Ultra-low Latency Continuous Block-parallel Stream Windowing using FPGA On-chip Memory

Justin S. J. Wong [*†], Runbin Shi [*], Maolin Wang [*] and Hayden K.-H. So [*]
[*] Department of Electrical and Electronic Engineering, The University of Hong Kong
[†] Conzeb Limited, Hong Kong
justin270f@gmail.com, {rbshi, mlwang, hso}@eee.hku.hk

***Abstract*—In this paper, we propose and demonstrate a real-time ultra-fast multi-data stream processing methodology on FPGA called “SWIM” (Stream Windowing on Interleaved Memory). The method exploits the flexible on-chip block memory fabric on existing FPGA architectures to achieve ultra-low-latency and fully pipelined continuous data flow while maintaining linear spatial locality of data for efficient data addressing and processing. The SWIM method is directly applicable to many practical applications such as real-time stencil computing, streaming image data processing, as well as closed loop-control systems that require ultra-low latency interleaved access and processing of high-speed sensor data. We demonstrate two practical cases on actual FPGA for generic 3-by-3 2-D convolution filter and image super-resolution method using pixel interleaving. Both memory usage and latency scales linearly with window height, or width of the 2-D input data set. The generic implementation of SWIM on FPGA showed impressive worst-case operation frequency of 410 MHz and uses $9.0\times$ and $5.6\times$ less Register and LUT resources respectively compared with a high-level synthesis solution.**

I. Introduction

Recent advancements in high-speed I/O capabilities have made modern FPGAs prime candidates to accelerate some of the most I/O demanding applications. With ample of high-speed serial connections and parallel configurable I/O pins, FPGAs promise to play a central role in processing data from high-speed ADCs, computer network and memory devices. However, the underlying clock speed of even the most advanced FPGAs remains relatively limited, and real-time processing of such high-speed data remains a daunting technical challenge.

It is particularly challenging in scenarios where high bandwidth data arrive as a continuous stream and requires cycle-accurate real-time stream processing to avoid data loss. A generic system on FPGA is shown in Fig. 1, where data stream enters via high-speed I/O with serial-to-parallel hardware (deserializer) available on most FPGAs [1], [2]. The deserializer collects data into blocks of words allowing parallel processing of data at a slower clock rate while maintaining the same overall throughput. An example of pixel data stream from a high-speed image sensor is shown in Fig. 2. If the input pixel clock is at 4 GHz, and the deserializer outputs 16 parallel pixels per block, then the FPGA can process each block at a slower clock rate of 250 MHz with the same overall pixel processing rate. There is, however, one potential complication when considering the underlying data structure of the source

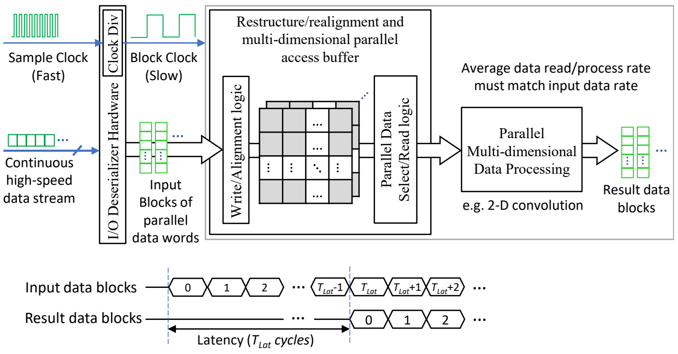

Sample Clock (Fast)
Clock Div
Block Clock (Slow)
I/O Deserializer Hardware
Continuous high-speed data stream
Input Blocks of parallel data words
Restructure/realignment and multi-dimensional parallel access buffer
Write/Alignment logic
Parallel Data Select/Read logic
Average data read/process rate must match input data rate
Parallel Multi-dimensional Data Processing
e.g. 2-D convolution
Result data blocks
Input data blocks
Result data blocks
Latency (T_{Lat} cycles)
0
1
2
T_{Lat}-1
T_{Lat}
T_{Lat}+1
T_{Lat}+2

Fig. 1. Example of a generic high-speed streaming data processing framework on FPGA with data realignment. The latency (T_{Lat}) of the result data blocks as shown by the timing diagram is defined by the number of clock cycles between the first input data block and the first result data block.

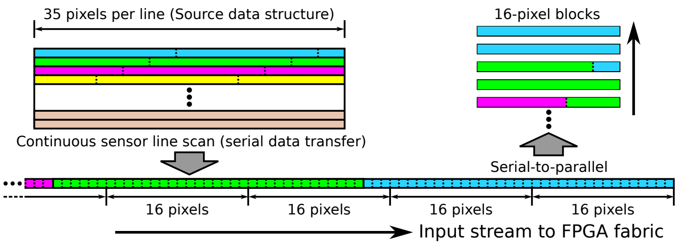

35 pixels per line (Source data structure)
Continuous sensor line scan (serial data transfer)
16-pixel blocks
Serial-to-parallel
16 pixels
16 pixels
16 pixels
16 pixels
Input stream to FPGA fabric

Fig. 2. Example serial pixel stream from image sensor line scan. Original 2-D data structure of the source image must be restored on FPGA from the pixel blocks for correct image processing.

data, such as the 2-D structure of image data in the example. If the width (line length) of the original input image is not a multiple of 16, such as 35 in Fig. 2, then some of the input data blocks read by the FPGA fabric will end up encapsulating data from two different lines of the original image. To realign the image lines within the continuous stream of data blocks at-speed, one must create a fully pipelined complex realignment network to restore the original image line structure without using additional clock cycles. Further complication arises when multiple lines from the original images must be used later for further processing, such as performing 2-D convolution operations. In this case, the realignment network must also be designed to accommodate multiple parallel read access that allows the processing operation to obtain data from relevant existing lines, as well as consuming data from new lines at the same rate as the input stream.

