Journal of Parallel and Distributed Computing 138 (2020) 32–47

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Scalable energy-efficient parallel sorting on a fine-grained many-core processor array

Aaron Stillmaker [a,b,*], Brent Bohnenstiehl [a], Lucas Stillmaker [a], Bevan Baas [a]

[a] Electrical and Computer Engineering Department, University of California, Davis One Shields Ave. Davis, CA 95616, USA
[b] Electrical and Computer Engineering Department, California State University, Fresno 2320 E. San Ramon Ave. Fresno, CA 93740, USA

ARTICLE INFO

Article history:
Received 27 July 2018
Received in revised form 10 July 2019
Accepted 21 December 2019
Available online 26 December 2019

Keywords:
Parallel processing
External sorting
Scalable sorting
Fine-grained many-core
Processor array

ABSTRACT

Three parallel sorting applications and two list output protocols for the first phase of an external sort execute on a fine-grained many-core processor array that contains no algorithm-specific hardware acting as a co-processor with a variety of array sizes. Results are generated using a cycle-accurate model based on measured data from a fabricated many-core chip, and simulated for different processor array sizes. The data shows most energy efficient first-phase many-core sort requires over 65× lower energy than GNU C++ standard library sort performed on an Intel laptop-class processor and over 105× lower energy than a radix sort running on an Nvidia GPU. In addition, the highest first-phase throughput many-core sort is over 9.8× faster than the std::sort and over 14× faster than the radix sort. Both phases of a 10 GB external sort require 6.2× lower energy×time energy delay product than the std::sort and over 13× lower energy×time than the radix sort.

1. Introduction

Energy efficiency in large data centers is an increasingly important concern as these centers increase in server density and power consumption [25] while often running on a power-supplier-imposed peak power budget [10]. In the United States during 2014, data centers used 1.8% of the country's total power consumption, consuming an estimated 70 billion kWh [26]. Processing units are a large contributing factor to a system's overall power usage, both directly and indirectly through required cooling [10].

Sorting is one of the most-used processing kernels in database systems [13], creating an interest in energy conscious sorting methods [23]. Data centers with large data sets generally perform *external sorts*, where data cannot fit in volatile memory necessitating two separate phases [16].

With ever shrinking transistors, extra area is being dedicated to multiple processing cores, instead of adding more transistors to a single core [4,34,36,41,42]. 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 [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 general energy-efficient, scalable sorting algorithms for an external sort, with the first phase completed by a fine-grained many-core array of low-power, simple multiple instruction, multiple data (MIMD) processors. The sorts consist of small modular programs operating on each core, making them scalable to different array sizes. A many-core array serving as a co-processor performs the first phase of the sort, while a laptop-class processor performs the second phase. The concepts presented can be transferred to future 2D-mesh many-core processor arrays, enabled by the ever increasing number of devices per chip.

2. Sorting

As computers shifted to multi-core processors, sorting research adapted sorting algorithms to take advantage of multiple processing cores. The progression to many-core processors [4,36,43] necessitates a shift to sorting with large processor arrays.

2.1. Related work

Internal sorting is the sorting of data sets which can fit completely in main memory. Common examples include the merge, quick, radix, and bitonic sorts [27].

The merge, quick, and radix sorts are computationally simple, and in a processor array, each parallel processor either computes

* Correspondence to: California State University, Fresno, USA.
E-mail addresses: astillmaker@mail.fresnostate.edu (A. Stillmaker), bvbohnen@ucdavis.edu (B. Bohnenstiehl), lstillmaker@ucdavis.edu (L. Stillmaker), bbaas@ucdavis.edu (B. Baas).

https://doi.org/10.1016/j.jpdc.2019.12.011
0743-7315/

independent parts of a list or is mapped into a data path to perform a part of a sort and then pass the data to another processing stage. Using clever methods of partitioning and redistribution, parallel sorts show promising results [21,31]. Specific parallel sorting algorithms, such as the odd–even and bitonic merge sorts, were explicitly developed to exploit parallelism [3,9,11].

These methods rely on each processor's ability to efficiently access system I/O, a shared memory or processors other than their neighbors. This is a reasonable expectation for a small array, but not always the case for large many-core arrays.

The single instruction, multiple data (SIMD) GPUTerraSort takes advantage of large shared memories to attain high throughput [12]. However, with the high power of the GPU and its memory system, the sort is not energy-efficient.

Theoretical and early work on sorting with mesh-connected processor arrays on arbitrarily-sized arrays is not implementable on our targeted fine-grained many-core architecture because these theoretical discussions target a specific and incompatible architecture to our own and therefore was not compared with our results [14,22]. Sorting on systolic arrays is data driven, similar to our implementation, but the arrays need to either operate in lock step or as a wavefront array with required handshaking [43]. Because of major architectural differences including different communication methods, memory system differences, and timing differences, these ideas cannot be transposed onto a fine-grained many-core array, and with little recent work in the field, no comparison with this work was made.

Work has been done on sorting on a fine-grained many-core architecture, namely the development of SAISort [28], and the implementation of the SAISort kernel in sorting on a many-core array on the AsAP2 chip to perform the first phase of an external sort [30]. This paper implements these sorts, as well as a new sorting application, different output protocols, scaling on an arbitrarily-sized general many-core platform, and reporting on an entire external sort, instead of just the first phase.

JouleSort [23] is a system-level benchmark for energy efficiency of a large external database sort. The JouleSort benchmark compares total system power of a physical system, while the proposed many-core sorts are designed to work on a simulated co-processor. Therefore the proposed sorts could not utilize the JouleSort benchmark directly, though many details from the guidelines were used and a comparison is made with the processing power of a comparable JouleSort winner in Section 6.3.

Vitter [37] utilizes common sorting methods using different I/O paradigms on traditional general purpose processors instead of a many-core array to complete an external sort. CloudRAMSort [15] used a large cloud of distributed high performance processors that perform an external sort entirely in DRAM. CloudRAMSort utilizes distributed traditional processors as well as SIMD arrays that maximizes performance. Neither of these sorts report data which could be meaningfully compared to the presented sorts.

Work by Solomonik and Kale [27] analyzes the scalability of parallel sorting algorithms on large many-core processor arrays in large distributed systems. They sort 64-bit keys and perform an internal sort, and therefore not compared with our external sort.

Other work has been performed on simulating the computational time of a distributed network of processors, using varied numbers of processors for the computation. Baddar and Mahafzah [2] explored mapping an efficient bitonic merge sort onto a chained-cubic tree interconnection network, and simulated results of computation time using between 16 and 1024 processors in their network. Their work simulated many separate chips working in a large network, and only published delay values including communication delay, and therefore was not compared with our single many-core chip with 10 to 10,000 simple processors on a single die.

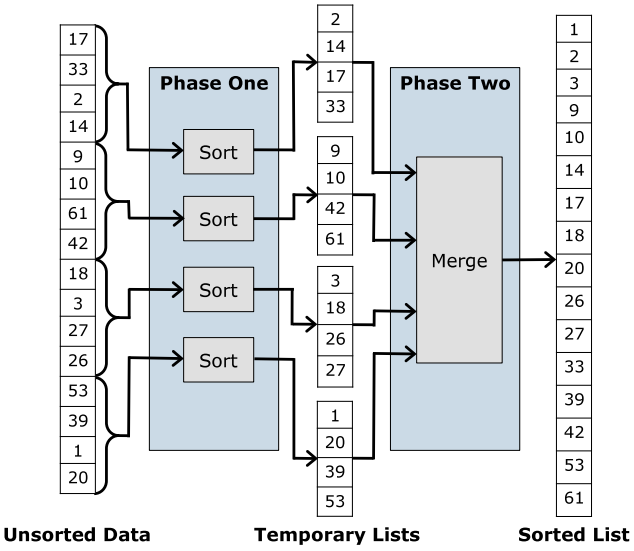


Fig. 1. Example of an external sort, where 16 unsorted numbers are sorted into 4 separate temporary sorted lists in phase 1, and merged together to form one sorted list. In this example, high-speed local memory is presumed to be of a size such that a maximum of 4 records can be sorted at a time.

2.2. External sorting

External sorting requires the use of a secondary storage system, like a hard drive, and is done in two phases. During the first phase, temporary sorted lists are created from unsorted data, which can fit inside main memory. In the second phase, these temporary lists are merged together to create one complete sorted list, as shown in Fig. 1. There are many efficient merging sorts on multi-core processors [24,31]. We implemented both first and second phases on a many-core array, and it was found the more complex first phase is computationally bound, while the second merging phase is generally I/O bound. Therefore, we focus on the first phase for a many-core implementation, and use a merge sort on a laptop-class processor to model the entire external sort.

2.3. Scaling to many-core

Challenges of implementing parallel sorts on large processor arrays include balancing the computational load amongst the processors and transmitting the data between processors and memories. Because of these challenges, most sorts do not scale well with more cores [27]. To get around these issues, many parallel processor sorts, such as GPU sorting algorithms, take advantage of shared caches or shared memories to easily access a list, and have many different processors work on different parts of one list.

Most commonly-used multi-core sorts do not scale well to a many-core architecture because they require access to a large shared memory, or high speed networks to allow for shifting and swapping of records, as in the bitonic merge sort. These more complex sorts are not suited for many-core array architectures, and are not explored in this paper. Our target architecture is a low power, small area, many-core system with 10's to 1000's of cores per chip [1].

