Memory Interface Design for 3D Stencil Kernels on a Massively Parallel Memory System

ZHEMING JIN and JASON D. BAKOS, University of South Carolina, Columbia

Massively parallel memory systems are designed to deliver high bandwidth at relatively low clock speed for memory-intensive applications implemented on programmable logic. For example, the Convey HC-1 provides 1,024 DRAM banks to each of four FPGAs through a full crossbar, presenting a peak bandwidth of 76.8GB/s to the user logic. Such highly parallel memory systems suffer from high latency, and their effective bandwidth is highly sensitive to access ordering. To achieve high performance, the user must use a customized memory interface that combines scheduling, latency hiding, and data reuse. In this article, we describe the design of a custom memory interface for 3D stencil kernels on the Convey HC-1 that incorporates these features. Experimental results show that the proposed memory interface achieves a speedup in runtime of 2.2 for 6-point stencil and 9.5 for 27-point stencil when compared to a naive memory interface.

Categories and Subject Descriptors: C.2.2 [**Computer-Communication Networks**]: Network Protocols

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Memory latency hiding, memory access scheduling, data reuse, memory interface, 3D stencil

ACM Reference Format:
Zheming Jin and Jason D. Bakos. 2015. Memory interface design for 3D stencil kernels on a massively parallel memory system. ACM Trans. Reconfigurable Technol. Syst. 8, 4, Article 24 (September 2015), 23 pages.
DOI: http://dx.doi.org/10.1145/2800788

1. INTRODUCTION

The Convey HC-1 coprocessor contains four user-programmable FPGAs that each have access to a shared on-board memory. The memory system is composed of a set of eight off-chip DDR2 DRAM controllers, where each controller is connected to two DIMMs containing eight DRAM chips with eight banks each. The memory is thus organized as 1,024 banks across 128 DRAM chips across 16 DIMM modules.

To maximize bandwidth for nonconsecutive access patterns, each memory controller (MC), controlling one-eighth of the memory space, contains a dynamic out-of-order scheduler for each of its 16 DRAM chips. Each scheduler attempts to select requests to nonbusy banks (often ahead of memory accesses made earlier) and groups reads and writes into bursts to minimize state changes.

The crossbar and scheduler buffering together impose a latency of 256 to 512 cycles to every memory request. As such, memory-bound kernels must explicitly support latency hiding to maintain high throughput. Specifically, the user logic must be able to support

This material is based upon work supported by the National Science Foundation under grant 0844951.
Authors' address: Z. Jin and J. D. Bakos, Department of Computer Science and Engineering, University of South Carolina; emails: jinz@email.sc.edu, jbakos@cse.sc.edu.

DOI: http://dx.doi.org/10.1145/2800788

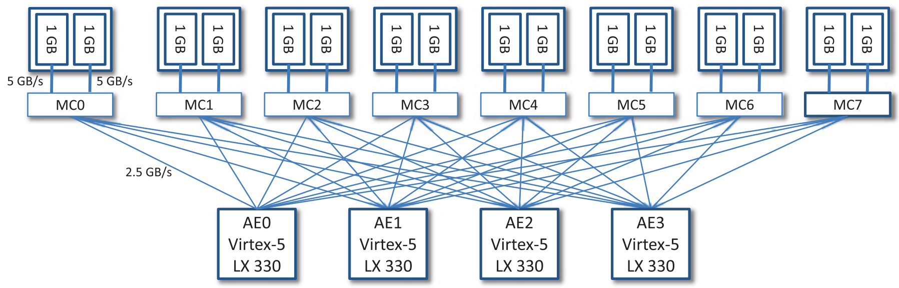

Fig. 1. Memory architecture of the Convey HC-1.

512 outstanding memory requests. Even then, the MCs will stall whenever a scheduler buffer becomes full, which occurs often during periods of bank contention.

When designing fully custom kernel pipelines, the memory interface should be optimized for the kernel, so it is crucial to develop best practices for interface design. In this article, we use 3D stencils—a common memory-bound scientific kernel—as a benchmark to explore how interface parameters affect various performance factors.

We propose three metrics with which to evaluate interface designs. For memory latency hiding, we use *interface efficiency*—the percentage of execution cycles in which the memory interface performs a memory access as opposed to being stalled due to its own internal dependencies. For example, a double buffer may need to wait for a buffer to fill before it can swap, even after making all of its requests. Using our interface design, we are able to achieve nearly 100% interface efficiency.

For scheduling, we use *DRAM controller efficiency*—the proportion of cycles in which the interface is ready and able to access memory and is not stalled by the MC due to bank contention. We improve DRAM controller efficiency from 69% to 74% for the 6-point 3D stencil and from 52% to 72% for the 27-point 3D stencil as compared to an unscheduled implementation.

For data reuse, we use *data reuse rate*—the ratio of the number of reused memory requests to the total number of memory requests. We are able to achieve a reuse rate of 0.6 and 0.9 for the 6- and 27-point 3D stencil, respectively.

We evaluate the designs on the Convey HC-1, although this approach would be beneficial for any platform with a similar memory system. Compared to the baseline memory interface designs, we are able to achieve average speedup of 2.2 for the 6-point 3D stencil and 9.5 for the 27-point 3D stencil.

The rest of the article is organized as follows. Sections 2 and 3 introduce the background and related work. Section 4 describes the memory interface designs. Section 5 shows the experimental results of memory interface designs, and Section 6 concludes the article.

2. BACKGROUND

2.1. Convey HC-1

The Convey HC-1 is a high-performance reconfigurable computer containing an FPGA-based coprocessor attached to a host motherboard through a socket-based front-side bus interface.

As shown in Figure 1, the coprocessor board contains four user-programmable Virtex-5 LX 330 FPGAs called *application engines* (AEs). The coprocessor board also contains eight discrete MCs, each of which is implemented on its own Virtex-5 LX 110 FPGA. The memory space of the coprocessor board is physically partitioned into eight equal-size address spaces, and each space is accessible only from one of the eight MCs. Each

AE is connected to all eight MCs through a crossbar switch that is instanced on each MC.

The interface between each user-programmable FPGA and each MC allows up to two independent 64-bit memory transactions (read or write) per cycle on a 150MHz clock, giving a peak theoretical bandwidth of 2.4GB/s between each AE-to-MC interface, or 19.2GB/s per user-programmable FPGA, or 76.8GB/s aggregate bisection bandwidth [Convey 2012].

Memory addresses are virtual and mapped to 4MB pages. Each MC contains a translation look-aside buffer (TLB) to cache the page table. Within each 4MB page, the memory address is divided into the following physical fields:

—Bits 21:20 (2 bits): DRAM row
—Bits 19:13 (7 bits): DRAM column
—Bits 12:10 (3 bits): DRAM bank
—Bit 9 (1 bit): DIMM selector
—Bits 8:6 (3 bits): MC selector
—Bits 5:3 (3 bits): DRAM selector (eight DRAMs per DIMM)
—Bits 2:0 (3 bits): Always 000, as accesses are aligned on 8-byte boundary

Each of the eight MCs is connected to two DIMMs selected by bit 9. Both DIMMs contain eight DRAM chips (i.e., eight 8-bit busses), and each DRAM chip has eight banks that can be accessed independently by the MC. Memory addresses are interleaved across the MCs as well as across the banks within the DIMMs attached to each MC. This ensures that sequential accesses will be spread across all 16 DIMMs, maximizing concurrency and bandwidth. Each of the MCs attempts to reduce the performance impact of nonconsecutive accesses by scheduling incoming memory requests to each of the banks on its DIMMs, routing requests to nonbusy banks, and grouping reads and writes into bursts to minimize bus turns. As a result, memory accesses are performed in a different order in which they were requested by the programmable logic. In practice, the achieved bandwidth ranges from 2% to 93% of the peak bandwidth depending on access orders. In an exhaustive test, we found that only 18% of all possible access patterns that access all of the unique addresses within a page achieve greater than 50% of peak bandwidth, and more than half of the patterns resulted in less than 20% of peak bandwidth [Jin and Bakos 2013].

2.2. Memory Access Ordering

As described previously, the Convey HC-1's FPGA-based MCs include a proprietary, out-of-order dynamic memory scheduler. This scheduling is performed in the opaque MCs that are not user programmable. Despite the presence of a dynamic scheduler, the effective bandwidth still depends on the memory access pattern provided by the programmable logic. Thus, our proposed interface introduces an additional layer of static memory access scheduling from the perspective of the programmable logic.

Figure 2 depicts the various memory access orderings that occur within this design. The left outer box represents the user-programmable logic, whereas the right outer box represents nonprogrammable entities in the HC-1's onboard memory system, composed of DRAMs and nonprogrammable logic (including buffers and schedulers).

The kernel's memory interface, designed by the user, performs memory requests using the access order A. In general, this ordering should be optimized for the memory system as opposed to the kernel pipeline. In other words, although the kernel pipeline expects inputs to arrive in the order C that corresponds to the structural arrangement of the kernel's functional units, it can request the data in another order that seeks to minimize nonconsecutive DRAM accesses. For example, if the kernel is adding two vectors, the computational logic may expect inputs to arrive as interleaved pairs of

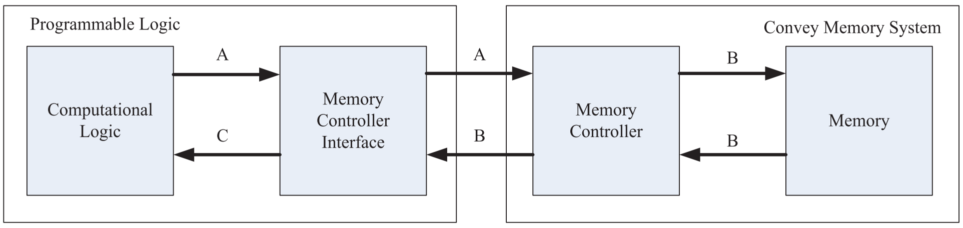

Fig. 2. Memory access orderings on the Convey HC-1.

