

Sorting and Permuting without Bank Conflicts on GPUs

Peyman Afshani^{*1} and Nodari Sitchinava²

¹ MADALGO, Aarhus University, Denmark

² University of Hawaii – Manoa, HI, USA

Abstract. In this paper, we look at the complexity of designing algorithms without any bank conflicts in the shared memory of Graphical Processing Units (GPUs). Given input of size n , w processors and w memory banks, we study three fundamental problems: sorting, permuting and w -way partitioning (defined as sorting an input containing exactly n/w copies of every integer in $[w]$).

We solve sorting in optimal $O(\frac{n}{w} \log n)$ time. When $n \geq w^2$, we solve the partitioning problem optimally in $O(n/w)$ time. We also present a general solution for the partitioning problem which takes $O(\frac{n}{w} \log_{n/w}^3 w)$ time. Finally, we solve the permutation problem using a randomized algorithm in $O(\frac{n}{w} \log \log \log_{n/w} n)$ time. Our results show evidence that when working with banked memory architectures, there is a separation between these problems and the permutation and partitioning problems are not as easy as simple parallel scanning.

1 Introduction

Graphics Processing Units (GPUs) over the past decade have been transformed from special-purpose graphics rendering co-processors, to a powerful platform for general purpose computations, with runtimes rivaling best implementations on many-core CPUs. With high memory throughput, hundreds of physical cores and fast context switching between thousands of threads, they became very popular among computationally intensive applications. Instead of citing a tiny subset of such papers, we refer the reader to gpgpu.org website [11], which lists over 300 research papers on this topic.

Yet, most of these results are experimental and the theory community seems to shy away from designing and analyzing algorithms on GPUs. In part, this is probably due to the lack of a simple theoretical model of computation for GPUs. This has started to change recently, with introduction of several theoretical models for algorithm analysis on GPUs.

A brief overview of GPU architecture. A modern GPU contains hundreds of physical cores. To implement such a large number of cores, a GPU is designed hierarchically. It consists of a number of *streaming multiprocessors*

* Work supported in part by the Danish National Research Foundation grant DNRF84 through Center for Massive Data Algorithmics (MADALGO)

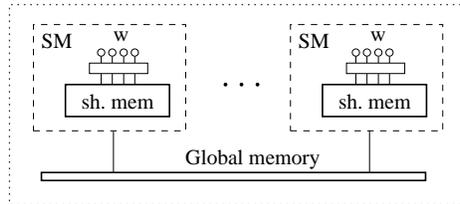


Fig. 1: A schematic of GPU architecture

(SMs) and a *global memory* shared by all SMs. Each SM consists of a number of cores (for concreteness, let us parameterize it by w) and a *shared memory* of limited size which is shared by all the cores within the SM but is inaccessible by other SMs. With computational power of hundreds of cores, latency of accessing memory becomes non-negligible and GPUs take several approaches to mitigate the problem.

First, they support massive hyper-threading, i.e., multiple logical threads may run on each physical core with light context switching between the threads. Thus, when a thread stalls on a memory request, the other threads can continue running on the same core. To schedule all these threads efficiently, groups of w threads, called *warps*, are scheduled to run on w physical cores simultaneously in *single instructions, multiple data (SIMD)* [9] fashion.

Second, there are limitations on how data is accessed in memory. Accesses to global memory are most efficient if they are *coalesced*. Essentially, it means that w threads of a warp should access contiguous w addresses of global memory. On the other hand, shared memory is partitioned into w *memory banks* and each memory bank may service at most one thread of a warp in each time step. If more than one thread of a warp requests access to the same memory bank, a *bank conflict* occurs and multiple accesses to the same memory bank are sequentialized. Thus, for optimal utilization of processors, it is recommended to design algorithms that perform coalesced accesses to global memory and incur no bank conflicts in shared memory [19].

Designing GPU algorithms. Several papers [13,17,18,22] present theoretical models that incorporate the concept of coalesced accesses to global memory into the performance analysis of GPU algorithms. In essence, all of them introduce a complexity metric that counts the number of blocks transferred between the global and internal memory (shared memory or registers), similar to the I/O-complexity metric of sequential and parallel external memory and cache-oblivious models on CPUs [2,3,5,10]. The models vary in what other features of GPUs they incorporate and, consequently, in the number of parameters introduced into the model.