2.4. Streaming co-processor

This paper presents sorting with a large array of processors that have communication only with nearest neighbors and

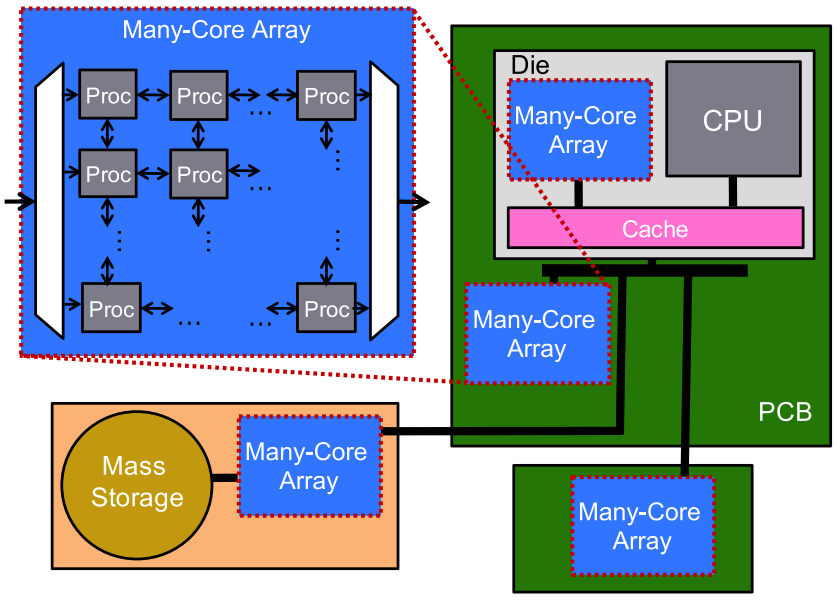


Fig. 2. Diagram showing possible locations for integration of a many-core co-processor into a computational system. A fine-grained many-core architecture processor array is shown with nearest-neighbor communication, chip data input on the left side of the array, chip data output on the right, and no global shared memory.

limited long-distance communication. The targeted architecture is a large streaming general purpose MIMD array, such as the AsAP2 [34] and KiloCore [5] chips. Only processors on the edge of the array have access to chip I/O and the chip contains no explicit global shared memory. The proposed sorts are designed to use a many-core array as a co-processor working in tandem with a general purpose processor, allowing it to use its complex high power circuits on more appropriate computations. While a sorting ASIC would have a higher performance than a general-purpose array, using a many-core array co-processor gives the benefit of configurability to perform other desirable tasks, such as compression [38] and encryption [17] with the same hardware. Fig. 2 shows a generic view of the fine-grained many-core architecture and different ways that it could be connected into a computer system.

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 traditional general purpose processor or another memory, not involving the traditional general purpose processor during the first phase.

In order to show quantitative results, it was necessary to use measured values from an existing architecture. Therefore, the architecture of the fabricated general-purpose many-core array, AsAP2 [34] was used for certain limitations and measured values. This chip was originally designed with DSP applications in mind, and the minimalist instruction set has no specialized instructions which could be used to accelerate sorting. Each processor contains its own clock domain and oscillator, and has the ability to turn off its oscillator when stalled, which causes active power dissipation to fall to zero (leakage only) while a processor is idle [39]. Processors are able to individually change their supply voltage to further reduce power dissipation [8], but this feature is unused in this work. Processors are connected by a 2D circuit-switched mesh network, allowing for nearest-neighbor communication to adjacent processors [40], as well as long-distance communication that bypasses cores to connect non-adjacent cores [33]. Using the AsAP many-core architecture, we were limited to its reduced instruction set architecture with sixty-three instructions, 128 words of instruction memory, 256 bytes of data memory, and two 128-byte first in, first out (FIFO) buffers, for inter-processor communication across clock boundaries per processor. For modeling purposes we used the physically measured traits from the fabricated 65-nm chip, where each processor takes up 0.17 mm^2 of area [35].

3. Proposed sorting kernels

The sorting applications, described in Section 4, utilize basic program kernels in each of the processors in the array, all of which are newly proposed sorting methods. Each kernel was designed for modularity with no knowledge about a processor's location or temporary list size. There were three specific limitations of our target architecture which prevented more complex or widely used parallel sorting methods to be explored. One is the limitation of 128 instructions, which prevented the use of more complex sorting algorithms which required more instructions. The second limitation was communication, which prevented the use of algorithms which require large communication networks, such as the bitonic merge sort network. The last limitation was the data memory size, which shaped the size of the possible temporary lists. All of these architectural artifacts do not diminish the relevance of these proposed sorts, as these algorithms may be implemented on a range of many-core arrays, from simple to more complex processing elements.

3.1. SAISort kernel

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 the sort on the micro scale is a traditional insertion sort, and when multiple kernels are serially linked together, a traditional bubble sort variant is created in the macro scale. The bubble sort, which is a worst case $\mathcal{O}(n^2)$ algorithm, will govern (as the insertion sort operates on a constant size) and make the worst case computational complexity of all sorts using the SAISort kernel to sort to also be $\mathcal{O}(n^2)$.

As shown in Algorithm 1, when a fresh list is sent through the SAISort kernel, it starts by filling up its internal memory with a sorted list, as records arrive. Our target architecture has a local memory of 256 bytes, so only two 100-byte records can be locally

stored at a time. Once the local memory is full, each new record is sorted into the local list and the lowest valued record is sent out. A code word preceding each record is reserved for triggering a reset at the end of a temporary list, allowing temporary lists to begin without foreknowledge of the list's size. When the kernel detects a reset signal, it passes the reset signal to its output and begins the record output streaming protocol described in Section 4.4.

3.2. Merge kernel

The merge kernel, shown in Algorithm 2, compares the keys received on each of its two inputs and passes the record with a lower key value to its output. When a reset signal a received from one input, records from the other input are passed directly to the output until a second reset signal is received. The kernel passes the reset signal and then resumes merging.

3.3. Split kernel

The split kernel, shown in Algorithm 3, is given a constant value at compile time for the number of split cores past the current core. The kernel passes that many records to the next split processor, then takes one record for its current row. It continues in this fashion until it receives the reset signal, at which point it forwards the reset signal to both the next split processor and the current row, then starts from the beginning of the program.

3.4. Distribution kernel

The distribution kernel, shown in Algorithms 4, 5, and 6, 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. When a record enters a distribution core, its key is compared to the core's unique threshold value, and passes the record to the next distribution core if above the threshold or is processed to be sent into a row otherwise, as shown in Algorithm 6. These thresholds progressively increase along the series of distribution cores such that a random key has an approximately equal chance to be sent to any row.

The rate at which records are admitted into the array is controlled by the first distribution core, using information sent from other distribution cores to avoid overflowing any lane. Records are streamed in a two stage algorithm, as described below.

During the first stage, records are streamed from the first distribution core up to a limit of the minimum remaining capacity of any row. With each record passed, a counter for records pending response is incremented. When the count has passed half of the target limit, a request is sent out for updated information. Distribution cores pass the request downstream if they have passed records downstream since the previous response, else they respond to the request. The response consists of the minimum capacity of either sorting row in this lane or the minimum of the next upstream core, and a count of the number of records processed since the last response was sent. Both of these values are updated by each core passing the response.

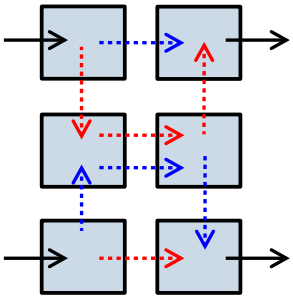


Fig. 3. A block of 6 Adaptive Sorting processors route data to dynamically change the data path. Either the shown blue or red routing is taken depending on need. These 6 processors constitute a part of a single lane, which contains two sorting rows. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

When the first distribution core reaches a number of records pending response equal to its stored minimum remaining capacity, it reads the response to update this capacity and reduce the counter for records pending response. Records resume streaming using this updated information. When the safe record limit is sufficiently large, the first distribution core may continue to stream records during the entire time the requested response is being processed and returned.

The second stage of distribution is entered when the minimum safe limit falls below a threshold value, set to approximately the number of distribution cores for optimization. A signal is sent downstream through the distribution cores indicating the change in stage. During this stage, whenever a distribution core processes a packet it automatically sends a response upstream containing an updated minimum capacity. The first distribution core progressively checks these responses and admits additional records into the array as it verifies there is guaranteed remaining capacity for them.

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 sends a reset request to the upstream and downstream distribution cores, as well as both of its lanes sorting rows. The core then resets its state and reprocesses the record that triggered the reset. Other distribution cores propagate the reset signal and reset. A reset signal may also be sent into the core array to force a reset.

The advantage of stage one is that it avoids excessive processing of responses, reducing the cycles needed per record and filling the array quickly when room is plentiful. The advantage of stage two is that it provides much finer grained response arrival to the first distribution core when the ability to hide response latency is limited by the low minimum safe remaining capacity of one or more lanes.

3.5. Dynamic routing

Assignment of the center SAISort processors in a lane to the upper or lower rows is performed in 3×2 blocks, with two cores in each row as shown in Fig. 3. Upon reset, a block defaults to assigning the central cores to the lower row; upon reception of a reconfiguration signal from the distribution core, the central cores are reassigned to the upper row. The 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

Algorithm 1 SAISort Kernel

```
1: RecCnt ← 0
2: while true do
3:     if Input ≠ Reset then
4:         if RecCnt < 2 then
5:             SrtdList[RecCnt] ← InRec
6:             if RecCnt == 0 then
7:                 RecCnt ← 1
8:             else if SrtdList[0].Key ≤ SrtdList[1].Key then
9:                 RecCnt ← 2
10:            else
11:                RecCnt ← 3
12:            end if
13:        else
14:            if InRec.Key ≤ SrtdList[RecCnt − 2].Key then
15:                Output ← InRec
16:            else
17:                Output ← SrtdList[RecCnt − 2]
18:                SrtdList[RecCnt − 2] ← InRec
19:                if SrtdList[0].Key ≤ SrtdList[1].Key then
20:                    RecCnt ← 2
21:                else
22:                    RecCnt ← 3
23:                end if
24:            end if
25:        end if
26:    else
27:        Output Procedure (See Section 4.4)
28:        RecCnt ← 0
29:    end if
30: end while
```

