Energy-Efficient Sorting on a Many-Core Platform

Aaron Stillmaker, Lucas Stillmaker, Brent Bohnenstiehl and Bevan Baas
University of California, Davis
{astillmaker, lstillmaker, bvbohnen, bbaas}@ucdavis.edu

***Abstract*—As processors move from multi-core to many-core architectures, opportunities arise for energy-efficient enterprise computations, such as sorting, on large arrays of processors. This paper proposes three different energy-efficient sorting methods for the first phase of an external sort simulated on a varying sized fine-grained many-core processor arrays used as a co-processor to an Intel CPU, which completes the second phase. The most energy efficient sort requires over 100× less energy than the comparable phase one GPU radix sort, and over 145× less energy than the comparable CPU quick sort to complete phase 1. Despite these large energy usage discrepancies, the highest throughput many-core sort was still over 17× faster than the comparable CPU sort and 9× faster than the GPU sort. The proposed sorting algorithms are scalable to any sized 2D mesh processor array while giving a large energy savings and increasing performance.**

I. Introduction

Demand for reduced power consumption while not reducing processing ability is high as energy efficiency in large database data centers is an increasingly important concern [1].

With ever shrinking transistors, extra area is being dedicated to multiple processing cores, instead of adding more transistors to a single core [2]. Increasing the number of cores on a single chip results in the bandwidth for shared memory systems becoming too large to be implemented with traditional architectures [3].

Sorting is one of the most used processing kernels in database systems [4], creating an interest in energy conscious sorting methods [5]. Datacenters with large data sets generally perform *external sorts*, sorting data that cannot fit in volatile memory, so the data needs to be sorted in two different phases [6]. With these larger arrays scaling to hundreds of cores, current sorting algorithms designed for traditional architectures cannot be used due to differences with architectural features such as intra-processor communication, shared memories, and off chip I/O.

This paper presents energy efficient, scalable sorting algorithms for an external sort, with the first phase completed by a fine-grained many-core array of low-powered, simple Multiple Instruction Multiple Data (MIMD) processors. The sorts consist of small modular program kernels operating on each core, making them scalable to different array sizes. A general purpose CPU performs the second-phase merge with the many-core array acting as a co-processor, performing the first phase. The merge sort does not benefit from parallelization onto a large many-core array on a single chip since the computation is I/O bound.

II. Sorting on a Many-Core Platform

As computers changed to multi-core processors, sorting research adapted, parallelizing sorting algorithms to take advantage of multiple processing cores. The progression to many-core processors necessitates a new shift to sorting with large processor arrays.

External sorting, which is commonly used in datacenters, requires the use of a secondary storage system and is done in two phases. During the first phase, runs of sorted lists are created from unsorted data, which can fit inside main memory. In the second phase, these sorted runs are merged together to create one sorted list.

The proposed sorts follow the Sort Benchmark [7] requirement of 100-byte records, where the first 10 bytes are the keys and the other 90 bytes are the record's payload.

A. Scaling to Many-Core

As arrays get larger, high bandwidth long-distance communication and access to shared memory becomes more difficult, and in some cases are removed in favor of more processing area and processing units, as done in the many-core array by Truong et al. [2].

This paper presents sorting with a large array of processors that have communication only with nearest neighbors and limited long-distance communication. Only processors on the edge of the array have access to chip I/O and the chip contains no global shared memory. The proposed sorts are designed to use this array as a co-processor working in tandem with a general purpose CPU, allowing it to use its complex high power circuits on more appropriate computations.

With only local communication and arbitrarily large arrays, the design of the proposed many-core phase 1 sorts was limited to streaming data through an array. Therefore the co-processor could be used to sort data as it streams from memory to the general purpose CPU or another memory, not involving the general purpose CPU during the first phase.

The sorts were evaluated on simulations of large arrays, but in order to show quantitative results, it was necessary to confine some limits to an existing architecture. Therefore, the AsAP2 architecture, developed by Truong et al. [2] was used for certain limitations and the physically measured traits from the fabricated 65-nm chip.

III. Sorting Kernels

The sorting variations, described in Section IV, utilize basic program kernels in each of the processors in the array. Each kernel was designed for modularity, so the kernels can be easily used in any processor on any part of the array, allowing for easy scaling. No specific information or knowledge about a processor's location or run size is required for the sorting or merge kernels, making different and changing sorts easily programmable. Each processor on the target platform has a 128x35-bit instruction memory, limiting each kernel to just 128 assembly instructions.