Once the data is in shared memory, the usual approach is to implement standard parallel algorithms in the PRAM model [14] or interconnection networks [16]. For example, sorting data in shared memory, is usually implemented

using sorting networks, e.g. Batcher’s odd-even mergesort [4] or bitonic merge-sort [4]. Even though these sorting networks are not asymptotically optimal, they provide good empirical runtimes because they exhibit small constant factors, they fit well in the SIMD execution flow within a warp, and for small inputs asymptotic optimality is irrelevant. On very small inputs (e.g. on w items – one per memory bank) they also cause no bank conflicts.

However, as the input sizes for shared memory algorithms grow, bank conflicts start to affect the running time.

The first paper that studies bank conflicts on GPUs is by Dotsenko et al. [8]. The authors view shared memory as a two-dimensional matrix with w rows, where each row represents a separate memory bank. Any one-dimensional array $A[0..n - 1]$ will be laid out in this matrix in column-major order in $\lceil n/w \rceil$ contiguous columns. Note, that *strided* parallel access to data, that is each thread t accessing array entries $A[wi + t]$ for integer $0 \leq i < \lceil n/w \rceil$, does not incur bank conflicts because each thread accesses a single row. The authors also observed that the *contiguous* parallel access to data, that is each thread scanning a contiguous section of $\lceil n/w \rceil$ items of the array, also incurs no bank conflicts if $\lceil n/w \rceil$ is co-prime with w . Thus, with some extra padding, contiguous parallel access to data can also be implemented without any bank conflicts.

Instead of adding padding to ensure that $\lceil n/w \rceil$ is co-prime with w , another solution to bank-conflict-free contiguous parallel access is to convert the matrix from column-major layout to row-major layout and perform strided access. This conversion is equivalent to in-place transposition of the matrix. Catanzaro et al. [6] study this problem and present an elegant bank-conflict-free parallel algorithm that runs in $\Theta(n/w)$ time, which is optimal.

Sitchinava and Weichert [22] present a strong correlation between bank conflicts and the runtime for some problems. Based on the matrix view of shared memory, they developed a sorting network that incurred no bank conflicts. They show that although compared to Batcher’s sorting networks their solution incurs extra $\Theta(\log n)$ factor in parallel time and work, it performs better in practice because it incurs no bank conflicts.

Nakano [18] presents a formal definition of a parallel model with the matrix view of shared memory, which is also extended to model memory access latency hiding via hyper-threading.³ He calls his model *Discrete Memory Model (DMM)* and studies the problem of *offline permutation*, which we define in detail later.

The DMM model is probably the simplest abstract model that captures the important aspects of designing bank-conflict-free algorithms for GPUs. In this paper we will work in this model. However, to simplify the exposition, we will assume that each memory access incurs no latency (i.e. takes a unit time) and we have exactly w processors. This simplification still captures the key algorithmic challenges of designing bank-conflict-free algorithms without the added complexity of modeling hyper-threading. We summarize the key features of the model below, and for more details refer the reader to [18].

³ Nakano’s DMM exposition actually swapped the rows and columns and viewed memory banks as columns of the matrix and the data laid out in row-major order.

Model of Computation. Data of size n is laid out in memory as a matrix \mathcal{M} with dimensions $w \times m$, where $m = \lceil n/w \rceil$. As in the PRAM model, w processors proceed synchronously in discrete time steps, and in each time step perform access to data (a processor may skip accessing data in some time step). Every processor can access any item within the matrix. However, an algorithm must ensure that in each time step at most one processor accesses data within a particular row.⁴ Performing computation on a constant number of items takes unit time and, therefore, can be completed within a single time step. The parallel time complexity (or simply *time*) is the number of time steps required to complete the task. The work complexity (or simply *work*) is the product of w and the time complexity of an algorithm.⁵

Although the DMM model allows each processor to access any memory bank (i.e. any row of the matrix), to simplify the exposition, it is helpful to think of each processor fixed to a single row of the matrix and the transfer of information between the processors being performed via “message passing”, where at each step, processor i may send a message (constant words of information) to another processor j (e.g., asking to read or write a memory location within row j). Next, the processor j can respond by sending constant words of information back to row i . Crucially and to avoid bank conflicts, we demand that at each parallel computation step, at most one message is received by each row; we call this the “Conflict Avoidance Condition” or CAC.