Algorithm 2 Merge Kernel

```
1: while true do
2:     ResetFlag ← 0
3:     while ResetFlag == 0 do
4:         if Input1Rec.Key < Input2Rec.Key then
5:             Output ← Input1Rec
6:             if Input1 == Reset then
7:                 ResetFlag ← 1
8:             end if
9:         else
10:            Output ← Input2Rec
11:            if Input2 == Reset then
12:                ResetFlag ← 2
13:            end if
14:        end if
15:    end while
16:    if ResetFlag == 1 then
17:        while Input2 ≠ Reset do
18:            Output ← Input2Rec
19:        end while
20:    else
21:        while Input1 ≠ Reset do
22:            Output ← Input1Rec
23:        end while
24:    end if
25: end while
```

Algorithm 3 Split Kernel

```
1: while true do
2:     CountPass ← 0
3:     while Input ≠ Reset do
4:         if CountPass ≠ Ratio then
5:             OutputPass ← InRec
6:             CountPass + +
7:         else
8:             OutputRow ← InRec
9:             CountPass ← 0
10:        end if
11:    end while
12:    OutputPass ← Reset
13:    OutputRow ← Reset
14: end while
```

Algorithm 4 Radix Distribution

```
1: while true do
2:     if ThisCore == FirstCore then
3:         if (AutoRespMode + PipedResponseReq) == 0 then
4:             if RecordsPending ≥ PipeMaxCountDiv2 then
5:                 DwnStrOutput ← ResponseRequest
6:                 PipedResponseRequests + +
7:             end if
8:         else
9:             if RecordsPending ≥ PipeMaxCount then
10:                Process Response (See Algorithm 5)
11:            else
12:                Process Packet (See Algorithm 6)
13:            end if
14:        end if
15:    else if UpStreamInput/neq Empty then
16:        Process Response (See Algorithm 5)
17:    else if DwnStrInput/neq Empty then
18:        Process Record (See Algorithm 6)
19:    end if
20: end while
```

Algorithm 5 Process Response

```
1: if DwnStrInput == Reset then
2:     (UpStream, UpRow, LowRow)Output ← Reset
3:     Perform Reset
4: end if
5: if ThisCore == FirstCore then
6:     RecordsPending− = InputAccountedRecordsTotal
7:     if RecordsPending ≤ AutoResponseCuttoff then
8:         AutoRespMode ← 1
9:         DwnStrOutput ← AutoRespEnSignal
10:    end if
11:    PipeMaxCount ← InputCapacity
12:    PipeMaxCountDiv2 ← InputCapacity >> 1
13: else
14:    MinCap ← MIN(UpRoom, LoRoom, DwnStrCap)
15:    UpStreamOutput ← MinCap
16:    UpStreamOutput ← InputAccountedRecordsTotal + AccountedRecords
17:    AccountedRecords ← 0
18: end if
```

Algorithm 6 Process Record

```
1: if UpStreamInput == Reset then
2:     (DwnStr, UpRow, LowRow)Outputs ← Reset
3:     Perform Reset
4: else if UpStreamInput == ResponseRequestSignal then
5:     if RecordPassedSinceLastResponse == 1 then
6:         RecordPassedSinceLastResponse ← 0
7:         DwnStrOutput ← ResponseRequestSignal
8:     else
9:         MinCap←MIN(UpRoom, LoRoom, DwnStrCap)
10:        UpStreamOutput ← MinCap
11:        UpStreamOutput ← AccountedRecords
12:        AccountedRecords ← 0
13:    end if
14: else if UpStreamInput == AutoRespEnSignal then
15:    AutoRespMode ← 1
16:    DwnStrOutput ← AutoRespEnSignal
17: else if Key > CuttoffValue then
18:    RecordsPassedSinceLastResponse ← 1
19:    DwnStrOutput ← Record
20:    if ThisCore == FirstCore then
21:        RecordsPending+ = 1
22:    end if
23: else
24:    if ThisCore ≠ FirstCore then
25:        AccountedRecords+ = 1
26:    end if
27:    if Key ≤ LowerRowCuttoffValue then
28:        SelectedRow ←LowerRow
29:        OtherRow ←UpperRow
30:    else
31:        SelectedRow ←UpperRow
32:        OtherRow ←LowerRow
33:    end if
34:    if SelectedRowRoom ≠ 0 then
35:        SelectedRowRoom − −
36:        if AutoRespMode == 1 then
37:            MinCap ← MIN(UpRoom, LoRoom, DwnStrCap)
38:            UpStreamOutput ← MinCap
39:            UpStreamOutput ← 1
40:        end if
41:        SelectedRowOutput ← Record
42:    else
43:        if SelectedRowUnassignedNodes ≠ 0 then
44:            if SelectedRow == UpperRow then
45:                UpperRowOutput ← ReconfigSignal
46:                LowerRowOutput ← ReconfigSignal
47:            end if
48:            SelectedRowRoom+ = 7
49:            OtherRowRoom+ = 4
50:            SelectedRowUnassignedNodes− = 1
51:            if AutoRespMode == 1 then
52:                MinCap ← MIN(UpRoom, LoRoom,
53:                                DwnStrRoom)
54:                UpStreamOutput ← MinCap
55:                UpStreamOutput ← 1
56:            end if
57:            SelectedRowOutput ← Record
58:        else
59:            AllOutput ← Reset
60:            Reprocess Latest Record
61:        end if
62:    end if
63: end if
```

central cores 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 delay reconfiguration until the signal is received.

4. Proposed sorting applications

Several different new sorting schemes are proposed with the goal of exploring applications that could be implemented in a wide range of simple 2D mesh processing arrays. The following sorting applications fit within our target architecture limitations, utilize simple modular kernels, and scale easily with processor array sizes.

4.1. 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 temporary list by two.

One final record can be added to the temporary list, giving the temporary list size Eq. (1) where n is the number of processors

in the array and R is the number of records that can fit inside a single processor's local memory.

$$TempListSize = n \times R + 1 \quad (1)$$

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 temporary list has been sent into the chip, a reset signal is sent through the snake triggering a flush in each processor. The output protocol is covered in Section 4.4.

A generalized mapping for the Snake Sort is shown in Fig. 4 which displays how more processors are used.

4.2. 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 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. 5 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 temporary list than snake sort. The total number of records in a temporary list can be given by Eq. (2) where r is the number of rows, c is the number of columns, and R is the number of records that can fit in a single processor's local memory.

$$TempListSize = (r \times c - 2 \times (r - 1)) \times R + r \quad (2)$$

The number of sorting processors is given by the total number of processors, r times c, minus $r - 1$ distribution and $r - 1$ merge processors. The bottom left and upper right processors are not required for distribution or merging respectively, as shown in Fig. 5. 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 were the merge column of processors.

4.3. Adaptive sort

In adaptive sort, the processor array is partitioned into a number of sorting lanes. Each lane begins with one 3×1 block containing one distribution processor and two basic SAISort processors. The lane is then expanded with some number of 3×2 adaptive sorting blocks, shown in Fig. 3, each containing the SAISort kernel with additional reconfiguration code, and connected as shown in Fig. 6. Each lane is responsible for sorting two rows of radixes.

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 that 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 temporary list. The disadvantages are a more complicated distribution algorithm and a non-deterministic number of records per temporary list, as each run ends when a radix is full.

The maximum potential temporary list size is given Eq. (3) where L is the number of sorting lanes, b is the number of adaptive 3×2 blocks per lane, and R is the number of records that can fit in one processor's local memory. Similar to snake sort, one extra record can be sorted for each radix, or potentially two per lane.

$$TempListSize = L \times R \times (2 + 6 \times b) \quad (3)$$

4.4. Two output streaming protocols

When the chip contains a completely sorted temporary list, it then needs to stream that data out of the array. Two such protocols were explored.

4.4.1. Flush

With this protocol, each processor will receive a count of how many records are held by upstream processors, output its two local stored records, and subsequently, directly pass the upstream records from input to output. While simple, the flush protocol requires the first sorting processor to hold its stored records until all downstream processors have passed their own stored records, delaying the beginning of a new temporary list. The delay would be even worse if the buffers used for inter-processor communication were not large enough to take an entire record. The pseudo code is shown in Algorithm 7.

4.4.2. Pass and store

This protocol is similar to flush except that input records are temporarily stored in a processor's local memory before being passed to output as shown in Algorithm 8. The processor will continually send out its lowest record and then store a new record into the vacated memory. When all upstream records have been read, the processor will output its final stored records. Pass and store effectively compresses a temporary list into the downstream processors while waiting for records to exit the chip, allowing the upstream processors to begin a new temporary list almost immediately. Compared to flush, the drawbacks of pass and store include increased code size, more than doubled operations per record, and additional memory reads and writes.

5. Experimental methodology

All of the proposed sorts follow the Sort Benchmark [19] requirement of 100-byte records, which is composed of a uniformly random 10-byte key and a 90-byte payload. As the Sort Benchmark only uses random data, no other types of data distributions are explored. The gensort program [18] was utilized to create random data sets for all experiments.

The most relevant Sorting Benchmark to the proposed work is the JouleSort [23], which measures the total system energy cost of sorting a data set. This paper proposes using a many-core array as a co-processor in a database system, freeing the general purpose CPU to process other workloads during the first phase of sorting. Therefore, it would be uninformative to try and add a system power to our sorting results, as required by the JouleSort, so we focus on just the processing energy, which is not reported for other sorts. There is no way to compare our first phase results to other results, as metrics separated by phase are rarely published, and even if they were, they would not match the size of our temporary lists, which would make any comparison uninformative.

