FIFO-based Hardware Sorters for High Bandwidth Memory

Koji Nakano, Yasuaki Ito
Department of Information Engineering
Hiroshima University
Kagamiyama 1-4-1, Higashi Hiroshima, 739-8527 Japan

Jacir L. Bordim
Department of Computer Science
University of Brasilia
70910-900, Brasilia - DF - Brazil

***Abstract*—The main contribution of this paper is to show efficient FIFO-based hardware sorters that sort n elements with w bits each stored in a high bandwidth memory with modest access latency. We assume that each address of the high bandwidth memory can store p elements of w bits each, which can be read or written at the same time. Thus, n elements to be sorted are stored in $\frac{n}{p}$ addresses. The access latency l of the high bandwidth memory is assumed to take l clock cycles to access p elements in a specified address. Furthermore, burst mode is supported and k (≥ 1) consecutive addresses can be accessed in $k+l-1$ clock cycles in a pipeline fashion. However, if k addresses are not consecutive, then memory access to each of them takes l clock cycles and so kl clock cycles are necessary to access all of them. Thus, all n elements arranged $\frac{n}{p}$ addresses can be duplicated in $2(\frac{n}{p}+l-1)$ clock cycles in burst mode. These assumptions capture fundamental characteristic of latest high bandwidth memories such as HBM2 and GDDR5X. We present two types of hardware sorters that sort $n=rc$ elements stored in an $r \times c$ matrix of the high bandwidth memory. We first develop Three-Pass-Sort and Four-Pass-Sort that sort an $r \times c$ matrix by reading from and witting in it three times and four times, respectively. We implement these two algorithms using FIFO-based mergers that can be configured as pairwise mode and sliding mode. Our hardware sorter based on Three-Pass-Sort runs in $6\frac{n}{p}+3\frac{c^2}{p^2}l+O(\frac{c}{p}(l+\log r)+r)$ clock cycles using a circuit of size $O(rwp)$ provided that $r \geq c^2$. Also, our hardware sorter based on Four-Pass-Sort runs in $8\frac{n}{p}+2c^2l+O(cl+\log r+p)$ clock cycles using a circuit of size $O(rw)$.**

***Keywords*—parallel sorting algorithms, hardware sorter, high bandwidth memory, burst memory access, big data analysis.**

I. Introduction

It is no doubt that sorting is one of the most important tasks in computer engineering, such as database operations, image processing, big data analysis, statistical methodology and so on. Hence, many sequential and parallel sorting algorithms have been studied in the past [1], [2]. We focus on comparison-based sorting, in which data movement for sorting is determined only by means of comparison operations for pairs of two elements. It is well-known that sequential sorting algorithms such as heap sort and merge sort can sort n elements in $O(n \log n)$ time [3], and they are optimal because at least $\Omega(n \log n)$ comparisons are necessary to sort n elements. Also, a work-time optimal parallel sorting algorithm running in $O(\log n)$ time using n processors on the PRAM has been presented [4]. Furthermore, sorting n elements can be done by a $O((\log n)^2)$-depth circuit [5] and a $O(\log n)$-depth circuit [6].

The main contribution of this paper is to introduce *a Hardware Sorter for High Bandwidth Memory (HS-HBM) model*, and show very efficient hardware algorithms on it. Figure 1 illustrates the HS-HBM model. We assume that n elements with w bits are stored in the high bandwidth memory with modest access latency. We assume that each address of the high bandwidth memory can store p elements of w bits each, which can be read or write at the same time. Thus, the n elements to be sorted are stored in $\frac{n}{p}$ addresses. We also assume access latency l of the high bandwidth memory such that it takes l clock cycles to access p elements in a specified address. Further, burst mode is supported and k (≥ 1) consecutive addresses can be accessed in $k+l-1$ clock cycles in a pipeline fashion. However, if k addresses are not consecutive, then memory access to each of them takes l clock cycles and so kl clock cycles are necessary to access all of them. Figure 1 also shows the timing chart of consecutive access and stride access. Memory access to consecutive 8 addresses 0, 1, 2, ..., 7 can be completed in $8+l-1=10$ clock cycles. However, stride access to 8 addresses 0, 2, 4, ..., 14 takes $8l=24$ clock cycles. Thus, memory access operations should be performed on consecutive addresses to maximize the performance. A hardware sorter is implemented using logic circuits using *basic logic gates* such as AND, OR, NOT, XOR, etc, *flip-flops*, *registers*, and *internal memories*. For example, latest FPGAs have a number of Look-Up-Tables (LUTs) [7], [8] and block memories [9], [10] in which these logic circuits can be implemented. An internal memory is a dual-port memory to which read and write operations can be performed to different addresses at the same time. Also, we use FIFOs to which enqueue and dequeue operations are performed at the same time. Note that a FIFO can be implemented as a ring buffer using a dual-port block memory and registers to indicate the head and tail of the FIFO. In particular, latest FPGAs support hardware FIFOs using block memories [10]. We evaluate the *circuit size* of a hardware sorter by the sum of the total number of logic gates and the total number of bits of internal memories.

We define sorting problem for the HS-HBM as follows.
Input: n input elements to be sorted are stored in consecutive $\frac{n}{p}$ addresses in the high bandwidth memory.
Output: sorted elements are stored in consecutive $\frac{n}{p}$ addresses in the high bandwidth memory as follows.
Let $\pi : \{0,1,\ldots,\frac{n}{p}-1\} \times \{0,1,\ldots,p-1\} \rightarrow \{0,1,\ldots,n-1\}$ be a pre-determined one-to-one mapping. After sorting, each j-th element of address i ($0 \leq i \leq \frac{n}{p}-1$ and $0 \leq i \leq p-1$) must

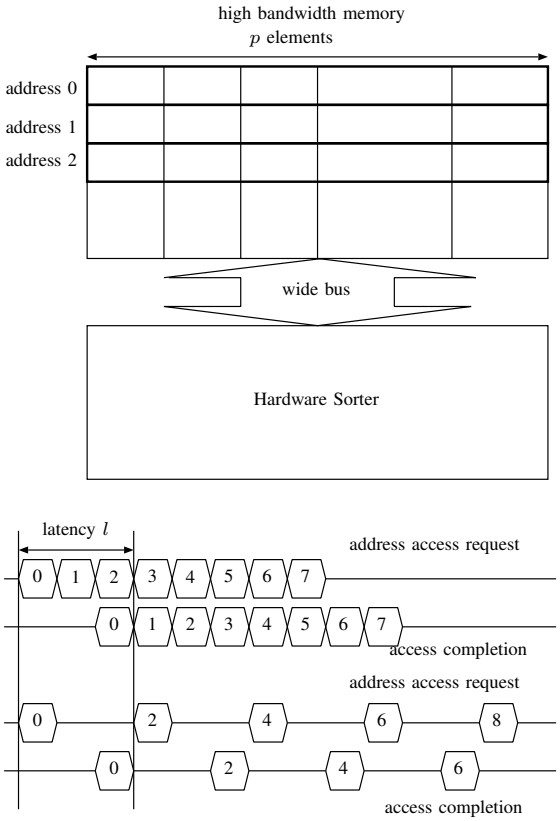

Fig. 1. A Hardware Sorter with high bandwidth memory (HS-HBM) and the timing chart of memory access with latency $l = 3$.

store $\pi(i, j)$-th largest element of the input. We say that the HS-HBM works for sorting correctly if the output is correctly sorted with respect to some fixed permutation π for any input matrix.

There are several works that present hardware sorters. For example, Marcelino *et al.* [11] implemented a FIFO-based merge sorter [12] and evaluated the performance on an FPGA. Koch and Torresen [13] presented FPGA implementations of various parallel sorting algorithms. Mueller *et al.* [14] presented implementations of sorting networks [15]. However, sorting networks are very costly and require a lot of comparators. Marcelino *et al.* [16] shows a simple architecture for parallel merge sort using one merge sorting unit. Since it repeats merging many times, latency is quite large. Matsumoto *et al.* [17] have presented a very efficient merge-based sorter, which minimizes the FIFO capacity used in FIFO-based merge sorter proposed in [12]. Harada *et al.* [18] have presented a timestamp sorter, which sorts almost sorted sequence using FIFO-based mergers. However, most of the above works support sorting of a one-port memory, in which each address stores an element. Hence, it is not possible to sort n elements in the high bandwidth memory faster than n clock cycles, which are necessary to read n elements. It is a challenging work to achieve a speedup factor of p using bandwidth of pw bits to sort n w-bit elements.