A naive solution to the above challenge is to rely on HLS tools to synthesize this complex realignment network using on-chip registers. However, doing so not only may consume a large portion of FPGA resources, but it may also create a highly complex routing problem for the implementation tools, greatly hindering the resulting timing performance of the design. Existing data processing systems tend to rely on on-chip frame buffers implemented by block memory as an intermediate storage where data realignment and processing may take place [3]. Similar buffering techniques have also been employed in processing of convolutional neural network [4] as well as in 2-D stencil computing applications [5], [6]. In addition, a generalized memory structure that can be customized to a particular memory access pattern for grid computing has also been proposed in [7]. However, all of the above techniques assume that the FPGA processing rate is much higher than the input frame rate, where the input data arrives as a stream of single data word, and that the additional latency introduced by the buffer is acceptable for the application. Furthermore, depending on the scope of data needed, the on-chip buffer size quickly increases and thus consumes a significant amount of on-chip memory, making such solutions difficult to scale. A systolic memory structure has also been proposed in [8], [9] for efficient distributed stencil computing. However, the scheme cannot be generalized to other similar applications.

In summary, a desirable generic processing framework as shown in Fig. 1 has 5 key requirements: 1) high input data rate; 2) fully-pipelined continuous real-time processing; 3) minimal input to output latency (T_{Lat}) as shown by the timing diagram in Fig. 1; 4) optimal resource usage for the data buffer, realignment and access logic; 5) high operating frequency, which in turn ensures high data throughput.

In this work, we present SWIM (Stream Windowing on Interleaved Memory), a parameterizable framework to systematically process a class of high-speed data realignment using on-chip memory blocks. By leveraging the flexibility of on-chip block memory resources on FPGAs, SWIM allows continuous streaming data to be captured and realigned on each cycle. SWIM also ensures that the captured data are organized in such a way that they can be processed in parallel for windowing or other complex interleaving operations that require data from multiple lines of input. When compared to previous works, SWIM is general purpose, parameterizable and scalable, and incurs only minimum latency in the processing pipeline. To illustrate the flexibility of SWIM, we demonstrate its operation through two practical examples: a generic 3-by-3 2-D convolution filter and an image super-resolution method using pixel interleaving.

The rest of the paper is organized as follows. An overall design principal for SWIM will first be presented in Section II. In Section III, implementation and optimization techniques of the proposed scheme on FPGA will be explored using 2 applications as case studies (2-D convolution, and image super resolution). Performance comparisons against other common 2-D data processing approaches are then discussed in Section IV in terms of resource usage and timing performance. Lastly, Section V concludes the paper with insights on future implications and development.

TABLE I
KEY VARIABLES AND PARAMETERS FOR SWIM.

Symbol	Description
L_{line}	Number of words in a line
L_{blk}	Number of words in an input data block
H	Window height of the 2-D stencil
n	Line index
r_n	Number of words in the tail input block remainder of line n
d_n	Number of input blocks in line n
P	Number of lines in an alignment pattern repeating period
N	Number of lines being buffered in the system

II. SWIM Overview

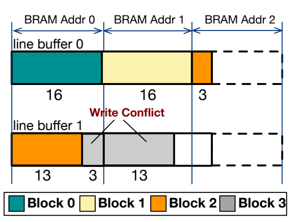
The main objective of SWIM is to enable continuous buffering and alignment of parallel data words from a high-speed stream for any arbitrary 2-D word array structure, and enables flexible parallel data access for processing such as sliding window operations in 2-D convolution. It achieves this flexibility by exploiting specific interleaving arrangements of independent Block RAM (BRAM) of different sizes. Spatial misalignment of the incoming data is handled automatically through such BRAM structure with no clock cycle overhead. As a result, SWIM incurs only minimum latency to the processing pipeline, and is able to maintain the throughput of its incoming data stream. While similar dedicated memory partition approaches have been widely proposed in both academia and industry, SWIM presents a novel general memory partition scheme for the proposed 2-D window stencil scenario that also supports block-parallel inputs with unaligned line boundaries. Table I summarizes the parameter symbols that are used in the rest of the paper for ease of discussion.

A. Data Misalignment Challenges

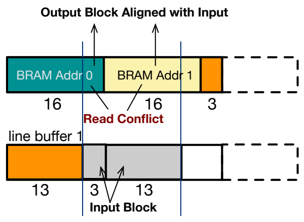
One of the key challenges for designing SWIM is to overcome the inherent limited I/O flexibilities of on-chip BRAMs. Unlike using distributed registers as storage, which provide arbitrary I/O flexibilities, BRAMs only allow simple dual-port read and/or write operations with predefined connection points. Fig. 3 illustrates this I/O limitation when a naive BRAM organisation is employed for the example depicted in Fig. 2.

In this naive design, the input data block stream is realigned into line buffers that temporarily store 1 line of the original input. Each line buffer is logically 35 pixels wide (L_{line}) and is physically implemented by 1 BRAM block. The BRAM block is configured with a data width of 16 pixels (L_{blk}) so as to accept one data block from the input data stream every cycle.

In the simple case when L_{line} is an integer multiple of L_{blk}, this naive arrangement works well — each line buffer will be implemented with exactly L_{line}/L_{blk} memory addresses in the BRAM. Furthermore, by using H line buffers in the system,



(a) Write Conflict



(b) Read Conflict

Fig. 3. FPGA BRAM data access conflicts.