We compare our proposed many-core sorting results with an Intel Core2 Duo T7700 and an Nvidia GeForce 9600 m because no published results are available for a wide variety of block sizes such as the ones in Table 1, and because hardware and software tools are widely available for them. In addition, although the design team sizes and resources are vastly different for the three platforms, they are all built from the same 65 nm fabrication technology generation which results in a very fair comparison

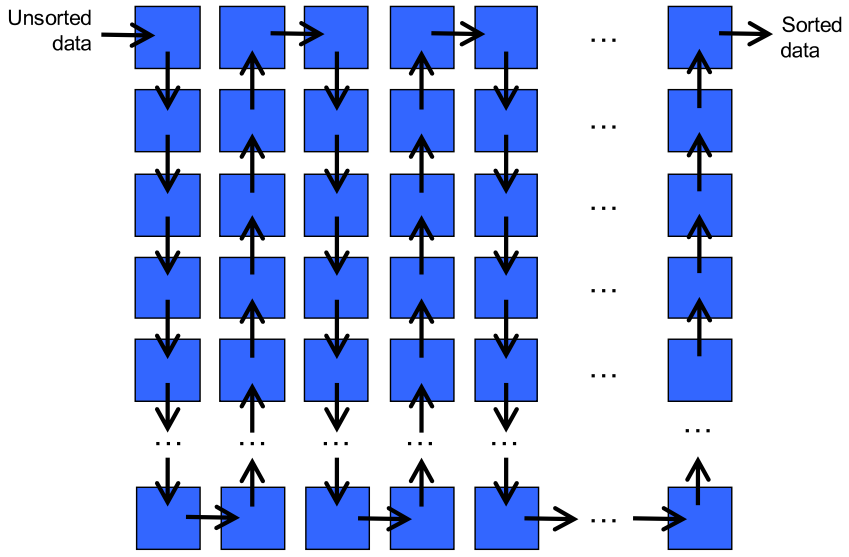


Fig. 4. An example snake sort mapping showing the communication path and how it can be scaled.

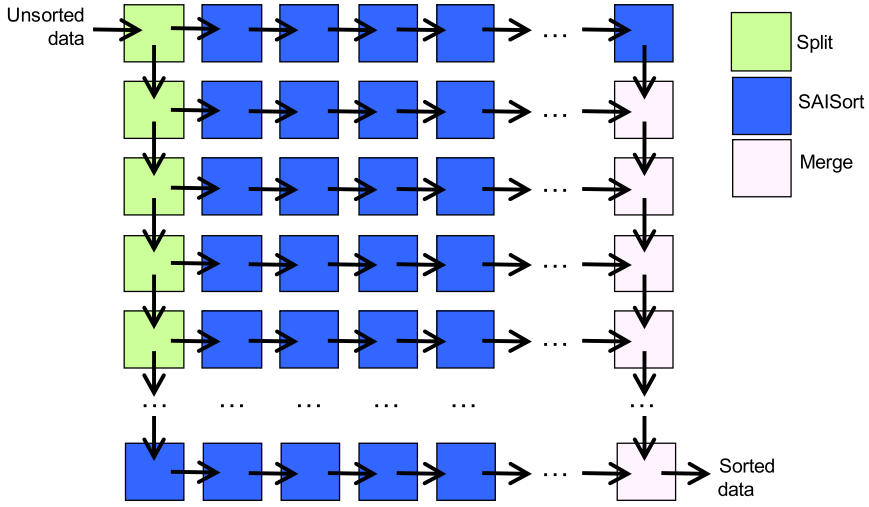


Fig. 5. An example row sort mapping showing the communication path and how it is scaled.

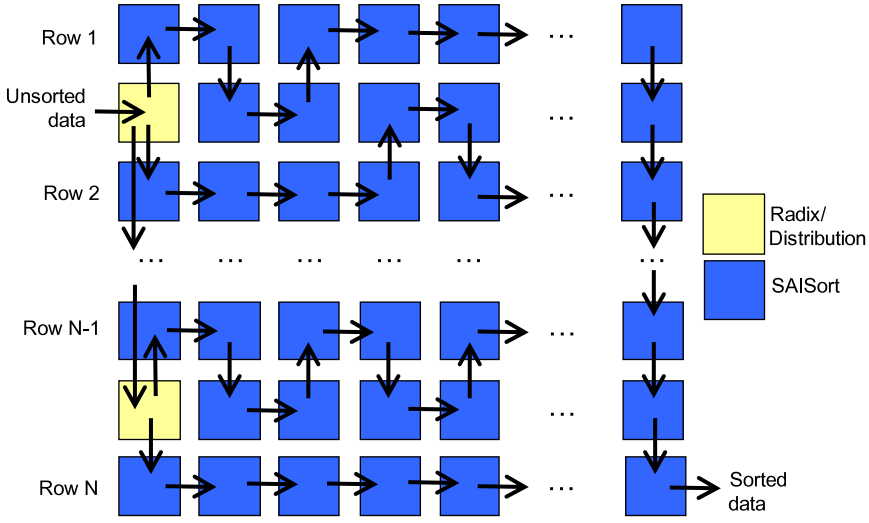


Fig. 6. An example adaptive sort mapping showing the communication path and how it is scaled. There are two radix rows assigned to every three rows of processors, where the central processors are dynamically allocated during runtime. In this example rows 1 and 2 filled up at a similar rate so they were allocated a similar number of middle processors, but row N-1 filled up faster than row N, so all of the middle processors were dynamically allocated to row N-1.

Algorithm 7 Flush

```
1: NumToPass ← Input
2: Output ← Reset
3: if RecCount == 2 then
4:     Output ← NumToPass + 2
5:     Output ← SortedList[0 : 1]
6: else if RecCount == 3 then
7:     Output ← NumToPass + 2
8:     Output ← SortedList[1]
9:     Output ← SortedList[0]
10: else if RecCount == 1 then
11:    Output ← NumToPass + 1
12:    Output ← SortedList[0]
13: end if
14: for i = 1 to NumToPass do
15:    Output ← InputRec
16: end forreturn
```

Algorithm 8 Pass and Store

```
1: NumToPass ← Input
2: Output ← Reset
3: if RecCount < 2 then
4:     if RecCount == 1 then
5:         Output ← NumToPass + 1
6:         Output ← SortedList[0]
7:     else
8:         Output ← NumToPass
9:     end if
10:    for i = 1 to NumToPass do
11:        Output ← InputRec
12:    end forreturn
13: else if RecCount == 2 then
14:    Output ← NumToPass + 2
15: else
16:    Output ← NumToPass + 2
17:    if NumToPass ≥ 1 then
18:        Output ← SortedList[1]
19:        SortedList[1] ← InputRec
20:        NumToPass − −
21:    end if
22: end if
23: for i = 1 to NumToPass do
24:    if i rem 2 == 1 then
25:        Output ← SortedList[0]
26:        SortedList[0] ← InputRec
27:    else
28:        Output ← SortedList[1]
29:        SortedList[1] ← InputRec
30:    end if
31: end for
32: if NumToPass rem 2 == 1 then
33:    Output ← SortedList[1]
34:    Output ← SortedList[0]
35: else
36:    Output ← SortedList[0 : 1]
37: end ifreturn
```

in this critical parameter. Technology scaling trends can give a first order prediction of how a device or ASIC would perform in a different technology, however, as smaller transistors allow for a higher transistor density, other architectural and system design considerations affect large scale chip and processor performance between technology nodes [29]. Therefore, scaling the performance of an external sort on a complex processor to a different technology node is not possible. Table 1 also contains a comparison with a JouleSort winner which used a 65 nm Intel Core2 T7600 processor, the CoolSort [23].

5.1. Many-core sorts

The scalable many-core architecture is modeled using a cycle-based, two-state simulator written in C++ which is able to model arbitrarily sized arrays. 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 circuit network. Energy usage was taken from physical chip measurements for each type of operation, memory accesses, network access, oscillator activity, and leakage. The simulator then matched every operation performed with the measured values to extrapolate an energy consumption value which is accurate to the fabricated and measured chip mentioned in Section 2.4. A 167-core sort was ran on the physical AsAP2 chip, which had matching results to the simulator.

Each of the proposed sorts is implemented on many-core arrays with the number of processors scaled from 10 to 10,000 processors with a clock speed of 1.2 GHz at 1.3 V. Phase 1 results are shown in Section 6.2. Some selected points for the size of the array are used to compare full external sorts in Section 6.3.

5.2. Intel GNU C++ standard sort

The GNU C++ standard library sort function was implemented on an Intel Core2 Duo T7700, a 65 nm chip with a TDP of 35 W and a clock frequency of 2.4 GHz. As accurate energy or power numbers are not given for Intel processors, 50% of the TDP for the Intel processors were used to determine the energy consumption [7]. The main memory was preloaded with records for multiple temporary lists of a size matched to the many-core sorts. Thousands of runs were timed to calculate the throughput. The program was compiled with the GCC compiler version 4.2 and optimization set to –O3.

5.3. Nvidia GPU radix sort

The radix sort was chosen for the GPU platform because there was a version publicly available with the Nvidia CUDA SDK version 2.3, written by Satish et al. [24]. The sort was implemented on an Nvidia GeForce 9600 m GT, a 65 nm chip with 32 processing elements, a clock speed of 500 MHz, and a power consumption of 23 W. This power consumption value was multiplied by the computation time to get the expended GPU energy. The older CUDA SDK version was chosen for its compatibility with our CUDA compute 1.1 GPU. The original sort was designed to efficiently work with the 32 bit data width, so it only sorted keys that

Table 1
Time, energy, area, and energy delay product for 65 nm platforms performing a 10-GB external sort of 100-B records. Phase 2 for each of the presented sorts is accomplished with a merge sort on an Intel T7700.