We present two sorting algorithms, Three-Pass-Sort and Four-Pass-Sort that sort $n = rc$ elements with w bits arranged in an $r \times c$ matrix. These algorithms are modifications of Leighton's Columnsort [19], [20] for efficient implementations in the HS-HBM model. We first show that the Three-Pass-Sort can be implemented in the HS-HBM model with bandwidth pw to run in $6\frac{n}{p} + \frac{n}{p^2}l + O(\frac{c^2}{p^2}l + \frac{c}{p}\log r + r)$ clock cycles [1]. Quite surprisingly, it reads and writes n elements to/from the high bandwidth memory only three times each. This hardware algorithm uses the following circuits:

- p FIFO-based hardware r-sorters with pairwise mode and sliding mode;
- an internal memory of size $r \times pw$; and
- a transposer of size $p \times p$, which is used to transpose $p \times p$ matrices.

An r-sorter in pairwise mode sorts sequences of r elements each in a pipeline fashion. When in sliding mode, it sorts an almost sorted sequence, called $\frac{r}{2}$-sorted sequence, in which a pair of two elements can be in wrong order if the difference of their positions is less than $\frac{r}{2}$. Finally, we implement the Four-Pass-Sort in the HS-HBM, which reduces the circuit size to $O(rw)$. Our implementation performs 4 read and 4 write operations per element and runs in $8\frac{n}{p}+2c^2l+O(cl+\log r+p)$ clock cycles.

This paper is organized as follows. In Section II, we present the Three-Pass-Sort and the Four-Pass-Sort, which sort an $r \times c$ matrix in three passes and four passes, respectively. We then go on to show three circuit elements: an r-sorter, a transposer, and an internal memory, which are used to implement the Three-Pass-Sort and the Four-Pass-Sort in the HS-HBM. Section IV implements the Three-Pass-Sort and the Four-Pass-Sort in the HS-HBM model. In Section V, we concludes our work.

II. Three-Pass-Sort and Four-Pass-Sort

The main purpose of this section is to present the Three-Pass-Sort and the Four-Pass-Sort, which sorts an $r \times c$ matrix with $n = rc$ elements such that $r \geq c^2$ in row-major order. For simplicity, we assume that r and c are powers of two.

A. Three-Pass-Sort

Three-Pass-Sort has three passes, each of which reads all elements stored in the input matrix, performs some operations on them, and writes the output matrix. The output matrix in each pass will be the input matrix of the following pass and the sorted results will be stored in the output matrix of Pass 3. Figure 2 shows how a 16×4 matrix with 0-1 elements is sorted. Note that from *the* 0-1 *principle for sorting* [21], comparison-based sorting can sort any input numbers if it can sort any 0-1 elements. So, it is sufficient to analyze how 0-1 elements are sorted for the purpose of understanding the correctness of the Three-Pass-Sort.

We say that a sequence $s_0, s_1, \ldots$ is *k-sorted* if $s_i \leq s_j$ $(i \leq j)$ holds for all i and j such that $j-i \geq k$. In other words, if $s_i > s_j$ then $j - i < k$ must be satisfied. For example, sequence 0, 1, 5, 3, 4, 6, 7 is 3-sorted. However, it is not 2-sorted, because $5 > 4$. In the context of 0-1 principle, sequence

[1] We use formulas such as $f+O(g)$ if $f \gg g$, for the purpose of clarifying dominant terms f and avoid cumbersome handling of small non-dominant terms g.

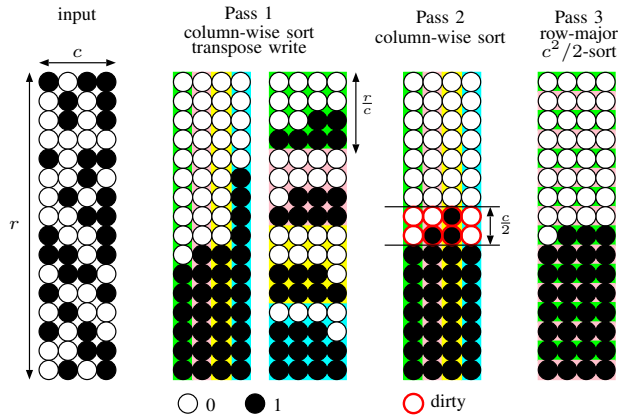

Fig. 2. Three-Pass-Sort for a 16×4 matrix.

0, 0, 1, 0, 0, 1, 1 is 3-sorted, but not 2-sorted. Intuitively, k-sorted sequence is "better sorted" if k is smaller.

The Three-Pass-Sort performs *column-wise sort*, which sorts r elements in each column independently, and *row-major* $\frac{c^2}{2}$*-sort*, which sorts $\frac{c^2}{2}$-sorted sequence of length n in row-major order as illustrated in Figure 2. The details are spelled out as follows.

Suppose that an $r \times c$ matrix is partitioned into c bands of $\frac{r}{c}$ consecutive rows each. Thus, each band has $c \times \frac{r}{c} = r$ elements. In Pass 1, column-wise sort and transpose write are performed as follows. All r elements in each column i ($0 \leq i \leq \frac{c}{2} - 1$) are sorted and written in band i in row-major order. Similarly, each column i ($\frac{c}{2} \leq i \leq c - 1$) is sorted and written in band i in inverse row-major order, in which elements are placed from right to left in each row. From Figure 2, we can see that at most one row in each band may have both 0 and 1 elements in the resulting matrix. Pass 2 simply performs column-wise sort. We say that a row of the resulting matrix of Pass 2 is *dirty* if it has both 0 and 1 elements. We will show that after Pass 2, the resulting matrix has at most $\frac{c}{2}$ dirty rows. In each band i ($0 \leq i \leq \frac{c}{2} - 1$) obtained in Pass 1, a dirty row has 0s followed by 1s. Similarly, each band $i + \frac{c}{2}$ ($0 \leq i \leq \frac{c}{2} - 1$) has 1s followed by 0s in a dirty row. Hence, if column-wise sort is performed for two bands i and $i + \frac{c}{2}$ combined, at most one row is dirty. Thus, the resulting matrix obtained by column-wise sort in Pass 2 has at most $\frac{c}{2}$ dirty rows, and so it is $\frac{c^2}{2}$-sorted in row-major order. Pass 3 simply performs $\frac{c^2}{2}$-sort to complete sorting of the matrix in row-major order.

B. Parallel Three-Pass-Sort

Next, we present the Parallel Three-Pass-Sort, which performs the Three-Pass-Sort using multiple processing units working in parallel. We assume a parallel machine with p processing units and a shared memory to store the matrix. We also assume that p is a power of two such that $p \leq c$. Similarly to the PRAM (Parallel Random Access Machine) model [21], p processing units work synchronously and can read from and/or write to any address of the shared memory at the same time. Each processing unit can perform:

r-wise sort: r elements in a column are read one by one and then the sorted results are written one by one. This can be done for multiple columns in a pipeline fashion.

$\frac{c^2}{2}$-sort: an $\frac{c^2}{2}$-sorted sequence is read one by one. After reading $\frac{c^2}{2}$ elements, the sorted sequence is written, element by element, in a pipeline fashion.

Figure 3 shows the timing charts for the above sorters. For simplicity, we assume latency r and $\frac{c^2}{2}$ respectively, because a processor unit has to read elements in the latency time necessary to output the first element. We do not discuss the details of processing units in this section. The processing units will be implemented by circuits later in this paper.

We will show how p processing units complete the Three-Pass-Sort. In Pass 1, we assign $\frac{c}{p}$ columns to each processing unit and perform column-wise sorting. The sorted results of each band with $\frac{r}{c}$ consecutive rows in row-major order or inverse row-major order. Since each processing unit works for $\frac{c}{p}$ columns, it runs in $\frac{c}{p} \cdot r + r = \frac{n}{p} + r$ clock cycles. Pass 2 can be done in the same way in $\frac{n}{p} + r$ clock cycles. For Pass 3, we consider a sequence S of length n obtained by picking elements in the matrix in row-major order. Suppose that the resulting matrix of Pass 2 are arranged in a 1-dimensional array of length $n = rc$ by picking elements in row-major order as illustrated in Figure 4. We partition the 1-dimensional array into p segments $S_0, S_1, \ldots, S_{p-1}$ of $\frac{n}{p}$ elements each. Furthermore, let $S'_0, S'_1, \ldots, S'_{p-1}$ be segments such that

- each S'_i ($0 \leq i \leq p-2$) is the concatenation of S_i and the first $\frac{c^2}{2}$ elements in S_{i+1}; and
- $S'_{p-1} = S_{p-1}$.

