# Fine-Grained Energy-Efficient Sorting on a Many-Core Processor Array

Aaron Stillmaker, Lucas Stillmaker and Bevan Baas
*Electrical and Computer Engineering Department*
*University of California, Davis*
*Davis, United States of America*
{*astillmaker, lstillmaker, bbaas*}*@ucdavis.edu*

***Abstract*—Data centers require significant and growing amounts of power to operate, and with increasing numbers of data centers worldwide, power consumption for enterprise workloads is a significant concern. Sorting is a key computational kernel in large database systems, and the development of energy efficient sorting capabilities would therefore significantly reduce data center power usage. We propose highly parallel sorting algorithms and mappings using a modular design for a fine-grained many-core system that greatly decreases the amount of energy consumed to perform sorts of arbitrarily large data sets. The memory, computational, and nearest-neighbor inter-processor communication hardware of the many-core processor array require relatively small die area. We present the design and implementation of several sorting variants that perform the first phase of an external sort. They are built using program kernels operating on independent processors in a many-core array with 256 bytes of data memory and fewer than 128 instructions per processor. The algorithms employed are simple and the vast majority of processors contain identical programs. Compared to a quicksort implementation on an Intel Core 2 Duo T9600 the highest throughput design achieves up to 27× higher throughput per chip area, and the most energy efficient sort yields a 330× reduction in energy dissipated per sorted block. Compared to a radix sort implementation on a GPU, the highest throughput design achieves up to 22× higher throughput per chip area, and the most energy efficient sort yields a 750× reduction in energy dissipated per sorted block.**

***Keywords*-parallel processing; external sorting; streaming sorting; fine-grained many-core; processor array; modular programing;**

## I. Introduction

Energy efficiency is becoming increasingly important in today's data centers with their growing size and power consumption [1]. Sorting functions are a critical computational kernel as they are one of the most used functions in database systems [2].

Data centers in the United States spent a combined $4.5 billion in 2006 for electrical power, accounting for 61 billion kilowatt-hours and 1.5% of the total United States electricity consumption [3]. Power consumption of database systems can be so high that after a few years of running a common database system, the amount of money spent to power the system can easily exceed the price of the hardware [4]. Power consumption is such a significant cost in database center operation that low power rates and weather conditions are sometimes factors in the selection of the location of new database centers, such as Google's selection of Finland for a new data center in 2009 [5] and Facebook's proposed data center in Sweden [6].

Data centers generally perform sorts on data sets that are too large to be contained inside of main memory. This type of sorting is called *external sorting* and is generally done in two phases. The first phase creates sorted lists that can be as large as the size of some memory. The second phase merges these lists into one final sorted list [7].

### A. Contributions

We present sorting approaches and algorithms for the first phase of an external sort, which are targeted to operate on a low power fine-grained many-core processor array. While CPUs attain ever increasing performance, their performance per watt has stayed relatively constant [4]. It is well known that workloads can be more efficiently executed on platforms with algorithm-specific hardware. The presented sorts operate on a many-core co-processor array which is fully programmable, contains no algorithm-specific hardware, and is smaller than a general purpose CPU.

The contributions of this paper are as follows:

- several high throughput, area efficient, and energy efficient sorts for fine-grained many-core systems
- sorts do not require a large shared memory
- sorts are simple to program on independent MIMD cores, they can run on a globally asynchronous system, and each processor's program is fewer than 128 instructions with identical programs on most processors and
- the proposed sorts are highly modular—they can be easily scaled without changes to the algorithm

### B. Related Work

External sorting has been well explored such as in work by Vitter [9], who looked specifically at making external sorting more efficient from an I/O perspective, though the work generally focuses on common CPUs with either single cores or multi-cores. Smaller sized internal sorting has been heavily researched on uni-core and multi-core systems resulting in fast sorting algorithms that process small 4-byte keys, which are smaller than what databases would typically utilize, such

as the work by Chhugani et al. [10] and Gedik et al. [11]. Both use sorts specialized for their target architectures, taking advantage of architecture-specific SIMD instructions, large shared memories, and special instructions that reduce computation time and access time, respectively, to achieve high throughputs. GPUs contain large processor arrays and have been explored as a platform to sort data [12].

Sorting theory of mesh-connected computers have been explored extensively [8]. Such works focus on theory of internal sorting on an arbitrary mesh-connected computer, which does not transfer to our chosen platform, and cannot be compared against our realized results.