Records Per Temp. List	Phase 1 Processor	Phase 1 Sort	Output Protocol	Phase 1 Time (s)	Phase 1 & 2 Time (s)	Phase 1 Energy (J)	Phase 1 & 2 Energy (J)	Phase 1 & 2 Area×Time (mm^2 s 10^3)	Phase 1 & 2 E×D (J s 10^4)
468		Row		9.054	50.22	14.5	893	7.58	4.43
501		Snake	Flush	21.39	50.22	116	995	8.11	4.66
365	256 Many-Core	Adaptive		13.32	53.01	15.2	943	8.16	4.94
468	Processors	Row	Pass	7.190	50.22	14.8	894	7.49	4.44
501		Snake	and	13.08	50.22	146	1030	7.75	4.61
365		Adaptive	Store	12.79	53.01	19.1	947	8.14	4.94
468	Intel Core2 T7700	std::sort		54.41	104.6	952	1831	15.0	9.59
468	Nvidia 9600M GT	Radix		66.67	72.25	1530	2410	16.8	14.6
3 919		Row		8.516	41.85	42.9	775	8.92	3.10
4 001		Snake	Flush	21.89	41.89	926	1660	13.5	5.09
3 657	2,025 Many-Core	Adaptive		13.14	41.85	46.4	779	10.5	3.13
3 919	Processors	Row	Pass	7.163	41.85	44.3	777	8.45	3.10
4 001		Snake	and	13.18	41.85	1170	1900	10.5	4.61
3 657		Adaptive	Store	13.23	41.85	58.4	791	10.5	3.14
3 919	Intel Core2 T7700	std::sort		70.74	112.59	1240	1970	16.1	11.8
3 919	Nvidia 9600M GT	Radix		104.2	109.7	2400	3130	21.0	28.0
19 704		Row		8.796	36.28	102	737	20.1	2.39
20 001		Snake	Flush	22.60	36.46	4670	5310	43.6	12.9
16 737	10,000 Many-Core	Adaptive		14.39	36.29	118	753	29.6	2.47
19 704	Processors	Row	Pass	7.478	36.27	105	739	17.9	2.38
20 001		Snake	and	13.59	36.29	5880	6520	28.3	10.3
16 737		Adaptive	Store	14.78	36.30	149	783	30.3	2.52
19 704	Intel Core2 T7700	std::sort		84.80	121.1	1480	2120	17.3	14.9
19 704	Nvidia 9600M GT	Radix		114.9	120.5	2640	3280	21.7	32.7
–	Intel Core2 T7600	CoolSort [23]		–	86.6	–	1470	12.4	12.7

were 32 bits or smaller. Therefore it was necessary to modify the code to work with the 80-bit keys and 100-byte total records. The main memory of the graphics card was preloaded with records for multiple temporary lists in sizes matching the inputs for the many-core sorts before the timing was started. Thousands of runs were processed through the platform to get an average throughput result.

Because of the SIMD nature of the GPU processing array, if a data set does not completely pack the array, excess processing elements will have wasted cycles. This can be reduced with large temporary list sizes, but with some of the smaller temporary list sizes tested, this became a factor causing sub-optimal results.

After the GPU used in this comparison was created, Nvidia developed GPUs which have architectural advancements targeted at GPGPU programming, as well as a toolkit which is not backwards compatible and contains a sort that is more efficient because it uses these new advances. Contemporary comparisons between the 65 nm many-core chip and newer GPU hardware using the thrust library are given in Section 6.4.

5.4. Intel merge sort

In order to simulate an entire external sort, it was necessary to write a merge sort for phase 2. This was implemented in C++ on the same Intel Core2 Duo processor described in Section 5.2. One GB of sorted lists were loaded onto main memory and timed separately from the sort where thousands of records were merged to get a throughput number. As large arrays of hard drives are used to negate the access and read/write times, we used this assumption in our models after determining our bandwidth was comfortably under the theoretical bandwidth of 68 Gb/s in the previous JouleSort winner, FAWNSort's setup [20]. The throughput number was used to model the total time to complete phase two for the different temporary list sizes. The modeled phase 2 was performed with multiple passes, hierarchically merging each of the temporary lists into larger temporary lists, until the final sorted list contains the entirety of the input data set.

6. Results

Performance results are presented for a 10 GB sort, where the first phase of the external sort is performed on many-core arrays of varying sizes, as well as comparison sorts on an Intel laptop-class processor and an Nvidia GPU. The second phase of the sort for all of the experiments was performed on a laptop-class Intel laptop-class processor using a hierarchical multi-pass merge sort. Results presented for the many-core array were generated using a cycle-accurate model of a many-core architecture, which was developed using measured results from a fabricated chip using that architecture.

6.1. SAISort kernel

An array of 500 SAISort cores was simulated with uniform random input data to characterize an individual processing core. A single sort requires an average of 149.12 single issue clock cycles per record per core to complete a sort. Many of these cycles are dedicated to data movement, as each record contains 50 data words. This requires an average of 8.37 nJ/rec per core. The flush output protocol requires an average of 61.02 CC/rec per core and 2.75 nJ/rec per core, while the pass and store output protocol requires an average of 108.89 CC/rec per core and 5.32 nJ/rec per core.

6.2. Processor scaling sorting results

The three proposed many-core sort applications were simulated using between 10 and 10,000 processors in a single square array.

6.2.1. Data distributions

Fig. 7 displays the throughput of the three proposed sorts with different data distributions: uniform random, all zeros, three uniform random keys ANDed together, already sorted data where

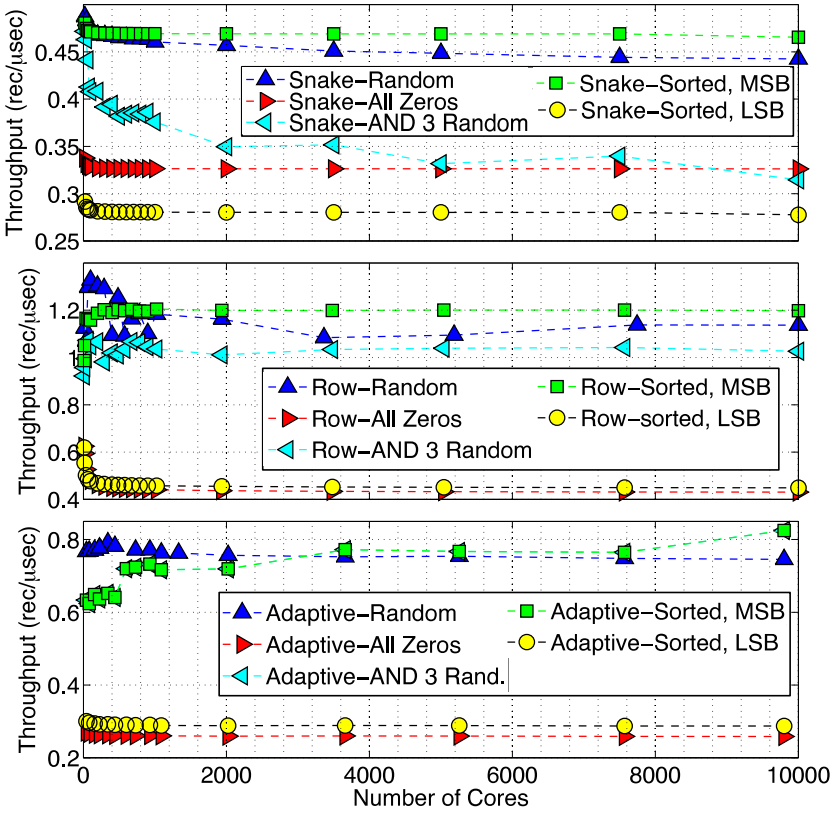


Fig. 7. Throughput of sorts using a many-core array while scaling the number of processors. With each new processor, the temporary list size increases, as described in Eqs. (1)–(3), which means that the data set, or amount of work per run, increases as the number of processors on chip are increased.

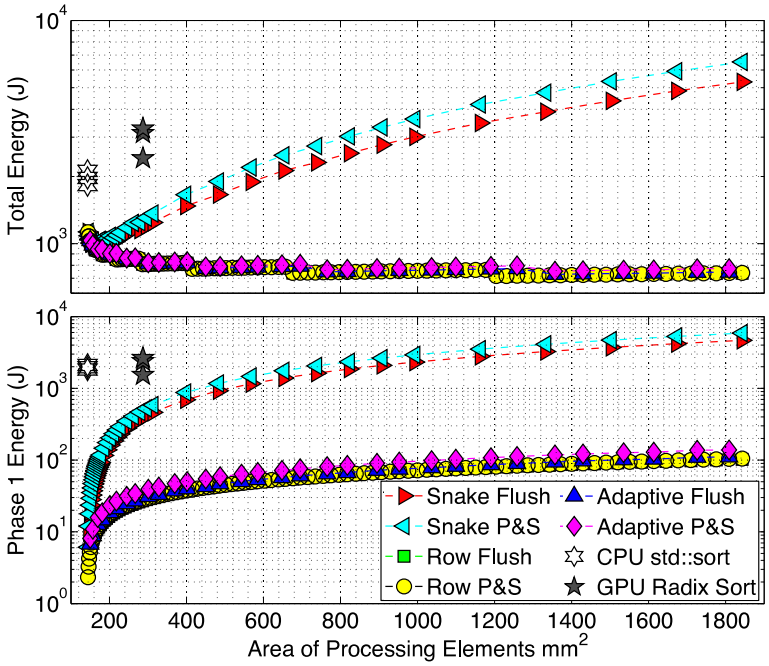


Fig. 8. Energy used by the many-core array to sort a 10 GB data set while scaling the number of processors in the array. With each new processor, the temporary list size increases, as described in Eqs. (1)–(3), which means that the data set, or amount of work, increases as the number of processors on chip is increased. The top plot includes energy for phase 1 + phase 2, where the bottom plot includes only energy for phase 1. Each of the comparison sorts was ran with three different temporary list sizes to make a meaningful comparison to the proposed sorts, as shown in Table 1. All data are from the 65 nm platforms mentioned in Section 5.

the first 16 bits contain the changing data, and already sorted data where the last 16 bits contain the changing data. The data set with uniform random has an entropy of 1.0 bits, the all zeros have an entropy of 0 bits, and the three ANDed keys have an entropy of 0.544 bits [32]. All of the data in Fig. 7 use the flush output protocol. As the sort benchmark uses 10 byte keys, and the