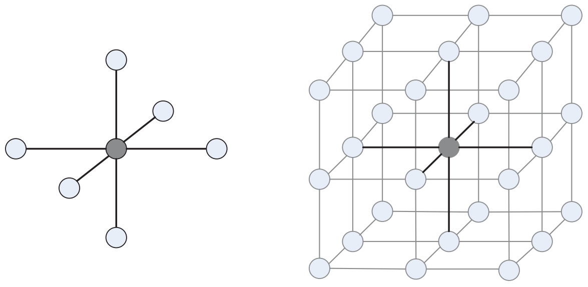

Fig. 3. The 6-point and 27-point 3D stencils.

elements from each of the two vectors (order C). This ordering would require a strided access pattern, so the programmable logic may group requests from consecutive elements from each vector into a block (order A).

The kernel's memory interface passes these requests to the HC-1's off-chip MC, which will dynamically schedule the requests and thus generate a new, indeterminate access order B. The memory data is returned to the programmable logic in this same order.

Convey's Design Kit includes an optional memory reorder buffer IP block that ensures data returned from the memory is delivered to the programmable logic in the same order it was originally requested (changes the order from B to A). Generally, this module is integrated into the user memory interface. The kernel's memory interface must buffer and reorder this data again so that it can present the input data to the kernel pipeline in the order that it expects—order C.

Since both Convey's reorder buffer and our proposed scheduler exist within the block labeled "Memory Controller Interface," this block changes order from B to C (B to A, then A to C).

2.3. Stencil Computation

As shown in Figure 3, a stencil—a commonly used operation in scientific codes—reads N input points to calculate one output point. In this article, we choose a 6-point and 27-point stencil computation as representative kernels:

—*6-point 3D stencil*: Each point (element) in the output block is updated by the six neighbors offset by 1 on each direction. Each point is a 64-bit word. The six neighbors are accumulated to generate the output point, so there are five additions for each point.

—*27-point 3D stencil*: Each point computation involves all points in a $3 \times 3 \times 3$ cube surrounding the center output point. The 27 64-bit points are accumulated to generate the output point, so there are 26 additions for each point.

We assume that the stencil computation of each output point is independent of every other output points and can thus be computed in any order.

2.4. Performance Measurement

When implemented on an FPGA, stencil computations that have sufficiently low operations per byte are memory bound, in which their performance is determined by achieved memory bandwidth, as shown in Equation (1).

$$\textit{execution time} = (\textit{size of data transacted})/(\textit{achieved memory bandwidth}) \tag{1}$$

The achieved memory bandwidth is determined by the peak bandwidth (memory clock speed multiplied by memory width) and the memory efficiency, as shown in Equation (2).

$$\textit{achieved memory bandwidth} = (\textit{peak memory bandwidth}) * (\textit{memory efficiency}) \tag{2}$$

2.4.1. Memory Efficiency. Memory efficiency is determined by the ratio, as shown in Equation (3), where *rc* is the number of memory read cycles, *wc* is the number of write cycles, and *ec* is the observed number of execution cycles.

$$\textit{memory efficiency} = (rc + wc)/ec \tag{3}$$

This efficiency metric characterizes the relative impact of the number of cycles in which the programmable logic does not access memory. These *idle cycles* (*ic*) can be expressed as the difference in execution cycles and the memory reference cycles, as shown in Equation (4).

$$ic = ec - (rc + wc) \tag{4}$$

There are two factors that determine the number of idle cycles. The first factor is the number of *stall cycles* (*sc*), in which the memory interface in the programmable logic must temporarily cease making memory requests after receiving a stall request from Convey's memory scheduler. These stall requests occur when the scheduler buffers become full due to bank contention.

The second factor is *interface idle cycles* (*icc*), which are caused by inefficiency in the memory interface of the programmable logic. The cause of these is specific to the memory interface design.

$$ic = sc + icc \tag{5}$$

Because stall cycles and interface idle cycles are caused by different aspects of the memory interface design, we define two efficiency metrics that characterize them separately.

The first is *interface efficiency*, shown in Equation (6). Interface efficiency characterizes the memory interface's ability to sustain memory requests despite long memory latencies. For example, in a typical double buffer, the interface will seek to fill one side of the buffer while flushing the other side. It will experience idle cycles from the time it requests all words until all requests return (since the buffer cannot be switched until all requested words arrive).

$$\textit{interface efficiency} = (rc + wc)/(rc + wc + icc) \tag{6}$$

The second is *DRAM controller efficiency*, shown in Equation (7). DRAM controller efficiency characterizes how effectively the latency of requests to DRAM banks is hidden with requests to other banks.

$$\textit{DRAM controller efficiency} = (rc + wc)/(rc + wc + sc) \tag{7}$$

2.4.2. Reuse Rate. The total amount of data transacted with off-chip memory depends on access locality and how effectively the data is cached. The relationship between the amount of data transacted and the total amount of input and output data required is related using *memory reuse rate*, shown in Equation (8).

$$data\ transacted = (input\ and\ output\ data) * (1 - (reuse\ rate)) \tag{8}$$

To measure data reuse, reuse rate is computed as the ratio of the number of reused memory requests to all memory accesses.

$$reuse\ rate = M2/M1, \tag{9}$$

where *M1* is the total number of memory references and *M2* is the number of memory references that are reused.

2.4.3. Performance. Each of these three metrics—interface efficiency, DRAM controller efficiency, and reuse rate—has a direct relationship with execution time. As such, improving any of these metrics will lead to a corresponding kernel speedup.

3. RELATED WORK

3.1. Convey HC-1 Memory System

Augustin et al. [2011] explored the challenges in designing efficient 3D stencils for the Convey HC-1. Using a 3D 7-point stencil for solving the Laplace equation on grids of different sizes, they compared the peak floating-point performance on the Convey HC-1 with a two-way 2.53GHz Intel Nehalem processor. They showed that the stencil performance on the Convey HC-1 is lower on smaller grids due to the lack of caching.

Cong et al. [2011b] demonstrated that the Convey HC-1 suffers from low DRAM efficiency for memory-bound kernels having a complex access patterns. In this case, the authors achieved only 30% memory efficiency on both the HC-1 and a GPU.

3.2. Optimizing Memory Performance for 3D Stencils

He et al. [2004, 2005, 2006] developed a series of FPGA-based 2D and 3D stencils for the FDTD simulation required for seismic imaging. These designs exploit data reuse by sending grid values from the input cache to a set of cascaded FIFOs, where each FIFO buffers 2D pages. This work emphasized the trade-off between reuse and on-chip memory bandwidth.

Datta et al. [2008] explored double-precision 3D single-point stencil computations on multicores by developing numerous optimization strategies and an autotuning environment. To hide memory latency, they employed hardware prefetching, software prefetching, DMA, and multithreading. They used circular buffers to reduce cache conflict misses for both read and write planes.

3.3. DRAM Scheduling

DRAM scheduling received much attention in the early 2000s as a result of DARPA investment in advanced multicore architectures. In this work, the scheduling was usually performed dynamically since tasks are loosely synchronized across cores. For example, Dally's group at Stanford developed a dynamic DRAM bank and row scheduling technique for streaming applications to maximize effective memory bandwidth [Rixner et al. 2000]. Fang et al. [2009] developed a core-aware memory access scheduler that prioritizes consecutive accesses to the same row or bank to minimize DRAM row misses and thus reduce memory latency. Ahn et al. [2009] used a similar approach targeting DRAM ranks in a technique called *rank subsetting*. Liu et al. [2008] developed a page hit aware write buffer (PHA-WB) for the purpose of reducing DRAM power consumption without affecting performance.

Static scheduling techniques on CPUs, on the other hand, often rely on instruction scheduling (e.g., software pipelining) and associating each array entry with specific time slots that can then be scheduled for specific DRAM banks. The arrays are then allocated to banks to minimize conflicts. Lyuh and Kim [2004] used this technique for the purpose of DRAM power management. Their scheduler allocates each array such that during execution the standby time for each bank is maximized, considering the overhead time required to switch banks from active to standby mode. Chang and Lin [2000] previously developed a similar technique of array-bank allocation but targeted memory latency.

3.4. Data Reuse

The design of specialized on-chip memories to maximize data reuse is often explored in the context of compiler-generated management of program-controlled scratchpads [Wang et al. 2012; Banakar et al. 2002; Panda et al. 1997; Kandemir et al. 2004; Issenin et al. 2007; Cong et al. 2011a; Cong et al. 2011d]. One of the main challenges is that increasing reuse inevitably leads to port contention on the on-chip RAM and replication is constrained by RAM size. In addition, memory mapping and partitioning approaches often fail when data reuse is performed in a circular manner to save buffer size [Tatsumi and Mattausch 1999; Ho and Wilton 2004; Benini et al. 2002; Baradarn and Diniz 2008; Cong et al. 2011c; Ben-Asher and Rotem 2010].

