Q100: The Architecture and Design of a Database Processing Unit," in *Proceedings of ASPLOS*, 2014.

[2] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch, "Disaggregated Memory for Expansion and Sharing in Blade Servers," in *Proceedings of ISCA*, 2009.

[3] S. Pugsley, J. Jestes, H. Zhang, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li, "NDC: Analyzing the Impact of 3D-Stacked Memory+Logic Devices on MapReduce Workloads," in *Proceedings of ISPASS*, 2014.

[4] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Proceedings of OSDI*, 2004.

[5] R. Balasubramonian, J. Chang, T. Manning, J. Moreno, R. Murphy, R. Nair, and S. Swanson, "Near-Data Processing: Insight from a Workshop at MICRO-46," in *IEEE Micro's Special Issue on Big Data*, 2014.

[6] S. Pugsley, J. Jestes, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li, "Comparing Different Implementations of Near Data Computing with In-Memory MapReduce Workloads," in *IEEE Micro's Special Issue on Big Data*, 2014.

[7] J. Jeddeloh and B. Keeth, "Hybrid Memory Cube – New DRAM Architecture Increases Density and Performance," in *Symposium on VLSI Technology*, 2012.

[8] A. Gutierrez, M. Cieslak, B. Giridhar, R. Dreslinski, L. Ceze, and T. Mudge, "Integrated 3D-Stacked Server Designs for Increasing Physical Density of Key-Value Stores," in *Proceedings of ASPLOS*, 2014.

[9] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan, "Meet the Walkers: Accelerating Index Traversals for In-Memory Databases," in *Proceedings of MICRO*, 2013.

[10] L. Wu, R. Barker, M. Kim, and K. Ross, "Navigating Big Data with High-Throughput Energy-Efficient Data Partitioning," in *Proceedings of ISCA*, 2013.

[11] A. Farmahini-Farahani, J. Ahn, K. Morrow, and N. Kim, "NDA: Near-DRAM Acceleration Architecture Leveraging Commodity DRAM Devices and Standard Memory Modules," in *Proceedings of HPCA*, 2015.

[12] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing," in *Proceedings of ISCA*, 2015.

[13] M. Arlitt and T. Jin, "1998 World Cup Web Site Access Logs," http://www.acm.org/sigcomm/ITA/, August 1998.

[14] "PUMA Benchmarks and dataset downloads," https://sites.google.com/site/farazahmad/pumadatasets.

[15] "Wind River Simics Full System Simulator," 2007, http://www.windriver.com/products/simics/.

[16] N. Chatterjee, R. Balasubramonian, M. Shevgoor, S. Pugsley, A. Udipi, A. Shafiee, K. Sudan, M. Awasthi, and Z. Chishti, "USIMM: the Utah SImulated Memory Module," University of Utah, Tech. Rep., 2012, UUCS-12-002.