A. SAISort

The Serial Array of Insertion Sorts, or SAISort, is the fundamental sorting block used in all variations of the presented sorts. The name is taken from the observation that this sort on the micro scale is a insertion sort, and when multiple of these kernels are serially linked together, it creates a bubble sort in the macro scale.

When a fresh run is sent through the SAISort kernel, it will start by filling up its internal memory with a sorted list, as records arrive. Once the local memory is full, each new record will be sorted into the local list and the lowest valued record will be sent out. A code word preceding each record is reserved for triggering a reset at the end of a sorting run, allowing runs to begin without foreknowledge of the run size. When the kernel detects a reset signal, it will pass the reset signal to its output and begin output streaming.

B. Merge

The merge kernel will compare the keys received on each of its two inputs and will pass the record with a lower key value to its output. When a reset signal a received from one input, records from the other input will be passed directly to the output until a second reset signal is received. The kernel will pass this reset signal and then resume merging.

C. Split

The split kernel will be given a constant value at compile time for the number of split cores past the current core. The kernel will pass that many records to the next split processor, then take one record for its current row. It continues in this fashion until it receives the reset signal, at which point it will forward the reset signal to both the next split processor and the current row, then start from the beginning of the program.

D. Distribution

The distribution kernel presorts records by the MSBs of their key and sends them to rows of the array for finer sorting. When a reset is triggered, records are streamed one row at a time in increasing radix order. The radix value used by each core is set at compile time.

Three rows of cores are combined into a single lane that processes two radixes and is managed by a single distribution core. The upper and lower rows are each dedicated to a radix, while the middle row processors are dynamically assigned to the upper or lower row as those rows fill. Control signals for the middle row configuration are generated by the distribution core for a lane. The sorting cores use the SAISort kernel.

A series of distribution cores handle the routing of records to the appropriate lane and row, as well as detecting when a row is full and triggering a reset. Sorting of a set of records ends when a lane receives a record and determines it should be placed into a row that is already full. The core will send a reset request to the upstream and downstream distribution cores, as well as both of its lanes sorting rows. This core will then reset its state and reprocess the record that triggered the reset. Other distribution cores will propagate the reset signal and reset. A reset signal may also be sent into the core array to force a reset.

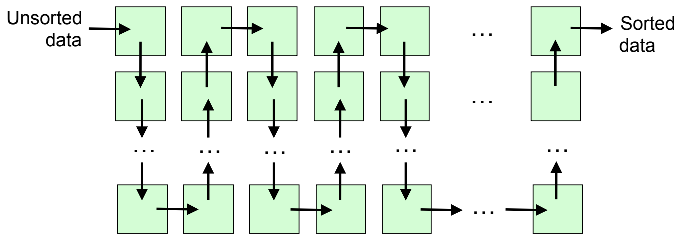


Fig. 1. A sample Snake Sort mapping showing the data path and how processors are used.

E. Dynamic Routing

Assignment of the center SAISort processors in a lane to the upper or lower rows is performed in 3x2 blocks, with two cores in each row. Upon reset, a block will default to assigning the central cores to the lower row; upon reception of a reconfiguration signal from the distribution core, the central cores will be reassigned to the upper row. This reassignment is performed by reconfiguration of the circuit network control signals in each core of the block, allowing records to be diverted from the upper or lower row into the central cores for additional capacity. To prevent reconfiguration of the block when it is still active, the central cores will send a signal to the upper and lower rows to indicate when they have completed a flush and are ready for reset. The upper and lower rows will delay reconfiguration until this signal is received.

IV. Proposed Sorting Variations

Several different sorting schemes were explored. The following sorting variations fit within our target architecture limitations, utilize simple, modular kernels, and scale easily with processor array sizing

A. Snake Sort

The snake sort is the simplest of the proposed sorting variations. This sort uses the SAISort kernel on each processor in the array, linking them together using a single input and output per processor. Each processor will take an input record, determine where it fits in that processor's sorted list, and will then output the lowest record. To fit within the processor memory limitation of 256 bytes on our target architecture, each processor core can hold up to two 100 byte records. Each processor added to the array increases the number of records sorted per run by two. One extra record can be sent into the array, but any additional records added will be sorted incorrectly if they have a value lower than that of a record previously pushed out of the array After a full run has been sent into the chip, a reset signal is sent through the snake triggering a flush in each processor.