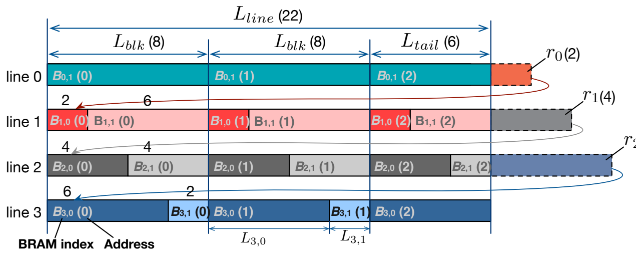


Fig. 4. Example of memory partition with basic SWIM scheme with $L_{blk} = 8$, $L_{line} = 22$ and $H = 4$. Seven BRAM blocks ($B_{0,1}$, $B_{1,0}$, $B_{1,1}$, $B_{2,0}$, $B_{2,1}$, $B_{3,0}$, $B_{3,1}$) are used to implement 4 line buffers.

parallel block reading from H different lines for 2-D stencil access is possible. Unfortunately this organization does not work in the general case when L_{line} is not an integer multiple of L_{blk}, like the example in Fig. 2 . In these cases, as shown in Fig. 3(a), the last data block of an input line (called a *tail block*) is going to include data from 2 different lines. Even if the tail block is duplicated and is written to the correct locations in both line buffers, the subsequent write to the second line buffer remains impossible because all subsequent data block will contain data that are destined for 2 different memory addresses on the same line buffer. Configuring each BRAM to have 2 write ports may partially relieve this data writing problem, but will leave no port to read from this buffer subsequently.

B. Memory Organisation

To facilitate the cases where L_{line} is not an integer multiple of L_{blk}, let the size of the remaining data in the tail block of line n that is destined for line $n+1$ be r_n, and let the number of data blocks to be stored in line n be d_n. Since the remainder from the previous line affects the alignment as well as the number of data blocks needed for the next line, r_n and d_n will vary according to the following relationship:

$$d_n = \begin{cases} \left\lceil \frac{L_{line}}{L_{blk}} \right\rceil & n = 0 \\ \left\lceil \frac{L_{line} - r_{n-1}}{L_{blk}} \right\rceil & n \in [1, N-1] \end{cases} \tag{1}$$

and

$$r_n = \begin{cases} d_n \times L_{blk} - L_{line} & n = 0 \\ r_{n-1} + d_n \times L_{blk} - L_{line} & \text{otherwise} \end{cases} \tag{2}$$

The values of r_n and d_n continue to vary with each new line but the alignment pattern will eventually repeats after P lines of input when $r_{P-1} = 0$ where

$$P = \frac{L_{blk}}{\text{GCD}(r_0, L_{blk})} \tag{3}$$

and GCD denotes the greatest common divisor of the two operands. Finally, we define the number of line buffers in the system as N, where

$$N = \left\lceil \frac{H}{P} \right\rceil \times P \tag{4}$$

With the given design parameters, the proposed memory partition scheme and data arrangement are carried out as follows. In the basic solution where $N = P$, N dedicated line buffers are set for one periodic storage pattern. Each line buffer is split into unique BRAM partitions. One line buffer contains two BRAMs, represented as $B_{n,0}$ and $B_{n,1}$. The width of each BRAM partition is equal to r_{n-1} and $L_{blk} - r_{n-1}$ words, which is represented by $L_{n,0}$ and $L_{n,1}$ respectively. During the tail block writing process of line n, a copy of the remainder part (data words that belong to the next line) is written onto the first BRAM on the next line ($B_{n+1,0}$).

Fig. 4 shows an example in the basic SWIM memory split mode for $L_{blk} = 8$, $L_{line} = 22$, $H \leq 4$. Hence, with the proposed formulations (1)(2)(3)(4), the design parameters are $N = P = 4$, $r_{0,1,2,3} = \{2, 4, 6, 0\}$, $d_{0,1,2,3} = \{3, 3, 3, 2\}$.

Detail arrangement of memory partitions is illustrated in Fig. 4 with 4 independent lines of BRAM blocks each labelled with $B_{n,p}(addr)$, where n represents the line buffer index, p represents the index of independent BRAM partitions within a line buffer, and $addr$ represents the access address in each corresponding partition. The partition width is indicated on top of each partition block in Fig. 4. Since $r_{n-1} = 0$ when $n = 0$, the zero size partition $B_{0,0}$ is omitted, and hence line buffer 0 only has one partition $B_{0,1}$ with width L_{blk}. The input block at the end of line 0 contains 2 remainder words belonging to line 1, which are written to $B_{1,0}$ at address 0. Hence, the next input block is split and stored across $B_{1,1}(0)$ and $B_{1,0}(1)$ separately. For line 2 and 3, input blocks are stored with the same rules as above. However, since line 3 will align perfectly with the input block at the end, where the width of partition $B_{3,0}$ is exactly L_{tail}, it forms the final line in the repeating storage pattern. Subsequent input block from the next line will be directed back to line 0, where the write alignment pattern restarts and each line will be overwritten accordingly as before.