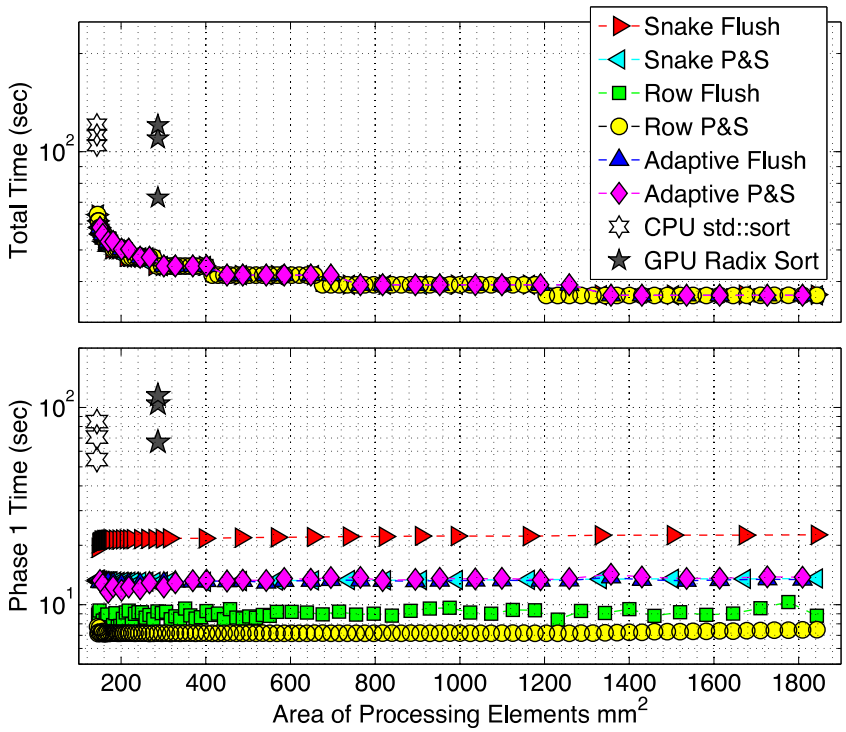


Fig. 9. Time to sort a 10 GB data set using a many-core array while scaling the number of processors in the array. The number of records sorted per temporary list is dependent on the number of processors, as described in Eqs. (1)–(3). This means that the data set, or amount of work, increases as the number of processors on chip is increased. The top plot includes time for phase 1 + phase 2, where the bottom plot includes only time for phase 1. Each of the comparison sorts was ran with three different temporary list sizes to make a meaningful comparison to the proposed sorts, as shown in Table 1. All data are from the 65 nm platforms mentioned in Section 5.

target architecture contains a 2 byte word size, the throughput of each of the proposed sorts is decreased when the MSBs of two keys match, requiring multiple comparison instructions for a single record comparison. Sorted keys with the MSBs storing the changing data is the optimal data set for the proposed sorts, as only one comparison must be made, and there are no repeated keys. The adaptive sort has greatly decreased throughput for both the already sorted LSBs and all zeros data sets, as this sends all of the incoming data into a single sorting bin, which forces very small run sizes, with only a fraction of the array utilized. There are minor differences in how the sorting applications deal with identical numbers, which results in the worst throughput for row and adaptive to be all zeros, and the snake's worst case to be already sorted with LSBs.

6.2.2. Alternative linear scaling

Each individual sort on the many-core array has a specific throughput, shown in Fig. 7, where the throughput of the single sort stays relatively similar no matter how big the sorted list is. Therefore an alternative scaling would be to use the new processing elements to create multiple parallel sorts, creating multiple smaller sorted lists, as opposed to using new elements to increase the phase one sorted list size. With this replication method, the throughput would scale linearly with the number of parallel sorts being executed.

6.2.3. 10 GB sort

Fig. 8 shows the energy usage to sort an entire 10 GB uniform random data set for the different implementations. The snake sort energy usage is considerably greater than the row sort consumption and to a lesser extent, the adaptive sort consumption. The snake sort has a greater rate of increase of energy consumption, where the other two have a diminishing rate of increased energy consumption. This happens because with the snake sort every record must be operated on by every processor in the sort, where the other two sorts divide the workload. With the row and adaptive sorts, the individual workload of a SAISort processor only goes up with each additional processor added to a row, and would go down with the addition of another row.

Fig. 9 shows time required for sorting a 10 GB uniform random data set, using snake sort, row sort, and adaptive sort utilizing both pass and store and flush output protocols. The throughput is held relatively steady as the number of processors and corresponding list size grows. Pass and store requires less time than flush for snake and row sorts but slightly increased time for adaptive sort. In the latter case, often only one sorting row will be full at the end of a sorting a temporary lists; other rows will have room to move records downstream and free upstream processors for a new temporary list, mitigating the benefit of pass and store. This benefit is then overwhelmed by the increased operations per record for the pass and store protocol, requiring more time for records to exit the upstream processors and delaying a new temporary list.

The row sort is the most energy-efficient and has the highest throughput, when compared to the other sorts. The row sort with flush scaling results shows the most variation in throughput, this is because the row sort is most susceptible to large changes in sort time with random data. Depending on the distribution of the random records, one row could potentially take a long time to flush all of its records before it can start sorting the next temporary list. The pass and store output protocol was able to smooth out the throughput of the row sort because it removed the penalty of the random records clogging a row, and slowing down the sort.

6.3. Complete external sort results

An entire external database sort is modeled to compile data on the time and energy required to compute a 10-GB external

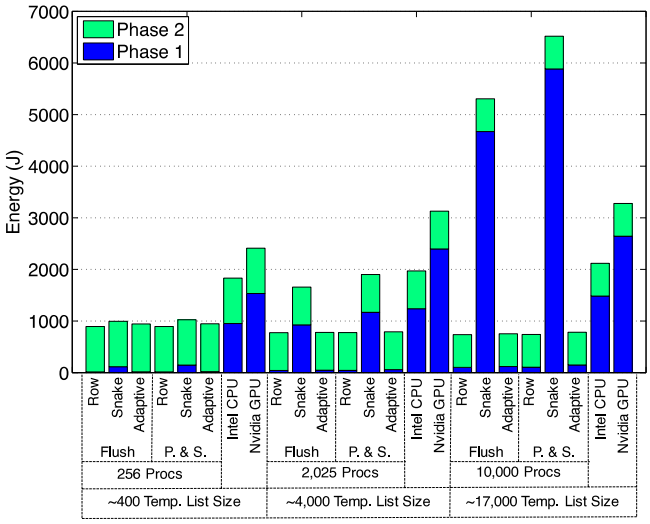


Fig. 10. Total energy required by processors on a complete 10-GB sort.

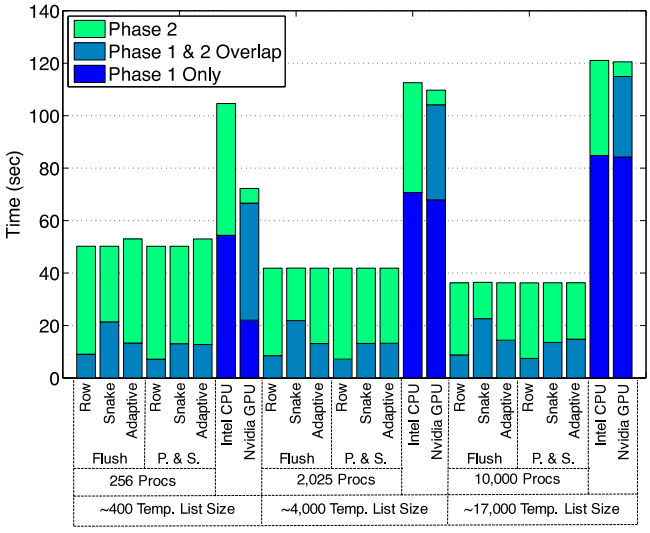


Fig. 11. Time required to sort a complete 10-GB data set. With two chips, there is some amount of time, marked Phase 1 & 2 in the legend, where both phase 1 and phase 2 are being computed at the same time on different chips.

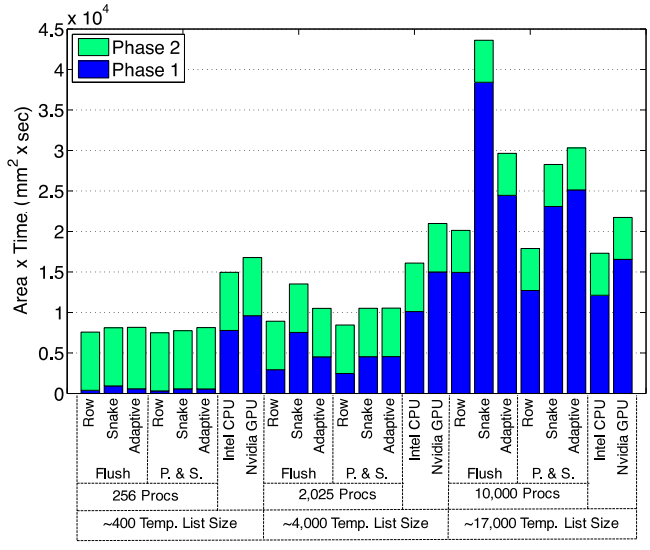


Fig. 12. Area×time required by processors of a complete 10-GB sort.

sort. The phase 2 merge as well as other administrative tasks are performed on an Intel CPU, and phase 1 computations are performed on the given computational platform. The results can be found in Table 1. If multiple processing platforms were used, overlap can occur where parts of Phase 1 & 2 process concurrently if required data is ready. Since this particular many-core architecture has the ability to completely halt each individual idle processor's clock, and because leakage power is very low, the relative core utilization can be closely approximated by dividing the average application power by the peak fully-active core power of 31.84 mW [34].

The results of the full external sort models show the time to complete phase 1, the total time to complete the sort, the energy required by the processors to complete the sort, the total time multiplied by the area of the processing unit to highlight the tradeoff of smaller sized computational platforms, and the energy×delay, also known as energy-delay product (EDP), to highlight energy usage and execution time together.

Shown in Figs. 10–12, the phase 1 is a significant portion of the cost of an external sort performed on the Intel CPU and the Nvidia GPU. Fig. 10 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 65× less energy than the Intel CPU GNU C++ standard sort library, and over 105× less energy than the Nvidia GPU radix sort. The phase one efficiency is overshadowed by the phase 2 energy, making the total energy cost for the most efficient many-core sort require more than 2.8× less energy than the comparable Intel T7700 sort and 4.4× less energy than the comparable GPU sort.