Work has been done on creating systolic processor arrays for sorting [13]. While the idea of a data driven processor array is shared in this implementation, systolic arrays are generally not fully general in their inter-processor communication which makes them a much less flexible platform than what was utilized in this work. Design and fabrication of a integrated circuit chip is costly, so programable processors are able to spread out the cost, making them much cheaper than Application Specific Integrated Circuits (ASICs) [14]. A sorting ASIC could also undesirably limit configurability.

With multi-core and many-core systems becoming more prevalent, there has been some research in parallel sorting on database systems, as presented by Taniar and Rahayu [15], [16]. The work focuses on general purpose CPUs, assuming each processor has large available memories. The GPUTeraSort [17] was shown to be a effective many-core sort, but their platform benefited from large shared memories. The JouleSort benchmark [18] specifically considers the energy efficiency of large database sorts.

## II. External Sorting and the Targeted Many-Core Architecture

### A. External Sorting Overview

External sorting is a sort in which the data set is too large to fit into main memory [7]. This means that the data set must be stored on a secondary storage, such as a hard drive. With access times to secondary storage typically far longer than to main memory, it is generally advantageous to sort the large data sets into smaller sorted lists, or runs, which fill the available main memory. This is termed the *first phase* of the external sort. All of these data sets are combined into a final complete sorted data set in secondary memory, which is termed the *second phase* of the external sort, as shown in Figure 1.

The second phase is typically executed by a type of merge sort, of which there are many serial implementations that accomplish this efficiently as well as several multi-core implementations [12], [15], [16]. Because the first phase is more likely CPU-bound and the second merging phase is likely to be I/O bound, we focus on the first phase only performed on a co-processor. With the main processor performing the second phase, we leave it out of this discussion.

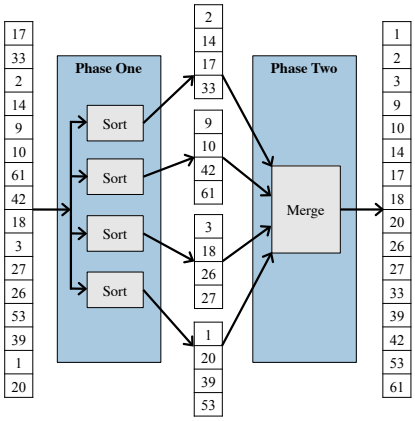



Figure 1. An example *external sort* where 16 data values are first sorted into four lists of four items each in the first phase, and the four intermediate lists are merged into a final sorted list in the second phase.

### B. Sorting a Stream of Data

Most of the effort put into researching sorting on many-core systems has been spent on arrays with access to large local memories [10], [17], [12], [15], [16]. Conversely, this work looks into sorting with many-core arrays where the data is streamed through a processor array that is not attached to a large global memory. In particular, we utilize a large 2-D mesh-connected processor array with nearest neighbor communication and without global access to a large shared memory. Many sorts that require swaps from opposite sides of a list, such as the bitonic merge sort, are not efficiently implementable on this type of platform. Our target platform contains many small, low-power processors, with limited instruction and data memories. With a reduced instruction set, many of the more complicated sorting methods such as quicksort were ruled out. The sort should be modular; making it simple to change the size of the processing array without having to create a new set of programs. Along with being simple, the processor array should operate without any knowledge of how large the list length is going to be, allowing for modules, such as regular-expression filters or statistics calculators, to be added to the processing array.

We target database systems, so we use standard 100-byte records with 10-byte keys, which are commonly used to benchmark large database sorts [19].

### C. Targeted Platform: Fine-Grained Many-Core Processor Array

The targeted platform for this work is a fine-grained many-core system. These sorts were implemented on the second version of the Asynchronous Array of Simple Processors, or AsAP2 [20]. The chip contains 164 general

purpose processors that use a simple instruction set with only sixty-three instruction types. The chip also includes three 16 kB memories with the entire chip connected via a 2D-mesh, allowing for nearest neighbor communication and long distance communication. The chip was fabricated using 65 nm CMOS technology, with each programmable core in the array occupying 0.17 $mm^2$ of area, and the processors can operate up to 1.2 GHz at 1.3 V. Each processor contains $128\times35$-bit instruction memory, $128\times16$-bit data memory, and two dual clock $64 \times 16$-bit FIFO buffers for communication between processors. This chip has Globally Asynchronous Locally Synchronous (GALS) clocks where each of the 167 processors in the array has its own fully-unconstrained oscillator. Each processor has the ability to turn off its oscillators while stalled, require ring negligible energy when stalled.