Note that this view of interprocessor communication via message passing is only done for the ease of exposition, and algorithms can be implemented in the DMM model (and in practice) by processor i directly reading or writing the contents of the target memory location from the appropriate location in row j . Finally, CAC is equivalent to each memory bank being accessed by at most one processor in each access request made by a warp.

Problems of interest. Using the above model, we study complexity of developing bank conflict free algorithms for the following fundamental problems:

- *Sorting*: The matrix \mathcal{M} is populated with items from a totally ordered universe. The goal is to have \mathcal{M} sorted in row-major order.⁶
- *Partition*: The matrix \mathcal{M} is populated with labeled items. The labels form a permutation that contains m copies of every integer in $[w]$ and an item with label i needs to be sent to row i .
- *Permutation*: The matrix \mathcal{M} is populated with labeled items. The labels form a permutation of tuples $[w] \times [m]$. And an item with label (i, j) needs to be sent to the j -th memory location in row i .

⁴ This is analogous to how EREW PRAM model requires the algorithms to be designed so that in each time step at most one processor accesses any memory address.

⁵ Work complexity is easily computed from time complexity and number of processors, therefore, we don’t mention it explicitly in our algorithms. However, we mention it here because it is a useful metric for efficiency, when compared to the runtime of the optimal sequential algorithms.

⁶ The final layout within the matrix (row-major or column-major order) is of little relevance, because the conversion between the two layouts can be implemented efficiently in time and work required to simply read the input [6].

While the sorting problem is natural, we need to mention a few remarks regarding the permutation and the partition problems. Often these two problems are equivalent and thus there is little motivation to separate them. However rather surprisingly, it turns out that in our case these problems are in fact very different. Nonetheless, the permutation problem can be considered an “abstract” sorting problem where all the comparisons have been resolved and the goal is to merely send each item to its correct location. The partition problem is more practical and it appears in scenarios where the goal is to split an input into many subproblems. For example, consider a multi-way quicksort algorithm with pivots, $p_0 = -\infty, p_1, \dots, p_{w-1}, p_w = \infty$. In this case, row i would like to send all the items that are greater than p_{j-1} but less than p_j to row j but row i would not know the position of the items in the final sorted matrix. In other words, in this case, for each element in row i , we only know the destination row rather than the destination row and the rank.

Prior work. Sorting and permutation are among the most fundamental algorithmic problems. The solution to the permutation problem is trivial in the random access memory models – $\mathcal{O}(n)$ work is required in both RAM and PRAM models of computation – while sorting is often more difficult (the $\Omega(n \log n)$ comparison-based lower bound is a very classical result). However, this picture changes in other models. For example, in both the sequential and parallel external memory models [2, 3], which model hierarchical memories of modern CPU processors, existing lower bounds show that permutation is as hard as sorting (for most realistic parameters of the models) [2, 12].

In the context of GPU algorithms, matrix transposition was studied as a special case of the permutation problem by Catanzaro et al. [6] and they showed that one can transpose a matrix in-place without bank conflicts in $\mathcal{O}(n)$ work. While they didn’t explicitly mentioned the DMM model, their analysis holds trivially in the DMM model with unit latency. Nakano [18] studied the problem of performing arbitrary permutations in the DMM model *offline*, where the permutation is known in advance and we are allowed to pre-compute some information before running the algorithm. The time to perform the precomputation is not counted toward the complexity of performing the permutation. Offline permutation is useful if the permutation is fixed for a particular application and we can encode the precomputation result in the program description. Common examples of fixed permutations include matrix transposition, bit-reversal permutations, and FFT permutations. Nakano showed that any offline permutation can be implemented in linear work. The required pre-computation in Nakano’s algorithm is coloring of a regular bipartite graph, which seems very difficult to adapt to the *online* permutation problem.

As mentioned earlier, Sitchinava and Weichert [22] presented the first algorithm for sorting $w \times w$ matrix which incurs no bank conflicts. They use Shearsort [21], which repeatedly sorts columns of the matrix in alternating order and rows in increasing order. After $\Theta(\log w)$ repetitions, the matrix is sorted in column-major order. Rows of the matrix can be sorted without bank conflicts. And since matrix transposition can be implemented without bank conflicts, the



Fig. 2: Consider one row that is being converted to column-major layout. The elements that will be put in the same column are bundled together. It is easily seen that each row will create at most one dirty column.

columns can also be sorted without bank conflicts via transposition and sorting of the rows. The resulting runtime is $\Theta(t(w) \log w)$, where $t(w)$ is the time it takes to sort an array of w items using a single thread.