There have been recent efforts to automatically generate and allocate minimum size on-chip scratchpads to reuse data and guarantee freedom from bank conflicts given a specific access pattern extracted from a high-level language kernel [Wang et al. 2013]. However, this approach does not consider DRAM scheduling (changing the ordering of outgoing memory accesses to optimize against DRAM topology) or latency hiding (maximizing the number of inflight memory requests) for the purpose of decreasing DRAM control and interface stalls. In other words, our proposed designs have different objectives. Our buffer duplication design targets DRAM controller efficiency by minimizing the frequency of nonconsecutive memory access. In this case, reducing the buffer size is detrimental to DRAM scheduling, as it will increase the frequency of nonconsecutive memory requests (i.e., decreasing the block size). Our numerical ordering design sacrifices the size of contiguous block sizes to achieve data reuse, but it still achieves a trade-off between DRAM controller efficiency and reuse.

4. MEMORY INTERFACE DESIGNS

4.1. Memory Access Scheduling with No Data Reuse

As described in Sections 2.1 and 2.2, the effectiveness of Convey's dynamic DRAM scheduler depends on the memory access pattern provided by the programmable logic. Each of the eight DRAM controllers is connected to 128 DRAM banks. The schedulers attempt to achieve high bandwidth under any arbitrary nonconsecutive access pattern by scheduling pending requests out of order to nonbusy banks and grouping sets of reads and writes to minimize bus turns. However, access patterns that exhibit higher bank conflict rates will eventually lead to the scheduler queue holding too many outstanding requests and become full, at which point the scheduler stalls the programmable logic, throttling the request rate and reducing effective bandwidth.

The rate at which this throttling behavior occurs can be reduced if the memory access pattern is prescheduled before being presented to Convey's DRAM scheduler. In other words, our proposed interface introduces an additional layer of memory access scheduling from the perspective of the computational logic.

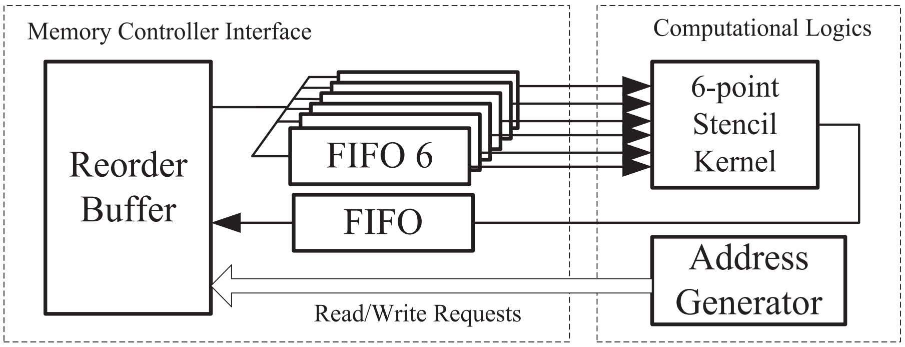

Fig. 4. Block diagram of the 6-point 3D stencil for each PE.

To demonstrate the impact of this prescheduling, we present two access orders—loop order and array order—as two memory access orders in the baseline memory interface designs.

Loop order refers to a conceptual memory access order in which each element from each of the input and output arrays is referenced as if the kernel is being executed in software with no compiler optimization on a single thread on an in-order processor. In other words, loop order is the order that data would be accessed in a naive pseudocode description of the kernel's algorithm. Stated yet another way, loop order is the ordering implied by the abstract high-level code.

For example, for a kernel that adds two vectors C = A + B, loop order would describe the case when each element from each of the two input vectors and output vector (e.g., A[i], B[i], and C[i]) are accessed in an element-wise interleaved fashion without considering any cache blocking or loop tiling behavior. Loop order will naively perform nonconsecutive accesses within each loop iteration. For example, in the 6- and 27-point stencils, each iteration has to access 6 and 27 elements from the memory to compute an output element, respectively.

Array order, on the other hand, uses on-chip buffering to reduce the frequency at which nonconsecutive addresses are referenced by the kernel by prefetching a block of consecutive addresses from each referenced array. When processing a multidimensional array, the kernel will buffer data along the major dimension. Array order also leads to access patterns containing nonconsecutive addresses, but S times less frequently than loop order, where S equals the size of the buffer.

Figure 4 shows the design of a processing element for the 6-point 3D stencil. In this design, a FIFO is instanced for each stencil point (element from input array). The FIFOs allow the stencil points to be requested from memory using an arbitrary access pattern (request order), which is determined by the address generator. This access pattern could be loop order if each of the stencil points for each output point is requested in consecutive cycles. Alternatively, the access order can be optimized to balance the requests across DRAM banks and/or maximize the number of accesses per DRAM row. In either case, the FIFOs are used to reconcile the difference between the order in which inputs are requested from memory (received from memory) and the order in which they are expected by the kernel logic (pipeline), such as by delivering all stencil input points for each output in parallel in a single clock cycle.

The address generator (which is essentially a DMA engine) produces the memory addresses in either loop order (in which $S = 1$) or array order (for an arbitrary S limited by the FIFO depth). In loop order, the six inputs needed to compute each output element are requested consecutively within its group. When each input arrives,

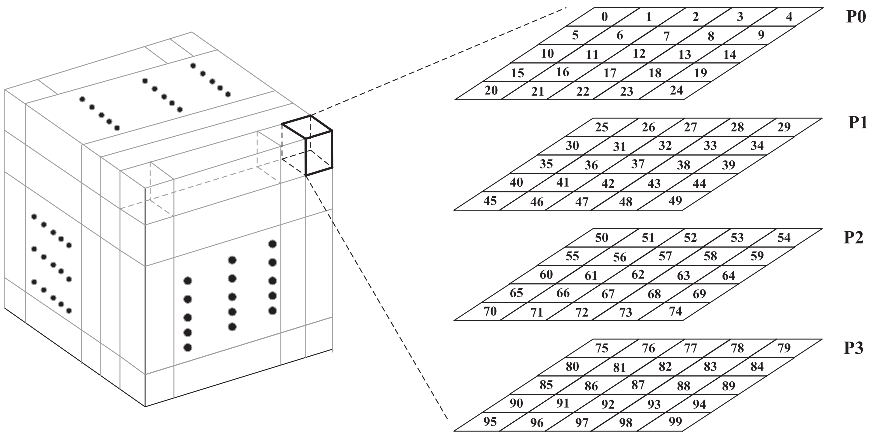

Fig. 5. A 5 × 5 × 4 block with four 5 × 5 planes with elements numbered numerically.

it is stored in the FIFO corresponding to its stencil point. In array order, a block of S elements from consecutive addresses is requested, and each block of S elements is stored in a single FIFO. No FIFOs are read until no FIFOs are empty.

When only using one of the HC-1's FPGAs (and thus being constrained to only 25% of the available memory bandwidth), the DRAM controller efficiency is 67% for loop order and 90% for the array order with a sufficiently high value of S. Using additional FPGAs would produce additional demand for the DRAM banks and further reduce efficiency.

4.2. Memory Access Scheduling with Data Reuse

To combine memory latency hiding, memory access scheduling, and data reuse, we propose *numerical order*—a memory access order that emphasizes consecutive memory address access while taking advantage of the stencil's static access pattern to reuse words already received from memory. Numerical ordering does this by associating buffered input elements with a local address.

Figure 5 illustrates numerical order by showing how a subblock of 5 × 5 × 4 input elements from the stencil input space are conceptualized as four 2D planes (P0, P1, P2, and P3). Each element in the planes is numbered numerically starting from 0 in P0 and ending at 99 in P3. The points are used as inputs and are inclusive of the halo region. As such, plane P1 has nine output points corresponding to the locations at 31, 32, 33, 36, 37, 38, 41, 42, and 43, and plane P2 has nine output points (56, 57, 58, 61, 62, 63, 66, 67, and 68). Since each output element is accumulated by N neighbors offset by 1 on each direction in the stencil computation, the 6-point stencil computation of an output point in P1 requires the neighboring elements in P0, P1, and P2, whereas the computation of an output point in P2 requires the neighboring elements in P1, P2, and P3. For example, the output element corresponding to location 31 in P1 is a function of elements at locations 6 (P0), 26 (P1), 30 (P1), 32 (P1), 36 (P1), and 56 (P2).

4.2.1. DRAM Latency Hiding in Numerical Ordering. As described earlier, our 6-point stencil design contains six input FIFOs corresponding to each of the stencil points and—depending on the address generator—can use loop or array order when reading input points. Each FIFO receives the appropriate data with the FIFO select control signal. The control signal specifies into which of the six FIFOs the requested element is to be stored. When the address generator requests the elements numerically, it is not feasible to decode the location of each element to know in which FIFO the element is to be stored. However, if all requested elements are stored in the same on-chip buffer

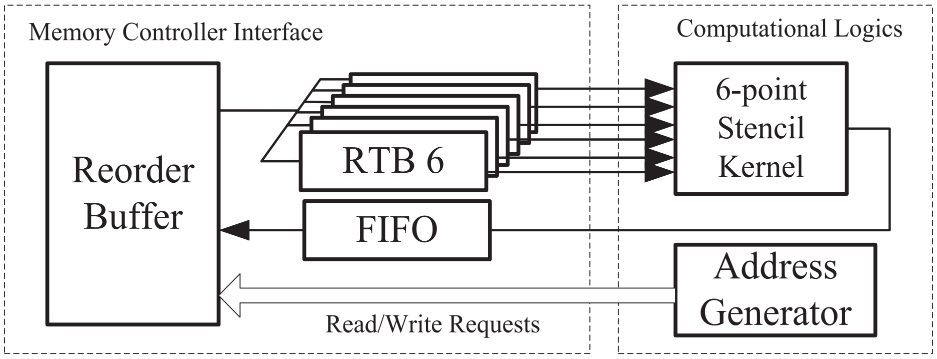

Fig. 6. Block diagram of the proposed 3D 6-point stencil for each PE.

with a single read port, it would take six cycles to read six elements for computing an output element.