## III. Sorting Program Kernels

Many different methods of sorting were tested; the most interesting are presented here. With the desire to keep the program modular, different program kernels were created. Each kernel needs no information about the run, they will reset when a reset signal is sent at the end of a run. In this way, the sort does not require reconfiguration for different runs, and can be easily scale to any sized array without the need for a different program.

### A. SAISort

The main programing kernel is a Serial Array of Insertion Sorts, or SAISort. It is a simple program that is an insertion sort in the micro view (one processor) and a bubble sort in the macro view (entire chip) [21]. As long as a reset signal is not received by the processor, inputs into each processor will populate a sorted list on that processor's data memory. Once the data memory is full, the processor passes the lowest entry to the next processor in the snake. In an effort to make sure the sort can be executed without knowledge of how large the sort is, a tag system was implemented. Each record was preceded by a non-reset tag, any bit combination other than the reset tag. At the end of the run, a reset tag was attached. When a reset signal is received by the SAISort kernel, it will pass along the reset signal to the next processor, as well as the number of entries that will be passed without the need to sort. Then it will flush all entries, starting with the lowest. After the internal memory has been flushed, it will pass the number of inputs it was told to pass, without sorting, to the output. The pseudo code of the algorithm of the SAISort is shown in Algorithm 1. Any number of processors can be connected in a chain and can use the exact same program to produce a sorted list. SAISort will compare only two bytes at a time, the size of one word in the AsAP2 processor. So it would not need to compare subsequent lower significant bits if a result was achieved from more significant bits. A significant portion of the code length was dedicated to dealing with administrative overhead of sorting keys that are larger than our platform's 16-bit data word length. The algorithm, when implemented on the AsAP2 chip, utilized 126 assembly instructions out of the available 128, and has a throughput of around 160 clock cycles per record. Keep in mind that 50 clock cycles are taken just to transmit the 100 byte (50 word) record between cores. The code is unoptimized, and was written in a relatively short amount of time. Further optimization could lead to decreased code length, decreased energy consumption and higher throughput.

**Algorithm 1** SAISort

```
while true do
  if inputTag ≠ Reset then
    if recCount ≠ Full then
      Place input in appropriate position
      recCount++
    else if input ≤ lowest then
      output ← input
    else
      output ← Lowest record on processor
      Place input in appropriate position
    end if
  else
    Save the incoming recsToPass
    output ← Reset
    output ← (recsToPass + recCount)
    for int i; i < recCount; i++ do
      output ← Lowest record on processor
    end for
    for int i; i < recsToPass; i++ do
      output ← input
    end for
    Reset appropriate variables
  end if
end while
```

### B. Distribution

The second modular program kernel is the distribution kernel, shown in Algorithm 2. This program will evenly distribute entries between one of the outputs from the processor. The program would be set up prior to run time with the ratio of how many SAISort kernels are going to use each of the output streams. If for example there were four times as many SAISort kernels using the records from the south output, it would output four out of five entries to the south. This algorithm was written with 50 assembly instructions providing a throughput of 55 clock cycles per record.

### C. Merge

The last modular program kernel is the merge kernel, shown in Algorithm 3. The merge kernel will output the lower of its two inputs. If it receives a reset signal from either input, then it will just pass the input which is not finished until that input also gives a reset signal, at which point the

**Algorithm 2** Distribution

```
while true do
  if inputTag ≠ Reset then
    if count < ratio then
      outputdirection = south
      output ← input
      count++
    else
      outputdirection = east
      output ← input
      count = 0
    end if
  else
    outputdirection = south
    output ← Reset
    outputdirection = east
    output ← Reset
  end if
end while
```

**Algorithm 3** Merge

```
while true do
  if input1Tag = Reset then
    while input2Tag ≠ Reset do
      output ← input2
    end while
  else if input2Tag = Reset then
    while input1Tag ≠ Reset do
      output ← input1
    end while
  else
    if input1Key ≥ input2Key then
      output ← input2
    else
      output ← input1
    end if
  end if
end while
```

kernel will output a reset signal and wait for another set of inputs. This kernel was written in 80 assembly instructions and found to have a throughput of around 70 clock cycles per record.

## IV. Sorting Implementations and Variations

The kernels are used as building blocks to construct the two presented sorting schemes, described in Sections IV-A, IV-B and IV-C. Thusly little extra programing was required above writing the previously mentioned kernels, as identical programs were simply repeated across the array. Each processor works independently of the others throughout the sort. Unsorted data will be streamed into the chip, and sorted data is streamed out. In each version, each chain of SAISort processors can take one extra record, which will be stored in the input FIFO of the first processor in the chain. This allows for 329 records in the Snake Sort.

