IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 16 September 2025, accepted 6 October 2025, date of publication 13 October 2025, date of current version 23 October 2025.

*Digital Object Identifier 10.1109/ACCESS.2025.3620475*

RESEARCH ARTICLE

# Optimization and Implementation of DRAM-Based Interleavers for Free-Space Optical Communication Beyond 100 Gbit/s

**LUKAS STEINER, (Graduate Student Member, IEEE), UWE WASENMÜLLER, (Member, IEEE), AND NORBERT WEHN, (Senior Member, IEEE)**
Microelectronic Systems Design Research Group, University of Kaiserslautern-Landau, 67663 Kaiserslautern, Germany
Corresponding author: Lukas Steiner (lukas.steiner@rptu.de)

This work was supported in part by the German Federal Ministry of Research, Technology, and Space (BMFTR) under Grant 16KIS1608 (FACTOR).

**ABSTRACT** Optical transmission links from low earth orbit (LEO) satellites to earth require interleaving to ensure reliable data transmission. With data rate requirements beyond 100 Gbit/s and channel coherence times in the millisecond range, the capacity of on-chip memory (SRAM) is insufficient to store the symbols of one full interleaving window, and external memory (DRAM) must be used instead. However, with DRAM, bandwidth utilization depends heavily on the access pattern. In the present application case of the right triangle interleaver, a two-dimensional array in the shape of an isosceles triangle is accessed both row-wise and column-wise. When this array is mapped to linear memory addresses in either row-major or column-major order, the bandwidth utilization for one of the two access directions is far below the theoretical maximum. This limits the overall throughput of the interleaver. We present an optimized mapping solution that is tailored to the internal DRAM architecture and minimizes the factors that restrict bandwidth utilization in both access directions simultaneously. Furthermore, this mapping solution can be applied to other problem instances involving alternating row- and column-wise accesses, including the more commonly used block interleaver. On the demonstrator platform, the AMD Alveo U280 accelerator card, the worst-case bandwidth utilization across both access directions is increased by 2.45 $\times$ for the DDR4 DRAM and 1.38 $\times$ for the HBM2 DRAM, while the overhead in FPGA resources is negligible. The complete interleaver design is then implemented on the platform, achieving a maximum data rate of 131.1 Gbit/s with the optimized mapping.

**INDEX TERMS** Address mapping, Alveo, DRAM, HBM, interleaving, optical communication.

## I. INTRODUCTION

Optical transmission links from *low earth orbit* (LEO) satellites to earth have experienced strong growth in commercial satellite communication in recent years [1], [2]. They are particularly suitable for transmitting large amounts of data because they offer significantly higher data rates (>100 Gbit/s) compared to conventional radio transmission and operate in a freely available signal spectrum. However, due to atmospheric effects, which are very effective at high frequencies as in optical transmission links, the communication channel is highly error prone and experiences large dynamics in the channel quality and maximum possible data rates. To ensure reliable data transmission, powerful error correction methods with advanced channel codes must be employed. One important step required for a high error-correction performance is interleaving, which mitigates the large dynamics in channel quality. With data rate requirements beyond 100 Gbit/s, channel coherence times in the millisecond range [3], and a minimum interleaving time interval that must be a multiple of the coherence time, very large data blocks in the range of hundreds of megabytes to several gigabytes must be stored. This exceeds the memory capacity of SRAM, which is typically used to implement

The associate editor coordinating the review of this manuscript and approving it for publication was Xueqin Jiang.

interleavers. The only memory type that can provide both sufficient capacity and bandwidth is DRAM. However, unlike SRAM, the achievable bandwidth utilization of DRAM is highly dependent on the pattern of accessed addresses. Sequential or strided accesses (i.e. with a fixed increment) achieve high bandwidth utilization because only a few bits are toggled between consecutive addresses. Random accesses, on the other hand, achieve much lower bandwidth utilization because the number of toggled bits is much higher, resulting in time-consuming row misses within the DRAM. In the present case, we opt for a right triangle interleaver, which has shown promising results in initial studies [4]. It uses a two-dimensional logical memory array in the shape of an isosceles triangle. For interleaving, multiple consecutive code words are first written to the array row-wise and then read from the array column-wise. When the two-dimensional index space is mapped to linear DRAM addresses in row-major order, the row-wise access results in a sequential pattern, while the column-wise access results in a non-unit strided pattern with many address bit toggles. The same problem applies to column-major order. This has a negative impact on bandwidth utilization and significantly reduces the overall throughput of the interleaver. To meet data rate requirements, the theoretical peak bandwidth of the DRAM must be greatly oversized, resulting in higher costs and power consumption.

In this paper, we present a novel optimized mapping of the right triangle interleaver to DRAM. Unlike the row-major or column-major order, it achieves high bandwidth utilization in both access directions by taking into account the internal DRAM architecture. The mapping works with any JEDEC-compliant DRAM device (e.g. DDR, LPDDR or HBM) and it can also be applied to other problem instances involving alternating row- and column-wise accesses. This includes the more common block interleaver, which uses a memory array in the shape of a rectangle instead of an isosceles triangle. To the best of our knowledge, we are the first to investigate the efficient implementation of right triangle interleavers that are based on DRAM. In our previous paper [5], we introduced a first version of the optimized mapping and presented first simulation results. This work makes the following new contributions:

- We introduce a complete mathematical model of the mapping.
- We present a new optimization step that eliminates the need for costly multiplications in the address calculation.
- We evaluate the increase in bandwidth utilization by measurements on real hardware using the DDR4 and HBM2 DRAM of the AMD Alveo U280 accelerator card.
- On the same device, we implement the complete interleaver design, once with the DDR4 DRAM and once with the HBM2 DRAM. In addition, we analyze the maximum throughput, utilization of FPGA resources and memory overhead of the optimized mapping.

The remainder of the paper is organized as follows. Section II provides the reader with the necessary background on DRAM and interleaving. Section III discusses relevant related work. In Section IV, the optimized mapping is derived step by step. The increase in bandwidth utilization of the optimized mapping is evaluated in Section V. Afterwards, the implementation of the complete interleaver design is presented in Section VI. Finally, Section VII concludes the paper.

## II. BACKGROUND

This section begins with a brief introduction to DRAM and interleaving, followed by the challenges of implementing a right triangle interleaver with DRAM.

### A. DRAM ARCHITECTURE AND FUNCTIONALITY

DRAM is a type of memory optimized for high density and low cost per bit. To achieve this high density, a DRAM device uses an advanced internal architecture. Memory cells, which consist of a storage capacitor and an access transistor, are densely packed and organized in *rows* and *columns* with shared sensing logic. The DRAM device is operated by an external memory controller. Before data can be accessed, the target row must be activated (sensed) first. This activation process is initiated by the memory controller, which sends an activate command over the command/address bus. Afterwards, columns of the row can be read or written by issuing read and write commands. Accesses to a row that is already active are called row hits and experience a short latency. If data from a different row should be accessed, a precharge and an activate of the new target row have to be initiated first by the memory controller. These internal operations take a relatively long time to complete. Thus, accesses to an inactive row, also called row misses, experience a longer latency.