C. Reading from SWIM Memory Buffers

Fig. 5 shows an overview of the data read process from the SWIM memory buffer divided into two phases. Since the data write process already aligns the data blocks with the natural boundaries of the partitioned BRAMs as shown earlier in Fig. 4, reading from the same memory addresses of these BRAMs will return exactly $H \times L_{blk}$ words of data that are vertically aligned to the lines in the original 2-D image,

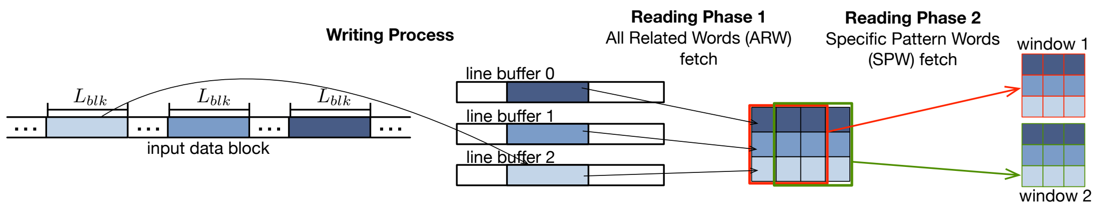


Fig. 5. Overview of the 2-phase data reading process in SWIM. In the first phase (ARW), aligned blocks of data from the original input are fetched into an output buffer. In the second phase (SPW), the specific data needed for stencil operation is read from this output buffer.

where H is the height of the designated output window. Taking advantage of this, reading from the SWIM can be divided into 2 phases that allows arbitrary parallel access pattern. The first phase is defined as All Related Words (ARW) fetch, where $H \times L_{blk}$ words of data is fetched into an addition output buffer. Subsequently, the necessary data used to form the required stencil operation are read from this output buffer into the second phase, defined as Specific Pattern Words (SPW) fetch. Depending on the application requirement, different reading schemes during the SPW fetch phase can be produced. In Fig. 5, two 3×3 windows with stride 1 for 2-D convolution is shown. In later sections, we demonstrate how this scheme can also be applied to a pixel interleaving super resolution application. Also, we will demonstrate that the overlapping 3×3 window that crosses boundary of different data blocks in the ARW fetch phase can also be handled effectively by simple logic.

One important design consideration for the SPW phase is that the overall data reading throughput must match the data writing throughout of the input data stream, so that continuous operation can be maintained without under-running or over-running the input buffers.

III. Implementation

In order to show how SWIM can be adopted in practical applications on FPGA, we first describe our generic SWIM implementation in terms of the main SWIM alignment hardware, and low-latency read logic for continuous parallel data access in applications such as 2-D window stencil computing. Next, we explain a scalable optimization method effective at minimizing overall logic and BRAM usage.

A. Generic Implementation

1) SWIM alignment hardware: A block diagram of the generic SWIM implementation is illustrated in Fig. 6. A simple example (Fig. 7) will be used to walk through the main operation of SWIM in terms of write addresses (ADDR), write enable (WE), and data access sequence at specific locations of BRAMs. As shown in Fig. 7, data are written to each BRAM in Line 0 according to the write sequence index denoted below each block. In the hardware, this corresponds to incrementation of the write address counter and the subsequent $\mathrm{ADDR}_{n,p}$, as well as assertion of $\mathrm{WE}_{n,p}$, where the subscripts n and p are the line index and BRAM partition index respectively. As the write address approaches the end partition of Line 0 on the right where a data block can no longer fit completely, remainder data words are redirected to the first BRAM partition on the next line (Line 1). The BRAM is partitioned to match the exact size of the remainder r_0, and the write operation occurs simultaneously within the same clock cycle to ensure continuous storage of incoming data blocks. Since the input data bus is common to all lines, this is achieved simply by asserting $\mathrm{WE}_{1,0}$ for Line 1.

Notice in line 1 of Fig. 7 that the block labeled “aligned write” covers BRAM partitions 1 and 0 in reverse order in relation to the physical order of the partitions within the same address. To compensate this effect, the input words are swapped in advanced along the partition boundary as illustrated in Fig. 6. Data write continues sequentially along each line from left to right and line by line until it reaches the bottom right BRAM partition, where the data block aligns perfectly with the right boundary.

2) Read logic: As described earlier in Section II-C, the read access has two phases, and the first phase that fetches all the related words (ARW) in parallel from the BRAM line buffer is the most critical and resource intensive part. Since data read of tail block at the end of each line may contain less number of words than the usual block size (L_{blk}) being written continuously to the buffers. Therefore, the read process must read addition words periodically to maintain the balance. This is achieved by utilizing both read ports on the BRAM. Although the first read port is limited by the write address control, it allows minimum latency read of the newly written data, allowing it to be used immediately for the purpose of catching up with the latest data in the read process. Fig. 9 shows the data read logic and buffers for the first read phase on one line. It allows the newly written data words from the tail block to be read one cycle ahead of time when needed. The second read ports on the BRAM partitions are driven by an independent read address bus that is common to all BRAM partitions. Therefore, data in a particular address across all lines can be read at once in parallel. When combined with the data from the first read port for maintaining the rate of reading, the buffered output contains the two most recent blocks of words from all lines.

For implementation of the phase 2 (SPW) read process, we assume the input data are lines of pixels for 2-D stencil image processing. The buffered lines are first passed onto the select and order logic described in Fig. 9 where lines relevant to the stencil window are selected. Since the input data are being

