# Lower Bounds in the Asymmetric External Memory Model

Riko Jacob
IT University of Copenhagen
Rued Langgaards Vej 7
DK-2300 København S, Denmark
rikj@itu.dk

Nodari Sitchinava[*]
University of Hawaii at Mānoa
Information and Computer Sciences
1680 East West Road, POST 317
Honolulu, HI 96822, USA
nodari@hawaii.edu

## ABSTRACT

Motivated by the asymmetric read and write costs of emerging non-volatile memory technologies, we study lower bounds for the problems of sorting, permuting and multiplying a sparse matrix by a dense vector in the asymmetric external memory model (AEM). Given an AEM with internal (symmetric) memory of size $M$, transfers between symmetric and asymmetric memory in blocks of size $B$ and the ratio $\omega$ between write and read costs, we show $\Omega(\min\{N, \frac{\omega N}{B} \log_{\frac{\omega M}{B}} \frac{N}{B}\})$ lower bound for the cost of permuting $N$ input elements. This lower bound also applies to the problem of sorting $N$ elements. This proves that the existing sorting algorithms in the AEM model are optimal to within a constant factor for reasonable ranges of parameters $N$, $M$, $B$, and $\omega$. We also show a lower bound of $\Omega\left(\min\left\{H, \frac{\omega H}{B} \log_{\frac{\omega M}{B}} \frac{N}{\max\{\delta, M\}}\right\}\right)$ for the cost of multiplying an $N \times N$ matrix with at most $H = \delta N$ non-empty entries by a vector with $N$ elements.

## CCS CONCEPTS

**•Theory of computation → Sorting and searching; •Hardware** → Memory and dense storage;

## KEYWORDS

Asymmetric external memory; AEM; asymmetric read-write; NVMs; sorting; permuting; lower bounds; sparse matrix vector multiplication; SpMxV

## 1 INTRODUCTION

Recent advances in Phase-Change Memory, Memristor-based Resistive RAM, and Spin-Torque Transfer Magnetic RAM technologies place non-volatile memory (NVM) technology on the path to become dominant memory technology in the near future. Relative to current DRAM, NVM provides better energy usage and higher density. However, the most significant difference between NVM and DRAM is the asymmetry in the cost of reading and writing data, sometimes by orders of magnitude [8, 9, 12, 15]. Such drastic difference in read and write costs motivated recent studies on the effects of read-write asymmetry on algorithm design.

Most of the studies in the past focused largely on the effects of read-write asymmetry in the NAND flash memories [2, 3, 10, 13, 14, 17]. But recently, several papers focused on models designed specifically for the NVM [4, 6, 7].

Blelloch et al. [6] define $(M, \omega)$*-Asymmetric RAM (ARAM)* model to analyze algorithms in the setting with asymmetric read and write costs. The model consists of a small symmetric memory of size $M$ and unbounded asymmetric memory, which contains the input. To perform any computation on the data, it must be first brought into the symmetric memory. Write accesses to the asymmetric memory cost $\omega$ times more than read accesses. If an algorithm performs $Q_r$ read accesses and $Q_w$ write accesses to the asymmetric memory, the *cost* $Q$ of an algorithm is defined as $Q = Q_r + \omega Q_w$. The model also defines *time* $T$ of an algorithm to be equal to the total number of read and write accesses to the symmetric memory, which represents the time it takes to perform computations to within a constant factor, similar to how time complexity is defined in the standard (symmetric) RAM model.

A related (symmetric) *external memory (EM)* model was introduce by Aggarwal and Vitter [1]. The EM model consists of two-level memory hierarchy, with the faster *internal memory* of size $M$ and the slower *external memory* of unbounded size, but symmetric unit cost, *I/O cost*, of accessing the external memory. To perform computation on data, it has to be first brought into the internal memory, and the transfer is performed in contiguous blocks of size $B \geq 1$ elements. The cost $Q$ of an algorithm in the EM model is defined to be the total number of read and write accesses to the external memory. It is assumed that the cost of accessing the external memory is significantly larger than accessing internal memory, therefore, the EM model focuses only on the I/O cost and does not count the number of accesses to the internal memory.

The natural generalization of the EM model to the asymmetric setting [7] defines the $(M, B, \omega)$*-Asymmetric External Memory (AEM)* model as an EM model with each write access to the external memory costing $\omega$ times more than a read access. The cost $Q$ of an algorithm, which performs $Q_r$ read and $Q_w$ write accesses (I/Os) to the external memory of the $(M, B, \omega)$-AEM model, is defined as $Q = Q_r + \omega Q_w$. Similarly to the (symmetric) EM model, this definition of the AEM model does not consider accesses (and, equivalently, computation) in the internal memory to be part of the cost. Also, notice that the $(M, \omega)$-ARAM model is equivalent to the $(M, 1, \omega)$-AEM model.

[*]This material is based upon work supported by the National Science Foundation under Grant No. 1533823.

*SPAA '17, Washington DC, USA*
© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM. 978-1-4503-4593-4/17/07…$15.00
DOI: 10.1145/3087556.3087583

Blelloch et al. study a number of algorithms in the $(M, B, \omega)$-AEM [7] and $(M, \omega)$-ARAM [6] models. In addition they prove lower bounds to a number of problems in the $(M, \omega)$-ARAM model [6].

## 1.1 Our contributions

In this paper we focus on the problems of sorting, permuting and multiplying a sparse matrix by a dense vector in the $(M, B, \omega)$-AEM model.

Of the three previously published [7] sorting algorithms in the $(M, B, \omega)$-AEM model, sample sort and heapsort achieve the cost $O(\omega n \log_{\omega m} n)$ unconditionally. However, to achieve the same bound, the mergesort relies on the assumption that $\omega < B$. In Section 3, we show that we can implement mergesort in the $(M, B, \omega)$-AEM model with the same cost without relying on any assumptions.

In Section 4 we show an $\Omega(\min\{N, \omega n \log_{\omega m} n\})$ lower bound for the problem of permuting $N$ elements. Since every sorting algorithm must be able to perform an arbitrary permutation, this lower bound also applies to the problem of sorting $N$ elements and matches the sorting upper bound to within a constant factor for reasonable ranges of the parameters $\omega$, $B$, $M$ and $N$. In addition to proving the permutation lower bound via the standard counting method, we also prove it via reduction to the unit-cost flash memory model of Ajwani et al. [2]. This result shows a close relationship between the $(M, B, \omega)$-AEM and the unit-cost flash model, which might make this reduction of independent interest.