DRAM devices use a prefetch technique to compensate for the difference in operating speed between the slow memory core and the much faster interface. With each access, multiple columns are fetched in parallel from the memory array to the interface. At the interface, the data is serialized and transferred over the narrower data bus in a burst fashion. To fully utilize the data bus, i.e., to achieve the maximum transfer bandwidth, prefetches must always be performed seamlessly, which is not possible when row misses occur. For this reason, all modern DRAM devices are internally partitioned into multiple concurrently operating *banks*, which allows hiding the latency caused by row misses and increasing the overall data bus utilization. Each bank has its own sensing logic, i.e., it can perform activations and precharges independently from all other banks. Only the I/O circuitry that connects the device to the memory controller is shared by all banks. To maximize performance, DRAM traffic should be distributed across multiple banks so that if a row miss occurs in one bank, a prefetch can still be performed from another bank.

The data interface of most DRAM devices is very narrow, such as 4, 8, or 16 bits. By connecting multiple devices to the same memory controller and sharing the command/address bus among them, a DRAM *channel* with a wider data bus (e.g. 64 bits) and higher maximum bandwidth is formed. A common organization is the *dual in-line memory module* (DIMM), where multiple DRAM devices are soldered onto a small printed circuit board that is plugged into a socket for connection to the memory controller. *High Bandwidth Memory* (HBM) is a special type of DRAM that uses a silicon interposer to connect to the memory controller. This allows higher pin count devices to be realized, e.g. with 16 channels, 64-bit data buses per channel and a total of 1024 data pins.

Newer DRAM standards such as DDR4/5, LPDDR5 or HBM internally partition multiple banks into so-called *bank groups*. With older standards (e.g. DDR3 or LPDDR4), the prefetch was identical to the ratio of interface speed to core speed. For DDR4, this would have resulted in a minimum prefetch of 16 and 128 bytes of data transferred per access when keeping the 64-bit data bus per channel from previous generations. In contrast, the typical processor cache line size is only 64 bytes. To maintain a prefetch of 8 at twice the interface speed, bank groups were introduced. Each bank group consists of multiple banks and uses its own I/O gating, while the device has a global I/O gating that is shared by all bank groups. This allows prefetches in different bank groups to happen in parallel and the data bus to be fully utilized, even though a prefetch takes more time than a burst transfer. For maximum bandwidth utilization, it is essential to switch bank groups after each access.

Another factor that negatively impacts DRAM performance is refresh. Each DRAM cell stores 1 bit of information in the form of a charged or discharged capacitor. Over time, capacitors leak their charge, so a DRAM cell must be refreshed every 32 to 64 ms. Refresh operation is triggered by the memory controller at the appropriate time by issuing special refresh commands. Internally, the refresh commands successively activate each row of the device, which couples the cells to the sense amplifiers and recharges them. Depending on the standard, refresh can be performed at different granularities, i.e. all banks at once (*all-bank refresh*, supported by all standards), only a single bank (*per-bank refresh*, supported e.g. by LPDDR or HBM), or one bank per bank group (*same-bank refresh*, supported by DDR5 only). Since the target bank(s) cannot be accessed during the refresh, the performance is negatively affected. This is particularly noticeable with the all-bank refresh.

### B. DRAM CONTROLLER AND ADDRESS MAPPING

As explained in the previous section, DRAM is internally partitioned into rows, columns, banks and - for newer standards - bank groups. To access a specific location in DRAM, the memory controller issues appropriate DRAM command sequences for the read and write accesses it received at the front end. For proper device operation, it ensures that both the order of commands and the timing between commands are in compliance with the respective standard that is issued by JEDEC. It also maps the system physical addresses to DRAM addresses consisting of a row, column, bank and bank group. An optional task of the memory controller is to optimize the DRAM performance in terms of bandwidth utilization or access latency, e.g. by minimizing row misses and evenly utilizing all banks and bank groups. This can be achieved either by reordering the incoming requests or by determining an optimized address mapping. Reordering is done on the fly by a so-called *scheduler* and does not require prior knowledge of the application's memory access scheme. However, the scheduler works on a limited window of accesses and adds additional hardware complexity to the memory controller. When an application's memory access scheme is fixed and known in advance, an optimized address mapping can be determined and implemented with minimal hardware overhead. The effectiveness of this optimization depends heavily on the received access pattern. For a sequential or strided pattern, the toggling rates of the physical address bits differ greatly. DRAM performance is optimized by mapping physical address bits with high toggling rates to bank group, bank, or column bits, and physical address bits with low toggling rates to row bits in the DRAM address. In contrast, when the access pattern is random, all physical address bits experience identical toggling rates and no optimized mapping can be determined.

### C. INTERLEAVING

Interleaving is a common technique in coding theory to improve the reliability against burst errors. The error correction capability of many channel codes is designed for random errors. As an example, a Hamming code can correct one erroneous bit in each code word. When a burst errors occurs and multiple consecutive bits of the same code word are received erroneous, the correction fails. One solution to this problem is to move to a more complex group of codes (e.g. Reed-Solomon codes) that can correct multi-bit errors. However, this usually goes hand in hand with a more complex decoding algorithm and a lower code rate, i.e. more redundancy. An alternative is to apply interleaving. Before transmitting the data over a noisy channel, a window of multiple consecutive code words is permuted such that encoded bits belonging to the same code word are not placed directly one after the other on the channel, but are spread over a larger time distance. After transmission, the inverse permutation (deinterleaving) is performed. Ideally, this transforms burst errors that occur during transmission into random errors placed in different code words, which can be corrected individually afterwards. Fig. 1 shows an example where three code words of a (7,4) Hamming code (marked in red, blue and brown) are interleaved. As a result, a three-bit burst error marked with dashed lines can be corrected after transmission, while it leads to uncorrectable errors in the non-interleaved case. The penalties of interleaving are a hardware overhead and an increase in latency because decoding can

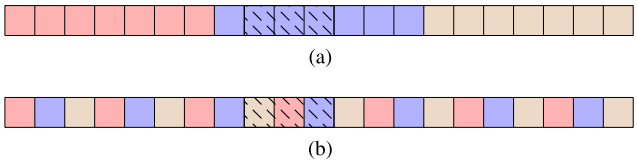


**FIGURE 1.** Example of interleaving to correct burst errors: (a) Non-correctable burst error without interleaving, (b) Correctable burst error with interleaving.