To hide memory latency, we duplicate the buffer six times for the 6-point stencil so that each buffer receives the same data from the MC interface. As such, the kernel can read six elements from the buffers in one cycle, as the case in the FIFO-based design. Each buffer must be designed as a modified FIFO, where words are always enqueued onto the tail but words can be read from any location. Once the buffer is full, the tail pointer wraps around and entries in the memory are replaced. We refer to this behavior as a rotation buffer (RTB). In other words, unlike a FIFO, the data in the reorder buffer is still available after reading as long as it is not overwritten. This behavior allows entries in the buffer to be reused as long as possible until they need to be replaced due to lack of capacity.

Figure 6 shows the block diagram of the design. Input elements are enqueued into the RTBs in numerical order. When reading each group of six elements, we must know their addresses in each of the six RTBs. The address of each element is specified by the numerical order of each element in a block.

For example, referring to Figure 5, to compute the output element corresponding to location 31, the six RTBs are accessed at addresses 6, 26, 30, 32, 36, and 56, respectively. Observe that if address 31 of the output point is assumed to be the reference address, then the differences between the reference address and the addresses of six neighbors are –25, –5, –1, 1, 5, and 25.

For an N-point stencil, we take advantage of the static nature of the address differences to simplify the implementation of read address generation. As long as the location of an output element and the size of the plane are known, the addresses of the N neighbors can be calculated using the relative addresses.

4.2.2. DRAM Scheduling in Numerical Ordering. When reading each plane to store in the RTBs, there is an address stride from the end of a row to the start of the next row within each plane. For example, in Figure 5, elements in plane P0 at location 4 and location 5 are not stored consecutively in DRAM. Thus, the number of consecutive accesses is limited by the width of the plane, which is generally a smaller number than the block size S used in array order. Therefore, we expect DRAM efficiency for numerical order to be less than that of array order. This represents a trade-off between DRAM controller efficiency and data reuse rate.

4.2.3. Data Reuse in Numerical Ordering. In the memory interface designs with loop and array order, the address generator always requests six input elements for each output element, and there is no data reuse. Thus, computing the block of $3 \times 3 \times 2$ outputs that corresponds to each block of $5 \times 5 \times 4$ inputs requires 108 ($3 \times 3 \times 2 \times 6$) requests. In numerical order, the address generator will request all elements in the $5 \times 5 \times 4$ block for computing 18 output elements in planes P1 and P2. In this case, the number of requested elements is 100 ($5 \times 5 \times 4$) elements.

Since the size of each RTB is limited, it has to be rotationally updated with the new data. The update is based on the knowledge of regular access patterns of a block in the stencil space. As the buffers always store data in numerical order, the old data in a buffer can be updated with new data when all elements for computing all output elements in a plane have been read by the kernel. For example, after all elements in P0 for computing all output elements in P1 have been read from the buffers, all elements in P0 in the buffers can be replaced safely. Since the RTBs are written and read in parallel, synchronization must be enforced to ensure that no elements are replaced that still need to be read.

Assume a block size of four 5×5 planes. The RTBs require at least 25 cycles to store each of the four input planes. The kernel requires 9 cycles to process each of the constituent two planes. Control logic ensures that the kernel does not begin processing each output plane until its three dependent input planes have been loaded into the RTBs.

RTB loading and kernel processing are pipelined; the kernel will be processing with planes P0 through P2 in parallel to P3 being loaded. As such, to avoid replacing data in the RTBs that is still needed, the buffers must have sufficient capacity to hold at least four planes.

The maximum size of each buffer is limited by the available memory resources on the target FPGA. Assuming the Virtex-5 LX330 FPGA on the Convey HC-1, and considering 16 PEs per FPGA and six RTBs and one output FIFO per PE, the maximum size of each buffer is 512×64, whereas the size of 64-bit output FIFO is 2,048 for a 6-point stencil and 1,024 for a 27-point stencil. The size of the buffer enables us to evaluate the performance impact of a relatively large plane.

4.2.4. Proposed 27-Point Stencil Kernel Design. For larger stencils, such as the 27-point stencil, it is not feasible to instantiate 27 RTBs. Instead, we use 9 shared RTBs that require three cycles to read 27 elements into the kernel.

For a relatively large $10 \times 10 \times 4$ block, each plane (P1 or P2) has 64 output elements. It takes 100 (10×10) cycles for RTBs to be loaded with each plane. On the other hand, since three cycles are required to read 27 input elements, the kernel requires 192 (64×3) cycles for each plane (e.g., P1). Since RTB write is faster than RTB read, RTB overflow would eventually occur. If RTB read time can be reduced below 100, then the overflow can be prevented. This can be accomplished by allowing the kernel to read all 27 points in each cycle and adding an additional layer of buffering between the RTBs and kernel.

Figure 7 shows a $3 \times 8 \times 3$ block composed of eight planes. The output element O1 is computed by reading nine input elements of the first plane, followed by nine inputs in the second plane, and then nine inputs in the third plane. When the second or third plane is read from RTBs and delivered to the kernel for computing the first output element O1, it is also used for computing the second output element O2. Thus, it requires 8 cycles to read all input elements for six output elements as compared to 18 cycles. For a $10 \times 10 \times 4$ block, using this technique, the RTB read time is reduced from 192 cycles to 80 cycles for each plane.

4.3. Data Reuse Analysis of 3D Stencils

Our benchmark 6-point stencil is described in pseudocode in Figure 8. As shown in line 4, six references to the array A are located in the innermost loop. The references are

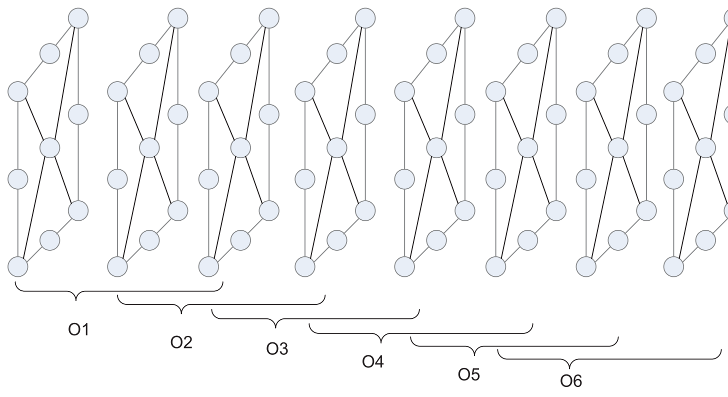

Fig. 7. Stencil computation of six output elements in a 27-point 3D stencil.

```
for i=1 to x-2 do
  for j=1 to y-2 do
    for k=1 to z-2 do
      S[i, j, k] = A[i-1, j, k] + A[i+1, j, k] + A[i, j-1, k] + A[i, j+1, k] + A[i, j, k-1] + A[i, j, k+1];
    endfor
  endfor
endfor
```

Fig. 8. The 6-point 3D stencil loop kernel.

R0(A[i-1, j, k]), R1(A[i+1, j, k]), R2(A[i, j-1, k]), R3(A[i, j+1, k]), R4(A[i, j, k-1]), and R5(A[i, j, k+1]). An array element accessed by one reference in an iteration may be accessed again by other references in other iterations. For example, the array element accessed by the reference R3 in iteration (i, j+1, k) can be reused by the reference R2 in iteration (i, j+2, k).

4.3.1. Ideal Reuse Rate. To evaluate the ideal reuse rate of the stencil kernel assuming a limited buffer capacity, we evaluate the kernel's access pattern against an idealized cache, which has one-word blocks, is fully associative, and has a perfect replacement policy that achieves minimum miss rate. During operation, a block will only be evicted if it is never to be reused. In our experiments, the cache is always large enough that this guarantee can be enforced. For this kernel, this ideal replacement policy can be realized using FIFO ordering. In other words, when the cache cannot hold a new memory reference due to a conflict, the oldest reference is evicted for storing the new reference.

We use this cache model as a means to analyze the best possible data reuse rate for different sizes of on-chip buffers and stencil space. When using this cache model, we assume that words are referenced by the kernel logic using loop order and that the loops and addressing match those from Figure 8.

We calculate reuse rate by counting the total number of reused memory references and dividing by the total number of memory references without data reuse. The reuse rate of the fully associative cache can be computed using Equation (10).

$$\text{Reuse rate of the cache model} = \frac{\textit{total reused references}}{(D-2)\times(D-2)\times(I-2)\times P} \qquad (10)$$

Table I. Relationship between Buffer Size and Reuse Rate

		Reuse Rate (6-point stencil)		Reuse Rate (27-point stencil)	
Buffer Size (Bytes)	Plane Size $D \times D$	*Numerical Ordering*	*Fully Associative Cache*	*Numerical Ordering*	*Fully Associative Cache*
512	4×4	0.33	0.50	0.85	0.85
1K	5×5	0.53	0.61	0.90	0.90
2K	8×8	0.62	0.72	0.92	0.93
4K	11×11	0.75	0.76	0.94	0.94
8K	16×16	0.78	0.79	0.95	0.95
16K	22×22	0.79	0.80	0.96	0.96
32K	32×32	0.81	0.81	0.96	0.96
64K	45×45	0.82	0.82	0.96	0.96

Note: The input set has a size of $D \times D \times I$ for 6- and 27-point stencils. Buffer is assumed to behave as a fully associative cache having one-element blocks and FIFO replacement policy. $I = 512$.

where I is the depth of the input space, P is the number of stencil points, and D is the plane size, which is sized as a function of the buffer size to guarantee that four planes can fit in the buffer, $D = \sqrt[2]{\frac{buffer\ size}{word\ size \times 4}}$.