Finally, in Section 5 we present a lower bound on the cost of multiplying a sparse matrix with a dense vector in the $(M, B, \omega)$-AEM model.

## 2 PRELIMINARIES

To simplify notation throughout this paper, we define the following variables, which are standard in the literature on the EM model:

- $N$ : size of the input vector/array or a dimension of a matrix
- $H = \delta N$ : number of non-zero entries in a sparse matrix
- $M$ : internal (symmetric) memory size
- $B$ : block size
- $m = \lceil M/B \rceil$
- $n = \lceil N/B \rceil$
- $h = \lceil H/B \rceil = \Theta(\delta n)$
- $\omega$ : ratio in cost between a write and a read access to the external (asymmetric) memory

Similar to [5], to prove our lower bounds, we distinguish between an *algorithm* and a *program* in the following way. An algorithm is one description of how the $(M, B, \omega)$-AEM model handles an arbitrary input. In particular the size of the input, the permutation, or the structure of the sparse matrix is not fixed, and there are loops and branches. In contrast, a program is a fixed sequence of I/O operations and internal memory move operations (and additions/multiplication in the semi-ring). Accordingly, a program implements one particular permutation or one particular conformation (structure of the non-zero entries, together with a layout) of a sparse matrix. Each such program is a straight line program that performs the same set of operations no matter what values are given as input. We can view the inputs and outputs to such programs as an ordered list of numbers and their meaning is given solely by their position in the ordering, rather than their values.

Clearly, an algorithm gives rise to a family of programs – one for each permutation or sparse matrix conformation. Hence, a lower bound on the cost of any program for some permutation or sparse matrix of a certain size induces the corresponding lower bound on the cost of any algorithm. For all considered problems we give upper bounds on the cost by means of describing an algorithm and lower bounds by arguments about programs.

## 3 MERGE SORT

In this section we present a multi-way mergesort which achieves $O(\omega n \log_{\omega m} n)$ read and $O(n \log_{\omega m} n)$ write I/Os for any value of $\omega$.

Our AEM mergesort follows the standard framework for multi-way mergesort algorithms: We divide the array into $d = \omega m$ subarrays, each of size $O(N/d)$, recursively sort each one and merge $d$ newly sorted subarrays into a single sorted array. At the base case, using the algorithm for sorting small arrays by Blelloch et al [7, Lemma 4.2], we can sort each subarray of size $N' \leq \omega M$ elements in $O(\omega n')$ read I/Os and $O(n')$ write I/Os (for a total cost of $O(\omega n')$ each), where $n' = N'/B$. If the cost of performing $d$-way merging of the subarrays is $O(\omega n)$, then the cost of the overall algorithm is defined by the following recurrence:

$$Q(N, M, B, \omega) = \begin{cases} d \cdot Q(N/d, M, B, \omega) + O(\omega n) & \text{if } N > \omega M \\ O(\omega n) & \text{if } N \leq \omega M \end{cases}$$

which solves to $Q(N, M, B, \omega) = O(\omega n \log_d n) = O(\omega n \log_{\omega m} n)$.

Thus, it remains to show how to perform $\omega m$-way merging of sorted subarrays, which collectively contain $N$ elements, within the desired $O(\omega n)$ cost.

## 3.1 Merging $\omega m$ sorted arrays

For ease of exposition, let $M$ be a constant fraction of the available internal memory. This does not affect the asymptotic bounds above and provides us with sufficient space to store a constant number of additional words of auxiliary data with each element in the internal memory.

To merge $\omega m$ sorted arrays, we proceed in $R = \lceil N/M \rceil$ rounds. At the end of each round we write a batch of the next $M$ smallest elements across all $\omega m$ arrays in sorted order in contiguous addresses of external memory. Next, we show how to perform each round in $O(\omega m)$ read and $O(m)$ write I/Os, for a total of $O(N/M \cdot \omega m) = O(\omega n)$ read and $O(N/M \cdot m) = O(n)$ write I/Os.

Let $A_i$, $0 \leq i < \omega m$, be the $i$-th sorted array to be merged. If $\omega > B$, then we do not even have enough space in the internal memory to maintain the pointers to the next element of $A_i$ in external memory. Therefore, we must maintain these pointers in external memory. To reduce the number of times we have to perform a write I/O to update these pointers, rather than maintaining a pointer $ptr[i]$ to the next element $e$ of $A_i$ that is not in internal memory, we maintain the pointer $b[i]$ to the *block* of $A_i$ that contains $e$. This way, we have to update each $b[i]$ in external memory only after $B$ elements are merged from $A_i$, i.e., at most once for each of $n$ blocks. Therefore, the total write I/Os required to update the pointers $b[i]$ for merging $N$ elements of $\omega m$ arrays is at most $O(n)$. Let $\pi$ denote the largest element stored in internal memory at the time. The next element to be considered from an array $A_i$ is the smallest element that is larger than $\pi$. Note that it is always in the block pointed to by

$b[i]$. Thus, we can always determine the correct element within the $b[i]$-th block to be processed next. Initially, all $b[i]$s are set to point to the first block of each array $A_i$. This initialization takes $O(\lceil \omega m/B \rceil) = O(\omega m)$ write I/Os.

In each round we scan the current elements of the arrays $A_i$ (as detailed later) and maintain $M$ smallest elements in internal memory. Since internal computations are not counted in the cost of an AEM algorithm, we can, for example, maintain them in a sorted array $\mathcal{M}$. The round ends when the next unprocessed element of **every** $A_i$ is larger than the largest element of $\mathcal{M}$, at which point we write $\mathcal{M}$ to external memory using $O(m)$ write I/Os.

*Initializing $\mathcal{M}$:* Starting with an empty array $\mathcal{M}$, we read two blocks from each array $A_i$, $0 < i < \omega m$, starting with the $b[i]$-th block. Every time a block is read, its elements that are larger than $\pi$ are merged into $\mathcal{M}$. If $\mathcal{M}$ grows beyond the size $M$, it is truncated down to exactly $M$ elements.

*Identifying active arrays:* We call an array $A_i$ *active*, if more blocks from that array might need to be loaded, i.e., the largest element of the last block read from $A_i$ (a) is not the last element of $A_i$ and (b) is among the $M$ smallest in the internal memory so far. Otherwise, we call $A_i$ *inactive*. Observe that once $\mathcal{M}$ contains $M$ elements the values of the elements of $\mathcal{M}$ can only decrease during that round. Therefore, once an array becomes inactive, it does not need to be considered again until the next round. Observe, that it is easy to identify active arrays by re-reading the blocks from the initialization phase and comparing the last element read from each array with the largest element in $\mathcal{M}$. For every inactive array we can also update $b[i]$, if it has changed.