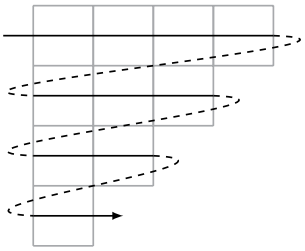

**FIGURE 2.** Row-wise access to isosceles triangle memory array.

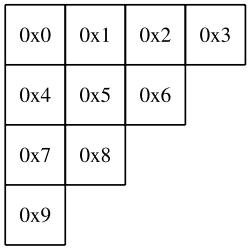


**FIGURE 3.** Addresses with row-major order.

only start after all code words of one interleaving window have been received.

One common interleaver type is the block interleaver. Multiple consecutive code words are first written to a rectangular logical memory array row-wise and afterwards read from the array column-wise. For deinterleaving, the access directions of writing and reading are swapped. In order to interleave a continuous stream of data, two memory arrays are required. While the input data stream is written to the first one, the output data stream is read from the second one. After completion, the memories are swapped like in double buffering. A similar interleaver type, e.g. employed in 5G New Radio, is the right triangle interleaver [6], [7], [8], [9]. It uses a memory array in the shape of an isosceles triangle instead of a rectangle. Fig. 2 shows the row-wise access order to such an array.

### D. DRAM-BASED RIGHT TRIANGLE INTERLEAVER

In the present application case, interleaving is used to mitigate the large channel quality dynamics of the optical transmission link, which are caused by atmospheric effects [4]. The coherence time of the channel, i.e., the time during which the channel quality can be assumed as constant, can range from below 1 ms to more than 20 ms depending on the weather conditions and the receiver aperture size [3]. As an example, with a data rate requirement of 100 Gbit/s, two additional soft information bits used for decoding, and a coherence time of 2 ms, the interleaver has to achieve a throughput of at least 300 Gbit/s and more than 70 MiB of data are received within one coherence time. In order to achieve sufficient balance in channel quality, interleaving must be performed over an interval that is a multiple of the channel coherence time [4]. The resulting data blocks that need to be stored are in the range of hundreds of megabytes up to several gigabytes. This exceeds the memory capacity of SRAM, which is typically used to implement block interleavers. The only memory alternative that offers higher capacities, but can still handle the target data rates is DRAM. However, when mapping the two-dimensional logical memory array of the right triangle interleaver to linear DRAM addresses in row-major order as shown in Fig. 3, the row-wise access leads to a sequential address pattern, while the column-wise access leads to non-unit strided accesses (the other way round for column-major order). The stride is decremented by one after each access, resulting in high toggling rates of all address bits. As explained in Section II-B, no optimized mapping from system physical address bits to DRAM address bits can be determined in this case although the access scheme is fixed and known in advance. The solution we propose in this work is a direct mapping of the two-dimensional index space to DRAM addresses, which maximizes bandwidth utilization both for row-wise and column-wise accesses simultaneously. Fig. 4 shows a block diagram of the DRAM-based interleaver. It uses an address generator to calculate the addresses of the optimized mapping, and two DRAM channels, one for writing and one for reading, as a single DRAM channel cannot be written and read at the same time. The channels are swapped after one interleaving window is completely written and read.

An additional constraint imposed by using DRAM instead of SRAM is its much higher access granularity of at least 32 or 64 bytes per burst (see Section II-A). While an ideal interleaver works on individual symbols, i.e. in our case, one bit for the binary symbol and two additional bits for the soft information that is added at the receiver side,[1] this cannot be done with DRAM. As a solution, interleaving is divided into two stages. First, a small block interleaver realized in SRAM interleaves a window of multiple DRAM bursts such that symbols within one DRAM burst belong to different code words. Afterwards, the interleaving with larger granularity is done. The performance of the two-stage interleaver has been analyzed and compared to the ideal right triangle interleaver and a random interleaver in [4].

## III. RELATED WORK

As explained in Section II-D, the basic problem is to find a mapping from a two-dimensional index space to the DRAM address space that maximizes bandwidth both for row-wise and column-wise accesses. The right triangle interleaver is a

[1] Placeholder bits need to be added already before interleaving such that deinterleaving with the added soft information restores the original symbol order.

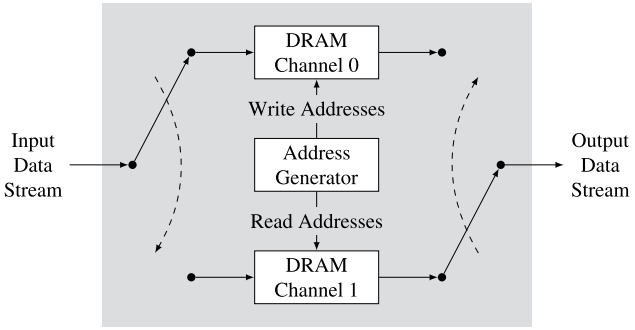


**FIGURE 4.** Block diagram of DRAM-based interleaver.

**TABLE 1.** Summary of related work.

| Paper | Target Application | Standard | Bandwidth Utilization | Performance Improvement |
|---|---|---|---|---|
| [10] | SAR processing | SDRAM | 91 % | 2.6 × |
| [11] | SAR processing | DDR2 | 91 – 95 % | 1.33 – 4.73 × |
| [12] | SAR processing | DDR | — | 2 × |
| [13] | data reorganization | 3D DRAM | — | up to 2 × |
| [14] | MPEG-2 video proc. | SDRAM | 99 % | — |
| [15] | 2D FFT | DDR | — | 1.22 × |
| [16] | SAR processing | SDRAM | 82 % | 1.32 × |
| [17] | 2D FFT | DDR2 | 90 % | 3.44 – 16.8 × |
| [18] | SAR processing | DDR | 96 – 98 % | — |
| [19] | 2D FFT | DDR3 | — | 3.9 – 4.5 × |
| [20] | 2D FFT | 3D memory | 28.8 – 40.0 % | 1.95 – 1.97 × |
| [21] | image processing | DDR3 | 86 – 95 % | 1.6 – 9.0 × |
| [22] | image processing | DDR3 | 97 – 98 % | up to 8.9 × |
| [23] | CPU & GPU benchmarks | DDR4, GDDR5 | — | 1.63 – 1.91 × |
| [24] | in-mem. databases, matrix-matrix mul. | DDR3 GS-DRAM | — | 2.0 – 3.0 × |

special case with an index space in the shape of an isosceles triangle. Alternating row-wise and column-wise accesses to large two-dimensional logical memory arrays are common in signal processing, for example, when working with a matrix and its transpose. The optimized mapping to DRAM is therefore a well-known research problem. By taking the internal DRAM architecture (i.e. banks, rows and columns) into consideration, large increases in bandwidth utilization were achieved in [10], [11], [12], and [13]. The resulting performance improvement and energy reduction was shown for different applications such as MPEG-2 video decoding [14] and 2D FFT [15], [16], [17], [18], [19], [20]. The studies were also carried out on different platforms with various DRAM standards (e.g. DDR [12], [15], [16], DDR2 [11], [17], DDR3 [19] or 3D DRAM [13], [20]). However, since all the papers relied on older DRAM standards, they did not consider the impact of bank groups. In addition, all optimizations were only done for rectangular memory arrays. Thus, the presented solutions cannot be applied to the application case of this work.