Our Contributions. In this paper we present the following results.

- *Sorting:* We present an algorithm that runs in $O(m \log(mw))$ time, which is optimal in the context of comparison based algorithms.
- *Partition:* We present an optimal solution that runs in $O(m)$ time when $w \leq m$. We generalize this to a solution that runs in $O(m \log_m^3 w)$ time.
- *Permutation:* We present a randomized algorithm that runs in expected $O(m \log \log \log_m w)$ time. Even though this is a rather technical solution (and thus of theoretical interest), it strongly hints at a possible separation between the partition and permutation problems in the DMM model.

2 Sorting and Partitioning

In this section we improve the sorting algorithm of Sitchinava and Weichert [22] by removing a $\Theta(\log w)$ factor. We begin with our base case, a “short and wide” $w \times m$ matrix \mathcal{M} where $w \leq \sqrt{m}$. The algorithm repeats the following twice and then sorts the rows in ascending order.

1. Sort rows in alternating order (odd rows ascending, even rows descending).
2. Convert the matrix from row-major to column-major layout (e.g., the first w elements of the first row form the first column, the next w elements form the second column and so on).
3. Sort rows in ascending order.
4. Convert the matrix from column-major to row-major layout (e.g., the first m/w columns will form the first row).

Lemma 1. *The above algorithm sorts the matrix correctly in $O(m \log m)$ time.*

Proof. We would like to use the *0-1 principle* [15] but since we are not working with a sorting network, we simply reprove the principle. Observe that our algorithm is essentially oblivious to the values in the matrix and only takes into account the relative order of them. Pick a parameter i , $1 \leq i \leq mw$, that we call the *marking value*. Based on the above observation, we attach a mark of “0” to any element that has rank less than i and “1” to the remaining elements. These marks are symbolic and thus invisible to the algorithm. We say a column (or row) is *dirty* if it contains both elements with mark “0” and “1”. After the

first execution of steps 1 and 2, the number of dirty columns is reduced to at most w (Fig. 2). After the next execution of steps 3 and 4, the number of dirty rows is reduced to two. Crucially, the two dirty rows are adjacent. After one more execution of steps 1 and 2, there would be two dirty columns, however, since the rows were ordered in alternating order in step 1, one of the dirty rows will have “0”s toward the top and “1”s toward the bottom, while the other will have them in reverse order. This means, after step 3, there will be only one dirty column and only one dirty row after step 4. The final sorting round will sort the elements in the dirty row and thus the elements will be correctly sorted by their mark.

As the algorithm is oblivious to marks, the matrix will be sorted by marks regardless of our choice of the marking value, meaning, the matrix will be correctly sorted. Conversions between column-major and row-major (and vice versa) can be performed in $O(m)$ time using [6] while respecting CAC. Sorting rows is the main runtime bottleneck and the only part that needs more than $O(m)$ time. \square

Corollary 1. *The partition problem can be solved in $O(m)$ time if $w \leq \sqrt{m}$.*

Proof. Observe that for the partition problem, we can “sort” each row in $O(m)$ time (e.g., using radix sort). \square

Using the above as a base case, we can now sort a square matrix efficiently.

Theorem 1. *If $w = m$, then the sorting problem can be solved in $O(m \log m)$ time. The partition problem can be solved in $O(m)$ time.*

Proof. Once again, we use the idea of the 0-1 principle and assume the elements are marked with “0” and “1” bits, invisible to the algorithm. To sort an $m \times m$ matrix, partition the rows into groups of \sqrt{m} adjacent rows. Call each group a *super-row*. Sort each super-row using Lemma 1. Each super-row has at most one dirty row so there are at most \sqrt{m} dirty rows in total. We now sort the columns of the matrix in ascending order by transposing the matrix, sorting the rows, and then transposing it again. This will place all the dirty rows in adjacent rows. We sort each super-row in alternating order (the first super-row ascending, the next descending and so on). After this, there will be at most two dirty rows left, sorted in alternating order. We sort the columns in ascending order, which will reduce the number of dirty rows to one. A final sorting of the rows will have the matrix in the correct sorted order.