As previously mentioned, smaller internal sorts are required to make sorted lists that are merged together to eventually make one large list. This sort would be run on a large data set, making numerous successive sorted lists, or runs, of the sorting program. As soon as a processor is done with one run, it can proceed to the next run, even if the following cores are still processing the previous run.

On the target AsAP2 chip, there are two 64 byte FIFO buffers between processors, which means that processors are not required to strictly synchronize read and writes. A processor will not stall so long as the FIFO is empty before a second entry is output. Therefore, processor throughput is limited by its ability to sort one entry.

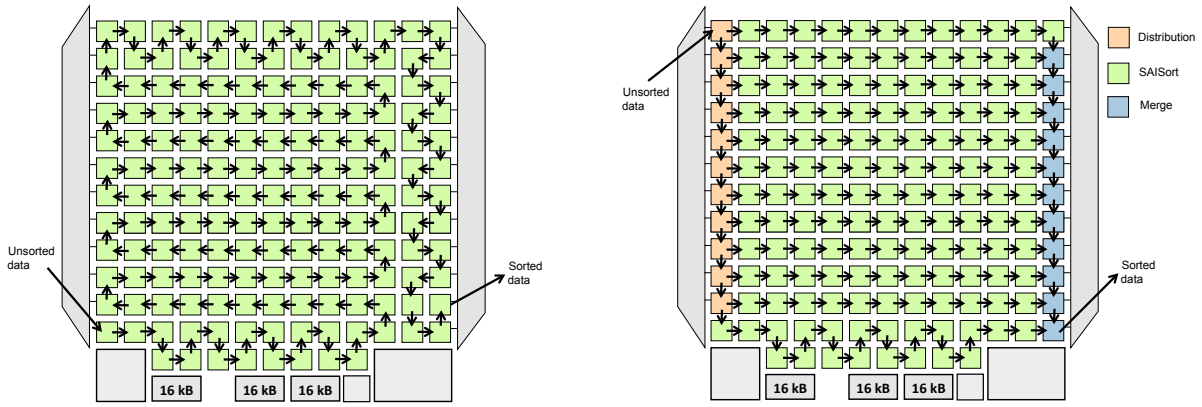
### A. Snake Sort

The simplest way to implement a sort using the SAISort is to use all available processors connected in one long chain, which resembles a coiled snake. The specific mapping is arbitrary as long as each processor is included in the chain. One such mapping on the AsAP2 chip is shown in Figure 2a. The same SAISort program is used on each of the 164 general purpose processors.

With the 256 byte internal data memory, two 100 byte records could be stored in each processor along with some control data. Different ways to use the FIFOs between processors as storage were explored. Initial attempts were made to use these FIFOs a storage space, but this cannot be implemented. Given a processor array, shown in Figure 2a, with each processor containing two records, when one more record is added, the last processor will have to output a record, which will be the lowest entry of that sorted list. Any subsequent attempts to push in more data will run the risk of creating an unsorted list. Other options, such as utilizing both FIFOs in the processor to allow the next processor to grab the smallest record, require more instruction memory than is available on the AsAP2 chip. This would also be counter productive towards the goal of a simple program and would have made the kernels platform specific. In one run through this Snake Sort, an AsAP2 chip processing 100 byte records can sort 329 records per run. The measured average throughput of the Snake Sort is about 240 clock cycles per record. The throughput is lower than the individual kernel throughput because the time required for flushing entries from the array depends on the length the sorted run.

### B. Row Sort

In an effort to make a sorting scheme with a shorter data path, the Row Sort is proposed. This version uses the distribution kernel in the first column of processors in the array to spread records evenly to each row in the array. Each row then uses the code as shown in Algorithm 1 to create a sorted list. After a reset signal is received, and records begin to flush through their respective rows. The last column of processors will merge the multiple lists into one larger list. This mapping is shown in Figure 2b.




(a) Data flow in the Snake Sort with each processor using the same SAISort kernel

(b) Data flow in the Row Sort using Distribution, SAISort, and Merge program kernels

Figure 2. Data flow diagrams for the two sorting algorithms mapped onto an AsAP2 many-core processor array.

With twenty two processors used for distribution and merging and the remaining 142 used to sort, 296 records can be sorted per run. The Row Sort takes about 120 clock cycles per record.

### C. Inclusion of Chip Memories

The target platform, AsAP2, has three 16 kB memory modules at the bottom of the chip, only accessible by two processors adjacent to each of the memories. An adjacent processor can be used to read and write data to a neighboring memory using certain configuration signals.