In [21], the minimization of DRAM row misses was formulated as a mathematical optimization problem. This way, the optimal mapping can be found for arbitrary applications with a fixed access pattern. Since the problem is proven to be NP-hard, it cannot be solved for huge instances, i.e. long access patterns. Therefore, the authors developed a heuristic to find a solution. Still, there exist two problems. First, as in the other works, no bank groups were considered. Second, since the mapping can be any possible bijective function, it must be implemented with a lookup table. This only makes sense for small address ranges and memory capacities, since the lookup table needs one entry for each input address. In [22], a more feasible solution was presented that uses a configurable address scrambler realized with multiplexers instead of a lookup table for the mapping. The scrambler can be implemented in hardware with low complexity, but it only allows to map one specific input address bit to one specific output address bit. This restricts the range of possible mappings significantly and only works well for access strides that are powers of two. However, when the triangular memory array is mapped to DRAM in row-major order and accessed column-wise, the stride decreases by one with each access, i.e., it is not always a power of two. In [23], a reinforcement learning approach was proposed to generate a binary invertible matrix for address mapping. The learning can be done both offline with prerecorded memory access traces and online at runtime where the mapping is changed over time and data is migrated. Although binary invertible matrices are a generalization of the configurable address scrambler, they can only express linear mappings over the Galois field $GF(2)^n$, so it is still not possible to handle non-unit strides.

The authors of [24] specifically addressed the problem of non-unit strided accesses, which also appears when processing in-memory databases. Their solution was to use a custom DRAM subsystem architecture where each device of the channel can access a different address. However, it can also only handle different power-of-two strides. In addition, we opt for a solution that works with off-the-shelf DRAM subsystems without requiring any modifications. Table 1 summarizes the aforementioned related work by showing the target application, the DRAM standard used, the maximum bandwidth utilization achieved, and the overall performance improvement.

Reference [25] presents an implementation of a DRAM-based block interleaver as part of a complete optical satellite uplink. It is based on the AMD Virtex 7 FPGA VC709 and uses DDR3 DRAM. However, the targeted data rate is only 10 Gbit/s instead of 100 Gbit/s and no details on the mapping in DRAM are provided. The work also suggests contemplating new memory technologies, such as HBM, for operation at higher data rates.

## IV. OPTIMIZED MAPPING

This section first formulates the search for an optimized mapping as a mathematical optimization problem and explains why finding the optimal solution is impractical. Afterwards, it explains the individual optimization steps that are performed.

### A. PROBLEM FORMULATION

Mathematically, the two-dimensional interleaver index space and the DRAM address space are two sets, which we refer to as $I$ and $D$, respectively. For an isosceles triangle with right-side length $s_{tri}$, the number of elements of the triangle is $n_{tri} = |I| = \frac{s_{tri} \cdot (s_{tri}+1)}{2}$. Each element of $I$ can be described by an $x$ and $y$ coordinate with $x \in \{0, 1, 2, \ldots, s_{tri} - 1\}$, $y \in \{0, 1, 2, \ldots, s_{tri} - 1\}$ and $x + y \in \{0, 1, 2, \ldots, s_{tri} - 1\}$. For DRAM, we make the assumption that older standards have a single bank group that includes all banks to avoid distinguishing between standards with and without bank groups. In addition, we predefine that the lower part of the bank index determines the bank group and the upper part determines the bank within the group. Each element of the DRAM address space $D$ is described by a bank index $i_{bank}$, a column index $i_{col}$ and a row index $i_{row}$. For a DRAM with $n_{bank}$ banks (which are distributed across one or multiple bank groups), $n_{col}$ columns and $n_{row}$ rows, it is $i_{bank} \in \{0, 1, 2, \ldots, n_{bank} - 1\}$, $i_{col} \in \{0, 1, 2, \ldots, n_{col} - 1\}$, $i_{row} \in \{0, 1, 2, \ldots, n_{row} - 1\}$ and $|D| = n_{bank} \cdot n_{col} \cdot n_{row}$. A mapping can be seen as an injective function $f$ that maps each element of $I$ to an element of $D$, i.e. $f : I \to D,\ (x, y) \mapsto (i_{bank}, i_{col}, i_{row})$ with $|I| \leq |D|$. Further, we define the set of all injective functions with the domain $I$ and codomain $D$ as $F$. The number of possible mappings can be calculated as $n_{map} = |F| = \frac{|D|!}{(|D|-|I|)!}$. To differentiate between the different mappings, each function $f$ is provided with an index $j$, i.e. $F = \{f_j \mid j \in \{0, 1, 2, \ldots, n_{map} - 1\}\}$.

During the process of interleaving and deinterleaving, the two-dimensional index space is written row-wise, read column-wise, written column-wise and read row-wise. The achieved bandwidth utilizations for these access schemes depend on the chosen mapping $f_j$ and are specified as $b_{RD,RW}(f_j)$ (read row-wise), $b_{RD,CW}(f_j)$ (read column-wise), $b_{WR,RW}(f_j)$ (write row-wise), and $b_{WR,CW}(f_j)$ (write column-wise). Since the maximum throughput of the interleaver is determined by the minimum bandwidth utilization across all four access schemes, our optimization problem is the following:

$$\underset{j \in \{0,1,2,\ldots,n_{map}-1\}}{\arg\max} \min\big(b_{RD,RW}(f_j), b_{RD,CW}(f_j), b_{WR,RW}(f_j), b_{WR,CW}(f_j)\big) \quad (1)$$

There are several difficulties in solving this problem.

- An exhaustive search is not feasible for determining the optimal mapping because the number of possible mappings is $n_{map} = \frac{|D|!}{(|D|-|I|)!} = \frac{(n_{bank} \cdot n_{col} \cdot n_{row})!}{(n_{bank} \cdot n_{col} \cdot n_{row} - n_{tri})!}$.
- There is no analytical model to calculate the DRAM bandwidth utilization for a given mapping due to the high internal complexity of DRAM subsystems.
- In [21], it was proven that already the problem of minimizing row misses for an arbitrary access pattern is NP-hard.
- Some mappings cannot be expressed as a computable function, which means they could only be implemented in hardware as a lookup table. However, the size of such a lookup table would not be feasible.