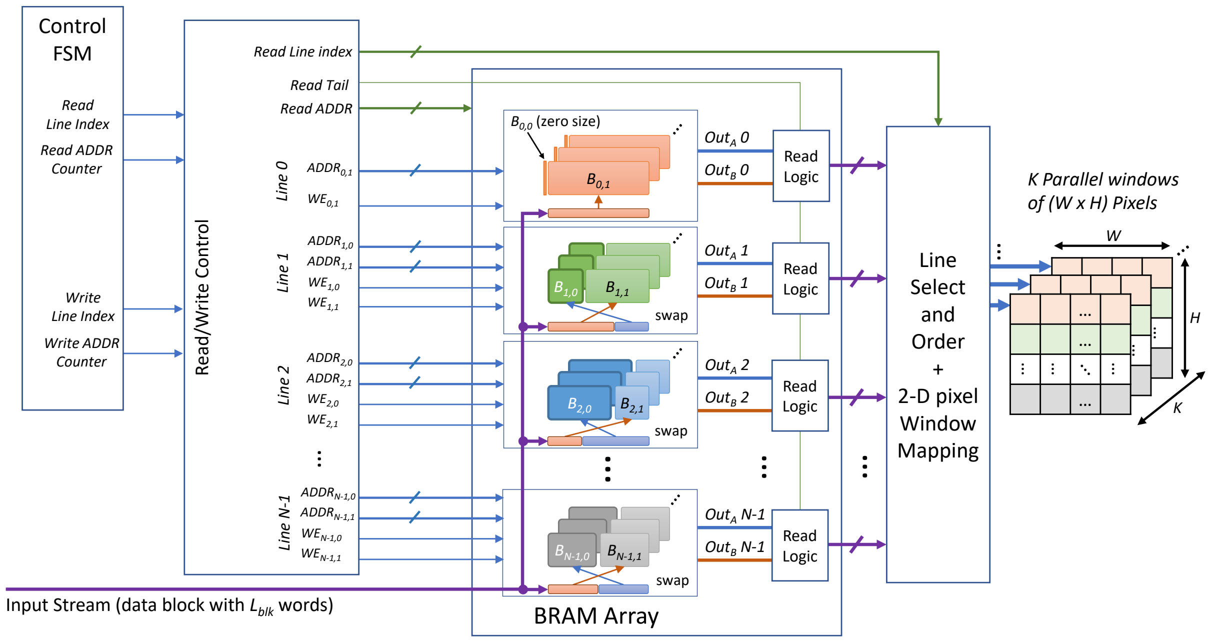


Fig. 6. Block diagram of SWIM implementation on FPGA using BRAM. W and H are the width and height of the stencil window for pixel selection, where $W \leq N$ and $H \leq L_{blk}$. K is the number of the resultant parallel 2-D pixel windows.

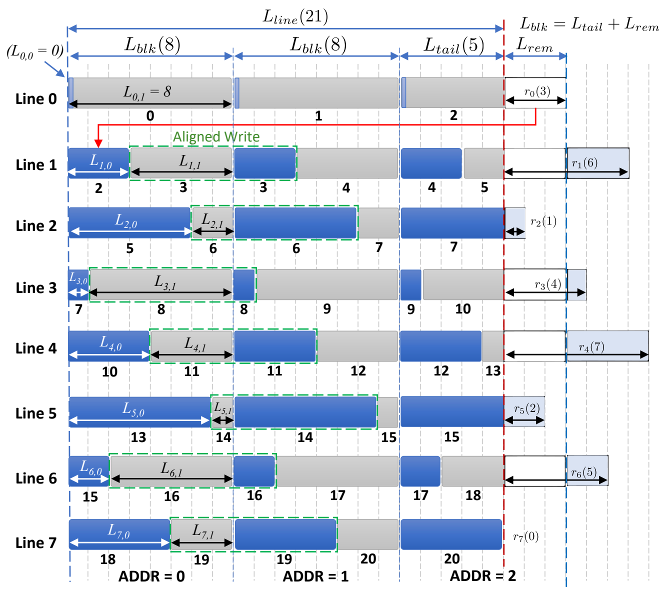


Fig. 7. Block diagram showing the structure and write sequence of data blocks onto BRAM partitions denoted by index numbers below each partition. Each partition has length $L_{n,p}$ where subscripts n and p correspond to the line index and partition index respectively. For Line n, $L_{n,0} = r_{n-1}$ and $L_{n,1} = L_{blk} - r_{n-1}$.

written to the BRAM lines in a circular buffer manner, the lines can be selected and arranged into consecutive order using basic multiplexers. The resultant pixels from the arranged lines can be accessed in parallel for further processing. As the lines already contain pixels from both the current "Target block" and the "Next block" as shown in Fig. 9, any pixel data for sliding window operation that span across two blocks can be

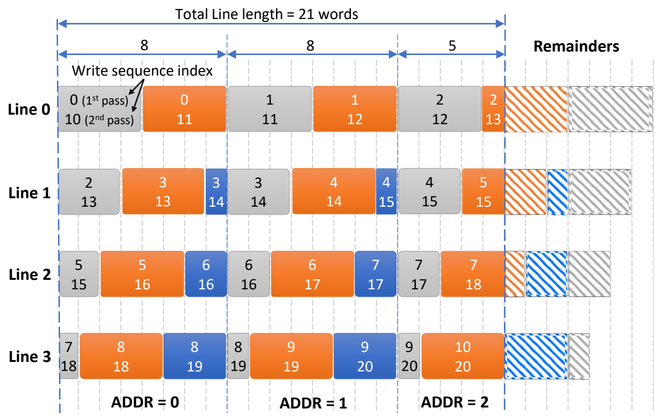


Fig. 8. BRAM usage optimization with 2-pass partitioning. The write sequence for 1st pass and 2nd pass are represented by the upper and lower index numbers respectively within each block .

easily obtained in parallel. The "Next block" will replace the "Target block" on the next clock cycle, while new data will shift into place as the new "Next block" to enable continuous and full coverage of pixels in each cycle.