Runs can be stored in these memories. When the memories are full, all of the separate sorted lists will merge and be sent off chip. To communicate with the memories, processors need to merge and transfer the data. With these memories, the size of the sorted list can be increased by 480 records (160 in each 16 kB memory). This allows for 785 records per run for the Snake Sort with 285 clock cycles per record and 753 records per run with the Row Sort at 162 clock cycles per record.

The mappings of the Snake and Row sort schemes using the on chip memories are similar to those shown in Figures 2a & 2b, except the bottom row of processors are required for communication between the array and the 16 kB memories. There are three added memory administrating processors, which deal with storing the data and reading the data from the memory modules, as well as five processors that are just used to transmit records to and from the memory administrative processors. Finally, there are three processors that use the merge kernel to merge the four runs into the final output. Each memory module can hold 160 records, so in the execution of the sort, three runs, each sized to 160 records, will be individually stored in the three memory modules. When the fourth run is finished sorting the maximum number of records, two of the stored runs will merge, which will then be merged with the third run, which in turn is merged with the fourth run. This method repeats for all of the data to be sorted.

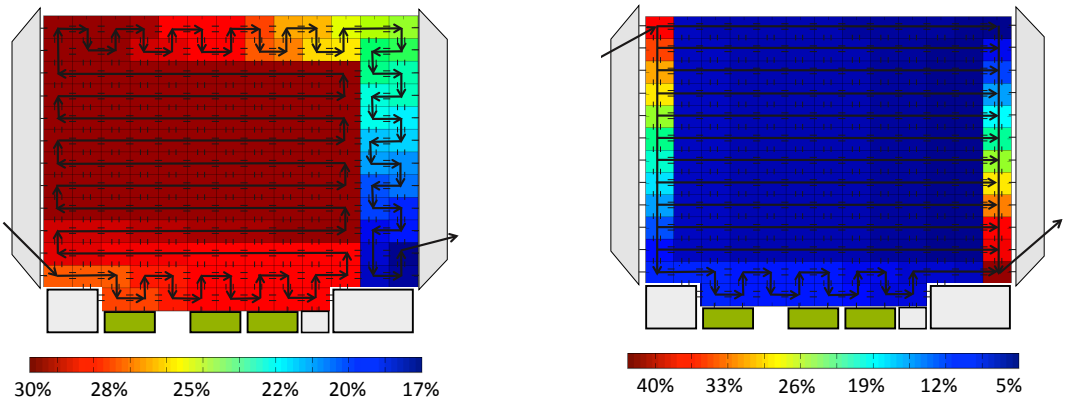
## V. Analysis

### A. Processor Activity Percentage

The percentage of time that each processor is actively processing information varies among the different processors in the array, highlighting bottlenecks in the sorts. Figure 3a shows the activity percentage of each processor when one run is sent through the Snake Sort. The first processors' lack of activity would be filled up by successive runs when running sorts on a large set of data, allowing runs to overlap. The processors are not fully utilized because they are stalled waiting for a new input or waiting to output a record. The activity level decreases starting at around the $120^{\text{th}}$ core reaching an activity percentage near 17% at the last processor in the snake. These last processors see a reduced amount of utilization because they pass a majority of the list through without needing to sort the data.

Figure 3b shows the activity percentage of the Row Sort while sorting one run of data. The merging processors are between 7% to 41% active. The 41% active merge processor at the bottom of that column is the bottleneck, as every entry needs to be merged through this one processor, while the other merging processors only work on subsets of the run. The distribution processors in Figure 3b are stalled waiting for more input records. With multiple runs of data, the activity of the distribution processors and the sorting processors would increase, again showing overlap that exists between runs.

The activity percentages are all below 100% utilization since they wait for neighboring processors, which decreases the throughput. However, as each processor in the array can completely halt its oscillator, the stalling of the processors has a negligible affect on the total energy consumed.




(a) Processor activity levels while executing the Snake Sort.

(b) Processor activity levels while executing the Row Sort.

Figure 3. "Heatmaps" showing processor activity levels highlighting bottlenecks and processors that are consuming low amounts of power while stalled.

### B. Comparisons

While there has been a lot of research on sorting, none that we found is straightforward to compare with this work. Most papers researching external sorting do not share their results for the first phase, nor do they contain a run size that would match those generated in these proposed sorts. Comparison sorts were written in C++ as well as CUDA and we generated our own data for comparison purposes.