Hence, each each S'_i ($0 \leq i \leq p-2$) has $\frac{n}{p} + \frac{c^2}{2}$ elements. In Pass 3, each S'_i ($0 \leq i \leq p-1$) is assigned processing unit i, which performs $\frac{c^2}{2}$-sort in parallel. Note that, when processing unit i ($0 \leq i \leq p-2$) operates on the first $\frac{c^2}{2}$ elements in S_{i+1}, the resulting elements obtained by $\frac{c^2}{2}$-sort by processing unit $i+1$ must be used. Since the 1-dimensional array stores $\frac{c^2}{2}$-sorted sequence, Pass 3 correctly sorts these elements in $\frac{n}{p} + c^2$ clock cycles. Thus, we have,

Lemma 1: Parallel Three-Pass-Sort sorts $n = rc$ elements in $3\frac{n}{p} + 2r + c^2$ clock cycles using p processing units if $r \geq c^2$.

C. Parallel Four-Pass-Sort

We will change Pass 3 in Parallel Three-Pass-Sort so that r-wise sort defined below is executed twice. Suppose that an $r \times c$ input matrix for Pass 3 is partitioned into $\frac{n}{r} = c$ bands $T_0, T_1, \ldots, T_{c-1}$ of r elements in $\frac{r}{c}$ rows each as illustrated in Figure 5. We perform r-wise sort in row-major order for the input matrix such that each T_i is sorted. For the resulting matrix of r-wise sort, imagine that the first $\frac{r}{2}$ elements in $\frac{r}{2c}$ rows and the last $\frac{r}{2}$ elements in $\frac{r}{2c}$ rows in row-major order are removed. We partition the remaining $(r - \frac{r}{c}) \times c$ matrix into $\frac{n}{r} - 1$ bands $T'_0, T'_1, \ldots, T'_{c-2}$ of $\frac{r}{c}$ rows each as shown in Figure 5. We perform r-wise sort in row-major order for the input matrix such that each T'_i is sorted. The input matrix of Pass 3 has at most $\frac{r}{2c}$ ($\geq \frac{c}{2}$) dirty rows if $r \geq c^2$. Thus, dirty rows are in at most two consecutive T_is. After sorting of each

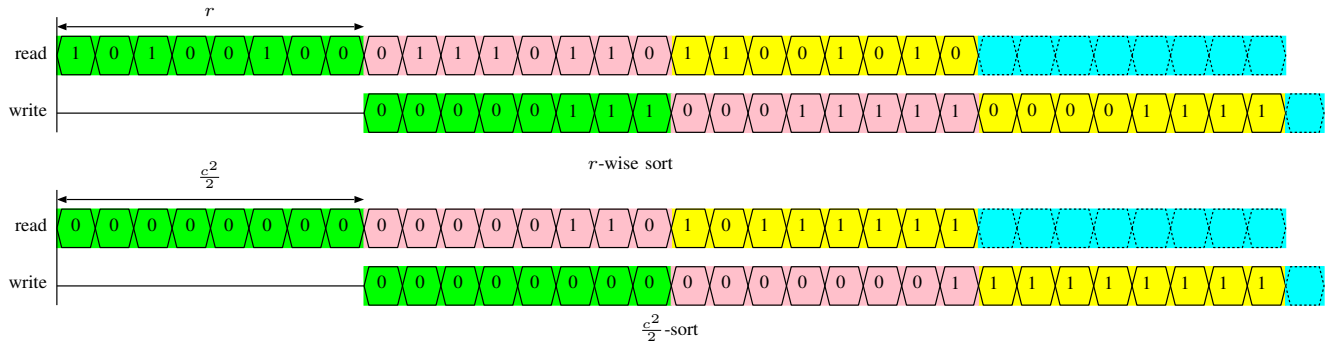

Fig. 3. Timing charts of r-wise sort and $\frac{c^2}{2}$-sort by a processing unit.

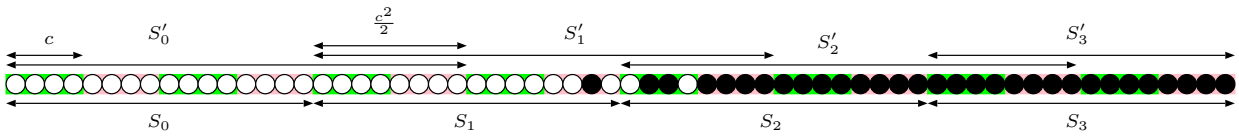

Fig. 4. Segments S'_0, S'_1, S'_2, and S'_3 for a 16×4 matrix.

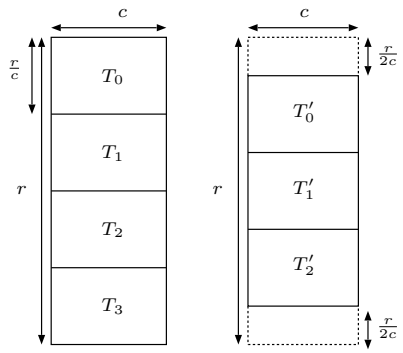

Fig. 5. T_i and T'_i of Pass 3 in Four-Pass-Sort.

T_i by r-wise sort, at most one T'_i has dirty rows. Hence, by sorting each T'_i by r-wise sort, sorting can be done correctly.

We assume that a processing unit can perform r-wise sort in row-major order. Since we use p processing units, we assign $\frac{c}{p}$ bands to each processing units and perform r-wise sort. Each processor can perform r-wise sort for $\frac{c}{p}$ bands in $\frac{n}{p} + r$ clock cycles. Since r-wise sort is executed twice, Pass 3 can be done in $2\frac{n}{p} + 2r$ clock cycles. Since Passes 1, 2, and 3 takes $\frac{n}{p} + r$, $\frac{n}{p} + r$, and $2\frac{n}{p} + 2r$, respectively, we have,

Lemma 2: Parallel Four-Pass-Sort sorts $n = rc$ elements in $4\frac{n}{p} + 4r$ clock cycles using p processing units if $r \geq c^2$.

III. Circuit elements for Three-Pass-Sort and Four-Pass-Sort

This section is devoted to show the following circuits: a d-sorter, a transposer, and an internal memory for implementing the Three-Pass-Sort and the Four-Pass-Sort to run in the HS-HBM. A d-sorter is a hybrid of the hardware sorter presented in [17] and the timestamp sorter shown in [18].

A. d-merger with pairwise mode and sliding mode

First, we show *a d-merger*, which has two FIFOs A and B of sizes $d + 1$ and d, respectively, and a comparator as illustrated in Figure 6. A FIFO is a first-in first-out memory, to which enqueue and dequeue operations are performed. Let $S_0, S_1, \ldots,$ be an input sequence such that each S_i $(i \geq 0)$ is a sorted sequence with d elements. A d-merger receives an element of an input sequence in every clock cycle and outputs an element in every clock cycle after some latency time. It has two modes, *pairwise mode* and *sliding mode*. In sliding mode, an input sequence must be d-sorted. In addition, to simplify treatment of two elements with the same value, we assume that, if the values of two elements are the same, then an element in smaller indexed S is smaller than the other. Hence, no two elements in different sequences are the same under this assumption and $S_i < S_{i+2}$ (i.e. all elemnts in S_i are small than those in S_{i+2}) always holds if the input is d-sorted.

In both modes, an element of an input sequence is enqueued in one of FIFOs in every clock cycle such that it is enqueued in FIFO A if it is in S_{2i} $(i \geq 0)$, and in FIFO B if it is in S_{2i+1}. In pairwise mode, each pair of two sorted sequences S_{2i} and S_{2i+1} is merged into one sorted sequence and is output in every clock cycle after $d + 1$ clock cycles. In sliding mode, all elements of $S_0, S_1, \ldots,$ are merged into one sorted sequence.