The running time is clearly $O(m \log m)$. In the partition problem, we use radix sort to sort each row and thus the running time will be $O(m)$. \square

For non-square matrices, the situation is not so straightforward and thus more interesting. First, we observe that by using the $O(\log w)$ time EREW PRAM algorithm [7] to sort w items using w processors, the above algorithm can be generalized to non-square matrices:

Theorem 2. *A $w \times m$ matrix \mathcal{M} , $w \geq m$, can be sorted in $O(m \log w)$ time.*

Proof. As before assume “0” and “1” marks have been placed on the elements in \mathcal{M} . First, we sort the rows. Then we sort the columns of \mathcal{M} using the EREW PRAM algorithm [7]. Next, we convert the matrix from column-major to row-major layout, meaning, in the first column, the first m elements will form the first row, the next m elements the next row and so on. This will leave at most m dirty rows. Once again, we sort the columns, which will place the dirty rows in adjacent rows. We sort each $m \times m$ sub-matrix in alternating order, using Theorem 1, which will reduce the number of dirty rows to two. The rest of the argument is very similar to the one presented for Theorem 1. \square

The next result might not sound very strong, but it will be very useful in the next section. It also reveals some differences between the partition and sorting problems. Note that the result is optimal as long as w is polynomial in m .

Lemma 2. *The partition problem on a $w \times m$ matrix \mathcal{M} , $w \geq m > 2\sqrt{\log w}$, can be solved in $O(m \log_m^3 w)$ time. The same bound holds for sorting $O(\log n)$ -bit integers.*

Proof. We use the idea of 0-1 principle, combined with radix sort, which allows us to sort every row in $O(m)$ time. The algorithm uses the following steps. **Balancing.** Balancing has $\lceil \log_m w \rceil$ rounds; for simplicity, we assume $\log_m w$ is an integer. In the zeroth round, we create w/m sub-matrices of dimensions $m \times m$ by partitioning the rows into groups of m adjacent rows and then sort each sub-matrix in column-major order (i.e., the elements marked “0” will be to the left and the elements marked “1” to the right). At the beginning of each subsequent round i , we have w/m^i sub-matrices with dimensions $m^i \times m$. We partition the sub-matrices into groups of m adjacent sub-matrices; from every group, we create m^i square $m \times m$ matrices: we pick the j -th row of every sub-matrix in the group, for $1 \leq j \leq m^i$, to create the j -th square matrix in the group. We sort each $m \times m$ matrix in column-major order. Now, each group will form an $m^{i+1} \times m$ sub-matrix for the next round. Observe that balancing takes $O(m \log_m w)$ time in total.

Convert and Divide (C&D). With a moment of thought, it can be proven that after *balancing*, each row will have between $\frac{w_0}{w} - \log_m w$ and $\frac{w_0}{w} + \log_m w$ (resp. $\frac{w_1}{w} - \log_m w$ and $\frac{w_1}{w} + \log_m w$) elements marked “0” (resp. “1”) where w_0 (resp. w_1) is the total number of elements marked “0” (resp. “1”). This means, that there are at most $d = 2 \log_m w$ dirty columns. We convert the matrix from column-major to row-major layout, which leaves us with $\frac{w}{md}$ adjacent dirty rows (each column is placed in $\frac{w}{m}$ rows after conversion). Now, we divide \mathcal{M} into md smaller matrices of dimension $\frac{w}{md} \times m$. Each matrix will be a new subproblem.

The algorithm. The algorithm repeatedly applies *balancing* and *C&D* steps: after the i -th execution of these two steps, we have $(md)^i$ subproblems where each subproblem is a $\frac{w}{(md)^i} \times m$ matrix, and in the next round, *balancing* and *C&D* operate locally within each subproblem (i.e., they work on $\frac{w}{(md)^i} \times m$ matrices). Note that before the divide step, each subproblem has at most $\frac{w}{(md)^i}$ adjacent dirty rows. After the divide step, these dirty rows will be sent to at

most two different sub-problems, meaning, at the depth i of the recursion, all the dirty rows will be clustered into at most 2^i groups of adjacent rows, each containing at most $\frac{w}{(md)^i}$ rows. Since, $m > 2\sqrt{\log_m w}$, after $2\log_m w$ steps of recursion, the subproblems will have at most m rows and thus each subproblem can be sorted in $O(m)$ time, leaving at most one dirty row per sub-problem. Thus, we will end up with only $2^{2\log_m w}$ dirty rows in the whole matrix \mathcal{M} .