We used quicksort [22] as it is simple, commonly known, and highly efficient. No attempt was made to optimize the quicksort code that was written, and no special processor-specific instructions were used. The C++ program was executed on an Intel Core 2 Duo T9600 with a clock frequency of 2.8 GHz and compiled using the gcc compiler, with the optimization flag set to -O3, utilizing both cores on the chip. As the average power of the Intel processor was not obtainable, the conservative value of one half the 35 W TDP, or 17.5 W was used as the power for the processor [23].

For a many-core comparison, we used the radixsort for the GPU, written by Satish et al. [12] and included in the Nvidia CUDA SDK. The program was modified to match our sorting parameters. 10 byte keys and 90 byte payloads were loaded and sorted completely on a laptop GPU, a Nvidia GeForce 9600M GT with a core clock frequency of 500 MHz and a maximum power draw of 23 W.

Thousands of runs with the same number of records per run as those of the proposed sorts were run through the program to record an average throughput. The program fills up main memory, and then a timer is started. The sorts are performed, and then the timer is stopped before the results are written to secondary memory, in an effort to remove system I/O speeds from the results. The random data was created by the gensort program [24]. The results of the Intel sorts were scaled from 45 nm to the 65 nm technology node to match the AsAP2 and Nvidia platforms [25]. Due to the simplicity and modular format of the many-core sorting versions, programing each of these comparison sorts took more time then programing each of the AsAP sort versions.

### C. Experimental Results

A version of the proposed many-core sort was run on a physical AsAP2 chip. Unfortunately, the laboratory setup does not have a method to stream data into the processor, as is required by the test conditions. The version of the sorts run on the physical chip contain a random record-generating program on the first processor on the chip, which has a lower throughput than what is required to avoid stalling the sorting processors. This occurs since at the beginning of each run, each processor will take in two records to fill its memory, which requires data to be input as fast as it can be transmitted. Thus, any timing numbers would be invalid. Therefore, all timing measurements are obtained from a cycle-accurate Verilog simulator, running the same Verilog code that was used to create the chip. It is possible to connect input and output to the chip with a different setup and could easily be implemented as a co-processor on a database system. All of the power numbers are calculated in the Verilog simulator, using measured power numbers for different operations. The metrics of throughput (rec/sec), throughput per area ((rec/sec)/mm$^2$), and energy dissipated per record (nJ/rec), are used. The chip die size of each platform was used to calculate the throughput per area.

The throughput and energy efficiency for the 100 byte record sorts are shown in Table I. Figure 4 compares the presented work with the sorts ran on the Intel and Nvidia platforms, with energy efficiency on the vertical axis and throughput on the horizontal axis. The AsAP2 processor operated with a clock speed of 1.07 GHz at 1.2 V and a clock speed of 260 MHz at 0.75 V. The highest throughput is over twice the number of records per second and over twenty seven times the number of records per second per mm$^2$

Table I
THROUGHPUT AND ENERGY DISSIPATION FOR 100-BYTE DATA RECORDS

| Records Per Block | Processor | | Throughput (1,000 rec/sec) | Throughput per Area ((rec/sec)/mm$^2$) | Energy per Rec (nJ/rec) |
|---|---|---|---|---|---|
| 296 | Intel Core 2 Duo quicksort | | 1,300 | 8,100 | 6,700 |
| | Nvidia 9600M GT radixsort | | 1,500 | 10,000 | 15,000 |
| | **AsAP2, Row Sort** | 1.2 V | **8,900** | **220,000** | 90.8 |
| | | 0.75 V | 2,200 | 55,000 | **19.8** |
| 329 | Intel Core 2 Duo quicksort | | 1,300 | 8,100 | 6,700 |
| | Nvidia 9600M GT radixsort | | 1,500 | 10,000 | 15,000 |
| | **AsAP2, Snake Sort** | 1.2 V | 4,400 | 110,000 | 670 |
| | | 0.75 V | 1,070 | 27,000 | 146 |
| 753 | Intel Core 2 Duo quicksort | | 1,200 | 7,500 | 7,200 |
| | Nvidia 9600M GT radixsort | | 1,500 | 10,000 | 15,000 |
| | **AsAP2, Row Sort w/ On-chip Memories** | 1.2 V | 6,600 | 170,000 | 108 |
| | | 0.75 V | 1,600 | 41,000 | 25.6 |
| 785 | Intel Core 2 Duo quicksort | | 1,200 | 7,500 | 7,200 |
| | Nvidia 9600M GT radixsort | | 1,500 | 10,000 | 15,000 |
| | **AsAP2, Snake Sort w/ On-chip Memories** | 1.2 V | 3,800 | 95,000 | 646 |
| | | 0.75 V | 910 | 23,000 | 143 |