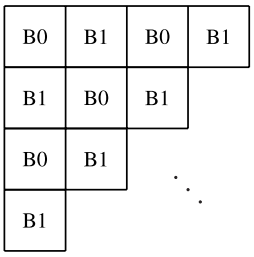


**FIGURE 5.** Example layout of bank indices for 2 banks.

We therefore take a heuristic approach. Each factor that limits DRAM bandwidth utilization is first addressed individually and a solution is determined. Afterwards, all solutions are combined into the final mapping. At the same time, the need for a low complexity hardware implementation is always kept in mind. The two Sections IV-B and IV-C independently show how to maximize the bank parallelism (inter-bank level) and minimize the row misses (intra-bank level). Section IV-D combines these optimizations. In order to reduce the hardware complexity, Section IV-E introduces an index transformation that eliminates the need for multiplications in the address calculation. Section IV-F focuses on optimizing the remaining row misses. Finally, Section IV-G explains how the interleaver can be implemented with more than two DRAM channels.

### B. MAXIMIZING BANK PARALLELISM

As explained in Section II-A, it is important to equally distribute DRAM traffic across all banks (and bank groups) to hide the latency of row misses and perform prefetches in parallel. For an ideal distribution of traffic across all banks, the bank index should be incremented after each access. In our application scenario, this needs to be the case when accessing the two-dimensional memory array both row-wise and column-wise. A diagonal layout for the bank indices as shown in Fig. 5 fulfills this requirement.

### C. MINIMIZING ROW MISSES

Within each bank, the number of row misses should be minimized. For an interleaver where each element of the memory is written or read only once per access phase, this is the case if all columns of an active DRAM row are addressed consecutively before a new row is activated. However, when this optimization is applied to one access direction, the row misses are maximized in the other access direction. The optimal solution with respect to our optimization problem is to balance the row misses between the two access directions. This is achieved with the layout shown in Fig. 6, where the columns are organized in a square shape.

Each square represents a complete DRAM row. The number of DRAM columns $n_{col}$ is evenly divided into the number of columns in the row-wise access direction $n_{col,x}$

| C0 | C1 |
|---|---|
| C2 | C3 |

**FIGURE 6.** Example layout of column indices for 4 columns.

| | | | |
|---|---|---|---|
| B0<br>C0<br>R0 | B1<br>C0<br>R0 | B0<br>C1<br>R0 | B1<br>C1<br>R0 |
| B1<br>C0<br>R1 | B0<br>C0<br>R1 | B1<br>C1<br>R1 | B0<br>C1<br>R1 |
| B0<br>C2<br>R0 | B1<br>C2<br>R0 | B0<br>C3<br>R0 | B1<br>C3<br>R0 |
| B1<br>C2<br>R1 | B0<br>C2<br>R1 | B1<br>C3<br>R1 | B0<br>C3<br>R1 |

**FIGURE 7.** Example layout of resulting basic block.

and in the column-wise access direction $n_{col,y}$ such that

$$n_{col,x} = n_{col,y} = \sqrt{n_{col}} = 2^k, k \in \mathbb{N} \tag{2}$$

holds. Powers of two result in a low complexity hardware implementation. If the number of columns is not a square number (i.e. the number of column address bits is odd), the highest column address bit is treated as a row address bit. This does not negatively impact the maximum throughput of the interleaver because the additional column bit would only increase the bandwidth utilization in one of the two access directions.

### D. FORMING BASIC BLOCKS

The optimizations from the previous two sections are now combined into what we call *basic blocks*. When accessing a basic block both in row- and column-wise order, the bank is switched with each access and all accesses target a different column of the same DRAM row. The side length of a basic block for a DRAM device with $n_{bank}$ banks and $n_{col} = n_{col,x} \cdot n_{col,y}$ columns is

$$s_{block} = n_{bank} \cdot n_{col,x} = n_{bank} \cdot n_{col,y}. \tag{3}$$

As this equations shows, one basic block contains $n_{bank} \cdot n_{bank} \cdot n_{col}$ different elements, while the number of different elements of a single row across all banks is only $n_{bank} \cdot n_{col}$. Thus, a basic block also needs to be formed of $n_{row,block} = n_{bank}$ different rows. An example of a basic block is shown in Fig. 7. Bank, column and row index 0 are color coded for a better understanding. Note that all accesses to a specific bank always target the same row in both directions. With the basic block defined, the formulas for bank index $i_{bank}$, column index $i_{col}$ and row index $i_{row}$ as functions of an $x$ and $y$ coordinate are derived as follows.

$$i_{bank} = (x + y) \mod n_{bank} \tag{4}$$

$$i_{col} = \left\lfloor \frac{y}{n_{bank}} \right\rfloor \cdot n_{col,x} + \left\lfloor \frac{x}{n_{bank}} \right\rfloor \tag{5}$$

$$i_{row} = y \mod n_{bank} \tag{6}$$

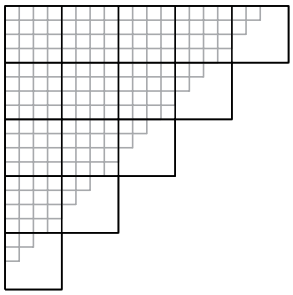

**FIGURE 8.** Right triangle interleaver covered by 4 × 4 basic blocks.

| | | | | |
|---|---|---|---|---|
| R0 | R1 | R3 | R6 | R10 |
| R2 | R4 | R7 | R11 | |
| R5 | R8 | R12 | | |
| R9 | R13 | | | |
| R14 | | | | |

**FIGURE 9.** Possible layout of outer row indices for triangle.

Since $n_{bank}$ as well as $n_{col,x}$ are powers of two, the modulo operations, multiplication, and integer divisions (divisions with flooring) can be implemented in hardware at zero cost.

When considering real interleavers and DRAM devices, the side length of a single basic block is much smaller than the side length of the complete interleaver. Thus, the two-dimensional index space has to be covered by many basic blocks as shown in Fig. 8. At the hypotenuse of the triangle, the basic blocks are not completely filled, resulting in a small memory overhead. In order to maintain unique addressing, the row indices used in the basic blocks are varied. We distinguish between an inner row index $i_{row,inner}$ (within one basic block, calculated with the formula of (6)) and an outer row index $i_{row,outer}$ (outside of basic blocks). The combined row index $i_{row}$ is calculated as

$$i_{row} = i_{row,outer} \cdot n_{row,block} + i_{row,inner}. \tag{7}$$

Because the number of row indices within a basic block $n_{row,block}$ is equal to the number of banks $n_{bank}$, and in current DRAM devices the number of banks is usually a power of two, this calculation translates to a concatenation in hardware. While (4) still holds for the new case, (5) has to be adapted to limit the range of possible column indices.

$$i_{col} = \left\lfloor \frac{y \mod s_{block}}{n_{bank}} \right\rfloor \cdot n_{col,x} + \left\lfloor \frac{x \mod s_{block}}{n_{bank}} \right\rfloor \tag{8}$$

The hardware complexity remains unchanged because $s_{block}$ is a power of two.