A generalized mapping for the Snake Sort is shown in Fig. 1 which displays how more processors are used.

B. Row Sort

In row sort, multiple lists are sorted in parallel and then merged together, shortening the data path for any given record. When data enters the processor array, split processors in the first column are used to evenly distribute the records to each of the rows in the array. Each row utilizes the SAISort to sort their given records similar to individual snake sorts, generating multiple sorted lists. When the reset signal is sent into the

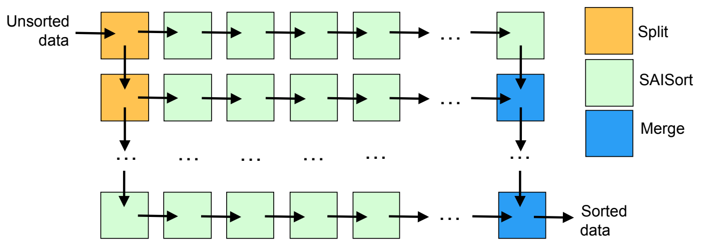


Fig. 2. Row Sort mapping showing the data path and how it is scaled.

chip, each row will send its sorted list to the final column of processors, where the merge kernel will join the lists to output a single sorted list. A mapping of the row sort is given in Fig. 2 where the sort can be scaled by adding more rows or columns.

Due to overhead for the split and merge kernels, this method sorts fewer records per run than snake sort. The bottom left and upper right processors are not required for distribution or merging respectively, as shown in Fig. 2. Two records are stored per sorting core and an additional record is added to each row, similar to snake sort. It was found that the highest activity processors with this method was the merge column of processors.

C. *Adaptive Sort*

In adaptive sort, the processor array is partitioned into a number of sorting lanes. Each lane begins with one 3x1 block containing one distribution processor and two basic SAISort processors. The lane is then expanded with some number of 3x2 adaptive sorting blocks, each containing the SAISort kernel with additional reconfiguration code. Each lane is responsible for sorting two radixes. The mapping is similar to the row sort, but the middle row of each set of three rows are dynamically allocated to the row filling up faster.

The distribution cores of each lane are connected together for passing records and control signals. The final sorting cores for each radix are similarly connected together for emptying records from the array. The lane connected to chip input runs additional distribution code for controlling the rate records are admitted into the array.

The advantage of radix sort over row sort is that it avoids a merging bottleneck at the end of each run. The disadvantages are a more complicated distribution algorithm and a non-deterministic number of records per run, as each run ends when a radix is full.

V. Simulation and Results

A. *Simulation*

This work explores sorting on a many-core platform as the size of the array is scaled to contain different numbers of processors. A simulator was used to model an arbitrarily sized many-core array. The gensort program [8] was utilized to create random datasets for all simulations and comparison simulations.

The scalable many-core architecture is modeled using a cycle based, two state simulator written in C++. Each processor core is simulated as a separate task, allowing multithreaded operation. Core simulations are synchronized based on the timing of data transfers through a modeled circuit network. Energy usage is estimated based on physical chip measurements for each type of operation, memory accesses, network access, oscillator activity, and leakage. These figures are scaled for voltage based on measured scaling characteristics.

The novel approach to sorting on many-core processors previously presented [9] was implemented on the AsAP2 platform. These sorts were designed to be modular and work on a large array of processors. Expanding on previous work, results for these sorts as well as the adaptive sort have been gathered for much larger and varying many-core array sizes.

The number of processors was scaled from 100 processors to 10,000 processors. Some selected points for the size of the array and the number of chips used are compared in Section V-B.

Comparison sorts were created which would benchmark our many-core results. Common unoptimized sorts were implemented on two different platforms, a laptop CPU, and a laptop GPU. For each comparison sort, thousands of executions were ran to get an average execution time.

1) Intel CPU Quick Sort: The quick sort [10] was chosen because it is a commonly used efficient sort. It was implemented in C++ on a Intel Core 2 Duo T7500, a 65-nm chip with a TDP of 35 W and a clock frequency of 2.2 GHz.