The efficiency of our merging algorithm relies on the following observation.

Lemma 3.1. *After $\mathcal{M}$ has been initialized, there are at most $m = M/B$ active arrays for the rest of the round of the algorithm.*

Proof. Since more than $B$ elements are read from each array, and $\mathcal{M}$ contains at most $M$ elements, there can be at most $M/B$ blocks from each array with the last element in $\mathcal{M}$, i.e., satisfying the condition (b) in the definition of active arrays. □

*Merging from active arrays:* According to Lemma 3.1, there is enough space in internal memory to maintain one block from each active array. Observe that the initialization step ensures that one block from each active array is already present in the internal memory. Let $\mu$ be the largest value that the algorithm should write out during the round. If $\mu$ were known to the algorithm, it could simply read from each $A_i$ all the elements smaller than $\mu$ and merge them. As it is not known, the algorithm uses a classical $M/B$-way merge on the active arrays to load the right elements.

More precisely, for any active $A_i$ let $s_i$ be the maximal element already loaded into the internal memory. Recall that if $s_i > \pi$, the array $A_i$ is no longer active. Let $j$ be be the index of the array with the smallest element among all $s_i$. In each step the algorithm loads the next block from $A_j$, merges it into $\mathcal{M}$, updates $s_j$ and repeats until there are no more active arrays, which means $\mu = \pi$. At this point, $\mathcal{M}$ contains $M$ smallest elements, which are written out into external memory in sorted order.

Theorem 3.2. *Merging $\omega m$ sorted arrays, containing in total $N$ elements, takes $O(\omega(n + m))$ read and $O(n + m)$ write I/Os.*

Proof. As has been argued earlier, since the $b[i]$ pointers are updated in external memory only once per block, the cost to maintain $b[i]$s is at most $O(n)$ write I/Os. Additional writes are only to output $M$ next smallest elements in each of $R = \lceil N/M \rceil$ rounds for a total of $R \cdot m = O\left(\left(\frac{N}{M} + 1\right) \cdot m\right) = O(n + m)$ write I/Os.

To compute the number of reads, observe that the use of $\pi$ by the algorithm implies that if in some round, array $A_i$ contains an element that is greater than $\pi$, then at most one block from $A_i$ (the one containing the smallest element larger than $\pi$) is read from external memory. Thus, the number of reads in each round is at most $\sum_{i=1}^{\omega m}\left(\frac{N_i}{B} + 1\right)$, where $N_i$ is the number of elements from $A_i$ that should be written out in that round. Since in each round $\sum_i N_i = M$ (only $M$ elements are to be written out per round), $\sum_i^{\omega m}(N_i/B+1) \leq m + \omega m$, and over $R = \lceil N/M \rceil$ rounds the number of reads adds up to $R \cdot (m + \omega m) = O\left(\left(1 + \frac{N}{M}\right) \cdot (m + \omega m)\right) = O(\omega(n+m))$ I/Os. □

## 4 PERMUTATION LOWER BOUNDS

As explained in the introduction, we formulate this lower bounds for programs and this implies the corresponding lower bound for algorithms. Our approach to proving permutation lower bound in the AEM model follows the framework of Hong and Kung [11]: We defined an $\omega m$-round to be a sequence of operations of an $(M, B, \omega)$-AEM program of cost at most $\omega m$. Thus, a round may consist of any combination of $r$ read and $w$ write I/Os, as long as $r + \omega w \leq \omega m$. Additionally, all but the last round must have the cost of at least $\omega(m - 1)$.

We say a program is *round-based*, if it performs computation in $\omega m$-rounds and at the beginning and the end of each round the internal memory is empty. We upper bound the progress that any round-based program can make in any particular round, which gives us a lower bound on the number of rounds that any program must perform for the whole computation, resulting in the lower bound for the whole program (and hence algorithm). We also show that every program $\mathcal{P}$ in the $(M, B, \omega)$-AEM model can be converted into a round-based program $\mathcal{P}'$, without increasing the cost by more than a constant factor (Lemma 4.1). Thus, a lower bound for $\mathcal{P}'$ implies a lower bound for $\mathcal{P}$ (Corollary 4.2).

Lemma 4.1. *Any program $\mathcal{P}$ in the $(M, B, \omega)$-AEM with the overall cost of $Q(N, M, B, \omega)$ can be implemented as a round-based program $\mathcal{P}'$ in the $(2M, B, \omega)$-AEM with cost*

$$Q'(N, 2M, B, \omega) = O(Q(N, M, B, \omega))$$

Proof. Split $\mathcal{P}$ into rounds of cost at least $\omega(m - 1)$ and at most $\omega m$. To simulate each round of $\mathcal{P}$ on the $(2M, B, \omega)$-AEM, $\mathcal{P}'$ logically partition the internal memory of size $2M$ into two halves: $\mathcal{M}'$ and $\mathcal{M}''$. $\mathcal{M}'$ will maintain the contents of the internal memory of $\mathcal{P}$, while $\mathcal{M}''$ will buffer the data being written by $\mathcal{P}$ to the external memory in each round. At the start of each round, $\mathcal{P}'$ initializes $\mathcal{M}'$ by reading the contents of the internal memory of $\mathcal{P}$ from the end of the previous round (this step is skipped during the first round.) For every write operation of $\mathcal{P}$, $\mathcal{P}'$ copies the block into $\mathcal{M}''$, instead of writing it to the external memory. For every read operation of $\mathcal{P}$, $\mathcal{P}'$ loads the block from the external memory if and only if it is not present in $\mathcal{M}''$ already; otherwise $\mathcal{P}'$ copies the block from $\mathcal{M}''$ to $\mathcal{M}'$. When the cost limit of a round is reached, $\mathcal{P}'$ writes the contents of $\mathcal{M}''$ to the external memory and deletes the contents of both $\mathcal{M}'$ and $\mathcal{M}''$. This completes the simulation of a round.