4.3.2. Actual Reuse Rate. For numerical ordering, comparing with the number of memory references without data reuse, we can compute the actual reuse rate using Equation (11). $D \times D \times I$ is the number of memory references in the numerical order.

$$\text{Reuse rate of the numerical ordering} = 1 - \frac{D \times D \times I}{(D-2) \times (D-2) \times (I-2) \times P} \qquad (11)$$

Table I compares the reuse rates of 6- and 27-point stencils using the numerical ordering and the cache for the input set size $D \times D \times I$.

The results of the cache model are optimal, as each cache can hold four $D \times D$ planes for stencil computation initially, and it always evicts the oldest data that will not be used in the future when new data needs to be loaded into the cache. In the 6-point stencil, the reuse rate of the numerical ordering is more than 10% lower than that of the cache model when the buffer size is smaller than 4K. As the buffer size further increases, the reuse rates are almost the same. For a small buffer that corresponds to a small plane size, the number of referenced input data in the numerical ordering that is not used for the stencil computation decreases the reuse rate. According to Equation (10), this effect gradually reduces as the buffer size increases. In the 27-point stencil, however, the reuse rates of the numerical ordering are very close to those of the cache model regardless of the buffer size, because all referenced input data is used for the 27-point stencil computation.

On the other hand, when I increases from 4 to 512, the reuse rate increases for both numerical ordering and cache model. For each I, the reuse rate of the numerical ordering will gradually reach that of the cache model when the plane size and buffer size increase.

5. EXPERIMENTAL RESULTS

In this section, we present and discuss the experimental results of the 6-point and 27-point 3D stencils using 64 PEs across four FPGAs on the Convey HC-1.

5.1. Performance Impact of the Loop and Array Orders (No Reuse)

We evaluate the performance of 3D stencils with the loop and array orders using a 3D space of $512 \times 512 \times 512$. The workload is divided along one of I, J, and K dimensions.

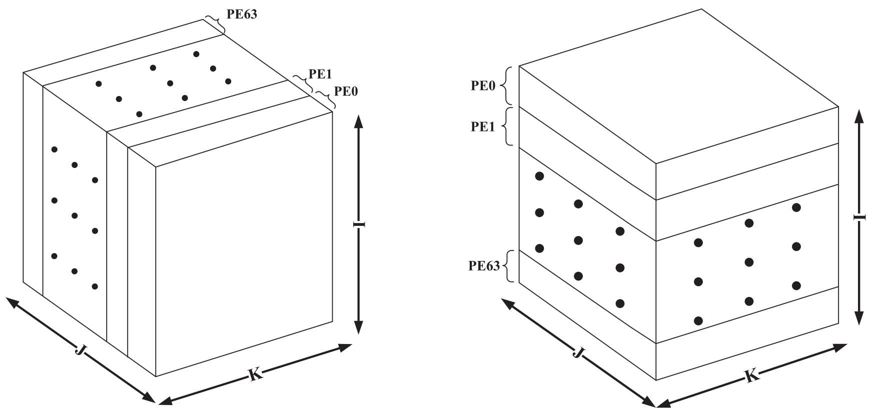

Fig. 9. Partitions along the J (left) or I (right) dimensions in 3D stencil space.

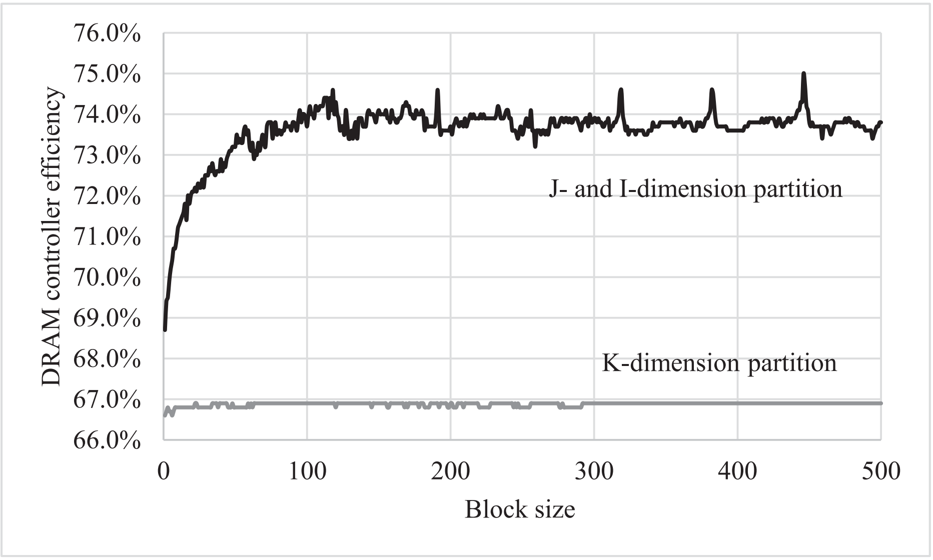

Fig. 10. DRAM controller efficiency versus block sizes in the 6-point 3D stencil.

5.1.1. The 6-Point Stencil. Figure 9 illustrates the workload partitions of stencil space along the J dimension (left) and I dimension (right). For the J-dimension partition, each PE is assigned a chunk of size 512 × 8 × 512 ($k = 512, j = 8, I = 512$) except PE63, which is assigned a chunk of size 512 × 6 × 512. For the I-dimension partition, each PE is assigned a chunk of size 512 × 512 × 8 except PE63, which is assigned a chunk of size 512 × 512 × 6. We assume that the stencil computation for each PE is based on the algorithm shown in Figure 8, where the innermost loop index is k and the outmost loop index is i.

Figure 10 shows the DRAM controller efficiency results of the 6-point stencil when the stencil space is partitioned along I, J, and K dimensions. As mentioned before, loop order corresponds to a block size of 1, whereas the array order has parameterized block sizes. For each partitioning, we use loop order and array order when accessing input elements from DRAM.

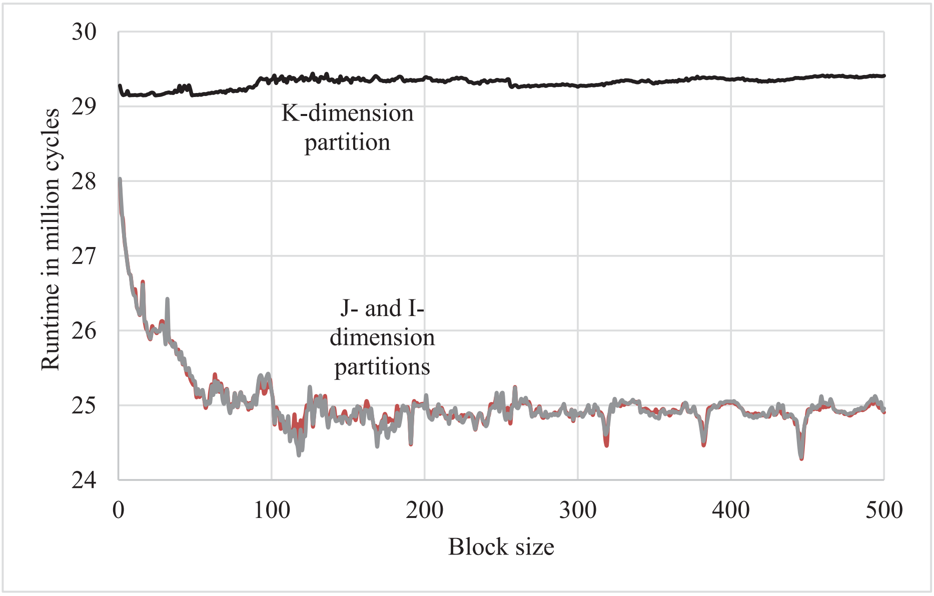

Fig. 11. Runtime vs. block sizes in the 6-point 3D stencil.

Figure 11 compares the average runtime in million cycles of all 16 PEs in an AE for the three partitions in the 6-point stencil. The minimum runtime is approximately 24.3 million cycles with a block size of 446, which represents a 1.15 speedup over the baseline design for the 6-point stencil.

The interface efficiency is nearly 100% regardless of how the stencil space is partitioned and the block sizes.

5.1.2. The 27-Point Stencil. We apply the same partitions to the 27-point stencil space, and results are shown in Figures 12 and 13. As with the 6-point stencil, the interface efficiency is nearly 100% for each partition.

As shown in Figure 12, DRAM controller efficiency of the K-dimension partition is approximately 67% for larger block sizes. DRAM controller efficiency of the I- and J-dimension partition increases to 72% until memory is exhausted at a block size of 27.

Figure 13 compares the runtime in million cycles of the three partitions in the 27-point stencil. The runtime of I-dimension partition is about 1% to 4% higher than the runtime of the J-dimension partition. The minimum runtime is about 104.5 million cycles when the block size is 27 using J-dimension partitioning, corresponding to a speedup of 1.60 as compared to using a block size of 1 using I-dimension partitioning.