A d-merger in both modes works as follows. After all d elements in S_0 and the first element S_1 are enqueued in FIFOs A and B, respectively, dequeue operation starts. One of the FIFOs is selected and dequeued in every clock cycle, and so two FIFOs always have $d + 1$ elements totally. In the figure, FIFOs always have 5 elements on and after 5 clock cycles. A FIFO for which dequeue operation is performed is selected as follows. In sliding mode, the heads of two FIFOs are compared and a FIFO with smaller head is dequeued and output. In pairwise mode, the head of FIFO A is an element in S_{2i} for some i and that of FIFO B is an element in S_{2i+1}

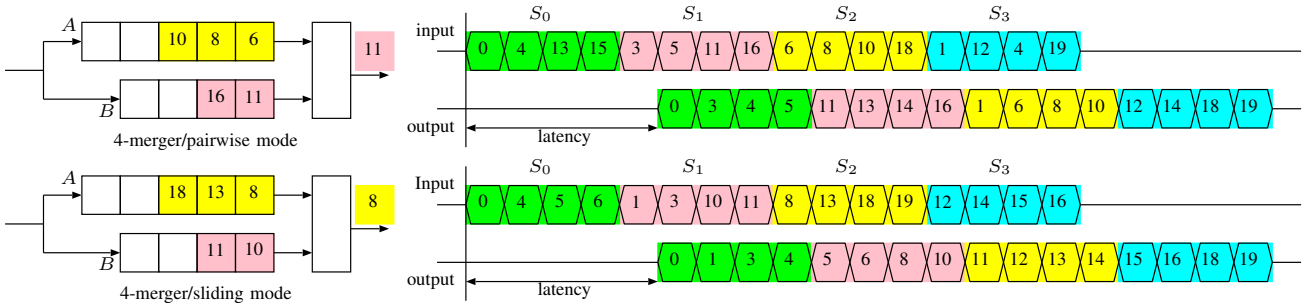

Fig. 6. Timing chart of 4-mergers with pairwise and sliding modes.

then a FIFO with smaller head is dequeued. If the head of A is an element in S_{2i+2} and that of B is an element in S_{2i+1} then FIFO B is dequeued, because no element in S_{2i+2} should be output before an element in S_{2i+1} is output. For example, in 4-merger with pairwise mode of Figure 6, element 11 in FIFO B is output because it is in S_2 and element 11 in FIFO B is an element in S_1. It should be clear that d-merger with pairwise mode can merge each pair of S_{2i} and S_{2i+1} of d elements each into one sorted sequence of $2d$ elements.

We show the correctness of a d-merger with each mode. It should be clear that a d-merger with pairwise mode correctly outputs a pairwise sorted sequence, because a smaller element in the heads is dequeued from the two sorted sequences. Note that FIFO A may store $d+1$ elements and so the size of FIFO A must be $d+1$. For example, all elements in S_0 is larger than S_1 then FIFO A is empty after the last element in S_1 is dequeued from FIFO B. If this is the case, FIFO A stores all d elements in S_0 and the first element in S_2 is enqueued. Thus, FIFO A stores $d+1$ elements. On the other hand, even if all d elements in S_1 is in FIFO B, the first element of S_2 is enqueued to FIFO A. Hence, FIFO B stores at most d elements. We can confirm the correctness of a d-merger with sliding modes as follows. Since each S_i is sorted and $S_i < S_{i+2}$ holds for all i, elements in the concatenated sequence of even-indexed sequences $S_0, S_2, S_4, \ldots$ enqueued in FIFO A are sorted. Similarly, elements in $S_1, S_3, S_5, \ldots$ enqueued in FIFO B are also sorted. Thus, they can be merged into one sorted sequence by dequeuing and outputting smaller one of heads of two FIFOs. We will show that FIFO B never stores more than d elements. The proof for FIFO A can be done in the same way. Recall that FIFOs A and B store $d+1$ elements totally in and after $d+1$ clock cycles. Hence, FIFO B stores $d+1$ elements if

- FIFOs A and B stores 1 and d elements, respectively, and
- dequeue and enqueue operation are performed for FIFOs A and B, respectively.

If the element in FIFO A is not the last element of S_{2i} for some i (≥ 1), then enqueue operation is performed for FIFO A to store the next element of S_{2i}. Thus, the element in FIFO A must be the last element of S_{2i}. Since dequeue operation is performed for FIFO A, the head of FIFO A is smaller than the head of FIFO B. Hence, the head of FIFO B is not an element from S_{2i-1} or earlier because an input sequence is d-sorted. Thus, all elements in FIFO B are from S_{2i+1} or later. Since no element in S_{2i+2} has been enqueued yet, all d elements in FIFO B are from S_{2i+1}, and S_{2i+2} must be enqueued in FIFO A, a contradiction. Therefore, FIFO B stores at most d elements. Consequently, we have,

Lemma 3: A d-merger with pairwise mode merges pairs of two sorted sequences with d elements each and outputs the sorted sequence with $2d$ elements in latency $d+1$. In sliding mode, a d-merger sorts a d-sorted sequence and outputs the sorted sequence in latency $d+1$.

B. *d-sorter with pairwise mode and sliding mode*

For simplicity, let d be a power of two. A d-sorter is a cascade of $\log_2 d$ mergers, 1-merger, 2-merger, 4-merger, ..., $\frac{d}{2}$-merger as illustrated in Figure 7. A d-sorter also has two modes, *pairwise mode* and *sliding mode*. In a d-sorter with pairwise mode, all mergers in it are pairwise mode. On the other hand, $\frac{d}{2}$-merger is sliding mode and the other mergers are pairwise mode, in a d-sorter with sliding mode. In other words, a d-sorter with sliding mode is $\frac{d}{2}$-sorter with pairwise mode followed by $\frac{d}{2}$-merger with sliding mode. Since the latency of i-merger is $i+1$, the latency of d-sorter is $(1+1)+(2+1)+(4+1)+\cdots+(\frac{d}{2}+1) = d+\log_2 d$. Figure 7 shows a 8-sorter with pairwise mode and sliding mode. The 8-sorter is pairwise mode and sliding mode if the 4-merger is pairwise mode and sliding mode, respectively.

A d-sorter receives an input sequence in every clock cycle. Let $S_0, S_1, \ldots$ be partitions of the input sequence, each of which has d elements. Clearly, d-sorter with pairwise mode sorts all d elements in each S_j and outputs them in turn, because pairwise merge is performed for two sequences with j element in i-merger ($i = 1, 2, 4, \ldots, d$). Figure 7 also shows a timing chart of 8-sorter with pairwise mode. We can see that the latency is $(1+1)+(2+1)+(4+1) = 10$ and S_0 and S_1 with 8 elements each are sorted.

Next, we will show that d-sorter with sliding mode outputs the sorted sequence of an input $\frac{d}{2}$-sorted sequence. It is sufficient to show that $\frac{d}{2}$-merger with pairwise mode in d-sorter outputs $\frac{d}{2}$-sorted sequence, because following $\frac{d}{2}$-merger with sliding mode correctly sorts it. Let $s_0, s_1, \ldots$ be an input sequence for $\frac{d}{2}$-sorter. By $\frac{d}{2}$-merger with pairwise mode, the input sequence is locally sorted. We will show that the output sequence of $\frac{d}{2}$-merger with pairwise mode is still $\frac{d}{2}$-sorted. We say that an interval $[i, j]$ ($i < j$) of an input sequence is *maximal reverse* if

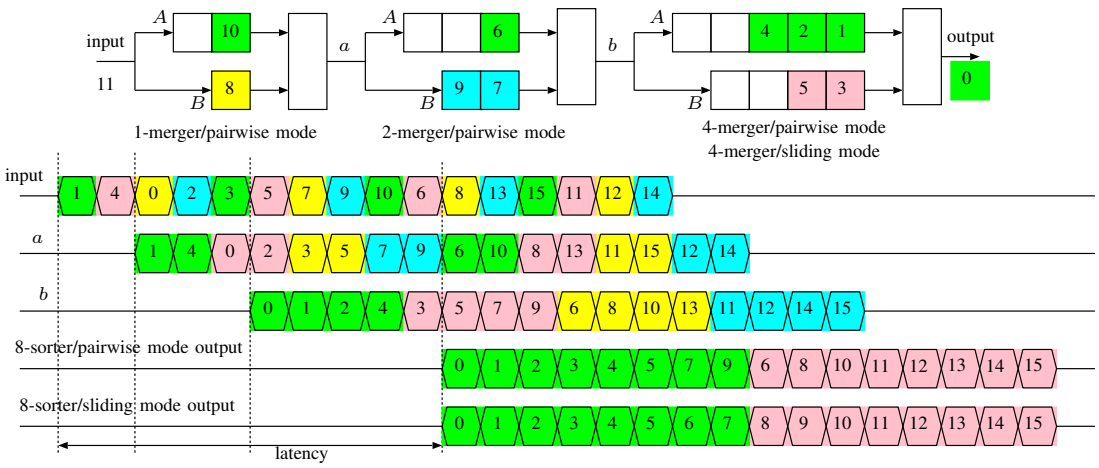

Fig. 7. Illustrating 8-sorter with pairwise mode and sliding mode and the timing chart.