Observe that $\mathcal{P}'$ fulfills the requirements of a round-based program. Since the cost of each round of $\mathcal{P}$ is at most $\omega m$, all elements to be written during the round of $\mathcal{P}$ fit in $\mathcal{M}''$. Excluding the reads required to initialize the contents of $\mathcal{M}'$ at the beginning of each round, the number of read and write accesses of $\mathcal{P}'$ is at most the same as those of $\mathcal{P}$. Additional reads required for initializing $\mathcal{M}'$ cost at most $\omega m$ in each round, starting with the second one. If $\mathcal{P}$ consists of at least two rounds, this additional cost in each round can be charged to the cost of the previous one, thus, increasing the overall cost only by a constant factor. If $\mathcal{P}$ consists of only one round, there is no additional cost, because the first round of $\mathcal{P}'$ does not perform the initialization. □

Corollary 4.2. *Any problem which requires cost $Q$ on $(M, B, \omega)$-AEM using a round-based program, requires cost $\Omega(Q)$ to be solved on the $(M/2, B, \omega)$-AEM.*

Similar to the (symmetric) EM model, to prove non-trivial lower bound for permutation, we must assume that each element is indivisible and new elements cannot be generated. To emphasize this point from now on we will refer to such indivisible elements as *atoms*.

With this, we are ready to prove the lower bound for the cost of permuting an array of $N$ elements, stored in $n = N/B$ consecutive blocks in the external memory. We take two approaches to prove the lower bound. In the first approach, we perform a simulation of the $(M, B, \omega)$-AEM permutation program in the unit-cost flash model [2]. Thus, the existing lower bounds in the unit-cost model will imply a lower bound in the $(M, B, \omega)$-AEM model. In the second approach, we prove the lower bound directly via the counting argument. The second approach provides a slightly stronger lower bound for some parameter ranges, due to some inefficiencies in the simulation. However, the simulation result might be of independent interest because it implies a close relationship between the $(M, B, \omega)$-AEM and the unit-cost flash model.

## 4.1 Lower bound via reduction to the unit-cost flash model

In the symmetric external memory model we could assume (by a simulation that has only a constant slowdown) that reading a block erases it on disc (and that there is no copying or deleting of elements). In the asymmetric model, this is not true because a simulation, which writes the contents of every block that is read back to the external memory, might increase the cost of the program by a factor of $\omega$. Hence the following arguments are based on a more refined trace of the program where we follow which copy of an atom is actually used in the output. This leads to the notion of a read operation 'using' some of the atoms of a block, meaning that the copies read are the ones eventually leading to the output.

In the following, we use a simulation of AEM in the unit-cost flash memory model of [2]. That model is an external memory model where the size of the blocks that are written is bigger than the size of the blocks that are read. This means that when one big block is written, it consists of several small blocks that can be read independently. Moreover, the cost of reading and writing is proportional to the number of elements in the block. Hence, similar to the AEM model, a single write operation is more expensive than a single read operation. Not too surprisingly, we choose a situation where the factor between the two is $\omega$, more precisely, the write blocks are of size $B$ as the AEM blocks, and the read blocks are of size $B/\omega$. For this to make sense, $B$ should be a multiple of $\omega$ (or somewhat bigger such that rounding is irrelevant). Still, the symmetry in the cost per element for reading and writing simplifies matters and we have (see [2]) upper and lower bounds for sorting and permuting as if all blocks were small. Interestingly, for the task of permuting, which is about moving around indivisible elements/atoms, there is a close connection between the two models. Observe that in a round-based program for permuting in the AEM model, only a $1/\omega$ fraction of the atoms read during a round can be written. Hence, the average number of *useful* atoms brought into the internal memory during a read operation is $B/\omega$, which is precisely the size of the read block of the associated flash model. The only challenge with simulating an AEM program in the flash model is that in a single read I/O the AEM program can choose an arbitrary subset of the $B/\omega$ atoms of the block, whereas in the flash model a read of $B/\omega$ elements must happen from the contiguous memory. It is easy to address this challenge if we know how a big block is going to be read in the future. However, this is well defined and easy to determine, because we are considering *programs* and not *algorithms*.

With this, we are ready to describe the simulation.

Lemma 4.3. *Assume there is a round-based program $\mathcal{P}_A$ for $(M, B, \omega)$-AEM that computes the permutation $\pi$ over $N$ elements with cost $Q$. Assume $B > \omega$ and $B$ is a multiple of $\omega$. Then there is a program $\mathcal{P}_F$ in the unit-cost flash memory model with read block of size $B/\omega$ and write block of size $B$ that performs I/Os of total volume of $2N + 2QB/\omega$. (i.e., an I/O volume corresponding to $Q$ small (read-) blocks).*

Proof. Observe that a read operation is an implicit copy operation: The atom is both in internal memory and in the block. Clearly, only one of the two copies will be moved further to become part of the output. If this is the copy that is in internal memory, we can think of the atom being deleted from the block. Because $\mathcal{P}_A$ is a program, at the time when the block is written, we can determine for all atoms the time when they will be removed from the block. We normalize $\mathcal{P}_A$ to write the block such that the atoms inside the block are ordered by the time they will be removed. We create $\mathcal{P}'_A$ from $\mathcal{P}_A$ by first doing one read and write scan over the input and then executing $\mathcal{P}_A$. This scan has I/O volume $2N$, can easily be round-based and has the effect that all read operation happen on normalized blocks, i.e., on blocks that are internally ordered by removal time. In particular, all read operations use only a contiguous interval of the atoms inside the block. To simulate $\mathcal{P}_A$ with $\mathcal{P}_F$ we keep the indices (names) of blocks. Every write operation of $\mathcal{P}_A$ is replicated directly in $\mathcal{P}_F$. A read operation of $\mathcal{P}_A$ leads to several read operations of small blocks, just enough to cover the interval of atoms that are actually removed from the block by this read operation. Observe that in this way every read operation of $\mathcal{P}_A$ induces at most 2 read operations in $\mathcal{P}_F$ that are not using (removing from the block) all of the read atoms. What remains is to determine the I/O volume of $\mathcal{P}_F$, which is simplified by $\mathcal{P}_F$ being round-based. Clearly, for any round the number $k$ of effectively read atoms is equal to the number of atoms written. Let $w$ be the number of block writes in the round, leading to a cost of $\omega w$, and the bound $k \leq Bw$. The I/O volume for writing in $\mathcal{P}_F$ is $Bw$. Let $r$ be the number of block read operations of $\mathcal{P}_A$, with cost $r$. All non-full read operations in $\mathcal{P}_F$ will have a volume of $2rB/\omega$. The full read operations have a total volume of at most $k \leq Bw$. Hence the overall volume of a round in $\mathcal{P}_F$ is $Bw + 2rB/\omega + k \leq 2Bw + 2rB/\omega \leq 2(\omega w + r)B/\omega$. Summing over all rounds and accounting for the initial read and write scan leads to the theorem. □

Using the classical lower bound on permuting [1] we get:

Corollary 4.4.

$$Q\left(N, M, B, \omega\right) = \Omega\left(\min\left\{N, \omega n \log_{\omega m} n\right\}\right) - 2\omega n$$

Note that the parameter range for which this lower bound is non-trivial depends on the constant factor of the $\Omega$.