For the outer row indices, there exist multiple possible layouts. One option is to increment the indices diagonally starting from the upper left corner of the triangle as shown in

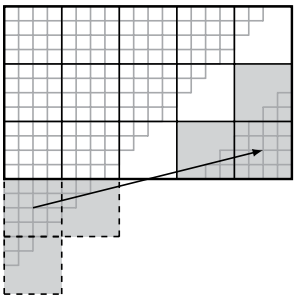

**FIGURE 10.** Index transformation from triangle to rectangle.

| | | | | |
|---|---|---|---|---|
| R0 | R1 | R2 | R3 | R4 |
| R5 | R6 | R7 | R8 | R9 |
| R10 | R11 | R12 | R13 | R14 |

**FIGURE 11.** Layout of outer row indices for rectangle.

Fig. 9. Alternatively, the indices can be counted up row-wise or column-wise.

### E. INDEX TRANSFORMATION

Because of the triangular shape, all layouts for the outer row indices proposed in the previous section result in a complex calculation that involves multiplying two variables. This is shown exemplarily for the layout of Fig. 9. First, the $x$ and $y$ coordinates in relation to basic blocks instead of individual interleaver elements are calculated using the formulas

$$x_{block} = \left\lfloor \frac{x}{s_{block}} \right\rfloor \tag{9}$$

and

$$y_{block} = \left\lfloor \frac{y}{s_{block}} \right\rfloor. \tag{10}$$

Afterwards, the outer row index can be calculated as

$$i_{row,outer} = \frac{(x_{block} + y_{block}) \cdot (x_{block} + y_{block} + 1)}{2} + y_{block}. \tag{11}$$

This multiplication makes it difficult to meet the timing requirements in hardware. To use a layout that does not require the multiplication of two variables, we transform the triangular index space into a rectangular index space. As shown in Fig. 10, the bottom tip of the triangle is mirrored in the empty space of a surrounding rectangle.

In a first step, we calculate the number of basic blocks the rectangle contains in the row-wise and column-wise access directions, called $n_{block,x}$ and $n_{block,y}$. The number of elements of an isosceles triangle with a right-side length $s_{tri}$ is

$$n_{tri} = \frac{s_{tri} \cdot (s_{tri} + 1)}{2}. \tag{12}$$

This formula can also be written as

$$n_{tri} = \frac{s_{tri}}{s_{block}} \cdot s_{block} \cdot \frac{s_{tri} + 1}{2 \cdot s_{block}} \cdot s_{block}. \tag{13}$$

The number of elements of the resulting rectangle with side lengths $s_{rect,x}$ and $s_{rect,y}$ is

$$n_{rect} = s_{rect,x} \cdot s_{rect,y} = n_{block,x} \cdot s_{block} \cdot n_{block,y} \cdot s_{block}. \tag{14}$$

Because the number of elements of the rectangle must be greater than or equal to the number of elements of the triangle and $n_{block,x}$ as well as $n_{block,y}$ should be integers to simplify calculations in hardware, we define them as

$$n_{block,x} = \left\lceil \frac{s_{tri}}{s_{block}} \right\rceil \tag{15}$$

and

$$n_{block,y} = \left\lceil \frac{s_{tri} + 1}{2 \cdot s_{block}} \right\rceil. \tag{16}$$

Next, the coordinates of the triangular index space $x$ and $y$ are transformed into coordinates of the rectangular index space $\bar{x}$ and $\bar{y}$ with

$$\bar{x} = \begin{cases} s_{rect,x} - 1 - x & \text{if } y \geq s_{rect,y} \\ x & \text{else} \end{cases} \tag{17}$$

and

$$\bar{y} = \begin{cases} 2 \cdot s_{rect,y} - 1 - y & \text{if } y \geq s_{rect,y} \\ y & \text{else} \end{cases} \tag{18}$$

while the new block coordinates $\bar{x}_{block}$ and $\bar{y}_{block}$ are calculated analogous to (9) and (10). Finally, when using the row-wise layout shown in Fig. 11, the outer row index is calculated as

$$i_{row,outer} = \bar{y}_{block} \cdot n_{block,x} + \bar{x}_{block}. \tag{19}$$

This calculation is much less complex than (11). Starting from the original coordinates $x$ and $y$, the calculation of $i_{row,outer}$ consists of two parallel comparisons and conditional subtractions of one constant and one variable, one multiplication of a variable with a constant and one final addition of two variables. Since the index transformation also changes the bank, column and inner row indices of the moved positions, (4), (8) and (6) have to be updated to rectangle coordinates $\bar{x}$ and $\bar{y}$ as well. As a side effect, in some cases the index transformation reduces the memory overhead that is caused by covering the interleaver index space with basic blocks. Furthermore, the optimized mapping can be applied directly to other problem instances with a rectangular index space. This includes the more common block interleaver, which is also used in [25], and the remaining applications discussed in Section III.

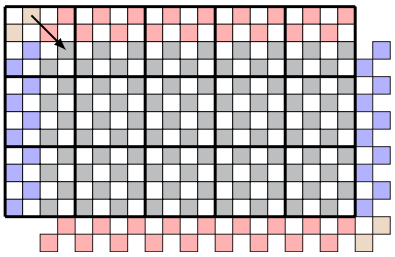

**FIGURE 12.** Bank-dependent shift.

### F. STAGGERING ROW MISSES OVER TIME

With the mapping proposed in the previous sections, the traffic is distributed across all banks and the row misses are minimized. However, there is still an unresolved issue that prevents optimal utilization of the DRAM bandwidth. Because the size of a DRAM row is limited, row misses still occur when moving from one basic block to the next. These row misses occur almost simultaneously on all banks, meaning that all banks are busy with precharging and activating a new row at the same time, and no data accesses can be performed in the meantime. A solution to this problem is to stagger the row misses on different banks over time. This can be achieved by introducing a bank-dependent column offset, i.e., the change from one basic block to the next is done with a time offset on each bank. All positions of the two-dimensional index space are shifted to the bottom right by an offset that depends on the bank index they are mapped to. The column indices $i_{col}$ and row indices $i_{row}$ are calculated with the shifted coordinates $\bar{x}^*$ and $\bar{y}^*$ instead of the rectangle coordinates $\bar{x}$ and $\bar{y}$. These coordinates depend on the bank index and are calculated as follows:

$$\bar{x}^* = (\bar{x} + (i_{bank} \cdot n_{bank} \mod s_{block})) \mod s_{rect,x} \quad (20)$$

$$\bar{y}^* = (\bar{y} + (i_{bank} \cdot n_{bank} \mod s_{block})) \mod s_{rect,y} \quad (21)$$

The maximum shift amount is limited to $s_{block}$ because the column indices repeat in each basic block and the row indices are irrelevant from a performance point of view. Furthermore, the shifting is done circular, i.e. positions that are shifted out of the rectangle at the bottom/right edge are moved back in at the top/left edge. Fig. 12 illustrates the shifting for an example interleaver with two banks. All colored elements (bank index 1) are shifted two positions to the bottom right, while all blank elements (bank index 0) remain in the same position. Circular shifts are highlighted in red, blue and brown, respectively.