3) Latency: The overall input to output latency (T_{Lat}) for 2-D stencil pixel read in terms of clock cycle is given by:

$$T_{Lat} = \left\lceil \frac{L_{line} \times (H-1) + L_{blk}}{L_{blk}} \right\rceil + 4 \quad (5)$$

$$= \left\lceil \frac{L_{line} \times (H-1)}{L_{blk}} \right\rceil + 5 \quad (6)$$

The additional 4 cycles in (5) correspond to the BRAM internal read latency and the read buffer latency. For a line

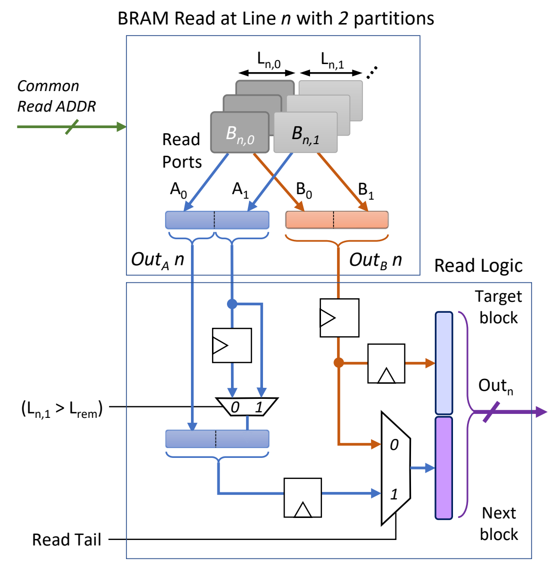


Fig. 9. Read logic for phase 1 data read. Each BRAM partition has dual read ports (A, B) and are labelled with corresponding partition index subscripts.

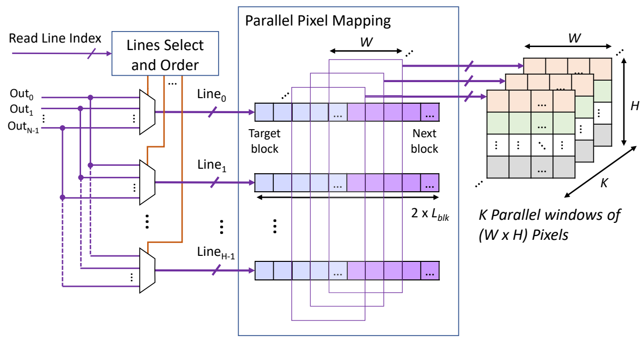


Fig. 10. Phase 2 data read with line select and order logic. W and H are the width and height of the stencil window for pixel selection, where $W \leq N$ and $H \leq L_{blk}$. K is the number of the resultant parallel 2-D pixel windows.

length of $L_{line} = 400$, $L_{blk} = 8$ and window height of $H = 3$, the latency is 105 clock cycles.

B. Multi-pass BRAM Partition Optimization

The example case in Fig. 7 takes up 8 lines of BRAM buffers. Therefore, if the window height H for data read is smaller, BRAM resources used in the additional lines exceeding H will serve little purpose other than maintaining the data alignment of incoming data. One solution to alleviate this is by employing a technique called multi-pass BRAM partitioning, where additional BRAM splits are applied and the lines are written through multiple times to reach data alignment. This allows the number of lines of buffer (N) to be reduced to specific integer factors of the alignment pattern's period P. For example, if H is less than or equal to 4, then $N = P = 8$ in the case of Fig. 7 can be reduced to $N = 4$ by a divisor of 2. The integer divisor indicates the number of passes in which the lines are written through to reach alignment. Fig. 8 shows the 2-pass optimised version for the case in Fig. 7.

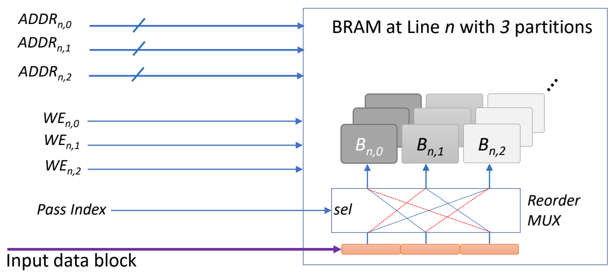


Fig. 11. Data write logic for 2-pass BRAM partitioning.

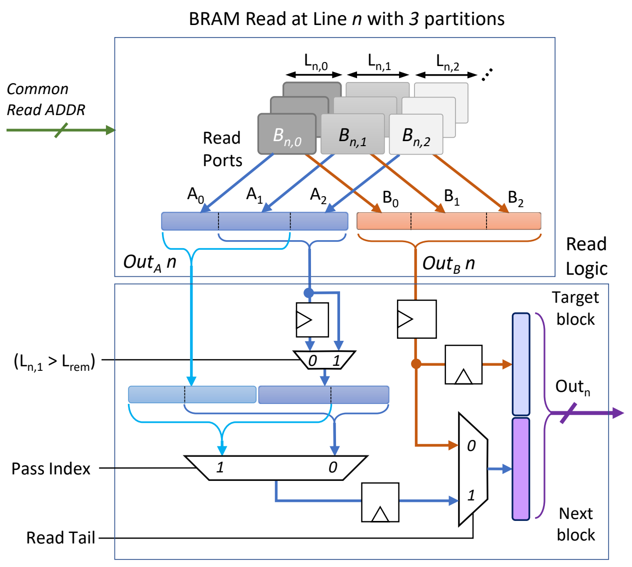
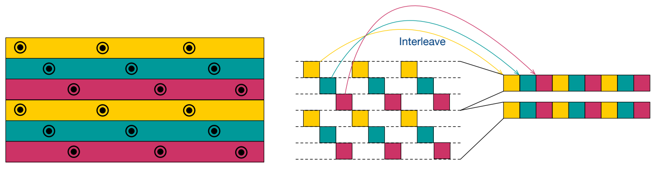


Fig. 12. Read logic for phase 1 data read with 2-pass BRAM partitioning. Each BRAM partition has dual read ports (A, B) and are labelled with corresponding partition index subscripts.