## 4.2 Lower bound via counting

In this section we prove the lower bound for permutation directly. It is a combination of the counting argument of [1] and the rounds introduced in the previous section.

Observe that when performing a permutation, any atom that is read from external memory but is not written back to the external memory does not contribute to generating a permutation. Therefore, we require that atoms are moved between the external memory and internal memory as follows. When reading a block $B_i$ from external memory, a program must decide which subset $S$ of atoms of $B_i$ will be kept in internal memory to be written later. Exact copies of the atoms in $S$ are created in internal memory, while destroying their copies in the external memory. The rest of the atoms of $B_i$ are left unchanged in the external memory. When writing an internal memory block $B_i'$ to the external memory, some atoms of $B_i'$ can be set to be empty. However, writing $B_i'$ to external memory replaces everything in the destination block $B_i$ with the contents of $B_i'$, i.e., any non-empty atom in $B_i$ is destroyed. Since an atom can exist either in the internal memory or in the external memory, but not both, and since there is no way to generate destroyed atoms, writing to external memory can only be performed into empty blocks (either a new block location or all atoms of the destination block had been destroyed via moving them into internal memory in prior reads).

We upper bound the number $P(R)$ of permutations a round-based algorithm can generate after $R$ rounds using the above rules. Since every correct algorithm must be able to generate every possible permutation, the inequality $P(R) \geq N!$ will provide us with the lower bound on the number of rounds $R$ required to generate these permutations. Finally, since every round (except possibly for the last one) costs $\Theta(\omega m) = \Theta(\omega M/B)$, every algorithm's cost to permute $N$ atoms is $\Omega(\omega m R)$. More precisely, we count the number of different normalized programs with $\ell$ I/Os, where we make sure that every permutation requires a different program.

Typically, we are interested in algorithms which at the end of computation, place the output within the first $\lceil N/B \rceil = \lceil n \rceil$ contiguous blocks of external memory. However, for our lower bound we only require the final output to reside in $\lceil n \rceil$ blocks of external memory, without the requirement for these blocks to be adjacent. Clearly, any lower bound with this relaxation holds in the more stringent setting of requiring the output to be in the fully contiguous space in external memory.

In the original permutation lower bound proof in the (symmetric) EM model, Aggarwal and Vitter [1] argued that the $B!$ permutation within each of $\lceil N/B \rceil$ blocks should be counted only once (see [1] for detailed argument). They count them only when a block $B_i$ is read for the first time. However, in the AEM model, a read of a block might not necessarily use all the atoms of a block, so we cannot argue that we can count those $B!$ permutations when we read the block for the first time. Instead, we can count them the last time a block is written. And since the final output is written into $\lceil N/B \rceil$ blocks and each final block (except the last one) contains exactly $B$ atoms, counting the $B!$ permutations within each final block $B_i$ during the final writing of $B_i$ to the external memory adds up to a total of $B!^{N/B}$ permutations. Thus, when counting the upper bound $P(R)$ on the number of generated permutations, we simply ignore the permutations within each block and require that $P(R) \geq \frac{N!}{B!^{N/B}}$. One can think of this as a normalization (selecting one out of many programs that create the same permutation), where until the block is finally written, the relative order of the atoms within that block is the same as in the input. This effectively reduces the content of a block to be an (unordered) subset of the input atoms.

Let us compute the multiplicative factor of permutations that the $r$-th round can generate. Let $\mathcal{N}_r$ be the set of non-empty blocks in the external memory at the beginning of the $r$-th round. Note, $|\mathcal{N}_r| \leq N$ because every atom can only be moved and no new atoms are generated.

Let us compute the number of ways to pick up to $M$ atoms in external memory to bring into the internal memory. There are $\binom{|\mathcal{N}_r|}{\omega M/B} \leq \binom{N}{\omega M/B}$ ways to pick $\omega M/B$ blocks from the input residing in the external memory. Out of the chosen $\omega M$ atoms in the $\omega M/B$ blocks, there are $\leq \binom{\omega M}{M} 2^M$ ways to pick up to $M$ atoms to keep in internal memory: $\binom{\omega M}{M}$ ways to pick exactly $M$ atoms out of $\omega M$ possible ones and 2 ways for each of these $M$ atoms to decide whether to keep it in the internal memory or not.

Since we empty the internal memory between the rounds, at the end of the $r$-th round we have to write all the atoms that we chose to bring into the internal memory back to the external memory. Even if we chose to bring all $M$ atoms into internal memory, there are $\frac{M!}{B!^{M/B}}$ ways to permute $M$ elements, without counting the permutations within each block, since we are ignoring those. Finally, each block in the internal memory can be written to $\leq 2|\mathcal{N}_r| + 1$ distinct locations in the external memory: one of $|\mathcal{N}_r|$ blocks of $\mathcal{N}_r$ which might have become empty due to reading at the beginning of $r$-th round, or one of the $|\mathcal{N}_r| + 1$ locations between the blocks of $\mathcal{N}_r$. So the choice of writing $\leq M/B$ blocks from the internal memory to the external memory provides a multiplicative factor of at most $(2|\mathcal{N}_r| + 1)^{M/B} \leq (3N)^{M/B}$.

Thus, the total number of permutations that can be generated after $R$ rounds is at most

$$P(R) \le \left[ \binom{N}{\frac{\omega M}{B}} \binom{\omega M}{M} 2^M \frac{M!}{B!^{M/B}} (3N)^{M/B} \right]^R \tag{1}$$

Using the inequalities $\binom{n}{k} \le \left(\frac{n \cdot e}{k}\right)^k$ and $\left(\frac{k}{3}\right)^k \le k! \le \left(\frac{k}{2}\right)^k$, the above expression simplifies to

$$\begin{aligned} P(R) &\le \left[ \left(\frac{Ne}{\omega M/B}\right)^{\omega M/B} (e\omega)^M 2^M \left(\frac{3M}{2B}\right)^M (3N)^{M/B} \right]^R \\ &\le \left[ \left(\frac{N^{1+\frac{1}{\omega}} \cdot 3^{\frac{1}{\omega}} e}{\omega M/B}\right)^{\omega M/B} (3e\omega m)^M \right]^R \\ &\le \left[ \left(\frac{N^{1+\frac{1}{\omega}} \cdot 3^{\frac{1}{\omega}} e}{\omega m}\right) (3e\omega m)^{B/\omega} \right]^{\omega m R} \end{aligned}$$