Shown in Fig. 11, the many-core sorts are able to sort their 10 GB of data in less time than any of the comparison sorts. Table 1 shows the fastest row sort takes over 9.8× less time to perform phase 1 compared to the comparable GNU C++ standard sort library on an Intel T7700 and over 14× less than the comparable radix sort on an Nvidia GPU. The largest difference in total time is found between the row sort with 10,000 processors showing over 3.3× smaller time that the comparable Intel T7700 sort, and over 3.3× smaller than the GPU sort. 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 area×time metric was used, where the total time is multiplied by the total chip area of the processing unit. Fig. 12 shows 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 Intel i7-5557U or GPU, except for the 10,000-processor many-core chip. The increased cost of area×time in phase 1 by increasing the number of processors to 10,000 is the penalty to decrease the phase 2 sorting time, and decrease total energy, due to the longer sorted lists from phase 1. With a 256-processor many-core array, the row sort has over 2.0× less area×time than the Intel CPU GNU C++ standard sort library, and over 2.2× less area×time than the GPU radix sort.

The energy×delay for the entire 10 GB sort is shown in Table 1, where the lowest energy×delay many-core sort was shown to require over 6.2× less energy×delay than the Intel CPU GNU C++ standard sort library, and over 13× less energy×delay than the GPU radix sort.

As previously discussed in Section 6.2, the store and pass output protocol increases the throughput of all of the sorts. Fig. 10 shows that with the phase 1 energy cost already at a small

Table 2
Time, energy, area, and energy delay product for platforms performing a 10-GB external sort of 100-B records. Phase 2 for each of the presented sorts is accomplished with a merge sort on an Intel 5557U.

Records Per Temp. List	Phase 1 Processor (Technology Node)	Phase 1 Sort	Output Protocol	Phase 1 Time (s)	Phase 1 & 2 Time (s)	Phase 1 Energy (J)	Phase 1 & 2 Energy (J)	Phase 1 & 2 Area×Time (mm^2 s 10^3)	Phase 1 & 2 E×D (J s 10^4)
468	**256 Many-Core Processors (65 nm)**	Row	Flush	9.054	28.26	14.5	410	4.15	1.13
			P & S	7.190	28.26	14.8	410	4.07	1.13
3 919	**2,025 Many-Core Processors (65 nm)**		Flush	8.516	23.55	42.9	373	6.06	0.813
			P & S	7.163	23.55	44.3	374	5.60	0.808
19 704	**10,000 Many-Core Processors (65 nm)**		Flush	8.796	20.43	102	388	17.7	0.673
			P & S	7.478	20.42	105	391	15.4	0.662
468	Intel	std::sort		11.97	40.23	167.4	662.1	5.75	1.32
3 919	Core2 5557U			16.26	39.81	227.6	639.7	5.69	1.15
19 704	(14 nm)			19.83	40.24	277.6	634.8	5.75	1.13
468	Nvidia	thrust::sort		81.94	85.08	2048	2444	13.4	17.9
3 919	750M GT			19.40	24.11	488	817	5.53	1.73
19 704	(28 nm)			19.40	22.54	485	771	5.00	1.52

amount, the small increase in energy cost from the store and pass protocol has little to no effect on the total energy usage.

The last entry in Table 1 displays the results from the 2007 JouleSort winner, CoolSort [23], in the Indy 10^8 record category, which was performed on a 65 nm Intel Core2 T7600. Because the JouleSort benchmark looks at system power, peripheral power and high I/O bandwidth is a large consideration, so fast processing was more important than low processing energy.

6.4. Contemporary processing comparison

Comparison sorts are evaluated in Section 6.3 for the most direct comparison with chips in the same technology node. However, for perspective, Table 2 shows modeled results of 10 GB external sorts on modern processing elements, compared to the proposed sorts using an Intel i7-5557U for the second phase. A comparison sort() from the thrust library was used on the Nvidia GeForce GT 750M, which was fabricated in 28 nm, operates at 967 MHz, and a TDP of 50 W with a die size of 118 mm^2. The GNU C++ standard sort library was used on an Intel i7-5557U, which was fabricated in 14 nm, operates at a maximum of 3.1 GHz, has a TDP of 28 W with a die size of 133 mm^2. The GPU is most efficient sorting data sets which saturate its array, as shown by the slower performance with a small list size. The proposed many-core sorts are still shown to be the best performing, but with smaller margins than when compared to other 65 nm chips.

7. Conclusion

We have presented three different sorting applications, with two different protocols for outputting data from each processor. This was accomplished on a varied number of processors in a many-core array acting as a co-processor to an Intel T7700 performing the phase 2 merge part of the external sort.

Many-core array sorts were modeled with the number of cores ranging from 10 to 10,000. We found the most energy efficient phase 1 sort required over 65× less energy than the comparable Intel laptop-class processor sort and over 105× less energy than the comparable Nvidia GPU. The highest throughput many-core phase 1 sort was over 9.8× faster than the comparable Intel laptop-class processor sort and over 14× faster than the comparable GPU sort. It was also shown that this same sort required over 5.5× less energy×delay than the JouleSort winner, CoolSort. The proposed sorts can be implemented on different sized arrays, and future many-core processing arrays in a large database system and recognize large energy savings without giving up performance. This study provides motivation for exploration of a many-core co-processor as an energy and throughput efficient alternative computation platform for datacenters.

Declaration of competing interest

No author associated with this paper has disclosed any potential or pertinent conflicts which may be perceived to have impending conflict with this work. For full disclosure statements refer to https://doi.org/10.1016/j.jpdc.2019.12.011.

Acknowledgments

This work was supported by ST Microelectronics, C2S2, Switzerland Grant 2047.002.014, National Science Foundation, USA Grant 0430090 and CAREER Award, USA 0546907, SRC GRC, USA Grants 1598, 1971, and 2321 and CSR, USA Grant 1659, Intel, UC Micro, Intellasys, and SEM.

References

[1] B. Baas, Z. Yu, M. Meeuwsen, O. Sattari, R. Apperson, E. Work, J. Webb, M. Lai, T. Mohsenin, D. Truong, J. Cheung, AsAP: a fine-grained many-core platform for DSP applications, IEEE Micro 27 (2) (2007) 34–45, http://dx.doi.org/10.1109/MM.2007.29.

[2] S.W.A.-H. Baddar, B.A. Mahafzah, Bitonic sort on a chained-cubic tree interconnection network, J. Parallel Distrib. Comput. 74 (1) (2014) 1744–1761, http://dx.doi.org/10.1016/j.jpdc.2013.09.008, http://www.sciencedirect.com/science/article/pii/S0743731513002049.

[3] K.E. Batcher, Sorting networks and their applications, in: ACM AFIPS (Spring), New York, NY, USA, 1968, pp. 307–314, http://dx.doi.org/10.1145/1468075.1468121.

[4] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C.-C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, J. Zook, TILE64 - processor: a 64-core SoC with mesh interconnect, in: 2008 IEEE International Solid-State Circuits Conference - Digest of Technical Papers, 2008, pp. 88–89, http://dx.doi.org/10.1109/ISSCC.2008.4523070.

[5] B. Bohnenstiehl, A. Stillmaker, J.J. Pimentel, T. Andreas, B. Liu, A.T. Tran, E. Adeagbo, B.M. Baas, Kilocore: a 32-nm 1000-processor computational array, IEEE J. Solid-State Circuits 52 (4) (2017) 891–902, http://dx.doi.org/10.1109/JSSC.2016.2638459.

[6] S. Borkar, Thousand core chips: a technology perspective, in: ACM Design Automation Conf., New York, NY, USA, 2007, pp. 746–749, http://dx.doi.org/10.1145/1278480.1278667.

[7] M. Butler, AMD Bulldozer Core a new approach to multithreaded compute performance for maximum efficiency and throughput, in: IEEE HotChips Symp. on High-Performance Chips, 2010.

[8] W.H. Cheng, B.M. Baas, Dynamic voltage and frequency scaling circuits with two supply voltages, in: IEEE Intl. Symp. on Circuits and Systems, 2008, pp. 1236–1239, http://dx.doi.org/10.1109/ISCAS.2008.4541648.

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

[10] A. Gandhi, M. Harchol-Balter, R. Das, C. Lefurgy, Optimal power allocation in server farms, in: SIGMETRICS '09, New York, NY, USA, 2009, pp. 157–168, http://dx.doi.org/10.1145/1555349.1555368.

[11] B. Gedik, R.R. Bordawekar, P.S. Yu, CellSort: high performance sorting on the cell processor, in: Int. Conf. on Very Large Data Bases, 2007, pp. 1286–1297.

[12] N. Govindaraju, J. Gray, R. Kumar, D. Manocha, GPUTeraSort: high performance graphics co-processor sorting for large database management, in: SIGMOD, New York, NY, USA, 2006, pp. 325–336, http://dx.doi.org/10.1145/1142473.1142511.

[13] G. Graefe, Query evaluation techniques for large databases, ACM Comput. Surv. 25 (1993) 73–169, http://dx.doi.org/10.1145/152610.152611.

[14] H.V. Jagadish, Sorting on an array of processors, in: Computer Design: VLSI in Computers and Processors, IEEE Intl. Conf. on, 1988, pp. 36–39, http://dx.doi.org/10.1109/ICCD.1988.25654.

[15] C. Kim, J. Park, N. Satish, H. Lee, P. Dubey, J. Chhugani, CloudRAMSort: fast and efficient large-scale distributed RAM sort on shared-nothing cluster, in: ACM SIGMOD, New York, NY, USA, 2012, pp. 841–850, http://dx.doi.org/10.1145/2213836.2213965.

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

[17] B. Liu, B.M. Baas, Parallel AES encryption engines for many-core processor arrays, IEEE Trans. Comput. 62 (3) (2013) 536–547, http://dx.doi.org/10.1109/TC.2011.251.

