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 \( k \geq 1 \) 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 writing 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 \( 6\frac{r}{p} + 3\frac{r^2}{p^2} + O(\frac{p}{r}(l + \log r + r)) \) clock cycles using a circuit of size \( O(rw^p) \) provided that \( r \geq c^2 \). Also, our hardware sorter based on Four-Pass-Sort runs in \( 8\frac{r}{p} + 2c^2l + O(cl + \log r + p) \) 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 \( k \geq 1 \) 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 \( 0, 1, 2, \ldots, 7 \) can be completed in \( 8 + l - 1 = 10 \) clock cycles. However, stride access to \( 8 \) addresses \( 0, 2, 4, \ldots, 14 \) 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 \( i \) \((0 \leq i \leq \frac{n}{p} - 1 \text{ and } 0 \leq j \leq p - 1) \) must...
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 \(\pi\) 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 \(6n + \frac{n}{p} l + O(\frac{l}{p} + \frac{c}{p} \log r + r)\) 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 \(\frac{c}{r}\)-sorted sequence, in which a pair of two elements can be in wrong order if the difference of their positions is less than \(\frac{c}{r}\). 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 \(\frac{8n}{p} + 2c^2 l + O(cl + \log r + p)\) 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 \(16 \times 4\) 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 \(s_0, s_1, \ldots, s_r\) 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 \(g\).
show that after Pass 2, the resulting matrix has at most \(c\) dirty rows, and so it is \(c\)-sorted because each 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 \(\frac{c}{p}\) consecutive rows in row-major order or inverse row-major order. Since each processing unit works for \(\frac{c}{p}\) columns, it runs in \(\frac{c}{p} \cdot r + r = \frac{c}{p} + r\) clock cycles. Pass 2 can be done in the same way in \(\frac{c}{p} + r\) 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 is 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 \(\frac{c}{p}\) elements each. Furthermore, let \(S_0^i, S_1^i, \ldots, S_{p-1}^i\) be segments such that

- each \(S_i^i\) (\(0 \leq i \leq p - 2\)) is the concatenation of \(S_i\) and the first \(\frac{c}{p}\) elements in \(S_{i+1}\); and
- \(S_{p-1}^i = S_0 - 1\).

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

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

\[\text{C. Parallel Four-Pass-Sort}\]

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 \(\frac{n}{p} = c\) bands \(T_0, T_1, \ldots, T_{c-1}\) of \(r\) elements in \(\frac{c}{p}\) 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 \(\frac{c}{p}\) elements in \(\frac{c}{p}\) rows and the last \(\frac{c}{p}\) elements in \(\frac{c}{p}\) rows in row-major order are removed. We partition the remaining \((r - \frac{c}{p}) \times c\) matrix into \(\frac{c}{p} - 1\) bands \(T_0', T_1', \ldots, T_{c-2}\) of \(\frac{c}{p}\) 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 \(\frac{c}{p} (\geq \frac{c}{p})\) dirty rows if \(r \geq c^2\). Thus, dirty rows are in at most two consecutive \(T_i\). 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 x 4 matrix.

Fig. 5. $T_i$ and $T_i'$ of Pass 3 in Four-Pass-Sort.

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 elements 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 $S_{2i+1}$

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 HSHBM. A $d$-sorter is a hybrid of the hardware sorter presented in [17] and the timestamp sorter shown in [18].
Thus, all elements in FIFO $A$ are from $S_{2i+2}$ 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 $i \geq 1$, 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-sor-ter with pairwise mode and sliding mode

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

A $d$-sor-ter 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, $d$-sor-ter with pairwise mode sorts all $d$ elements in each $S_i$ 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-sor-ter 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 $d$-sor-ter with sliding mode outputs the sorted sequence of an input $d$-sorted sequence. It is sufficient to show that $d$-merger with pairwise mode in $d$-sor-ter outputs $d$-sorted sequence, because following $d$-merger with sliding mode correctly sorts it. Let $s_0, s_1, \ldots$ be an input sequence for $d$-sor-ter. By $d$-merger with pairwise mode, the input sequence is locally sorted. We will show that the output sequence of $d$-merger with pairwise mode is still $d$-sorted. We say that an interval $[i, j]$ ($i < j$) of an input sequence is maximal reverse if
(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 \( s'_{i'} \) 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 \( \frac{d}{2} \)-sorter with sliding mode. We can see that \( \frac{d}{2} \)-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 \( 0, 1, \ldots \) are written in \( p \) rows and \( p \) columns of the transposer alternately, that is, at clock cycle \( t (t \geq 0) \), row \( t \mod p \) of matrix \( \left[ \frac{t}{p} \right] \) is written in row \( t \mod p \) of the transposer if \( \left[ \frac{t}{p} \right] \) is even and, in column \( t \mod p \) 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 \( t \mod p \) of matrix \( \left[ \frac{t}{p} \right] \). We can confirm this fact from Figure 8, which shows the transposing of matrices of size \( 4 \times 4 \). In this 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 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;
the matrix in wide column-major arrangement can be accessed.