5.1.3. Discussion. The 3D arrays are stored in DRAM using K-major ordering, meaning that reading data along the K dimension while keeping I and J constant will result in reading consecutive words from DRAM.

Regardless of which dimension is used for partitioning, each PE will still read data into its FIFOs along the K dimension (i.e., using array ordering). The partitioning strategy affects how many consecutives elements can be read into the FIFOs.

The partitioning strategy determines the K-dimension depth of the subset of data assigned to each PE. In K-dimension partitioning, each PE is assigned a block having a depth of 512/64 in the K dimension. In I- and J-dimension partitioning, each PE is assigned a block having a depth of 512 in the K dimension. For this reason, I/J partitioning always performs best for the 6-point stencil.

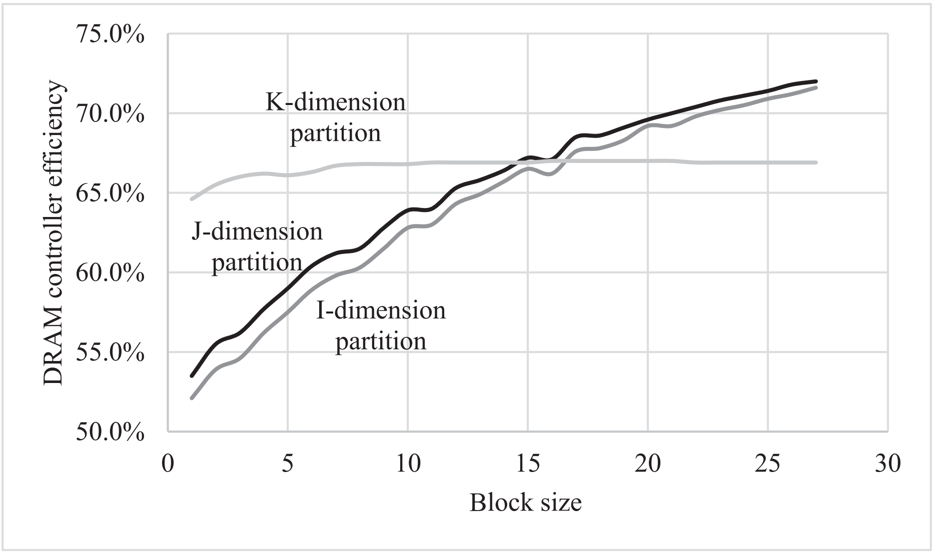

Fig. 12. DRAM controller efficiency versus block sizes in the 27-point 3D stencil.

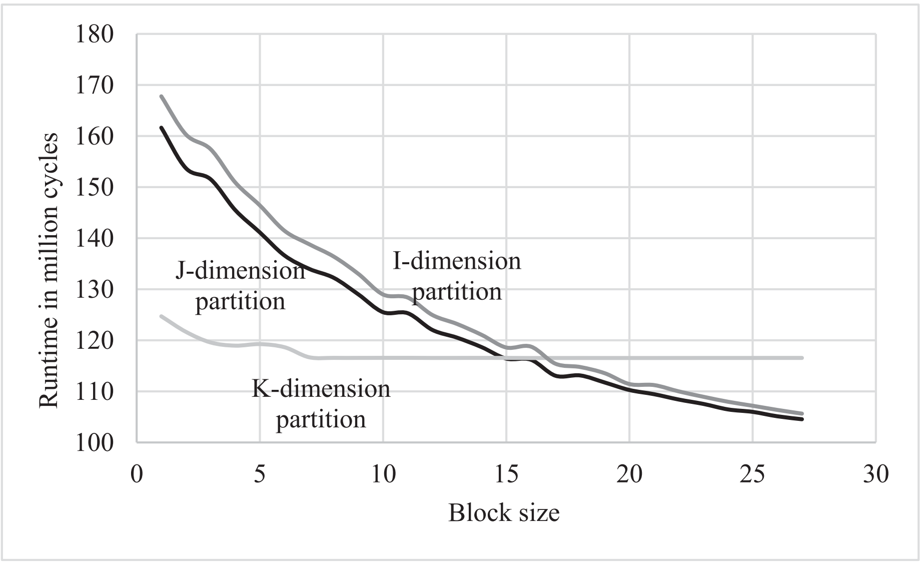

Fig. 13. Runtime versus block sizes in the 27-point 3D stencil.

Due to resource constraints, the 27-point stencil has smaller FIFOs and is limited to smaller buffer sizes (block sizes of 1 to 27, as compared to 1 to 500 for the 6-point stencil). Only larger block sizes (>17), allow the 27-point stencil design to take advantage of the longer runs in the K dimension offered by I/J partitioning.

5.2. Performance Impact of Numerical Order

We evaluate the performance of proposed memory interface designs for both stencils using the numerical ordering by a slightly larger 3D space of $514 \times 514 \times 512$. We

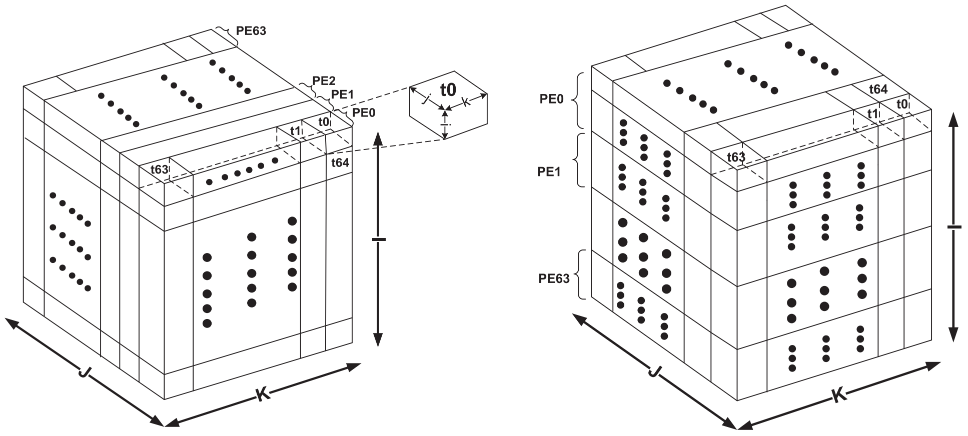

Fig. 14. Block partitions along the J and I dimensions in 3D stencil space.

adjust the stencil space from $512 \times 512 \times 512$ to $514 \times 514 \times 512$ so that each of 64 PEs can request a block composed of a number of 10×10 planes of elements without introducing load imbalance in the j or k dimension of the block.

As shown in Figure 14, we also divide the stencil space along I, J, or K dimension for workload assignment to 64 PEs. For the I-dimension partitions, each PE is assigned a chunk of size $514 \times 514 \times 10$ except PE63, which is assigned a chunk of size $514 \times 514 \times 8$. Each chunk is further divided into blocks. The size of each block (e.g., t0) is $j \times k \times i$, where $j = k = 10$. For each PE (e.g., PE0), it requests a block of elements in the order from block t0, through block t63, to block t64 until all blocks have been accessed in the chunk. The size of a block in the i dimension is a parameter that correlates with the data reuse. The larger the value of the parameter, the fewer the number of blocks that are accessed in the chunk, increasing the reuse rate. We choose the value of i such that the number of planes that contain the output points in the block is integer divisible by the number of planes that contain the output points in the chunk. In our experiment, the values of i are 4, 5, 7, 8, 32, 53, 87, 104, 172, 257, and 512 for the J- and K-dimension partitions, whereas they are 4, 6, and 10 for the I-dimension partition.

The proposed design achieves almost 100% interface efficiency for the three partitions. The reuse rate, as shown in Figure 15, increases from 0.41 to 0.63 for the J- and K-dimension partitions when the i dimension of the block increases from 4 to 512. For the I-dimension partition, the reuse rate increases from 0.41 to 0.58 when the i dimension of the block increases from 4 to 10.

As shown in Figure 16, although DRAM controller efficiency results are similar for the three partitions, the K-dimension partition yields slightly higher efficiency than the J- or I-dimension partition when the i dimension of the block is valid for the three partitions. In addition, small i dimensions of the block have more impact on the efficiency than large i dimensions of the block. When the i dimension of the block is greater than 32, the efficiency stays approximately at 66%.

Figure 17 shows that when the i dimension of the block increases from 4 to 512, the runtime decreases approximately from 18 million cycles to 11 million cycles. However, the runtime is almost the same when the i dimension of the block is greater than 32 for the J- and K-dimension partitions. On the other hand, the runtime results of the K-dimension partition are slightly lower than those of J- or I-dimension partition for each i dimension of the block.