Intuitively, the process of applying 2-pass optimization to the case in Fig. 7 is equivalent to combining the BRAM partition boundary locations in Line 0 with Line 4, Line 1 with Line 5, Line 2 with Line 6, and Line 3 with Line 7 to form the 4 lines in Fig. 8 respectively. The write sequence of the 2-pass case as depicted in Fig. 8 is similar to the original case with 8 lines. The major difference is that when writing through the 4 lines the first time (1st pass), the partition boundaries equivalent to the original line 0-3 in Fig. 7 is followed. Then on the 2nd pass, partition boundaries equivalent to line 4-7 in Fig. 7 is followed. This means that only minor alteration is needed to accommodate 2-pass optimization in the SWIM implementation in Fig. 6. The two main changes needed are 1) write address and write enable control to switch between the line 0-3 case (1st pass) and line 4-7 cases (2nd pass); 2) logic for handling the different data swapping pattern of input blocks between the 1st and 2nd pass. Fig. 11 depicts such modification with a selectable reorder multiplexer (MUX) for choosing the desirable data swapping pattern depending on which pass the write process is going through. Similarly, the implementation of the phase 1 read logic for 2-pass requires a minor modification of the logic for handling the tail block read



(a) Interleaved sampling. (b) Interleave process

Fig. 13. Principle of image super-resolution with line-interleaving.

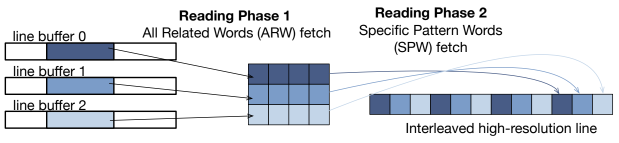


Fig. 14. The read phases of the super-resolution case using SWIM.

special case. The phase 2 read process is independent from the multi-pass optimization, so the exact same implementation as in Fig. 10 can be used.

Given the optimization principle of the 2-pass case, the number of passes can be scaled further with little hardware alteration and overhead, as long as $N \geq H$ is satisfied and the resultant N is an integer. An upper bound is reached when N is reduced to 1 with P passes, where the only permitted window height is $H = 1$.

C. Application Case Studies

The actual SWIM framework is designed as a parameterized Verilog module capable generating optimal FPGA implementations for different applications with specific word size, L_{line}, L_{blk}, H, and multi-pass optimization.

1) Convolution Filter: 2-D convolution filter is widely used in image processing and convolutional neural network (CNN) processing. The principle explained in Section III-A shows how a 3-by-3 2-D window stencil for convolution can be implemented with the generic SWIM architecture. Input blocks are written continuously onto the SWIM line buffers. Once 3 lines are filled, aligned data blocks from the lines are collected to form a phase 1 2-D array. Then in phase 2, multiple 3-by-3 overlapping windows offset by specific horizontal stride are used to select pixels in parallel from the 2-D array for actual convolution computations. Windows crossing two neighboring data blocks are handled correctly as illustrated in Fig. 10.

2) Image Super-Resolution: Another common case for SWIM is image super-resolution. In horizontal scanline based imaging system, the vertical sampling density orthogonal to the direction of line scan can often be higher than the horizontal sampling density limited by the sensor's physical bandwidth. To improve the horizontal resolution, the extra vertical resolution can be exploited by using a line-interleave sampling approach [10], [11] as shown in Fig. 13(a). Every three lines in a line-interleaving period are composed into one line with three times the resolution of the origin as in Fig. 13(b). When implementing line-interleaving with the SWIM architecture, only the phase 2 read is different from the convolution filter case. As shown in Fig. 14, phase 2 reorganizes the 2-D array output from phase 1 into interleaved-lines through wire connections only. Hence, super-resolution can be accomplished with SWIM without extra logic.

IV. Performance Evaluation and Comparisons

The performance of the generic SWIM implementation is evaluated in terms of resource usage and operating frequencies on an Xilinx Virtex-7 FPGA (xc7vx690tffg1157-2) against two other implementations: 1) a naive approach that uses distributed memory, and 2) a High level Synthesis (HLS) approach. In addition, the SWIM is implemented with and without 2-pass optimization to evaluate the performance difference in both resource usage and operating frequencies.

The line lengths used to evaluate each implementation approach are determined by the following equation:

$$L_{line} = 2^M \times L_{blk} - 3, \quad M \in \{3, 4, 5\} \tag{7}$$

where L_{blk} is fixed to 8, and the minus 3 at the end is added to ensure misalignment of the data blocks with any derived line lengths. The window size for data read is set to 3-by-3.

A. Baseline Method with Distributed Memory

A naive RTL implementation with different L_{line} has been used. Line buffers is implemented with register based distributed memory and is functionally identical to SWIM. Table II shows that the LUT usage increases extremely rapidly as L_{line} increases. At $L_{line} = 253$, the LUT usage alone reaches over 50% of the target Virtex-7 FPGA. This is likely caused by the non-linear increase in complexity of the data addressing logic when distributed memory is use. Register usage, on the other hand, appears to be directly proportional to the increase in L_{line}.

The SWIM data access behavior has also been modeled with HLS. The code structure is depicted in Fig. 15. Unlike the previous case using distributed memory, HLS requires a significantly lower number of LUTs. But interestingly, its register usage is considerably higher than the naive RTL case with distributed memory despite the fact that each HLS case is able to utilize 3 BRAMs. Moreover, even though the pipeline initiation interval is set to 1 clock cycle per data block to match SWIM, the actual HLS synthesis forced the block initiation intervals to be at least $5, 9, 17$ cycles for $L_{line} = \{61, 125, 253\}$ respectively. This means the HLS results failed to have the same continuous streaming behaviour as the SWIM implementation.

B. High-level Synthesis Implementation

C. SWIM Implementation

The implementation of SWIM is also carried out using the values of L_{line} derived from (7). For distinguishing the resource utilization between the generic alignment buffer/logic portion and the application specific read logic portion of SWIM, implementations with and without reading logic are evaluated.