and size sliding mode can sort an 

cobtain a implement Passes 1 and 2, and as sliding mode for Pass 3. We

size is O p < c < r < n

high bandwidth memory.
The matrix in the high bandwidth memory; numbers represent addresses in the

Fig. 9. Wide column-major and wide row-major arrangements of an 

H arranged in consecutive addresses from 0 to

in only

the matrix in wide row-major arrangement can be accessed

column-major arrangement. Similarly, the topmost

p is used in Passes 1 and 2.

Three-Pass-Sort runs very efficiently if the matrix is in wide

column-wise read operation performed in Pass 1 of the Parallel

Three-Pass-Sort, which performs read operation in

row-major order.

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×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 × 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 U0, U1, . . . , Ûp−1 be strips of r × p elements in the input

matrix of Pass 1 such that each U+i (0 ≤ i < p−1) has p columns from column ip to ip + p − 1. Also, let

U0′, U1′, . . . , Ûp−1′ be bands of size p W × c in the output matrix of

Pass 1 such that each Ûi′ (0 ≤ i < p−1) has W c rows from

ip W to (i + 1) W − 1. Clearly, each Ui and each Ûi′ has rp elements. Pass 1 reads each Ui, operates on it, and

write the results in Ûi′. 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 U0 and U0′ to see their locations in the

matrices. Also, Figure 11 illustrates how each Ui is processed.

First, U0 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 U0 is sorted. Let Ûi′ (0 ≤ i < p−1) denote the

sorted result of Ui by p r-sorters. Next, Û0′ are written in the

internal memory through the transposer, and at the same
time Ui is read and sent to the p r-sorters. After that, Û0′ of

the internal memory is read and written in Û0′ such that

the output matrix. The same operation is repeated for the remaining

Ui’s in a pipeline fashion as shown in Figure 11. Note that, transposer

write is a bit different from the original Three-Pass-Sort. This

implementation performs block-wise transpose write, which is

transpose in p × 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 Ûi′ are written in p W · p c consecutive addresses W p times, it takes at

most (p W + l − 1) · W p = r + W c (l − 1) clock cycles to write each

Ûi′ in the output matrix. On the other hand, reading of each Ûi

can be done in burst mode, which takes r + l + 1 clock cycles.

Also, the latency overheads of “log2 r” and “p” are imposed

for p r-sorters and the transposer, respectively. Thus, Pass 1

takes no more than (r + l + 1 + r + W c (l − 1) + log2 p) · W p =

2 p + W p + log2 r + O(W c (l + log2 r) + c) time. Note that, since

W p ≪ p, the

r × 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.


which 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 $V_0'''$ in the high bandwidth memory takes $(p + l - 1) \cdot \frac{p}{c} = r + \frac{p}{c} (l - 1)$ clock cycles. Thus, including the latency overheads of $p$ r-sorters and the transposer, Pass 2 takes at most $(r + l - 1 + r + \frac{p}{c} (l - 1) + \log_2 r + p) \cdot \frac{p}{c} = 2n + \frac{p^2}{c} + l + O\left(\frac{p^2}{c} \log_2 r + c\right)$ 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 $r \geq c^2$, an r-sorter can perform $\frac{c^2}{2}$-sort, because we assumed $r \geq c^2$. Each segment $S_i$ ($0 \leq i \leq p - 1$) is arranged in rows $i$, $i + p$, $i + 2p$, ... in the resulting matrix of Pass 2 by the shuffle write. Similarly, let $S_i'''$ ($0 \leq i \leq p - 1$) be a segment in the output matrix of Pass 3 corresponding to $S_i$. Thus, by simple reading operation for addresses from 0 to $\frac{p}{c}$ of the input matrix in the wide row-major arrangement, $p$ segments $S_0, S_1, ..., S_{p-1}$ 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 $\frac{c^2}{p}$ elements in $S''_1, S''_2, \ldots, S''_{p-1}$ in the output matrix are read. In Figure 10, $S_0$ and the first $\frac{c^2}{p}$ 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 output matrix of Pass 2 into $c$ sequences. Also, a larger internal memory of size $pwr$ can be used. The idea is to sort using an output matrix of size $2c \times \frac{c}{p}$ 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 $V'_0, V''_0, \ldots, V''_{p-1}$ of size $2c \times \frac{2p}{c}$ each. Hence, the width of strips is increased from $p$ to $\frac{2p}{c}$, and the writing for $\frac{2p}{c}$ consecutive addresses is performed in $\frac{c}{p}$ times. The wide shuffle write is performed so that elements in a single row of $V''_i$ contains $\frac{c}{p}$ consecutive rows of $V'_i$. Thus, each row of $V''_i$ has $2c \cdot \frac{c}{p} = \frac{2c^2}{p}$ consecutive rows of $V'_i$ with $\frac{2c}{p} \cdot p = \frac{2c^2}{p}$. From $r \geq c^2$, dirty rows with at most $\frac{c}{p}$ elements. From $r \geq c^2$, dirty rows with at most $\frac{c}{p}$ ($\geq \frac{c}{p}$) 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 \((\frac{2c^2}{p} + l - 1) \cdot \frac{c}{p} = r + \frac{2c^2}{p} (l - 1)\) and so Pass 2 takes \((r + l - 1 + r + \frac{2c^2}{p} (l - 1) + \log_2 r + p) \cdot \frac{c}{p} = 2\frac{2c^2}{p} + 2c^2 + O(\frac{c}{p} \log r + c)\). Pass 3 performs $r$-sort in the same way. Thus, we have,