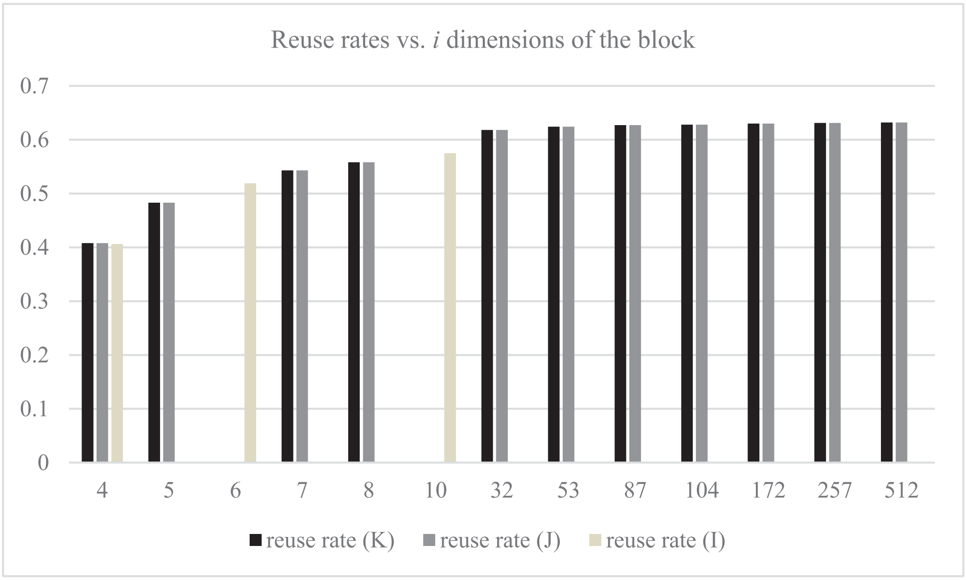

Fig. 15. Reuse rates of the proposed memory interface for the 6-point 3D stencil.

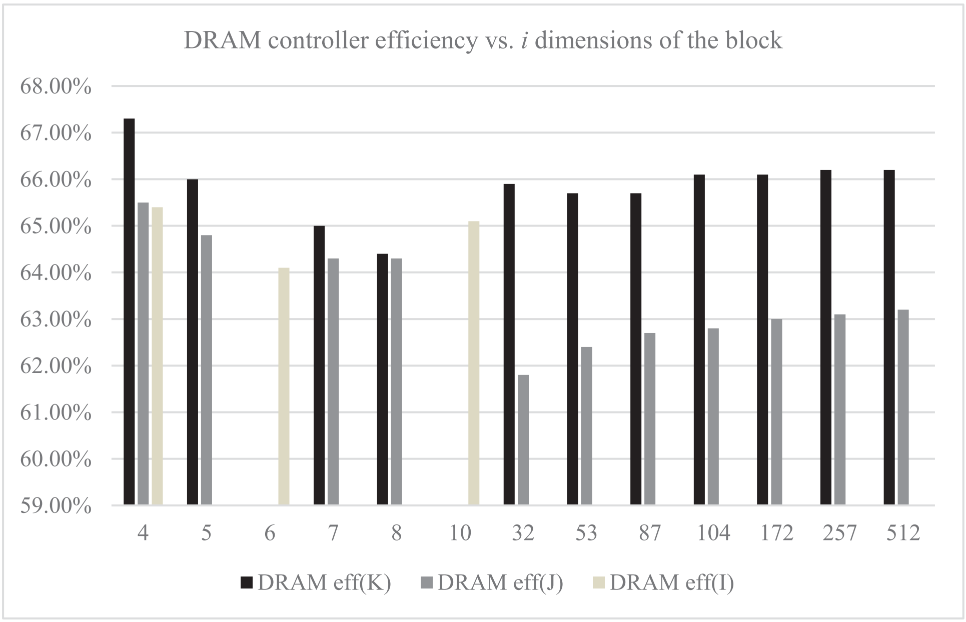

Fig. 16. DRAM controller efficiency of the proposed memory interface for the 6-point 3D stencil.

To evaluate the impact of reuse, we define the *ideal speedup* as the ratio of the number of memory accesses without data reuse to the number of memory accesses with data reuse. Note the total number of memory requests without data reuse is $510 \times 510 \times 8 \times 7$. For each i dimension of the block, the actual speedup is calculated by the ratio of the lowest runtime when the block size is 446 in the I-dimension partition to the corresponding runtime in Figure 17.

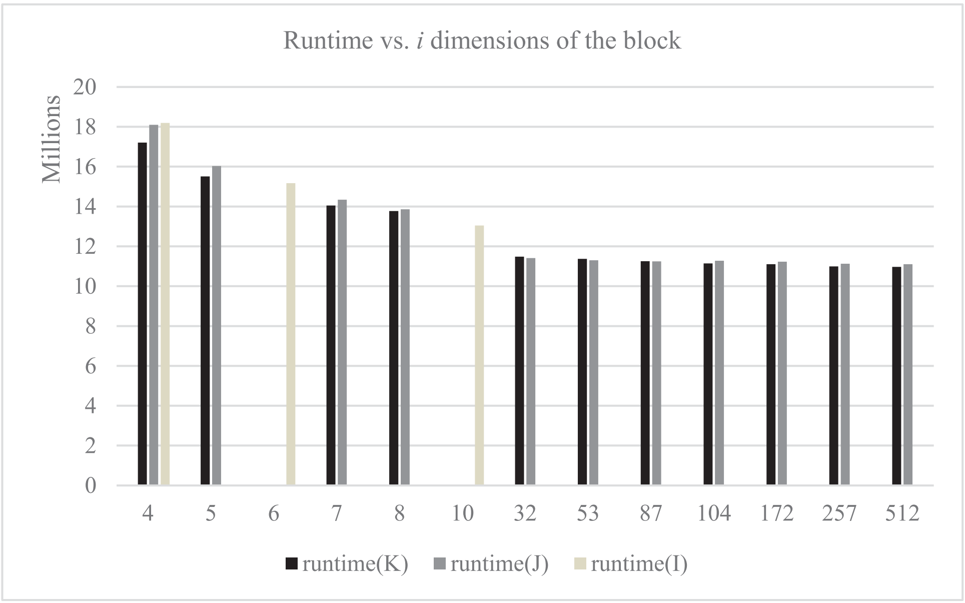

Fig. 17. Runtime of the proposed memory interface for the 6-point 3D stencil.

Table II. Comparison of the Ideal and Actual Speedup in the Proposed Memory Interface for the 6-Point 3D Stencil

i Dimension	Ideal Speedup	Actual Speedup	Actual Speedup vs. Ideal Speedup
4	1.69	1.41	83.48
5	1.93	1.57	80.92
7	2.19	1.73	79.03
8	2.26	1.76	77.99
32	2.61	2.12	80.91
53	2.66	2.14	80.35
87	2.68	2.16	80.46
104	2.69	2.18	81.06
172	2.70	2.19	80.96
257	2.71	2.21	81.56
512	2.71	2.21	81.58%

Table II compares the ideal and best actual speedup of 6-point stencil. With the increase of i value, the ideal speedup increases from 1.7 to 2.7, whereas the actual speedup increases from 1.4 to 2.2. The actual speedup achieves on average 80.76% of the ideal speedup.

Figure 18 compares DRAM controller efficiency results of the proposed 6-point 3D stencil design using the numerical order with the best DRAM controller efficiency using the array order. The best DRAM controller efficiency is 74%, as described in Section 5.1. Since DRAM controller efficiency results across all i dimensions of the block are close, we average the DRAM efficiency results in Figure 16. As shown in Figure 18, the averaged DRAM controller efficiency of the K-dimension partition is about 11% lower than the best.

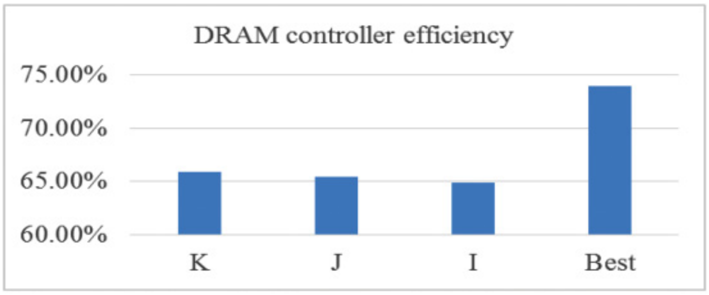

Fig. 18. Comparison of DRAM controller efficiency results for the 6-point 3D stencil.

Table III. Comparison of the Ideal and Actual Speedup in the Proposed Memory Interface for the 27-Point 3D Stencil

i Dimension	Ideal Speedup	Actual Speedup	Actual Speedup vs. Ideal Speedup
4	6.76	6.00	88.73
5	7.74	6.75	87.17
7	8.75	7.44	85.00
8	9.05	7.65	84.57
32	10.46	9.13	87.28
53	10.63	9.18	86.40
87	10.73	9.25	86.20
104	10.76	9.31	86.59
172	10.81	9.36	86.57
257	10.83	9.46	87.29
512	10.86	9.48	87.26%

The interface and DRAM controller efficiency results of the 27-point stencil design are similar to those of the 6-point stencil design, as the access patterns of the numerical order are the same. Because the results of the K-dimension partition are best in the proposed 6-point stencil design, we choose the same partition and compare the results with the best results in Section 5.1.

Table III compares the ideal and actual speedup of the proposed 27-point stencil design across all i dimensions. As defined before, the ideal speedup is the ratio of the number of memory accesses without data reuse to the number of memory accesses with data reuse for each PE. The total number of memory requests without data reuse is $510 \times 510 \times 8 \times 28$. For each i dimension of the block, the actual speedup is calculated by the ratio of the lowest runtime when the block size is 27 in the I-dimension partition to the corresponding runtime in Figure 17. With the increase of i value, the ideal speedup increases from 6.76 to 10.86, whereas the actual speedup increases from 6 to 9.48. The actual speedup achieves on average 86.64% of the ideal speedup.