[18] C. Nyberg, Sort benchmark data generator and output validator, 2011, http://ordinal.com/gensort.html.

[19] C. Nyberg, M. Shah, N. Govindaraju, Sort benchmark home page, 2016, http://sortbenchmark.org/.

[20] P. Pillai, M. Kaminsky, M.A. Kozuch, D.G. Andersen, FAWNSort: energy-efficient sorting of 10GB, 100GB, and 1TB, Tech. rep., Intel Labs and Carnegie Mellon University, 2012.

[21] D.M.W. Powers, Parallelized quicksort and radixsort with optimal speedup, in: Intl. Conf. on Parallel Computing Technologies, Novosibirsk, 1991, pp. 167–176.

[22] S. Rajasekaran, Mesh connected computers with fixed and reconfigurable buses: packet routing and sorting, IEEE Trans. Comput. 45 (1996) 529–539, http://dx.doi.org/10.1109/12.509905.

[23] S. Rivoire, M.A. Shah, P. Ranganathan, C. Kozyrakis, Joulesort: a balanced energy-efficiency benchmark., in: C.Y. Chan, B.C. Ooi, A. Zhou (Eds.), SIGMOD 2007, 2007, pp. 365–376, http://dx.doi.org/10.1145/1247480.1247522.

[24] N. Satish, M. Harris, M. Garland, Designing efficient sorting algorithms for manycore GPUs, in: IEEE Intl. Symp. on Parallel & Distributed Processing, Washington, DC, USA, 2009, pp. 1–10, http://dx.doi.org/10.1109/IPDPS.2009.5161005.

[25] R.K. Sharma, R. Shih, C. Bash, C. Patel, P. Varghese, M. Mekanapurath, S. Velayudhan, M. Kumar, On building next generation data centers: energy flow in the information technology stack, in: COMPUTE '08, New York, NY, USA, 2008, pp. 8:1–8:7, http://dx.doi.org/10.1145/1341771.1341780.

[26] A. Shehabi, S. Smith, D. Sartor, R. Brown, M. Herrlin, J. Koomey, E. Masanet, N. Horner, I. Azevedo, W. Lintner, United states data center energy usage report, Tech. Rep. LBNL-1005775, Lawrence Berkeley National Laboratory, Berkeley, CA, 2016.

[27] E. Solomonik, L.V. Kale, Highly scalable parallel sorting, in: Parallel Distributed Processing, IEEE Intl. Symp. on, 2010, pp. 1–12, http://dx.doi.org/10.1109/IPDPS.2010.5470406.

[28] L. Stillmaker, Saisort: an energy efficient sorting algorithm for many-core systems (Master's thesis), University of California, Davis, CA, USA, 2011.

[29] A. Stillmaker, B. Baas, Scaling equations for the accurate prediction of CMOS device performance from 180 nm to 7 nm, Integr. VLSI J. 58 (2017) 74–81, http://dx.doi.org/10.1016/j.vlsi.2017.02.002.

[30] A. Stillmaker, L. Stillmaker, B. Baas, Fine-grained energy-efficient sorting on a many-core processor array, in: IEEE Intl. Conf. on Parallel and Distributed Systems, 2012, pp. 652–659, http://dx.doi.org/10.1109/ICPADS.2012.93.

[31] D. Taniar, J.W. Rahayu, Sorting in parallel database systems, in: IEEE Conf. on High-Performance Computing in the Asia-Pacific Region, vol. 2, Washington, DC, USA, 2000, pp. 830–835, http://dx.doi.org/10.1109/HPC.2000.843555.

[32] K. Thearling, S. Smith, An improved supercomputer sorting benchmark, in: Proceedings of the 1992 ACM/IEEE Conference on Supercomputing, Los Alamitos, CA, USA, 1992, pp. 14–19, http://dx.doi.org/10.1109/SUPERC.1992.236714.

[33] A.T. Tran, D.N. Truong, B.M. Baas, A reconfigurable source-synchronous on-chip network for GALS many-core platforms, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 29 (6) (2010) 897–910, http://dx.doi.org/10.1109/TCAD.2010.2048594.

[34] D.N. Truong, W.H. Cheng, T. Mohsenin, Z. Yu, A.T. Jacobson, G. Landge, M.J. Meeuwsen, C. Watnik, P.V. Mejia, A.T. Tran, J.W. Webb, E.W. Work, Z. Xiao, B.M. Baas, A 167-processor 65 nm computational platform with per-processor dynamic supply voltage and dynamic clock frequency scaling, in: VLSI Circuits, 2008 IEEE Symp. on, 2008, http://dx.doi.org/10.1109/VLSIC.2008.4585936.

[35] D.N. Truong, W.H. Cheng, T. Mohsenin, Z. Yu, A.T. Jacobson, G. Landge, M.J. Meeuwsen, C. Watnik, A.T. Tran, Z. Xiao, E.W. Work, J.W. Webb, P.V. Mejia, B.M. Baas, A 167-processor computational platform in 65 nm CMOS, IEEE J. Solid-State Circuits 44 (4) (2009) 1130–1144, http://dx.doi.org/10.1109/JSSC.2009.2013772.

[36] S.R. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, V. Erraguntla, C. Roberts, Y. Hoskote, N. Borkar, S. Borkar, An 80-tile sub-100-W TeraFLOPS processor in 65-nm CMOS, IEEE J. Solid-State Circuits 43 (1) (2008) 29–41, http://dx.doi.org/10.1109/JSSC.2007.910957.

[37] J.S. Vitter, External memory algorithms and data structures: dealing with massive data, ACM Comput. Surv. 33 (2) (2001) 209–271, http://dx.doi.org/10.1145/384192.384193.

[38] Z. Xiao, S. Le, B. Baas, A fine-grained parallel implementation of a H.264/AVC encoder on a 167-processor computational platform, in: IEEE Asilomar Conference on Signals, Systems and Computers, ACSSC, 2011, http://dx.doi.org/10.1109/ACSSC.2011.6190391.

[39] Z. Yu, B.M. Baas, High performance, energy efficiency, and scalability with GALS chip multiprocessors, IEEE Trans. Very Large Scale Integr. Syst. 17 (1) (2009) 66–79, http://dx.doi.org/10.1109/TVLSI.2008.2001947.

[40] Z. Yu, B.M. Baas, A low-area multi-link interconnect architecture for GALS chip multiprocessors, IEEE Trans. Very Large Scale Integr. (VLSI) Syst. 18 (5) (2010) 750–762, http://dx.doi.org/10.1109/TVLSI.2009.2017912.

[41] Z. Yu, M.J. Meeuwsen, R. Apperson, O. Sattari, M. Lai, J. Webb, E. Work, T. Mohsenin, M. Singh, B.M. Baas, An asynchronous array of simple processors for DSP applications, in: IEEE Intl. Solid-State Circuits Conf., 2006, pp. 428–429, http://dx.doi.org/10.1109/ISSCC.2006.1696225.

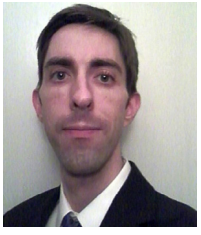
[42] Z. Yu, M. Meeuwsen, R. Apperson, O. Sattari, M. Lai, J. Webb, E. Work, D. Truong, T. Mohsenin, B. Baas, AsAP: an asynchronous array of simple processors, IEEE J. Solid-State Circuits 43 (3) (2008) 695–705, http://dx.doi.org/10.1109/JSSC.2007.916616.

[43] Y. Zhang, S. Zheng, Design and analysis of a systolic sorting architecture, in: Parallel and Distributed Processing, IEEE Symp. on, 1995, pp. 652–659, http://dx.doi.org/10.1109/SPDP.1995.530744.

Aaron Stillmaker received the B.S. degree in computer engineering from the California State University, Fresno, CA, USA, in 2008, and the M.S. and Ph.D. degrees in electrical and computer engineering from the University of California, Davis, CA, USA, in 2013 and 2015, respectively.

In 2013, he was an intern with the Circuit Research Lab, Intel Labs, Hillsboro, OR, USA. From 2008 to 2015, he was a Graduate Student Researcher with the VLSI Computation Laboratory, UC Davis.

Since 2017, he has been an Assistant Professor with the Electrical and Computer Engineering Department, California State University, Fresno, CA, USA. His current research interests include manycore processor architecture, many-core applications, and VLSI design.



Brent Bohnenstiehl received the B.S. degree in electrical engineering from the University of California, Davis, CA, USA, in 2006, where he is currently pursuing the Ph.D. degree in electrical and computer engineering.

He has been a Graduate Student Researcher with the VLSI Computation Laboratory, Davis, since 2011. His current research interests include processor architecture, VLSI design, hardware–software codesign, dynamic voltage and frequency scaling algorithms and circuits, and many-core compilers and other programming and simulation tools.

Lucas Stillmaker received the B.S. degree in computer engineering from the California State University, Fresno, CA, USA in 2008, the M.S. degree in electrical and computer engineering from the University of California, Davis, CA, USA in 2011, and the M.B.A degree in business administration from the California State University, Sacramento, CA, USA in 2016.

From 2008 to 2011 he was a Graduate Student Researcher with the VLSI Computation Laboratory. Since 2011 he has been a Graphics Hardware Engineer for Intel in Folsom, CA, USA.

Bevan M. Baas received the B.S. degree in electronic engineering from California Polytechnic State University, San Luis Obispo, CA, USA, in 1987, and the M.S. and Ph.D. degrees in electrical engineering from Stanford University, Stanford, CA, USA, in 1990 and 1999, respectively.

From 1987 to 1989, he was with Hewlett-Packard, Cupertino, CA, USA. In 1999, he joined Atheros Communications, Santa Clara, CA, USA, as an early employee and a core member of the team which developed the first commercial IEEE 802.11a Wi-Fi wireless LAN solution. In 2003, he joined the Department of Electrical and Computer Engineering, University of California, Davis, CA, USA, where he is currently a Professor. He leads projects in architectures, hardware, applications, and software tools for VLSI computation.