### G. MULTI-CHANNEL CONSIDERATIONS

As explained in Section II-D, at least two DRAM channels are required to interleave a continuous data stream. While the input data is written to one channel, the output data is read from the other channel. After each interleaving window, the channels are swapped. In order to reach a throughput of at least 300 Gbit/s, the DRAM's interface speeds must be very high. As an example, channels with a 64-bit data bus (e.g. DDR4 DIMMs or HBM2) would require interface speeds of at least 4687.5 MT/s, while with only 32 bits per channel (e.g. DDR5 DIMMs), the requirements would double to 9375 MT/s. Additionally, these calculations assume a bandwidth utilization of 100 %, which is challenging to reach even with the optimized mapping as will be shown later in this work. If the available devices do not achieve such high interface speeds, more than two channels must be used. This multi-channel case can be implemented in two different ways. First, multiple channels can be treated like a single channel with a wider data bus. This approach results in low implementation complexity, but it increases the access granularity. Second, multiple channels can be treated like a single channel with more banks. The lowest part of the bank index then determines the channel, the middle part the bank group within the channel and the uppermost part the bank within the bank group. This keeps the access granularity the same, but increases the size of the basic blocks and the routing overhead as will be shown in Section VI-B.

## V. EXPERIMENTAL EVALUATION

In this section, we quantify the performance improvements of our presented optimized mapping compared to the default row-major order. This is first done through simulations of four different DRAM standards. Afterwards, two different DRAM subsystems of an FPGA accelerator card are benchmarked to verify the optimizations on real hardware.

### A. SIMULATION OF DIFFERENT DRAM STANDARDS

The simulations are performed with DRAMSys [26], a simulator that represents a complete DRAM subsystem consisting of memory controller and devices. The communication between controller and devices is modeled cycle-accurate and fully compliant with the simulated JEDEC standard. In addition to different device configurations (speed grade, capacity, etc.), DRAMSys also offers various configuration options for the controller to mimic the behavior of real hardware controllers. To stimulate the DRAM subsystem, a custom traffic generator issues read or write requests to the DRAM with one of the four following access patterns:

- Row-wise accesses with row-major order (results in sequential addresses)
- Column-wise accesses with row-major order (result in non-unit strided addresses)
- Row-wise accesses with optimized mapping
- Column-wise accesses with optimized mapping

This allows us to determine the average DRAM bandwidth utilization for the interleaver's access patterns and the resulting maximum throughput with and without optimizations. The simulated DRAM configurations are provided in Table 2. They comprise four different standards, namely DDR4, DDR5, LPDDR4 and LPDDR5. The impact of interface speed is analyzed by simulating both a slow and a fast speed grade of each standard. HBM and GDDR devices

**TABLE 2. Simulated DRAM configurations.**

| DRAM Configuration | Transfer Rate (MT/s) | #Bank Groups | #Banks | #Columns | #Rows | Data Bus Width (bit) | Burst Length | Capacity (GiB) | Refresh Type |
|---|---|---|---|---|---|---|---|---|---|
| DDR4-1600 | 1600 | 4 | 16 | 128 | 65,536 | 64 | 8 | 8 | All-Bank |
| DDR4-3200 | 3200 | 4 | 16 | 128 | 65,536 | 64 | 8 | 8 | All-Bank |
| DDR5-3200 | 3200 | 8 | 32 | 64 | 65,536 | 32 | 16 | 8 | Same-Bank |
| DDR5-6400 | 6400 | 8 | 32 | 64 | 65,536 | 32 | 16 | 8 | Same-Bank |
| LPDDR4-2133 | 2133 | - | 8 | 64 | 65,536 | 32 | 16 | 2 | Per-Bank |
| LPDDR4-4266 | 4266 | - | 8 | 64 | 65,536 | 32 | 16 | 2 | Per-Bank |
| LPDDR5-4267 | 4267 | 4 | 16 | 64 | 65,536 | 32 | 16 | 4 | Per-Bank |
| LPDDR5-8533 | 8533 | 4 | 16 | 64 | 65,536 | 32 | 16 | 4 | Per-Bank |

are missing because it is not possible to simulate them accurately, as the standards do not provide numeric values for the timings. However, their internal architecture is very similar to that of DDR or LPDDR devices, so the results should be transferable. In order to keep the refresh-related bandwidth degradation as low as possible, per-bank refresh or same-bank refresh is preferred over all-bank refresh when it is supported by the standard. For the row-major order, the lowest physical address bits are mapped to the DRAM's bank bits, the middle ones to column bits, and the highest ones to row bits as explained in Section II-B (RCB mapping). The remaining memory controller configuration options are selected to maximize bandwidth utilization, enabling the upper limit to be determined for each DRAM configuration. To make the refresh impact comparable, the number of rows is the same for all devices. The data bus width is chosen such that one burst access always transmits 512 bits of data, since the minimum burst length is either 8 or 16, depending on the standard. For the experiments, the right-side length of the interleaver is set to 4096, which results in $8.4 \cdot 10^6$ elements and a memory requirement of 0.5 GiB. This corresponds approximately to the amount of data received in 14 ms at the target data rate of 100 Gbit/s with two additional soft information bits. Table 3 provides the simulation results. The lower row of each configuration shows the optimized mapping and in each row, the minimum is underlined as it is the relevant metric.