(1) $s_i > s_j$;
(2) all s_k $(k < i)$ is smaller than s_j; and
(3) all s_k $(k > j)$ is larger than s_i.

Since the input sequence is $\frac{d}{2}$-sorted, $j - i \leq \frac{d}{2}$ holds from (1). Let $s'_{i'}$ and $s'_{j'}$ be two elements in the output sequence $s'_0, s'_1, \ldots$ of $\frac{d}{2}$-sorter with pairwise mode. From (2) and (3), both $i \leq i'$ and $j' \leq j$ are satisfied. Thus, $j' - i' \leq j - i \leq \frac{d}{2}$ holds. In general, for any interval $[u, v]$ $(u < v)$ satisfying $s_u > s_v$, there exists a maximal reverse interval $[i, j]$ such that $[u, v] \subseteq [i, j]$. Similarly, let $s'_{u'}$ and $s'_{v'}$ be elements s_u and s_v in the output sequence. For such maximal reverse interval $[i, j]$, both $i \leq u'$ and $v' \leq j$ hold. Thus, $v' - u' \leq j - i \leq \frac{d}{2}$ holds and $\frac{d}{2}$-sorter with pairwise mode outputs an $\frac{d}{2}$-sorted sequence, which is given to $\frac{d}{2}$-merger with sliding mode. Figure 7 also shows a timing chart for an 8-sorter with sliding mode. We can see that 4-sorted sequence is sorted correctly.

Lemma 4: A d-sorter with pairwise mode sorts input sequences with d elements each and outputs the resulting sequence in latency $d + \log_2 d$. When in sliding mode, d-sorter sorts a $\frac{d}{2}$-sorted sequence and outputs the sorted sequence in latency $d + \log_2 d$.

C. Transposer and internal memory

Since column-wise access to the high bandwidth memory has large overhead, we use *the transposer* and *the internal memory* for row-wise memory access in burst mode.

Similarly to the high bandwidth memory, the internal memory is a memory in which each address stores p elements with w bits. We evaluate the size of the internal memory by the total number of bits it can store. We can perform read or write p elements in a specified address at the same time. Since the internal memory is implemented on-chip, we assume that the memory access latency is only 1.

The transposer is a dual-port memory of size $p \times p$. We can specify a row address or a column address to access a row and a column, and p elements in a row/column can be read and written at the same time. We assume write-after-read mode, and so p elements in a specified row/column before writing are read. Using the transposer, we can transpose each of multiple matrices of size $p \times p$ in a pipeline fashion as follows. Matrices 0, 1, ... are written in p rows and p columns of the transposer alternately, that is, at clock cycle t $(t \geq 0)$, row $t \bmod p$ of matrix $\lfloor \frac{t}{p} \rfloor$ is written in row $t \bmod p$ of the transposer if $\lfloor \frac{t}{p} \rfloor$ is even and, in column $t \bmod p$ if odd. After p clock cycles from the beginning, the transposer is also read in row-wise and in column-wise alternately, that is, the same row or the same column is read before writing a row or a column at clock cycle $t + p$ $(t \geq 0)$. A row of the transposer is read at clock cycle $t + p$, column $t \bmod p$ of matrix $\lfloor \frac{t}{p} \rfloor$. We can confirm this fact from Figure 8, which shows transposing of matrices of size 4×4. In the figure, we assume that integers from 0 to 15, from 16 to 31, and from 32 to 47 are stored in matrices in 0, 1, and 2 in row-major order. In 4 clock cycles, rows 0, 1, 2, and 3 of matrix 0 are written in rows 0, 1, 2, and 3 of the transposer. After that, those stored in matrix 1 are written in columns 0, 1, 2, and 3 of the transposer in 4 clock cycles. At the same time, columns 0, 1, 2, and 3 of the transposer are read, and read columns are equal to columns 0, 1, 2, and 3 of matrix 0. Similarly, those stored in matrix 2 are written in rows 0, 1, 2, and 3 of the transposer and the same rows are read at the same time. The read values are equal to columns 0, 1, 2, and 3 of matrix 1. By repeating the same operation, all columns of multiple matrix can be read.

IV. Hardware algorithms for the HS-HBM

We will show hardware algorithms for the HS-HBM. We assume that $n = rc$ elements of w bits are stored in the high bandwidth memory with bandwidth pw and latency l and hardware sorter performs Parallel Three-Pass-Sort on a $r \times c$ matrix of n elements. We first show a hardware algorithm for the Three-Pass-Sort-1 on the HS-HBM. We then go on to show for the Three-Pass-Sort-2, which reduces the latency overhead.

A. Three-Pass-Sort-1

Three-Pass-Sort-1 uses following three types of circuits:

- p r-sorters configured as pairwise mode and sliding mode;

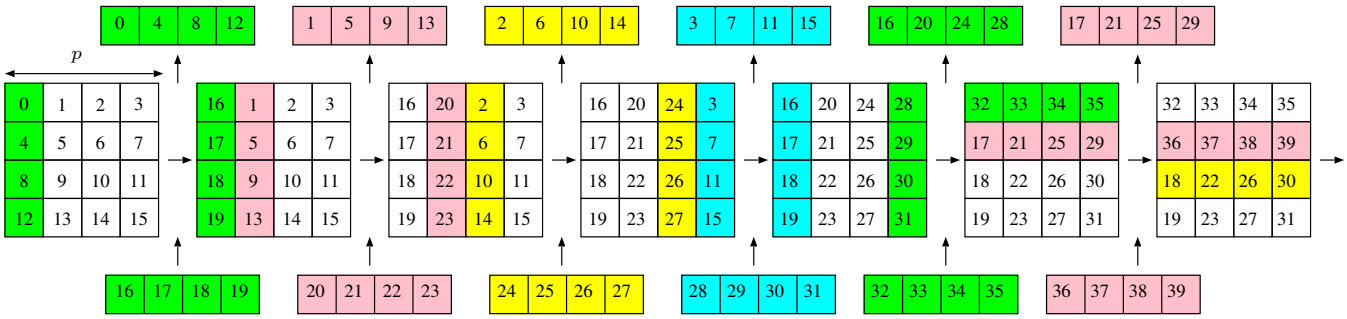

Fig. 8. Transposing multiple matrices of size 4×4 using the transposer.

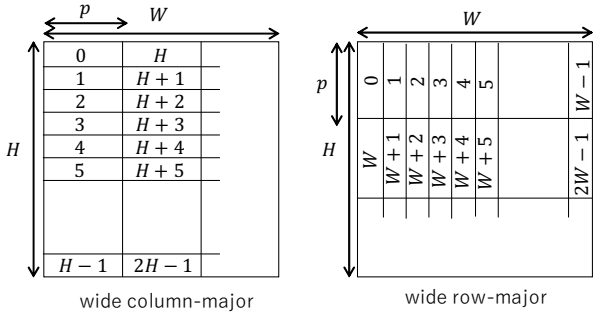

Fig. 9. Wide column-major and wide row-major arrangements of an $H \times W$ matrix in the high bandwidth memory; numbers represent addresses in the high bandwidth memory.

- a transposer of size $p \times p$; and
- an internal memory of size $p \times r$.

Note that $p < c < r < n$ and $r \geq c^2$ holds. Clearly, the circuit size is $O(rpw)$. We configure p r-sorters as pairwise mode to implement Passes 1 and 2, and as sliding mode for Pass 3. We obtain a $\frac{c^2}{2}$-sorted sequence after Pass 2 and an r-sorter with sliding mode can sort an $\frac{r}{2}$-sorted sequence. The transposer of size $p \times p$ is used in Passes 1 and 2.

We use two arrangements, *wide column-major arrangement* and *wide row-major arrangement* of matrix elements in the high bandwidth memory as illustrated in the Figure 9. The figure shows how an $H \times W$ matrix is arranged in a memory space of the high bandwidth memory. Each number corresponds to an address (or an address offset of the assigned memory space) in the high bandwidth memory. Hence, the leftmost p columns of the matrix in wide column-major arrangement can be accessed in burst mode in $H + l - 1$ clock cycles because they are arranged in consecutive addresses from 0 to $H - 1$. Thus, column-wise read operation performed in Pass 1 of the Parallel Three-Pass-Sort runs very efficiently if the matrix is in wide column-major arrangement. Similarly, the topmost p rows of the matrix in wide row-major arrangement can be accessed in only $W + l - 1$ time in burst mode. The wide row-major arrangement is used for efficient implementation of Pass 3 of the Parallel Three-Pass-Sort, which performs read operation in row-major order.