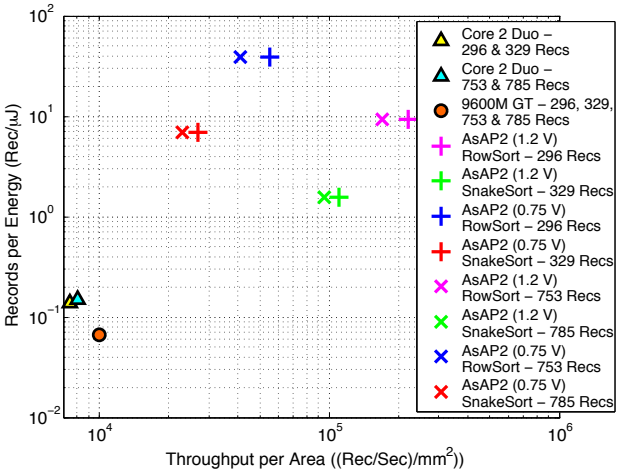



Figure 4. Comparison of sorting 100-byte records on AsAP2, a Nvidia 9600M GT, and a Core 2 Duo (scaled to 65 nm [25]).

higher than the sort on the Intel processor. From Table I, it can be seen that the AsAP2 throughput is higher than the Intel sort's, and it does this at less than half the clock speed of the Intel processor. The most energy efficient sort is 330 times more efficient than the sort on the Intel processor.

Table I shows that the fastest AsAP2 sort throughput is over twenty-two times higher than the GPU radixsort and the lowest energy consuming AsAP2 sort is over 750 times more efficient than the the Nvidia GPU sort. It should be noted that to make the GPU sort comparable to this work, it was necessary to sort a data set that is suboptimal for the SIMD architecture of the GPU. The GPU would preform best with all all 512 threads in a block full, so run sizes of either multiples of 512 or sufficiently large so that the blocks could be packed for multiple runs would have been more efficient. To change the run size to be optimal for the GPU would have given incomparable results to this work. In Table I it can be seen that all of the throughputs are the same for the GPU, this again can be attributed to the SIMD architecture. With similar run sizes, the number of blocks run in the CUDA program were nearly identical, meaning the effort for the GPU was about the same for all of the sorts with varying amounts of empty threads running.

Figure 4 clearly indicates that the Row Sort has the highest throughput and lowest energy usage per record. It can also be seen that adding the memories increases the size of the run without a significant change to throughput or energy usage per record. The Snake Sort recognizes a small decrease in energy used per record, which occurs as the sorting is shifted to fewer merge processors. The Row Sort sees a small increase in energy per record as it already uses series of of merges, The largest difference in the sorts which used the memories are the costs associated with powering the on chip memories, and merging more records.

## VI. CONCLUSION

In this paper, we have presented several sorts to produce sorted lists for the first phase of an external database sort using a fine-grained many-core processor array. The highly parallel sorts work on large 2D mesh processors without access to large shared memories. The program kernels presented are modular and independent of neighboring processors, allowing for scaling the sort to work on different sized processor arrays. The results show that using the proposed system, the highest throughput per area is up to 27× higher and the lowest energy per record sort is more than 330× smaller than the energy per record of a similar quicksort on a general purpose CPU. The highest throughput per area is also up to 22× higher and the lowest energy per record sort

is more than $750\times$ smaller than the energy per record of a similar radixsort on a GPU. Therefore, these sorts can be implemented on a co-processor in a large database system, providing a large energy savings.

## 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, and thank Kimberly Stillmaker, Zhibin Xiao, Jon Pimentel, Bin Liu, and Brent Bohnenstiehl.

## REFERENCES

[1] R. K. Sharma, R. Shih, C. Bash, C. Patel, P. Varghese, M. Mekanapurath, S. Velayudhan, and M. Kumar, V, "On building next generation data centers: energy flow in the information technology stack," in *Proceedings of the 1st Bangalore Annual Compute Conference*, ser. COMPUTE '08. New York, NY, USA: ACM, 2008, pp. 8:1–8:7. [Online]. Available: http://doi.acm.org/10.1145/1341771.1341780

[2] G. Graefe, "Query evaluation techniques for large databases," *ACM Comput. Surv.*, vol. 25, pp. 73–169, June 1993. [Online]. Available: http://doi.acm.org/10.1145/152610.152611