We now sort the columns of the matrix \mathcal{M} : we fit each column of \mathcal{M} in a $\frac{w}{m} \times m$ submatrix and recurse on the submatrices using the above algorithm, then convert each submatrix back to a column. We will end up with a matrix with $2^{2\log_m w}$ dirty rows but crucially, these rows will be adjacent. Equally important is the observation that $2^{2\log_m w} \leq m/2$ since $m > 2\sqrt{\log w}$. To sort these few dirty rows, we decompose the matrix into square matrices of $m \times m$, and sort each square matrix. Next, we shift this decomposition by $m/2$ rows and sort them again. In one of these decompositions, an $m \times m$ matrix will fully contain all the dirty rows, meaning, we will end up with a fully sorted matrix. If $f(w, m)$ is the running time on a $w \times m$ matrix, then we have the recursion $f(w, m) = f(w/m, m) + O(m \log_m^2 w)$ which gives our claimed bound. \square

3 A Randomized Algorithm for Permutation

In this section we present an improved algorithm for the permutation problem that beats our best algorithm for partitioning for when the matrix is “tall and narrow” (i.e., $w \gg m$). We note that the algorithm is only of theoretical interest as it is a bit technical and it uses randomization. However, at least from this theoretical point of view, it hints that perhaps in the GPU model, the permutation problem and the partitioning problem are different and require different techniques to solve.

Remember that we have a matrix \mathcal{M} which contains a set of elements with labels. The labels form a permutation of $[w] \times [m]$, and an element with label (i, j) needs to be placed at the j -th memory location of row i . From now on, we use the term “label” to both refer to an element and its label.

To gain our speed up, we use two crucial properties: first, we use randomization and second we use the existence of the second index, j , in the labels.

Intuition and Summary. Our algorithm works as follows. It starts with a preprocessing phase where each row picks a random word and these random words are used to shuffle the labels into a more “uniform” distribution. Then, the main body of the algorithm begins. One row picks a random hash function and communicates it to the rest of the rows. Based on this hash function, all the rows compute a common coloring of the labels using m colors $1, \dots, m$ such that the following property holds: for each index i , $1 \leq i \leq w$, and among the labels $(i, 1), (i, 2), \dots, (i, m)$, there is exactly one label with color j , for every $1 \leq j \leq m$. After establishing such a coloring, in the upcoming j -th step, each row will send one label of color j to its correct destination. Our coloring guarantees that CAC is not violated. At the end of the m -th step, a significant majority of the labels are placed at their correct position and thus we are left

only with a few “leftover” labels. This process is repeated until the number of “leftover” labels is a $1/\log_m^3 w$ fraction of the original input. At this point, we use Lemma 2 which we show can be done in $O(m)$ time since very few labels are left. Next, we address the technical details that arise.

Preprocessing phase. For simplicity, we assume w is divisible by m . Each row picks a random number r between 1 and m and shifts its labels by that amount, i.e., places a label at position j into position $j + r \pmod{m}$. Next, we group the rows into matrices of size $m \times m$ which we transpose (i.e., the first m rows create the first matrix and so on).

The main body and coloring. We use a result by Pagh and Pagh [20].

Theorem 3. [20] *Consider the universe $[w]$ and a set $S \subset [w]$ of size m . There exists an algorithm in the word RAM model that (independent of S) can select a family H of hash functions from $[w]$ to $[v]$ in $O(\log m(\log v)^{O(1)})$ time and using $O(\log m + \log \log w)$ bits, such that:*

- H is m -wise independent on S with probability $1 - m^{-c}$ for a constant c
- Any member of H can be represented by a data structure that uses $O(m \log v)$ bits and the hash values can be computed in constant time. The construction time of the structure is $O(m)$.

The first row builds H then picks a hash function h from the family. This function, which is represented as a data structure, is communicated to the rest of the rows in $O(\log w + m)$ time as follows: assume the data structure consumes $S_m = O(m)$ space. In the first S_m steps, the first row sends the j -th word in the data structure to row j . In the subsequent $\log(w)$ rounds, the j -th word is communicated to any row j' with $j' \equiv j \pmod{S_m}$, using a simple broadcasting strategy that doubles the number of rows that have received the j -th word. This boils down the problem of transferring the data structure to the problem of distributing S_m random words between S_m rows which can be easily done in S_m steps while respecting CAC.