Figure 10 illustrates how the Three-Pass-Sort-1 works. Each Pass has an input matrix and an output matrix of size $r \times c$ such that the output matrix of Pass i ($i = 1, 2$) is the input matrix of Pass $i + 1$ and the sorted result is obtained as the output matrix of Pass 3.

The details of Passes 1, 2, and 3 are spelled out as follows. Let $U_0, U_1, \ldots, U_{\frac{c}{p}-1}$ be *strips* of $r \times p$ elements in the input matrix of Pass 1 such that each U_i ($0 \leq i \leq \frac{c}{p} - 1$) has p columns from column ip to $ip + p - 1$. Also, let $U''_0, U''_1, \ldots, U''_{\frac{c}{p}-1}$ be *bands* of size $\frac{pr}{c} \times c$ in the output matrix of Pass 1 such that each U''_i ($0 \leq i \leq \frac{c}{p}$) has $\frac{pr}{c}$ rows from $i\frac{pr}{c}$ to $(i+1)\frac{pr}{c} - 1$. Clearly, each U_i and each U''_i has rp elements. Pass 1 reads each U_i, operates on it, and write the results in U''_i. We assume that both input and output matrices of Pass 1 are wide column-major arrangement in the high bandwidth memory. The reader should refer to Figure 10 illustrating elements of U_0 and U''_0 to see their locations in the matrices. Also, Figure 11 illustrates how each U_i is processed. First, U_0 are read in row-wise from the top to the bottom, and send to p r-sorters with pairwise mode, in which every column of U_0 is sorted. Let U'_i ($0 \leq i \leq \frac{c}{p} - 1$) denote the sorted result of U_i by p r-sorters. Next, U'_0 are written in the internal memory through the transposer, and at the same time U_1 is read and sent to the p r-sorters. After that, U'_0 of the internal memory is read and written in U''_0 of the output matrix. The same operation is repeated for the remaining U_is in a pipeline fashion as shown in Figure 11. Note that, transpose write is a bit different from the original Three-Pass-Sort. This implementation performs *block-wise transpose write*, which is transpose in $p \times p$ block-wise as shown in Figure 11. The block-wise transpose write is essentially the same as the transpose write in Figure 2, because the column-destination of each element is the same and Pass 2 performs column-wise sort. Note that reading and writing operations for the high bandwidth memory are performed alternately as shown in Fig 11, because the bus connecting the high bandwidth memory is unidirectional. To maximize writing performance in burst mode, writing operation from the internal memory to the high bandwidth memory are performed in column-wise. Each U''_i are written in $\frac{pr}{c}$ consecutive addresses $\frac{c}{p}$ times, it takes at most $(\frac{pr}{c} + l - 1) \cdot \frac{c}{p} = r + \frac{c}{p}(l-1)$ clock cycles to write each U''_i in the output matrix. On the other hand, reading of each U_i can be done in burst mode, which takes $r + l - 1$ clock cycles. Also, the latency overheads of "$\log_2 r$" and "p" are imposed for p r-sorters and the transposer, respectively. Thus, Pass 1 takes no more than $(r + l - 1 + r + \frac{c}{p}(l-1) + \log_2 + p) \cdot \frac{c}{p} = 2\frac{n}{p} + \frac{c^2}{p^2}l + O(\frac{c}{p}(l + \log_2 r) + c)$ time. Note that, since $\frac{c}{p}l \ll \frac{c^2}{p^2}l$,

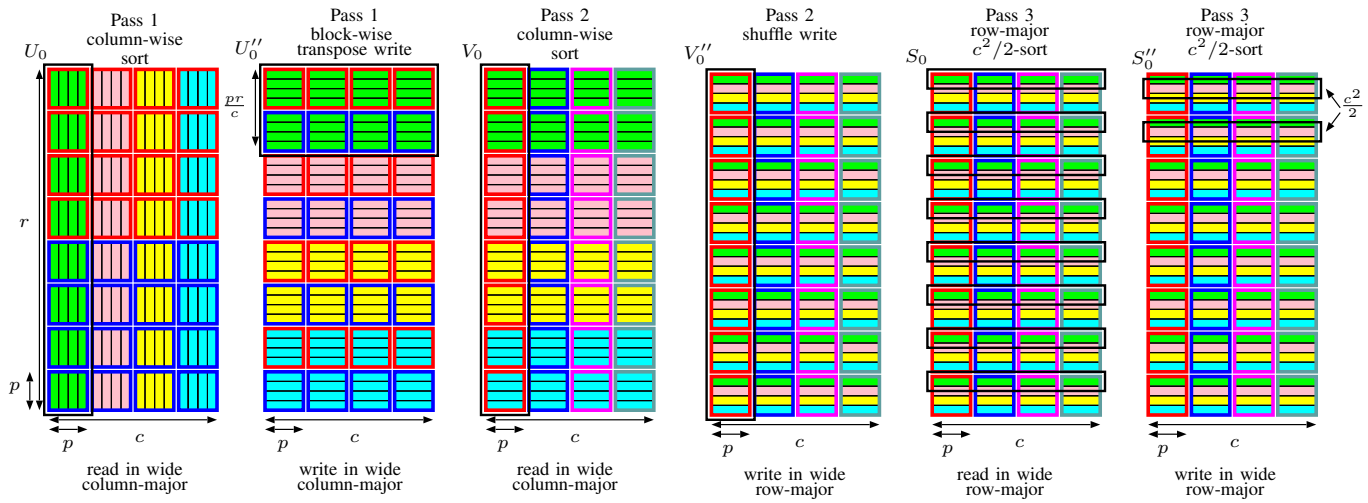

Fig. 10. Parallel Sorting algorithm on the HS-HBM based on Three-Pass-Sort.

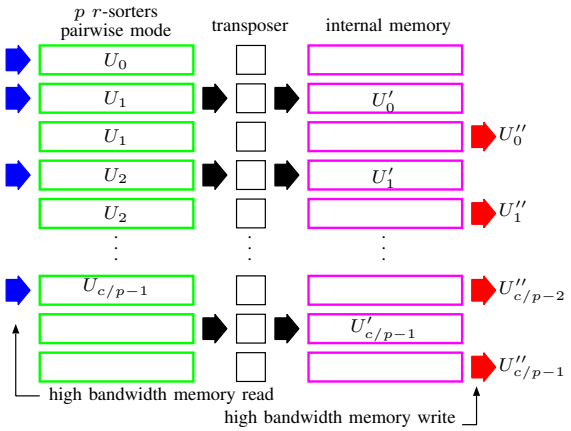

Fig. 11. Pipelined operation in Pass 1 of Three-Pass-Sort-1.

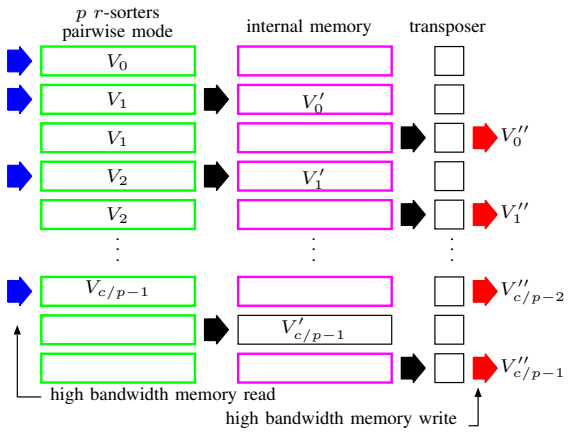

Fig. 12. Pipelined operation in Pass 2 of Three-Pass-Sort-1.

$\frac{c}{p}\log_2 r \ll \frac{n}{p}$, and $c \ll \frac{n}{p}$, we use big-O notation to non-dominant terms.