2) Nvidia GPU Radix Sort: The radix sort was chosen to be implemented on the GPU platform because there was an accessible version publicly available with the Nvidia CUDA SDK written by Satish et al. [11]. The sort was implemented on a Nvidia GeForce 9600m GT, a 65-nm chip with 32 processing cores, a clock speed of 500 MHz, and a power consumption of 23 W. It was necessary to extensively change the code to work with the 80-bit keys and 100-byte total records.

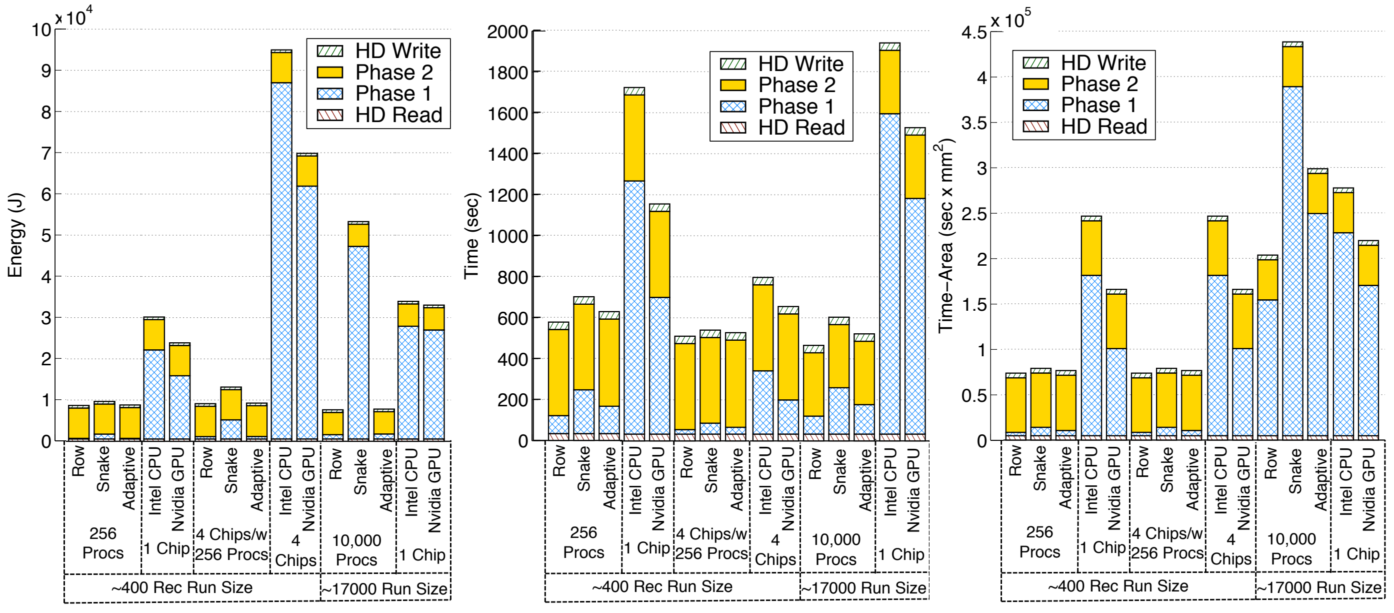
3) Intel CPU Merge Sort: In order to simulate an entire external sort, it was necessary to write a sort for phase 2. The merge sort was chosen due to it's simplicity [6]. This was implemented in C++ on the same Intel Core 2 Duo processor described in Section V-A1. Two GB of sorted runs were loaded onto main memory and timed separately from the sort where thousands of records were merged to get a throughput number. The throughput number was used to model the total time to complete phase 2 for the different run sizes.

B. *Results*

The different sorting methods were used to compute a 10-GB external sort. The phase 2 merge as well as other administrative tasks are performed on a Intel CPU, and phase 1 computations are performed on the given computational platform. All of the tests were run on or simulated with measured values from chips fabricated in 65-nm technology, so no scaling was necessary.

Fig. 3a shows the energy required by the processing unit to perform the different sorts. The energy cost of running the co-processor many-core chip is almost negligible in the graph when compared to the required power to operate the Intel processor for administrative tasks and phase 2 merge. This translates to extremely energy efficient sorts on the many-core system, with the phase 1 row sort on a single 256-processor array taking over 145$\times$ less energy than the Intel CPU quick sort, and over 100$\times$ less energy than the Nvidia GPU radix sort. This is overshadowed by the phase 2 energy, making the total energy cost for the most efficient many-core sort require 4$\times$ less energy than the comparable CPU and GPU sorts.