Since we must have $\frac{N!}{B!^{N/B}} \le P(R)$,

$$\begin{aligned} \omega m R &\ge \frac{\log \frac{N!}{B!^{N/B}}}{\log \left( \left( \frac{N^{1+\frac{1}{\omega}} \cdot 3^{\frac{1}{\omega}} e}{\omega m} \right) (3e\omega m)^{B/\omega} \right)} \\ &\ge \frac{N \log(2N/3B)}{\log \left( \frac{N^{1+\frac{1}{\omega}} \cdot 3^{\frac{1}{\omega}} e}{\omega m} \right) + (B/\omega) \log(3e\omega m)} \\ &\ge \frac{N \log(N/2B)}{2 \cdot \max \left\{ \log \left( \frac{N^{1+\frac{1}{\omega}} \cdot 3^{\frac{1}{\omega}} e}{\omega m} \right), \frac{B}{\omega} \log(3e\omega m) \right\}} \end{aligned}$$

Observe that $\log \left( \frac{N^{1+\frac{1}{\omega}} \cdot 3^{\frac{1}{\omega}} e}{\omega m} \right) = O\left(\frac{\omega+1}{\omega} \log N\right) \le c \log N$, for some constant $c > 0$ and sufficiently large $N$ (the last inequality following from the fact that $\omega \ge 1$, i.e., $\frac{\omega+1}{\omega} \le 2$).

Assuming $\omega \le N/B$ or, equivalently, $\omega B \le N$, we distinguish two cases, depending on which term in the denominator dominates:

(1) If $B \ge \frac{c\omega \log N}{\log(3e\omega m)}$ then $\omega m R = \Omega(n \log_{\omega m} n)$

(2) If $B < \frac{c\omega \log N}{\log(3e\omega m)}$ then
$\log \frac{N}{2B} = \log \frac{N}{2\sqrt{B \cdot B}} \ge \log \frac{N}{2\sqrt{Bc\omega \log N}}$
$\ge \log \frac{N}{2\sqrt{cN \log N}} = \Omega(\log N)$
and, therefore, $\omega m R = \Omega(N)$.

Thus, we obtain the following lower bound on the cost of any round-based algorithm:

$$Q(N, \omega, M, B) = \Omega(\omega m R) = \Omega\left(\min\left\{N, \omega n \log_{\omega m} n\right\}\right)$$

Combining with Corollary 4.2, we obtain the following theorem:

Theorem 4.5. *Assuming that $\omega \le N/B$, the cost of permuting $N$ elements of an array in the $(M, B, \omega)$-AEM model is at least*

$$\Omega\left(\min\left\{N, \omega n \log_{\omega m} n\right\}\right)$$

## 5 COMPLEXITY OF SPMXV, COLUMN MAJOR LAYOUT

In this section we will mainly prove a lower bound for computing the product $\mathbf{A} \cdot x$ between a sparse matrix $\mathbf{A}$ by a dense vector $x$ in the $(M, B, \omega)$-AEM model. This proof extends the ideas of [5] to the $(M, B, \omega)$-AEM model. In an attempt to be somewhat self contained, we present a complete but concise whole proof.

We consider an $N \times N$ matrix $\mathbf{A}$ stored in the column-major order in the external memory. This means that the non-zero entries of $\mathbf{A}$ are stored in the following way: Start with the first column, list the non-zero elements for increasing row index, then take the second column, and so on. The stored list consists of triples $(i, j, a_{ij})$. $\mathbf{A}$ consists of a total of $H = \delta \cdot N$ non-zero entries, for some integer $\delta \ge 1$, i.e., on average every column or row has $\delta$ non-zero entries.

We start by stating the upper bounds in the $(M, B, \omega)$-AEM model. They follow the well-known algorithms in the (symmetric) EM model. Similar to permuting, for each conformation of the matrix, there is a direct or naive algorithm that produces the output vector in its natural order. For each output element $y_i$, the program considers all entries $a_{ij}$ in the $i$-th row of $\mathbf{A}$, multiplying it by $x_i$ and adding the result to $y_i$. The cost of this program in the $(M, B, \omega)$-AEM model is $O(H + \omega n)$.

Alternatively, there is a sorting based algorithm. It performs a simultaneous scan of $\mathbf{A}$ (in column-major order) and $x$ and replaces the matrix entry $a_{ij}$ with the elementary product $a_{ij} x_j$. Next, it divides the matrix into $\delta$ meta columns, and sorts the $N$ entries of the meta column by the row indices of the entries, virtually reordering the meta columns into row-major layout. This essentially results in $\delta$ dense vectors, which need to be added up to yield the overall result.

The cost of this algorithm in the $(M, B, \omega)$-AEM model is dominated by sorting the input and writing the output for a total of $O\left(\omega h \log_{\omega m} \frac{N}{\max\{\delta, B\}} + \omega n\right)$. The $\max\{\delta, B\}$ term in the logarithm arises from the fact that each column is already sorted by the row indices and the base case of the mergesort is a sorted sequence of $\delta$ elements.

Hence, the upper bound for SpMxV problem in the $(M, B, \omega)$-AEM model is (note, the last term $\omega n$ is the cost of writing the output):

$$Q(N, H, \omega, M, B) = O\left(\min\left\{H, \omega h \log_{\omega m} \frac{N}{\max\{\delta, B\}}\right\} + \omega n\right)$$

When proving the permutation lower bounds in Section 4, we had to assume indivisibility of individual elements, i.e., that no new elements can be created by combining several elements or combining subset of bits from several elements. To that end, the only operation that was allowed was moving individual elements as atomic/indivisible entities between the external and internal memories. Since matrix multiplication inherently generates new elements via multiplications and additions, we must relax these constraints.

Following the approach of [5], we work with the semi-ring model, i.e., we consider only algorithms or programs that work over an arbitrary semi-ring. This means in particular that we restrict our attention to algorithms that do not rely on the existence of inverse elements and cancellation. This would disallow an algorithm like Strassen's matrix-matrix multiplication algorithm [16], but it is less of a restriction here because there are no known algorithms for SpMxV that use this. As for the other lower bounds, we consider programs that work only for one particular conformation of matrices, i.e., the structure of $\mathbf{A}$ is fixed, and the semi-ring atoms signifying the $a_{ij}$ are stored in the input in column major layout. This task is actually already difficult for the input vector being the all ones vector, meaning that we only have to compute the sum of the elements of each row.