Next, we will show Pass 2 of Thee-Pass-Sort-1. Let V_i $(0 \le i \le \frac{c}{p})$ be a strip of p columns in the input matrix (i.e. the output matrix of Pass 1). Also, let V_i'' be a strip of p columns of the output matrix. Similarly to Pass 1, every V_i is processed in a pipeline fashion as shown in Figure 12 and sorted result V_i'' is written. Reading of each V_i is exactly the same as that of Pass 1. Writing of each V_i'' is performed for the output matrix in the wide-row major arrangement for burst read operation in Pass 3. Each V_i is sorted in column-wise and the sorted result V_i' is written in the internal memory. The internal memory storing V_i' is read in *shuffle order* and written in the output matrix through the transposer. The shuffle order of r rows is 0, $\frac{r}{p}$, $\frac{2r}{p}$, $\frac{3r}{p}$, ..., that is, the i-th row number is $(i \bmod p) \cdot \frac{r}{p} + \lfloor \frac{i}{p} \rfloor$. Figure 10 illustrates the *shuffle write*, in which V_i'' is written in the shuffle order. Reading each V_i from the high bandwidth memory is performed on consecutive rows in burst mode, which takes $r + l - 1$ clock cycles. However, each V_i'' in the output matrix with wide row-major arrangement are not consecutive. For example, the first $2p$ rows of V_0'' are arranged in p addresses from 0 to $p-1$ and then in p addresses from c to $c+p-1$, respectively. Since p consecutive addresses can be written in $p+l-1$ clock cycles, writing of each V_i'' in the high bandwidth memory takes $(p+l-1)\cdot\frac{r}{p} = r+\frac{r}{p}(l-1)$ clock cycles. Thus, including the latency overheads of p r-sorters and the transposer, Pass 2 takes at most $(r+l-1+r+\frac{r}{p}(l-1)+\log_2 r+p)\cdot\frac{c}{p} = 2\frac{n}{p}+\frac{n}{p^2}l+O(\frac{c}{p}(l+\log_2 r)+c)$ time.

Recall that Pass 3 of Parallel Three-Pass-Sort performs $\frac{c^2}{2}$-sort for each segment S_i' in Figure 4. Since we assumed $r \ge c^2$, an r-sorter can perform $\frac{c^2}{2}$-sort, because we assumed $r \ge c^2$. Each segment S_i $(0 \le i \le p-1)$ is arranged in rows i, $i+p$, $i+2p$, ... in the resulting matrix of Pass 2 by the shuffle write. Similarly, let S_i'' $(0 \le i \le p-1)$ be a segment in the output matrix of Pass 3 corresponding to S_i. Thus, by simple reading operation for addresses from 0 to $\frac{n}{p}$ of the input matrix in the wide row-major arrangement, p segments $S_0, S_1, \ldots, S_{p-1}$ can be read in parallel. They are sent to p r-sorters with sliding mode for $\frac{c^2}{2}$-sort. The sorted results are written in the output matrix in wide row-major arrangement.

After p segments $S_0, S_1, \ldots, S_{p-1}$ in the input matrix are read, the first $\frac{c^2}{2}$ elements in $S''_1, S''_2, \ldots, S''_{p-1}$ in the output matrix are read. In Figure 10, S_0 and the first $\frac{c^2}{2}$ elements of S''_1 are indicated. Since they are sent to an r-sorter with sliding mode, Pass 3 of Parallel Three-Pass-Sort is emulated correctly. We can perform this operation using p r-sorters and the internal memory in a pipeline fashion. Reading and writing are performed for consecutive addresses, because the input and the output matrices are in wide row-wise arrangement. Hence, Pass 3 can be done in pipeline fashion similarly to Pass 1 shown in Figure 11. More specifically, we partition the input matrix into $\frac{c}{p}$ groups of rp elements each and they are processed using p r-sorters and the internal memory. After all groups are processed, the first r elements in $S''_1, S''_2, \ldots, S''_{p-1}$ in the output matrix are processed in the same way. Thus, we can think that $\frac{c}{p}+1$ groups are processed by p r-sorters and the internal memory. Thus, Pass 3 takes at most $(r+l-1+r+l-1+\log_2 r)\cdot(\frac{c}{p}+1) = 2\frac{n}{p}+2\frac{c}{p}l+O(\frac{c}{p}\log_2 r)$ clock cycles. By combining three passes of the Three-Pass-Sort-1, we have,

Lemma 5: Three-Pass-Sort-1 runs $6\frac{n}{p} + \frac{n}{p^2}l + O(\frac{c^2}{p^2}l + \frac{c}{p}\log_2 r + r)$ using a circuit of size $O(rpw)$.

Note that, when $l \ll p$, we have $\frac{n}{p^2}l \ll 6\frac{n}{p}$ and the latency overhead can be hidden in this implementation.

B. Three-Pass-Sort-2

We will show that the latency overhead $\frac{n}{p^2}l$ in Three-Pass-Sort-1 can be decreased. This latency overhead is involved in the shuffle write of Pass 2. Three-Pass-Sort-2 additionally uses p $2r$-mergers with sliding mode. By combining them and p r-sorters with pairwise mode, we have p $2r$-sorters with sliding mode, each of which performs r-sort for r-sorted input sequence. Also, a larger internal memory of size $p\times 2r$ is used. Note that the size of the circuit is still $O(rpw)$.

The idea is to use an output matrix of size $2c \times \frac{r}{2}$ in Pass 2 and to perform the wide shuffle write as illustrated in Figure 13. We partition the output matrix of Pass 2 into $\frac{c}{p}$ strips $V''_0, V''_1, \ldots, V''_{\frac{c}{p}-1}$ of size $2c \times \frac{rp}{2c}$ each. Hence, the width of strips is increased from p to $\frac{rp}{2c}$, and the writing for $\frac{rp}{2c}$ consecutive addresses is performed $\frac{2c}{p}$ times. The wide shuffle write is performed so that elements in a single row of V''_i contains $\frac{r}{2c}$ consecutive rows in each V'_i. Thus, each row of V''_i has $\frac{r}{2c}\cdot\frac{c}{p} = \frac{r}{2p}$ consecutive rows of V'_i with $\frac{r}{2p}\cdot p = \frac{r}{2}$ elements. From $r \geq c^2$, dirty rows with at most $\frac{r}{2}$ ($\geq \frac{c^2}{2}$) elements in V'_i are in at most two rows of V''_i. Since two consecutive rows of V''_i have r elements, we can think that V''_i is an r-sorted sequence. Hence, Pass 3 uses p $2r$-sorters to sort each V''_i.

Let us evaluate the running clock cycles. The wide shuffle write of each strip V''_i takes $(\frac{rp}{2c}+l-1)\cdot\frac{2c}{p} = r+\frac{2c}{p}(l-1)$ and so Pass 2 takes $(r+l-1+r+\frac{2c}{p}(l-1)+\log_2 r+p)\cdot\frac{c}{p} = 2\frac{n}{p}+2\frac{c^2}{p^2}l+O(\frac{c}{p}\log r+c)$. Pass 3 performs r-sort in the same way. Thus, we have,

Theorem 6: Three-Pass-Sort-2 runs in $6\frac{n}{p}+3\frac{c^2}{p^2}l+O(\frac{c}{p}(l+\log r)+r)$ clock cycles using a circuit of size $O(rpw)$.

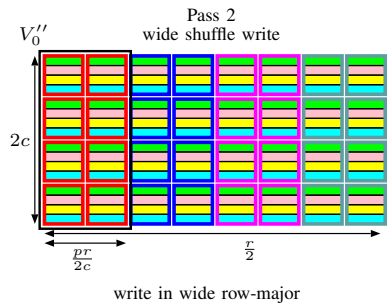

Fig. 13. The wide shuffle write performed in Pass 2.

When $l \ll \frac{rp}{c}$, we have $3\frac{c^2}{p^2} \ll 6\frac{n}{p}$, and the latency overhead can be hidden. From $r \geq c^2$, we can say that the latency overhead of Three-Pass-Sort-2 is smaller than that of Three-Pass-Sort-1.

C. Four-Pass-Sort

Three-Pass-Sort-1 and Three-Pass-Sort2 use circuits of size $O(rpw)$. We will show that Four-Pass-Sort uses a smaller circuit of size $O(rw)$ in compensation for a little increase of running clock cycles.

The idea is to perform r-wise sort using sorters, internal memories, and transposers by Parallel Four-Pass-Sort for $r'\times p$ matrices, where $r' = \frac{r}{p}$. Four-Pass-Sort repeatedly performs r-wise sort for $r\times c$ matrices in the high bandwidth memory to sort them by Parallel Four-Pass-Sort.

We first show how r-wise sort is performed using:

- four sets of p r'-sorters with pairwise mode;
- four internal memories of size $p\times r'$ each;
- three transposers of size $p\times p$.