Coloring. A label (i, j) is assigned color k where $j \equiv h(i) + k \pmod{m}$.

Communication. Each row computes the color of its labels. Consider a row i containing some labels. The communication phase has αm steps, where α is a large enough constant. During step k , $1 \leq k \leq m$, the row i picks a label of color k . If no such label exists, then the row does nothing so let’s assume a label (i_k, j_k) has assigned color k . The row i sends this label to the destination row i_k . After performing these initial m steps, the algorithm repeats these m steps $\alpha - 1$ times for a total of αm steps. We claim the communication step respects CAC: assume otherwise that another label (i'_k, j'_k) is sent to row i_k during step k ; clearly, we must have $i'_k = i_k$ but this contradicts our coloring since it implies $j'_k \equiv h(i_k) + k \equiv j_k \pmod{m}$ and thus $j'_k = j_k$. Note that the communication phase takes $O(m)$ time as α is a constant.

Synchronization. Each row computes the number of labels that still need to be sent to their destination, and in $O(\log w + m)$ time, these numbers are summed and broadcast to all the rows. We repeat the main body of the algorithm as long as this number is larger than $\frac{wm}{\log_m^3 w}$.

Finishing. We present the details in the full version of the paper [1] but roughly speaking, we do the following: first, we show that we can pack the remaining elements into a $w \times O(\frac{m}{\log_m^3 w})$ matrix in $O(m)$ time (this is not trivial as the matrix can have rows with $\Omega(m)$ labels left). Then, we use Lemma 2 to sort the matrix in $O(m)$ time. Finishing off the sorted matrix turns out to be relatively easy.

Correctness and Running Time. Correctness is trivial as at the end all rows receive their labels and the algorithm is shown to respect CAC. Thus, the main issue is the running time. The *finishing* and *preprocessing* steps clearly takes $O(m)$ time. The running time of each repetition of *coloring*, *communication* and *synchronization* is $O(\log w + m)$ and thus to bound the total running time we simply need to bound how many times the synchronization steps are executed. To do this, we need the following lemma (proof is in the full paper [1]).

Lemma 3. *Consider a row ℓ containing a set \mathcal{S}_ℓ of k labels, $k \leq m$. Assume the hash function h is m -wise independent on \mathcal{S}_ℓ . For a large enough constant α , let $\mathcal{C}_{\text{heavy}}$ be the set of colors that appear more than α times in ℓ , and $\mathcal{S}_{\text{heavy}}$ be the set of labels assigned colors from $\mathcal{C}_{\text{heavy}}$. Then $\mathbb{E}[|\mathcal{S}_{\text{heavy}}|] = m(k/2m)^\alpha$ and the probability that $|\mathcal{S}_{\text{heavy}}| = \Omega(m(k/2m)^{\alpha/10})$ is at most $2^{-\sqrt{m}(k/2m)^{\alpha/10}}$.*

Consider a row ℓ and let k_i be the number of labels in the row after the i -th iteration of the *synchronization* ($k_0 = m$). We call ℓ a *lucky* row if the hash function is m -wise independent on ℓ during all the i iterations. Assume ℓ is lucky. During the *communication* step, we process all the labels except those in the set $\mathcal{S}_{\text{heavy}}$. By the above lemma, it follows that with high probability, we will have $O(m(k_i/2m)^{\alpha/10})$ labels left for the next iteration on ℓ , meaning, with high probability, $k_{i+1} = O(m(k_i/2m)^{\alpha/10})$.

Observe that if $m \geq \log^{O(1)} w$ (for an appropriate constant in the exponent) and $k_i > m/\log_m^3 w$, then $\sqrt{m} \left(\frac{k_i}{2m}\right)^{\alpha/10} > 3 \log w$ which means the probability that Lemma 3 “fails” (i.e., the “high probability” fails) is at most $1/w^3$. Thus, with probability at least $1 - 1/w$, Lemma 3 never fails for any lucky row and thus each lucky row will have at most

$$O\left(m \left(\frac{k_0}{2m}\right)^{(\alpha/10)^i}\right)$$

labels left after the i -th iteration. By Theorem 3, the expected number of unlucky rows is at most n/m^c at each iteration which means after $i = O(\log \log \log_m w)$ iterations, there will be $O(\frac{mw}{\log_m^3 w} + \frac{mw^i}{m^c}) = O(mw \log_m^3 w)$ labels left. So we proved that the algorithm repeats the *synchronization* step at most $O(\log \log \log_m n)$ times, giving us the following theorem.