The proof of the following theorem is an adaptation of the proof of Theorem 6.2 in [5] to our situation. Note that the assumption $\omega \cdot \delta \cdot M \cdot B \leq N^{1-\varepsilon}$ is presumably stronger than necessary. We use it here to simplify the calculations significantly in comparison to the original proof in [5].

Theorem 5.1. *Consider any semi-ring program in the $(M,B,\omega)$-AEM model, where $B > 2$, $M > 4B$ and $\omega \cdot \delta \cdot M \cdot B \leq N^{1-\varepsilon}$, for a constant $\varepsilon > 0$. There is a sparse $N \times N$ matrix $\mathbf{A}$ with precisely $\delta \geq 1$ non-zero entries per column such that for $\mathbf{A}$ stored in column major layout, multiplying $\mathbf{A}$ with the all ones vector incurs a cost of at least*

$$Q(N,H,\omega,M,B) = \Omega\left(\min\left\{H, \omega h \log_{\omega m} \frac{N}{\max\{\delta,B\}}\right\}\right)$$

Proof. Assume the program is round-based, which by Corollary 4.2 is irrelevant for the statement of the theorem. A configuration of a program describes which atoms are stored in which memory cell at a certain time. We trace the computation backward, i.e., there is a unique final configuration, and we count how many different initial configurations are possible if $R$ many rounds are available. Since we are working with programs, rather than algorithms, we can assume that all atoms are actually used for the output. Because we consider multiplying with the all ones vector, all atoms used by the program are (partial) sums $\sum_{j \in S} a_{ij}$ for some $S$, including the input elements for $|S| = 1$ and the output elements for $S$ consisting of all (non empty) columns of that row. It is sufficient to trace the program by marking for each atom the row it belongs to. In this abstract trace, the final configuration is fixed. Additionally, the initial configuration of this trace identifies the conformation of the input matrix (because it is stored in column major layout and has precisely $\delta$ entries per column). Further we insist on the atoms stored in one block being sorted by the row they belong to. This effectively means that a block stores a subset of rows. To account for the additional choice of the program to change this order in the input blocks, we introduce the function $\tau$ below. So far we follow precisely the approach of [5]. Since the program is round-based, it suffices to focus on configurations with empty internal memory between the rounds. Also, note that the configurations can be described by the choices of atoms (initial or intermediate) within the blocks with the order within each block being irrelevant.

For permuting, in both the simulation in the flash model and in the counting lower bound it is important that the amount of data usefully read equals the amount (usefully) written. This is no longer true for the task of SpMxV because here the data read might be partial sums of the same row that are added together to form an atom that is written out. Fortunately, this reduction in volume is fairly limited: Every correct program for the task performs exactly $(\delta - 1)N$ additions. If a round of the computation does not perform any additions, the number of effectively read elements and written elements is identical. The number of additions performed during one round is precisely the difference between the number of elements read and written. Let $s_r, r = 1, \ldots, R$ be the number of additions in round $r$. We have $\sum_{r=1}^{R} s_r = (\delta - 1)N$. Because round $r$ writes at most $M$ atoms, it can usefully read at most $M + s_r$ atoms.

For a fixed configuration after executing the round, we are interested in the number of different possible configurations before the round. First, we fix the blocks this round uses for reading and writing; this gives a certain multiplicity we will see in the formulas. If an atom is written out and read in again within one round, we don't record this individual movement, but only the net effect of the round. Having fixed this, there is a well defined set of rows for which intermediate results are written in this round; its size is at most $M$.

In each round $r$, we have at most $\omega \frac{M}{B}$ read operations, each reading $c_k \leq B$ atoms, and in total reading $\sum_k c_k \leq M + s_r$ atoms. For a fixed choice of where the output is written to, the memory content (as a subset of size at most $M$ of rows) is fixed. We calculate the number of different configurations that are possible before the round, not (yet) accounting for the choice of block where input is read from (i.e. only considering content).

$$\begin{aligned}\prod_{k=1}^{\omega \frac{M}{B}} \binom{M}{c_k} \quad &\leq \quad \binom{\sum_{k=1}^{\omega \frac{M}{B}} M}{\sum_{k=1}^{\omega \frac{M}{B}} c_k} \leq \binom{\omega \frac{M}{B} M}{M + s_r} \\ &\leq \quad \left(\frac{e\omega \frac{M}{B} M}{M + s_r}\right)^{M+s_r} \leq \left(\frac{e\omega M}{B}\right)^{M+s_r}\end{aligned}$$

Here, the first inequality follows from a combinatorial argument: To describe all choices, we can create a marked/disjoint union of the individual universes. The individual subsets lead to a subset of size 'sum of the sizes'.

By choosing the addresses of the involved blocks, which is at most $H$, the total number of different preceding configurations for round $r$ is at most

$$H^{(\omega+1)\frac{M}{B}} \cdot \left(\frac{e\omega M}{B}\right)^{M+s_r} .$$

Multiplying this over $R$ rounds bounds the number of different set-wise starting configurations, which must be enough to handle all different conformations of sparse matrices:

$$\prod_{r=1}^{R} H^{(\omega+1)\frac{M}{B}} \cdot \left(\frac{e\omega M}{B}\right)^{M+s_r} = H^{R(\omega+1)\frac{M}{B}} \cdot \left(\frac{e\omega M}{B}\right)^{Mr+(\delta-1)N}$$

Additionally, the algorithm is free to choose the $s_r$ and the order of atoms within the input block, and we get the following inequality that bounds $R$:

$$H^{R(\omega+1)\frac{M}{B}} \cdot \left(\frac{e\omega M}{B}\right)^{Mr+(\delta-1)N} \cdot H^R \geq \binom{N}{\delta}^N / \tau(N,\delta,B),$$

using the definition of [5] for $\tau$:

$$\tau(N,\delta,B) = \begin{cases} 3^{\delta N} & \text{if } B < \delta, \\ 1 & \text{if } B = \delta, \\ (2eB/\delta)^{\delta N} & \text{if } B > \delta. \end{cases}$$

Solving for $R$ we get

$$R(\omega + 1)\frac{M}{B}\log H + (Mr + (\delta - 1)N)\log\left(\frac{e\omega M}{B}\right) + R\log H \geq$$
$$\geq \delta N \log\frac{N}{\delta} - \log\tau(N,\delta,B)\,,$$

$$R\left(2\omega\frac{M}{B}\log H + M\log\frac{e\omega M}{B} + \log H\right)$$
$$\geq \delta N\left(\log\left(\frac{N}{\delta}\,\frac{B}{e\omega M}\right)\right) - \log\tau(N,\delta,B)$$

implying that the cost is

$$Q(\cdot) = R\omega\frac{M}{B} \geq \frac{\delta N\log\frac{N}{\delta} - \log\tau(N,\delta,B) - \log\frac{B}{e\omega M}}{2\log H + \frac{B}{\omega}\log\frac{e\omega M}{B} + \frac{B}{\omega M}\log H}$$