TABLE II
RESOURCE USAGE AND FMAX COMPARISON TABLE

	Naive Verilog [a]			**HLS** [a]			**SWIM** [b]			**SWIM 2-pass** [b]			**SWIM 2-pass** [c]		
M	3	4	5	3	4	5	3	4	5	3	4	5	3	4	5
L_{line}	61	125	253	61	125	253	61	125	253	61	125	253	61	125	253
Register	1678	3414	6885	2378	4714	9380	166	169	172	169	170	177	1008	1022	1036
LUT	22265	96384	377966	1457	2858	5715	568	581	582	429	432	436	994	1010	1011
BRAM	0	0	0	3	3	3	18	18	18	9	9	9	9	9	9
Fmax(MHz)	210	216	202	224	224	224	456	500	567	442	588	462	410	448	420

[a] Synthesis result. [b] Implementation result without read logic. [c] Implementation result with read logic.

```
BLK: for (blk_idx=0;blk_idx<N_blk; blk_idx++){
 // Set the Initiation Interval between two BLK as 1 clclye
#pragma HLS PIPELINE II=1
    input.read(input_blk)
    INBLK: for(ii=0;ii<L_blk; ii++){
        ··· // Function(line buffer read & write)
    }
    output.write(output_blk)
    ··· // Function(control signal)
}
```

Fig. 15. HLS code structure.

As shown by the results in Table II, LUT and Register usage increases at an extremely slow rate as L_{line} increases. This is because the most resource expensive part for handling memory addressing is all done by dedicated hardware within each BRAM. Therefore, increase in BRAMs depth when L_{line} increases will hardly increase usage of LUT and Register. In fact, the tiny increase in LUT and Register usage is likely due to the increase of address bus width, causing the write address control logic described in Fig. 6 to slowly utilize more LUTs and registers. For the 2-pass case, we see an even lower usage in LUT, as it has half the number of buffer lines in the system, and hence less logic is needed to handle the reduced number of address buses and write enable signals.

Another interesting observation from Table II shows that BRAM usage of the single-pass and 2-pass SWIM remained constant at 18 and 9. This is due to the fact that all the cases with lower L_{line} only used a subset of the memory elements within each BRAM. Therefore, there is no need to utilize additional BRAMs until they are fully filled.

In terms of timing performance, both the single-pass and 2-pass SWIM showed impressive Fmax above 400MHz in all cases in Table II. This is again related to the fact that the most timing critical path lies within the BRAM's internal logic path and they are very well optimized to maintain consistent timing performance even when BRAM depth increases.

V. CONCLUSION

In this paper, we have presented SWIM, a framework that systematically generates FPGA implementations with data realignment and reorganization structures using on-chip block memory. SWIM leverages the flexibilities of block memories in modern FPGAs to create highly efficient structure tailored to the user's specific alignment and data access requirements. Using a generic 3×3 windowing operation and a line-interleaving pixel super resolution scheme as examples, we have demonstrated that SWIM is capable of creating low-latency designs that operate at close to the maximum clock frequency of the FPGA. The proposed scheme is generic and is applicable to any application that needs to process streams of input data blocks with tight timing requirements. The proposed scheme is scalable and works equally well with various windowing and interleaving pattern that involves multiple lines of the original input. In the future, we will continue to extend SWIM to accommodate other similar applications that operate on interleaving data input with stringent realignment requirements such as in high-throughput and low-latency networking.

ACKNOWLEDGEMENTS

This work is supported in part by the Research Grants Council of Hong Kong (Project GRF 17245716), and the Croucher Foundation (Croucher Innovation Award 2013).

REFERENCES

[1] *LVDS Source Synchronous DDR Deserialization (up to 1,600 Mb/s)*, Xilinx Inc., 2016.

[2] *Intel Stratix 10 High-Speed LVDS I/O User Guide*, Intel Corp., 2017.

[3] C. Johnston, K. Gribbon, and D. Bailey, "Implementing image processing algorithms on fpgas," in *Proceedings of the Eleventh Electronics New Zealand Conference, ENZCon04*, 2004, pp. 118–123.

[4] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2015, pp. 161–170.

[5] M. Frigo and V. Strumpen, "Cache oblivious stencil computations," in *Proceedings of the 19th annual international conference on Supercomputing*. ACM, 2005, pp. 361–366.

[6] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan, "Effective automatic parallelization of stencil computations," *ACM sigplan notices*, vol. 42, no. 6, pp. 235–244, 2007.

[7] T. VanCourt and M. Herbordt, "Application-Specific Memory Interleaving for FPGA-Based Grid Computations: A General Design Technique," in *2006 International Conference on Field Programmable Logic and Applications*. IEEE, 2006, pp. 1–7.

[8] W. Luzhou, K. Sano, and S. Yamamoto, "Domain-specific language and compiler for stencil computation on fpga-based systolic computational-memory array," in *International Symposium on Applied Reconfigurable Computing*. Springer, 2012, pp. 26–39.

[9] K. Sano, "Fpga-based systolic computational-memory array for scalable stencil computations," in *High-Performance Computing Using FPGAs*. Springer, 2013, pp. 279–303.

[10] A. C. Chan, H.-C. Ng, S. C. Bogaraju, H. K. So, E. Y. Lam, and K. K. Tsia, "All-passive pixel super-resolution of time-stretch imaging," *Scientific Reports*, vol. 7, 2017.

[11] R. Shi, A. C. Chan, E. Y. Lam, and H. K. So, "Image super-resolution for ultrafast optical time-stretch imaging," in *Proceedings of 24th Congress of the International Commission for Optics*, Aug. 2017, pp. W1F–08.