Theorem 4. *The permutation problem on an $w \times m$ matrix can be solved in $O(m \log \log \log_m w)$ expected time, assuming $m = \Omega(\log^{O(1)} w)$. Furthermore, the total number of random words used by the algorithm is $w + O(m \log \log \log_m w)$.*

References

1. Afshani, P., Sitchinava, N.: Sorting and permuting without bank conflicts on GPUs. CoRR abs/1507.01391 (2015), <http://arxiv.org/abs/1507.01391>
2. Aggarwal, A., Vitter, J.S.: The input/output complexity of sorting and related problems. *Commun. ACM* 31, 1116–1127 (1988)
3. Arge, L., Goodrich, M.T., Nelson, M.J., Sitchinava, N.: Fundamental parallel algorithms for private-cache chip multiprocessors. In: 20th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA). pp. 197–206 (2008)
4. Batcher, K.E.: Sorting networks and their applications. In: AFIPS Spring Joint Computer Conference. pp. 307–314
5. Blelloch, G.E., Chowdhury, R.A., Gibbons, P.B., Ramachandran, V., Chen, S., Kozuch, M.: Provably good multicore cache performance for divide-and-conquer algorithms. In: 19th ACM-SIAM Symp. on Discrete Algorithms. pp. 501–510 (2008)
6. Catanzaro, B., Keller, A., Garland, M.: A decomposition for in-place matrix transposition. In: 19th ACM SIGPLAN Principles and Practices of Parallel Programming (PPoPP). pp. 193–206 (2014)
7. Cole, R.: Parallel merge sort. In: 27th IEEE Symposium on Foundations of Computer Science. pp. 511–516 (1986)
8. Dotsenko, Y., Govindaraju, N.K., Sloan, P.P., Boyd, C., Manfredelli, J.: Fast Scan Algorithms on Graphics Processors. In: 22nd International Conference on Supercomputing. pp. 205–213 (2008)
9. Flynn, M.: Some computer organizations and their effectiveness. *IEEE Transactions on Computers* C-21(9), 948–960 (Sept 1972)
10. Frigo, M., Leiserson, C.E., Prokop, H., Ramachandran, S.: Cache-oblivious algorithms. In: 40th IEEE Symp. on Foundations of Comp. Sci. pp. 285–298 (1999)
11. GPGPU.org: Research papers on gpgpu.org, <http://gpgpu.org/tag/papers>
12. Greiner, G.: Sparse Matrix Computations and their I/O Complexity. Dissertation, Technische Universität München, München (2012)
13. Haque, S., Maza, M., Xie, N.: A many-core machine model for designing algorithms with minimum parallelism overheads. In: High Performance Computing Symposium (2013)
14. JáJá, J.: An Introduction to Parallel Algorithms. Addison Wesley (1992)
15. Knuth, D.E.: The Art of Computer Programming, Volume III: Sorting and Searching. Addison-Wesley (1973)
16. Leighton, F.T.: Introduction to Parallel Algorithms and Architectures: Arrays, Trees, and Hypercubes. Morgan-Kaufmann, San Mateo, CA (1991)
17. Ma, L., Agrawal, K., Chamberlain, R.D.: A memory access model for highly-threaded many-core architectures. *Future Generation Computer Systems* 30, 202–215 (2014)
18. Nakano, K.: Simple memory machine models for gpus. In: 26th IEEE International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW). pp. 794–803 (2012)
19. NVIDIA Corp.: CUDA C Best Practices Guide. Version 7.0 (March 2015)
20. Pagh, A., Pagh, R.: Uniform hashing in constant time and optimal space. *SIAM Journal on Computing* 38(1), 85–96 (2008)
21. Sen, S., Scherson, I.D., Shamir, A.: Shear Sort: A True Two-Dimensional Sorting Techniques for VLSI Networks. In: International Conference on Parallel Processing. pp. 903–908 (1986)
22. Sitchinava, N., Weichert, V.: Provably efficient GPU algorithms. CoRR abs/1306.5076 (2013), <http://arxiv.org/abs/1306.5076>