Simplifying the enumerator using the definition of $\tau$, this is:

$$Q(\cdot) = R\omega\frac{M}{B} \geq \frac{\delta N\log\left(\frac{N}{\max\{3\delta,2eB\}}\,\frac{B}{e\omega M}\right)}{2\log H + \frac{B}{\omega}\log\frac{e\omega M}{B} + \frac{B}{\omega M}\log H}$$

We distinguish cases by the leading term of the denominator. The last term is always dominated by the first, so we can ignore it for asymptotic considerations. If the first term in the denominator dominates the second one, we use the assumption $\omega\cdot\delta\cdot M\cdot B \leq N^{1-\varepsilon}$ to conclude $Q(\cdot) = \Omega(H)$. If, on the other hand, the second term in the denominator dominates the first one, then

$$\begin{aligned} Q(\cdot) \quad &\geq \quad \frac{\omega\delta N}{3B}\left(\frac{\log\frac{N}{\max\{3\delta,2eB\}}}{\log\frac{e\omega M}{B}} - 1\right) \\ &= \quad \Omega\left(\omega h \log_{\omega m}\frac{N}{\max\{\delta,B\}}\right) \end{aligned}$$

matching the bound of the sorting-based algorithm. □

## REFERENCES

[1] Alok Aggarwal and S. Vitter, Jeffrey. 1988. The Input/Output Complexity of Sorting and Related Problems. *Commun. ACM* 31, 9 (Sept. 1988), 1116–1127. DOI: http://dx.doi.org/10.1145/48529.48535

[2] Deepak Ajwani, Andreas Beckmann, Riko Jacob, Ulrich Meyer, and Gabriel Moruz. 2009. *On Computational Models for Flash Memory Devices.* Springer Berlin Heidelberg, Berlin, Heidelberg, 16–27. DOI: http://dx.doi.org/10.1007/978-3-642-02011-7_4

[3] Avraham Ben-Aroya and Sivan Toledo. 2011. Competitive Analysis of Flash Memory Algorithms. *ACM Trans. Algorithms* 7, 2, Article 23 (March 2011), 37 pages. DOI: http://dx.doi.org/10.1145/1921659.1921669

[4] Naama Ben-David, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Yan Gu, Charles McGuffey, and Julian Shun. 2016. Parallel Algorithms for Asymmetric Read-Write Costs. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '16).* ACM, New York, NY, USA, 145–156. DOI: http://dx.doi.org/10.1145/2935764.2935767

[5] Michael A. Bender, Gerth Stølting Brodal, Rolf Fagerberg, Riko Jacob, and Elias Vicari. 2010. Optimal Sparse Matrix Dense Vector Multiplication inthe I/O-Model. *Theory of Computing Systems* 47, 4 (2010), 934–962. DOI: http://dx.doi.org/10.1007/s00224-010-9285-4

[6] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Yan Gu, and Julian Shun. 2015. Efficient Algorithms with Asymmetric Read and Write Costs. In *Proceedings of the 24th Annual European Symposium on Algorithms (ESA '16).* 14:1–14:18.

[7] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Yan Gu, and Julian Shun. 2015. Sorting with Asymmetric Read and Write Costs. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '15).* ACM, New York, NY, USA, 1–12. DOI: http://dx.doi.org/10.1145/2755573.2755604

[8] Xiangyu Dong, Norman P. Jouppi, and Yuan Xie. 2009. PCRAMsim: System-level Performance, Energy, and Area Modeling for Phase-change Ram. In *Proceedings of the 2009 International Conference on Computer-Aided Design (ICCAD '09).* ACM, New York, NY, USA, 269–275. DOI: http://dx.doi.org/10.1145/1687399.1687449

[9] Xiangyu Dong, Xiaoxia Wu, Guangyu Sun, Yuan Xie, H. Li, and Yiran Chen. 2008. Circuit and microarchitecture evaluation of 3D stacking magnetic RAM (MRAM) as a universal memory replacement. In *2008 45th ACM/IEEE Design Automation Conference.* 554–559.

[10] Eran Gal and Sivan Toledo. 2005. Algorithms and Data Structures for Flash Memories. *ACM Comput. Surv.* 37, 2 (June 2005), 138–163. DOI: http://dx.doi.org/10.1145/1089733.1089735

[11] Jia-Wei Hong and H. T. Kung. 1981. I/O Complexity: The Red-blue Pebble Game. In *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing (STOC '81).* ACM, New York, NY, USA, 326–333. DOI: http://dx.doi.org/10.1145/800076.802486

[12] Hyojun Kim, Sangeetha Seshadri, Clement L. Dickey, and Lawrence Chiu. 2014. Evaluating Phase Change Memory for Enterprise Storage Systems: A Study of Caching and Tiering Approaches. *Trans. Storage* 10, 4, Article 15 (Oct. 2014), 21 pages. DOI: http://dx.doi.org/10.1145/2668128

[13] Suman Nath and Phillip B. Gibbons. 2008. Online Maintenance of Very Large Random Samples on Flash Storage. *Proc. VLDB Endow.* 1, 1 (Aug. 2008), 970–983. DOI: http://dx.doi.org/10.14778/1453856.1453961

[14] Hyoungmin Park and Kyuseok Shim. 2009. FAST: Flash-aware External Sorting for Mobile Database Systems. *J. Syst. Softw.* 82, 8 (Aug. 2009), 1298–1312. DOI: http://dx.doi.org/10.1016/j.jss.2009.02.028

[15] Moinuddin K. Qureshi, Sudhanva Gurumurthi, and Bipin Rajendran. 2011. *Phase Change Memory: From Devices to Systems* (1st ed.). Morgan & Claypool Publishers.

[16] Volker Strassen. 1969. Gaussian Elimination is Not Optimal. *Numer. Math.* 13, 4 (Aug. 1969), 354–356. DOI: http://dx.doi.org/10.1007/BF02165411

[17] Stratis D. Viglas. 2014. Write-limited Sorts and Joins for Persistent Memory. *Proc. VLDB Endow.* 7, 5 (Jan. 2014), 413–424. DOI: http://dx.doi.org/10.14778/2732269.2732277