As can be seen in Fig. 3b, the many-core sorts are able to sort their 10 GB of runs in less time than any of the comparison



(a) (b) (c)

Fig. 3. (a) Total energy required by processors on a complete 10-GB sort. (b) Time required to sort a complete 10-GB dataset. (c) Time-area required by processors of a complete 10-GB sort.

sorts. The row sort performed on 4 chips with 256 cores takes over 17× less time to perform phase 1 compared to the quick sort on a CPU, and over 9× less time to perform a comparable radix sort on the GPU. The largest difference in total time is found between the row sort with 10,000 processors showing over 4× smaller time that the comparable CPU sort, and over 3× smaller than the GPU sort. It can be seen that the sorting time is largely governed by the second phase merge with the proposed many-core sorts.

To highlight a benefit of using a small low powered many-core chip as a co-processor in a database system the total time-area metric was used, where the total time is multiplied by the total chip area of the processing unit. Shown in Fig. 3c, one can see the tradeoff of scaling the number of processors. This analysis highlights that all of the many-core sorts are faster and more area efficient than the CPU or GPU, except for the 10,000-processor many-core chip.

VI. Conclusion

We have presented three different sorting variations. This was accomplished on a varied amount of processors in a many-core array acting as a co-processor to a Intel CPU performing the phase two merge part of the external sort. The proposed sorts use kernels to program individual processors in the array in a manner that makes the sorts modular, easy to program, and easily scalable.

Many-core array sorts were modeled with the number of cores ranging from 100 to 10,000. We found the most energy efficient phase 1 sort required over 100× less energy than the comparable Nvidia GPU and over 145× less energy than the comparable Intel CPU sort. With these large energy differences, the highest throughput many-core phase 1 sort was still over 9× faster than comparable CPU and GPU sorts and more than 2× less time-area. The proposed sorts can be implemented on differing sized arrays in a large database system and recognize large energy savings without giving up performance.

Acknowledgment

The authors gratefully acknowledge support from C2S2 Grant 2047.002.014, ST Microelectronics, NSF Grant 1018972 and 0903549 and CAREER Award 0546907, SRC GRC Grant 1598 and 1971 and CSR Grant 1659, Intel, UC Micro, SEM, and Fudan University.

References

[1] A. Gandhi *et al.*, "Optimal power allocation in server farms," in *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, ser. SIGMETRICS '09. New York, NY, USA: ACM, 2009, pp. 157–168.

[2] D. Truong *et al.*, "A 167-processor computational platform in 65 nm CMOS," *Solid-State Circuits, IEEE Journal of*, vol. 44, no. 4, pp. 1130–1144, Apr. 2009.

[3] S. Borkar, "Thousand core chips: a technology perspective," in *Proceedings of the 44th annual Design Automation Conference*, ser. DAC '07. New York, NY, USA: ACM, 2007, pp. 746–749.

[4] G. Graefe, "Query evaluation techniques for large databases," *ACM Comput. Surv.*, vol. 25, pp. 73–169, June 1993.

[5] S. Rivoire *et al.*, "Joulesort: a balanced energy-efficiency benchmark." in *SIGMOD Conference*, C. Y. Chan, B. C. Ooi, and A. Zhou, Eds. ACM, 2007, pp. 365–376. [Online]. Available: http://dblp.uni-trier.de/db/conf/sigmod/sigmod2007.html#RivoireSRK07

[6] D. E. Knuth, *The Art of Computer Porgraming*. Reading, Massachusetts: Addison-Wesley, 1973, vol. 3 - Sorting and Searching.

[7] C. Nyberg, M. Shah, and N. Govindaraju, "Sort benchmark home page." [Online]. Available: http://sortbenchmark.org/

[8] C. Nyberg, "Sort benchmark data generator and output validator." [Online]. Available: http://www.ordinal.com/gensort.html

[9] A. Stillmaker, L. Stillmaker, and B. Baas, "Fine-grained energy-efficient sorting on a many-core processor array," in *Parallel and Distributed Systems (ICPADS), 2012 IEEE 18th International Conference on*, dec. 2012, pp. 652 –659.

[10] C. A. R. Hoare, "Algorithm 64: Quicksort," *Commun. ACM*, vol. 4, pp. 321–, July 1961.

[11] N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore GPUs," in *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–10.