As described in Section 5.1, the best DRAM controller efficiency for the 27-point stencil is 72% when the block size is 27. The averaged DRAM controller efficiency of the K-dimension partition is 65.8%, which is about 8.6% lower than the best, although the reuse rate is 0.9.

5.3. Resource Utilization

Table IV summarizes the resource utilization of the 6-point and 27-point stencils with the loop, array, and numerical orders when the stencil space is divided along the K dimension. All designs are compiled using the Xilinx ISE 14.4 with the same

Table IV. Resource Utilization of the 6-Point and 27-Point 3D Stencils

	6-Point Stencil			27-Point Stencil		
Resource	Loop (%)	Array (%)	Numerical (%)	Loop (%)	Array (%)	Numerical (%)
Slice	83	84	81	98	98	88
Slice register	57	58	53	72	73	58
Slice LUT	47	48	48	78	80	62
BRAM	80	80	80	97	97	86

constraints and settings for synthesis, map, place, and route. The resource utilization results of dividing the space along the other two dimensions are not listed, as they are approximately equal to those shown. As shown in the table, the proposed design with the numerical order is more resource efficient in terms of the number of slices, slice registers, slice LUTs, and block RAMs (BRAMs). Note that DSP48E macros are not used, as all arithmetic operations are implemented with slice LUTs. The 27-point stencil design with the loop or array order utilizes 98% of total FPGA slices due to the large number of 64-bit adders and the FSM controller in the address generator. For the 6-point stencil, the BRAM utilization is 80% for the three orderings. For the 27-point stencil, the BRAM utilization is 97% for the loop and array orderings, and 86% for the numerical ordering.

6. CONCLUSIONS

This article presents a set of custom memory interface designs on an FPGA-based platform for 3D stencil kernels. The target platform—the Convey HC-1—is a multi-FPGA platform whose memory system performs dynamic access scheduling but presents the programmable logic with three critical challenges: (1) its memory system does not perform caching or prefetching, (2) its memory performance depends on the access order given by the programmable logic, and (3) its DRAM latency is high and must be explicitly hidden in the programmable logic. To reconcile these problems and maximize overall memory performance, the interface of each kernel should provide explicit support for latency hiding, memory access scheduling, and data reuse.

To hide memory latency, we use address generation and large FIFO- and rotation-based buffers. For memory access scheduling, we propose three memory access orders—loop order, array order, and numerical order—to evaluate the effect of access scheduling on memory performance. Data reuse is realized by each PE reading 3D blocks composed of a set of planes. To combine memory latency hiding, memory access ordering, and data reuse, we develop custom circular buffering techniques, custom address generators, and kernel implementations for both stencils.

Our experimental results reveal a complex trade-off in the space of data reuse and scheduling. Using million output stencil points per second (MP/s) to measure throughput, we obtain 818MP/s and 190MP/s for the 6- and 27-point stencils with the array order and 1,823MP/s for both stencils with the numerical order. Taking into account DRAM controller efficiency and interface efficiency, array order achieves maximum effective memory bandwidth of 56.8GB/s and 55.3GB/s for the 6- and 27-point stencils, respectively, whereas numerical order achieves maximum effective bandwidth of 50.7GB/s. Compared to the loop and array orders with no data reuse, the reuse rate ranges from 0.33 to 0.82 and from 0.85 to 0.96 for the 6- and 27-point 3D stencils, respectively, for the numerical order. Although the number of consecutive accesses can improve the DRAM controller efficiency, on-chip memory constraints limit the number of consecutive accesses that can be achieved in the array and numerical orders. Despite this, we carefully adjust the sizes of the stencil space, FIFO, and RTB to achieve best performance results under the resource constraints.

REFERENCES

J. H. Ahn, N. P. Jouppi, C. J. Kozyrakis Leverich, and R. S. Schreiber. 2009. Future scaling of processor-memory interfaces. In *Proceedings of the Conference on High Performance Computing Networking, Storage, and Analysis (SC'09)*. Article No. 42.

W. Augustin, J. Weiss, and V. Heuveline. 2011. Convey HC-1 Hybrid Core Computer-The Potential of FPGAs in numerical simulation. In *Proceedings of the Second International Workshop on New Frontiers in High-Performance and Hardware-Aware Computing (HipHaC'11)*. San Antonio, Texas, USA.

R. Banakar, S. Steinke, and B. Lee. 2002. Scratchpad memory design alternative for cache on-chip memory in embedded systems. In *Proceedings of the 10th International Symposium on Hardware/Software Codesign (CODES'02)*. 73–78.

N. Baradaran and P. C. Diniz. 2008. A compiler approach to managing storage and memory bandwidth in configurable architectures. *ACM Transactions on Design Automation of Electronic Systems* 13, 4, Article No. 61.

Y. Ben-Asher and N. Rotem. 2010. Automatic memory partitioning: Increasing memory parallelism via data structure partitioning. In *Proceedings of the 2010 IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. 155–162.

L. Benini, L. Macchiarulo, A. Macii, and M. Poncino. 2002. Layout-driven memory synthesis for embedded systems-on-chip. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 10, 2, 96–105.

H. K. Chang and Y. L. Lin. 2000. Array allocation taking into account SDRAM characteristics. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC'00)*. 497–502.

J. Cong, H. Huang, C. Liu, and Y. Zou. 2011a. A reuse-aware prefetching scheme for scratchpad memory. In *Proceedings of the 48th Design Automation Conference (DAC'11)*. 960–965.

J. Cong, M. Huang, and Y. Zou. 2011b. 3D recursive Gaussian IIR on GPU and FPGAs: A case study for accelerating bandwidth-bounded applications. In *Proceedings of the 9th IEEE Symposium on Application Specific Processors*. 201.

J. Cong, W. Jiang, B. Liu, and Y. Zou. 2011c. Automatic memory partitioning and scheduling for throughput and power optimization. *ACM Transactions on Design Automation of Electronic Systems* 16, 2, Article No. 15.

J. Cong, P. Zhang, and Y. Zou. 2011d. Combined loop transformation and hierarchy allocation in data reuse optimization. In *Proceedings of the 2011 International Conference on Computer-Aided Design (ICCAD'11)*. 185–192.

Convey Corporation. 2012. *Convey Personality Development Kit Reference Manual*. Retrieved August 24, 2015, from http://www.conveysupport.com/alldocs/ConveyPDKReferenceManual.pdf.

K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. 2008. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. IEEE, Los Alamitos, CA, 1–12.

Z. Fang, X. H. Sun, Y. Chen, and S. Byna. 2009. Core-aware memory access scheduling schemes. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS'09)*. 1–12.

C. He, M. Lu, and C. Sun. 2004. Accelerating seismic migration using FPGA-based coprocessor platform. In *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'04)*. IEEE, Los Alamitos, CA, 207–216.

C. He, G. Qin, M. Lu, and W. Zhao. 2006. An efficient implementation of high-accuracy finite difference computing engine on FPGAs. In *Proceedings of the International Conference on Application-Specific Systems, Architectures, and Processors (ASAP'06)*. IEEE, Los Alamitos, CA, 95–98.

C. He, W. Zhao, and M. Lu. 2005. Time domain numerical simulation for transient waves on reconfigurable coprocessor platform. In *Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines.* IEEE, Los Alamitos, CA, 127–136.

W. K. C. Ho and S. J. E. Wilton. 2004. Logical-to-physical memory mapping for FPGAs with dual-port embedded arrays. In *Field Programmable Logic and Applications*. Lecture Notes in Computer Science, Vol. 1673. Springer, 111–123.

I. Issenin, E. Brockmeyer, M. Miranda, and N. Dutt. 2007. DRDU: A data reuse analysis technique for efficient scratch-pad memory management. *ACM Transactions on Design Automation of Electronic Systems* 12, 2, Article No. 15.

Z. Jin and J. D. Bakos. 2013. Memory access scheduling on the Convey HC-1. In *Proceedings of the 21st IEEE International Symposium on Field-Programmable Custom Computing Machines*. 237.

M. Kandemir, J. Ramanujam, M. J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. 2004. A compiler-based approach for dynamically managing scratch-pad memories in embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 23, 2, 243–260.

S. Liu, S. O. Memik, Y. Zhang, and G. Memik. 2008. A power and temperature aware DRAM architecture. In *Proceedings of the Design Automation Conference (DAC'08)*.

C. G. Lyuh and T. Kim. 2004. Memory access scheduling and binding considering energy minimization in multi-bank memory systems. In *Proceedings of the Design Automation Conference (DAC'04)*.

P. R. Panda, N. D. Dutt, and A. Nicolau. 1997. Efficient utilization of scratch-pad memory in embedded processor applications. In *Proceedings of the 1997 European Conference on Design and Test (EDTC'97)*. 7.

S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. 2008. Memory access scheduling. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA'00)*. 128–138.

Y. Tatsumi and H. Mattausch. 1999. Fast quadratic increase of multiport-storage-cell area with port number. *Electronics Letters* 35, 25, 2185–2187.

Y. Wang, P. Li, P. Zhang, C. Zhang, and J. Cong. 2013. Memory partitioning for multidimensional arrays in high-level synthesis. In *Proceedings of the 50th Annual Design Automation Conference (DAC'13)*. ACM, New York, NY, Article No. 12.

Y. Wang, P. Zhang, X. Cheng, and J. Cong. 2012. An integrated and automated memory optimization flow for FPGA behavioral synthesis." In *Proceedings of the 2012 17th Asia and South Pacific Design Automation Conference (ASP-DAC'12)*. 257–262.

Received November 2014; revised May 2015; accepted June 2015