Thus, the size of the circuit is $O(r'pw) = O(rw)$. These circuits are arranged as illustrated in Figure 14 to perform Parallel Four-Pass-Sort for an $r'\times p$ matrix. Using these circuit, we can perform Parallel Four-Pass-Sort in a pipeline fashion. For Pass 1, r elements in $\frac{r}{p}$ addresses of the high bandwidth memory are read and r'-wise sort is performed in parallel using p r'-sorters. The sorted results are written in the internal memory through the transposer in a similar way to Three-Pass-Sort-1. Pass 2 reads this internal memory and performs r'-wise sort in parallel using p r'-sorters and write them in another internal memory through the transposer similarly. Pass 3 reads this internal memory and performs r'-wise twice for simulating Parallel Four-Pass-Sort. For this purpose, two sets of p r'-sorters and the internal memory are used as illustrated in Figure 14. The sorted result in written in an internal memory. The transposer is used before writing if necessary. The circuit in Figure 14 works in a pipeline fashion to perform r-wise sort.

Next, we will show how r-wise sort is used to sort an $r\times c$ matrix stored in the high bandwidth memory. For this purpose, Parallel Four-Pass-Sort is used. We assume that the input matrices of Passes 1 and 2 are arranged in narrow column-major order and that of Pass 3 is in narrow row-major

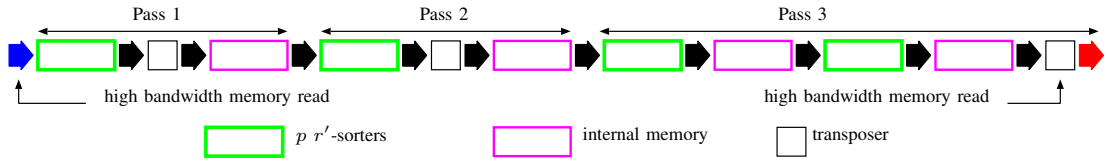

Fig. 14. Parallel Four-Pass-Sort using four sets of p r'-sorters, 3 $p \times p$-transposers, and four internal memory of size $r' \times p$.

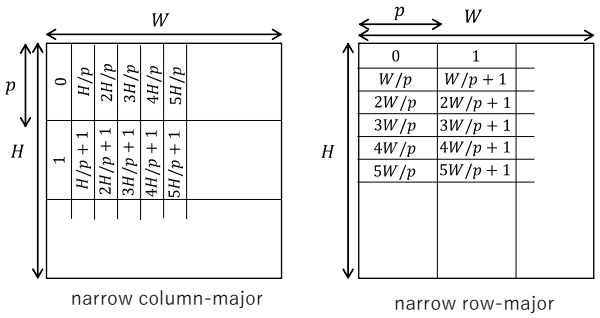

Fig. 15. Narrow column-major and narrow row-major arrangements of an $H \times W$ matrix in the high bandwidth memory; numbers represent addresses in the high bandwidth memory.

arrangement shown in Figure 15. In Pass 1, the $r \times c$ matrix in the high bandwidth memory are read in column major order. Each column is sorted by r-wise sort. Since narrow column-major order arrangement is used, r elements in a column can be read in $\frac{r}{p} + l - 1$ clock cycles. After that, the sorted result by r-wise sort of a column is written in a $\frac{r}{c} \times c$ band. Since a column of the band are stored in $\frac{r}{cp}$ consecutive addresses, this writing operation takes $(\frac{r}{cp} + l - 1) \cdot c = \frac{r}{p} + c(l-1)$ clock cycles. Since the latency of the circuit in Figure 14 is at most $4\log_2 r + 3p$, Pass 1 takes $(\frac{r}{p} + l - 1 + \frac{r}{p} + c(l-1)) \cdot c + 4\log_2 r + 3p = 2\frac{n}{p} + c(c+1)l + O(\log r + p)$ clock cycles. Pass 2 can be done in $2\frac{n}{p} + c(c+1)l + O(\log r + p)$ in the same way. Finally, Pass 3 performs r-wise sort in row-major order twice. Since the matrices are row-major order, both reading and writing operations for r elements are stored in the consecutive address of the high bandwidth memory. Thus, both reading and writing of r elements can be done in $\frac{r}{p} + l - 1$ clock cycles. Hence, the first r-wise sort for n elements can be done in at most $(\frac{r}{p} + l - 1 + \frac{r}{p} + l - 1) \cdot c + 4\log_2 r + 2p = 2\frac{n}{p} + 2cl + O(\log r + p)$ clock cycles. Similarly, the second r-wise sort performed for $n - r$ elements takes $(\frac{r}{p} + l - 1 + \frac{r}{p} + l - 1) \cdot (c-1) + 4\log_2 r + 2p = 2\frac{n-r}{p} + 2(c-1)l + O(\log r + p)$ clock cycles. By combining three passes, we have,

Theorem 7: Four-Pass-Sort runs at most $8\frac{n}{p} + 2c^2 l + O(cl + \log r + p)$ clock cycles using a circuit of size $O(rw)$.

V. Conclusion

In this paper, we have introduced the Hardware Sorter for High Bandwidth Memory (HS-HBM) model and presented FIFO-based hardware sorting algorithms. For $n = rc$ elements with w bits each stored in the high bandwidth memory with bandwidth pw, we present a hardware sorter of circuit size $O(rpw)$ that runs in $6\frac{n}{p} + 3\frac{c^2}{p^2} l + O(\frac{c}{p}(l + \log r) + r)$ clock cycles. We also present a hardware sorter of circuit size $O(rw)$ that runs in $8\frac{n}{p} + 2c^2 l + O(cl + \log r + p)$ clock cycles. Thus, the first and he second sorters are only three and four times slower than data duplication.

References

[1] D. E. Knuth, *The Art of Computer Programming. Vol.3: Sorting and Searching*. Addison-Wesley, 1973.

[2] S. G. Akl, *Parallel Sorting Algorithms*. Academic Press Inc., 1990.

[3] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. MIT Press, 1990.

[4] R. Cole, "Parallel merge sort," *SIAM J. Comput.*, vol. 17, no. 4, p. 770785, July 1987.

[5] K. E. Batcher, "Sorting networks and their applications," in *Proc. AFIPS Spring Joint Comput. Conf.*, vol. 32, 1968, pp. 307–314.

[6] M. Ajta, J. Komlós, and E. Szemerédi, "An 0(n log n) sorting network," in *Proceedings of the fifteenth annual ACM symposium on Theory of computing*, Dec. 1983, pp. 1–9.

[7] Xilinx Inc., *7 Series FPGAs Configurable Logic Block User Guide*, Nov. 2014.

[8] ——, "Ultrascale architecture configurable logic block: User guide," Feb 2017.

[9] ——, *7 Series FPGAs Memory Resources User Guide*, Nov. 2014.

[10] ——, "Ultrascale architecture memory resources," May 2017.

[11] R. Marcelino, H. Neto, and J. ao M.P. Cardoso, "Sorting units for FPGA-based embedded systems," *Distributed Embedded Systems: Design, Middleware and Resources*, vol. 271, pp. 11–22, 2008.

[12] S. Todd, "Algorithm and hardware for a merge sort using multiple processors," *IBM Journal of Research and Development*, vol. 22, no. 5, pp. 509–517, Sept. 1978.

[13] D. Koch and J. Torresen, "FPGASort: A high performance sorting architecture exploiting run-time reconfiguration on FPGAs for large problem sorting," in *Proc. of International Symposium on Field Programmable Gate Arrays*, 2011, pp. 45–54.

[14] R. Mueller, J. Teubner, and G. Alonso, "Sorting networks on FPGAs," *The International Journal on Very Large Data Bases*, vol. 21, no. 1, pp. 1–23, Feb. 2012.

[15] K. Batcher, "Sorting networks and their applications," in *in Proceedings of the AFIPS Spring Joint Computer Conference 32*, 1968, pp. 307–314.

[16] R. Marcelino, H. C. Neto, and J. ao M. P. Cardoso, "Unbalanced FIFO sorting for FPGA-based systems," in *Proc. of International Conference on Electronics, Circuits, and Systems*, Dec. 2009, pp. 431 – 434.

[17] N. Matsumoto, K. Nakano, and Y. Ito, "Optimal parallel hardware k-sorter and top k-sorter, with FPGA implementations," in *Proc. of International Symposium on Parallel and Distributed Computing*, June 2015, pp. 138–147.

[18] N. Harada, N. Matsumoto, K. Nakano, and Y. Ito, "A hardware sorter for almost sorted sequences, with FPGA implementations," in *Proc. of Fourth International Symposium on Computing and Networking (CANDAR)*, Nov. 2016, pp. 565–571.

[19] T. Leighton, "Tight bounds on the complexity of parallel sorting," *IEEE Transactions on Computers*, vol. C-34, no. 4, pp. 344 – 354, April 1985.

[20] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann, 1991.

[21] A. Gibbons and W. Rytter, *Efficient Parallel Algorithms*. Cambridge University Press, 1988.