[3] EPA, "EPA report to congress on server and data center energy efficiency," U.S. Environmental Protection Agency, Tech. Rep., 2007. [Online]. Available: http://www.energystar.gov/ia/partners/prod_development/downloads/EPA_Datacenter_Report_Congress_Final1.pdf

[4] L. A. Barroso, "The price of performance," *Queue*, vol. 3, pp. 48–53, September 2005. [Online]. Available: http://doi.acm.org/10.1145/1095408.1095420

[5] Google, "Hamina data center," 2011, http://www.google.com/datacenter/hamina/. [Online]. Available: http://www.google.com/datacenter/hamina/

[6] R. Miller, "Facebook goes global with data center in Sweden," 2011. [Online]. Available: http://www.datacenterknowledge.com/archives/2011/10/27/facebook-goes-global-with-data-center-in-sweden/

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

[8] S. Rajasekaran, "Mesh connected computers with fixed and reconfigurable buses: Packet routing and sorting," *IEEE Transactions on Computers*, vol. 45, pp. 529–539, 1996.

[9] J. S. Vitter, "External memory algorithms and data structures: Dealing with massive data," *ACM Comput. Surv.*, vol. 33, no. 2, pp. 209–271, 2001.

[10] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, and P. Dubey, "Efficient implementation of sorting on multi-core SIMD CPU architecture," *Proc. VLDB Endow.*, vol. 1, pp. 1313–1324, August 2008. [Online]. Available: http://dx.doi.org/10.1145/1454159.1454171

[11] B. Gedik, R. R. Bordawekar, and P. S. Yu, "Cellsort: high performance sorting on the cell processor," in *Proceedings of the 33rd international conference on Very large data bases*, ser. VLDB '07. VLDB Endowment, 2007, pp. 1286–1297. [Online]. Available: http://dl.acm.org/citation.cfm?id=1325851.1325998

[12] 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.

[13] Y. Zhang and S. Zheng, "Design and analysis of a systolic sorting architecture," in *Parallel and Distributed Processing, 1995. Proceedings. Seventh IEEE Symposium on*, oct 1995, pp. 652 –659.

[14] "International technology roadmap for semiconductors 2011 edition," ITRS, Tech. Rep., 2011.

[15] D. Taniar and J. W. Rahayu, "Sorting in parallel database systems," in *Proceedings of the The Fourth International Conference on High-Performance Computing in the Asia-Pacific Region-Volume 2 - Volume 2*, ser. HPC '00. Washington, DC, USA: IEEE Computer Society, 2000, pp. 830–. [Online]. Available: http://portal.acm.org/citation.cfm?id=822078.822580

[16] ——, "Parallel database sorting," *Inf. Sci. Appl.*, vol. 146, pp. 171–219, October 2002. [Online]. Available: http://portal.acm.org/citation.cfm?id=634802.634815

[17] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha, "GPUTerasort: High performance graphics co-processor sorting for large database management," in *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA: ACM, 2006, pp. 325–336.

[18] S. Rivoire, M. A. Shah, P. Ranganathan, and C. Kozyrakis, "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

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

[20] D. Truong, W. Cheng, T. Mohsenin, Z. Yu, A. Jacobson, G. Landge, M. Meeuwsen, C. Watnik, A. Tran, Z. Xiao, E. Work, J. Webb, P. Mejia, and B. Baas, "A 167-processor computational platform in 65 nm CMOS," *Solid-State Circuits, IEEE Journal of*, vol. 44, no. 4, pp. 1130–1144, Apr. 2009.

[21] L. Stillmaker, "Saisort: An energy efficient sorting algorithm for many-core systems," Master's thesis, University of California, Davis, CA, USA, Sep. 2011. [Online]. Available: http://www.ece.ucdavis.edu/vcl/pubs/theses/2011-2/

[22] C. A. R. Hoare, "Algorithm 64: Quicksort," *Commun. ACM*, vol. 4, pp. 321–, July 1961. [Online]. Available: http://doi.acm.org/10.1145/366622.366644

[23] M. Butler, "AMD Bulldozer Core - a new approach to multithreaded compute performance for maximum efficiency and throughput," in *IEEE HotChips Symposium on High-Performance Chips (HotChips 2010)*, Aug. 2010.

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

[25] A. Stillmaker, Z. Xiao, and B. Baas, "Toward more accurate scaling estimates of cmos circuits from 180 nm to 22 nm," VLSI Computation Lab, ECE Department, University of California, Davis, Tech. Rep. ECE-VCL-2011-4, Dec. 2011. [Online]. Available: http://www.ece.ucdavis.edu/cerl/techreports/2011-4/