\[ L_5: \text{Three-Pass-Sort-1 runs } 6\frac{2c^2}{p} + \frac{2c^2}{p} + O(\frac{2c^2}{p} + \frac{c}{p} \log r + r) \text{ using a circuit of size } O(rpwr). \]

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

\[ 2 \leq \frac{2c^2}{p} \leq \frac{2c^2}{p} \]

\[ \text{Lemma 5: Three-Pass-Sort-1 runs } 6\frac{2c^2}{p} + \frac{2c^2}{p} + O(\frac{2c^2}{p} + \frac{c}{p} \log r + r) \text{ using a circuit of size } O(rpwr). \]

\[ \text{Note that, when } l \leq p, \text{ we have } \frac{2c^2}{p} \leq \frac{2c^2}{p} \text{ and the latency overhead can be hidden in this implementation.} \]

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

The idea is to perform $r$-wise sort using sorters, internal memories, and transposers by Parallel Four-Pass-Sort for an $r^* \times p$ matrices,

\[ \text{C. Four-Pass-Sort} \]

Three-Pass-Sort-1 and Three-Pass-Sort-2 use circuits of size $O(rpwr)$. 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 a $r\times p$ matrix. Using this circuit, we can perform Parallel Four-Pass-Sort in a pipeline fashion. For Pass 1, $r$ elements in $p$ addresses of the high bandwidth memory are read and $r^*\times r$-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 order. The transposer is then used before writing if necessary.
cycles. We also present a hardware sorter of circuit size $O(r')$ with FIFO-based hardware sorting algorithms. For $n = rc$ elements with $w$ bits each stored in the high bandwidth memory 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 $\frac{r}{p} \times c$ band. Since a column of the band are stored in $r$ consecutive addresses, this writing operation takes $\left(\frac{r}{p} + l - 1\right) \cdot c = \frac{r}{p} + c(l - 1)$ clock cycles. Since the latency of the circuit in Figure 14 is at most $4 \log_2 r + 3p$, Pass 1 takes $(\frac{r}{p} + l - 1 + \frac{r}{p} + c(l - 1)) \cdot c + 4 \log_2 r + 3p = 2\frac{r}{p} + c(c+1)l + O(\log r + p)$ clock cycles. Pass 2 can be done in $2\frac{r}{p} + c(c+1)l + O(\log r + p)$ 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 done in $\frac{r}{c} + l - 1$ clock cycles. Hence, the first $r$-wise sort for $n$ elements can be done in $\frac{n}{p} \cdot l + l - 1$ clock cycles. Similarly, the second $r$-wise sort performed for $n - r$ elements takes $\left(\frac{n}{p} + l - 1 + \frac{r}{p} + l - 1\right) \cdot c + 4 \log_2 r + 2p = 2\frac{n}{p} + 2c(l + O(\log r + p))$ clock cycles. By combining three passes, we have,

**Theorem 7**: Four-Pass-Sort runs at most $8\frac{n}{p} + 2c^2l + O(cl + \log r + p)$ 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 $8\frac{n}{p} + 2c^2l + O(cl + \log r + p)$ clock cycles. Thus, the first and the second sorters are only three and four times slower than data duplication.

**References**