As expected, when the row-major order is accessed row-wise, all configurations achieve a high bandwidth utilization of over 90 %. Due to the RCB mapping, the traffic is evenly distributed across all banks and the number of row misses is minimal. The missing percentages can be traced back to refresh. However, the negative impact of per-bank and same-bank refresh commands (DDR5, LPDDR4 and LPDDR5) is smaller than that of all-bank refresh (DDR4) commands because they do not block the complete device at once (see Section II-A). When the access direction is column-wise, the bandwidth utilization always drops. Depending on the configuration, this drop is more or less significant. All standards have in common that the slow speed grade achieves a higher bandwidth utilization than the fast speed grade. This can be explained by the fact that a faster speed grade only results in faster interface speeds, while the DRAM core speed remains the same. In terms of clock cycles, the internal latencies for row activations and precharges increase. As explained in Section II-D, column-wise accesses of the row-major order lead to many row misses, i.e. activations and precharges. Thus, the negative impact on bandwidth utilization is higher for the fast speed grade. Furthermore, the column-wise accesses with row-major order benefit from a higher bank count (e.g. LPDDR4-4266 vs. LPDDR5-4267) and longer burst length (e.g. DDR4-3200 vs. LPDDR5-4267). Increased bank parallelism enables row misses to be hidden more effectively, while a longer burst length reduces the frequency of row misses. This is especially notable for DDR5, as it is optimized for random traffic due to its primary application in servers. With the optimized mapping, maximum bank parallelism is achieved in both access directions and row misses are equally balanced. This ensures they can be hidden almost perfectly, with a minimum bandwidth utilization of 90 % for all tested configurations. As for the row-major order, all-bank refresh has a larger negative impact than per-bank or same-bank refresh. In conclusion, only DDR5 achieves a bandwidth utilization close to the limits when using the row-major order. With the optimized mapping, however, all DRAM standards do so.

**TABLE 3. Simulated DRAM bandwidth utilization for different standards and speed grades.**

| DRAM Configuration | Read | | Write | |
|---|---|---|---|---|
| | Row-Wise | Col.-Wise | Row-Wise | Col.-Wise |
| DDR4-1600 | 92.61 % | 74.04 % | 92.02 % | 74.06 % |
| | 89.63 % | 89.66 % | 92.00 % | 92.00 % |
| DDR4-3200 | 92.52 % | 43.57 % | 91.80 % | 43.57 % |
| | 90.56 % | 90.54 % | 90.82 % | 90.73 % |
| DDR5-3200 | 100.00 % | 96.22 % | 100.00 % | 94.77 % |
| | 100.00 % | 100.00 % | 100.00 % | 100.00 % |
| DDR5-6400 | 99.90 % | 87.22 % | 99.90 % | 85.07 % |
| | 99.93 % | 99.92 % | 99.65 % | 99.56 % |
| LPDDR4-2133 | 98.82 % | 64.95 % | 98.46 % | 64.74 % |
| | 97.81 % | 97.75 % | 98.45 % | 98.37 % |
| LPDDR4-4266 | 98.94 % | 35.93 % | 98.27 % | 35.69 % |
| | 95.38 % | 95.32 % | 98.72 % | 98.53 % |
| LPDDR5-4267 | 98.46 % | 65.89 % | 97.79 % | 64.60 % |
| | 93.48 % | 93.35 % | 96.03 % | 95.80 % |
| LPDDR5-8533 | 98.88 % | 47.02 % | 98.46 % | 37.78 % |
| | 96.45 % | 96.30 % | 97.50 % | 97.08 % |

### B. MEASUREMENT OF REAL DRAM SUBSYSTEMS

The AMD Alveo U280 accelerator card is used for the hardware implementations. It features an UltraScale+ FPGA and various I/O interfaces, including two DDR4 channels with a total capacity of 32 GiB and a theoretical peak bandwidth of 307.2 Gbit/s as well as 32 HBM2 channels with a total capacity of 8 GiB and a theoretical peak bandwidth of 3.7 Tbit/s [27]. Both DRAM types are benchmarked separately in the following two sections.

**TABLE 4. Measured DDR4 bandwidth utilization on Alveo U280.**

| Address Mapping | Read | | Write | |
|---|---|---|---|---|
| | Row-Wise | Col.-Wise | Row-Wise | Col.-Wise |
| BRC | 30.71 % | 6.34 % | 29.26 % | 4.52 % |
| RBC | 31.55 % | 28.56 % | 31.07 % | 23.97 % |
| RCB | 93.31 % | 39.01 % | 88.57 % | 33.72 % |
| RCB INTLV | 93.71 % | 35.96 % | 89.10 % | 29.30 % |
| RoBaCoBg* | 93.70 % | 35.88 % | 89.15 % | 28.89 % |
| OPT 1 | 25.30 % | 25.30 % | 18.07 % | 18.07 % |
| OPT 2 | 84.14 % | 84.05 % | 82.95 % | 82.75 % |

#### 1) DDR4

The two DDR4 channels support a maximum interface speed of 2400 MT/s and feature 64-bit data buses. Internally, a channel is composed of 16 banks distributed over 4 bank groups, each with 131,072 rows and 128 columns per row. They can be accessed from the FPGA via two distinct memory controllers. Each controller provides an AXI4 slave interface with 512-bit-wide data channels and a maximum operating frequency of 300 MHz. With a proper alignment, therefore, each AXI beat translates into a complete DDR4 burst access of length eight, and the DRAM bandwidth can be fully utilized through the AXI4 slave interface. As in the simulation-based evaluation, a custom traffic generator issues read or write requests using the four access patterns described previously. It is also operated at 300 MHz. The right-side length of the triangle is set to 4096 again. Benchmarking only one channel is sufficient because both are identical. All measurement results are shown in Table 4 and are explained in more detail below.

The DDR4 controller provided by AMD/Xilinx is a soft IP core, i.e., it is built from FPGA resources and can also be replaced with a custom implementation. It supports four different mappings from physical addresses to DRAM addresses. All these mappings were tested separately for the row-major order and the results are shown in the first four rows of the table. BRC (BgBaRoCo) achieves the lowest performance because the highest bits of the physical address are mapped to the bank bits of the DRAM. These bits never toggle as the interleaver with a right-side length of 4096 only uses a small portion of the memory. Section VI-A4 will show this in more detail. Thus, all memory accesses target the same bank. While the row-wise accesses still maximize the number of row hits on that bank, the column-wise accesses also lead to many row misses. This explains the significant performance gap. RBC (RoBgBaCo) also achieves low bandwidth utilizations, but the gap between row-wise and column-wise accesses is much smaller. The row-wise accesses still achieve low bank parallelism and a high row-hit ratio, while the column-wise accesses achieve high bank parallelism and a low row-hit ratio. These two factors roughly balance each other out. RCB (RoCoBaBg) achieves a bandwidth utilization close to the specified limits of the memory controller (94 % for read and 89 % for write [28]) for row-wise accesses because both the bank parallelism and the row-hit ratio are maximized. However, the performance for column-wise accesses is much lower due to the low row-hit ratio. Interestingly, the RCB INTLV mapping, which places one column bit between the bank group and bank bits, achieves an even higher bandwidth utilization for the row-wise accesses although the bank parallelism is lower. The same result can be observed when all the column bits are placed between the bank group and bank bits, which is done by our custom RoBaCoBg* mapping. This effectively reduces the bank parallelism from 16 to 4. When comparing the measured bandwidth utilizations to the simulated ones, the results for row-wise accesses match very closely. For column-wise accesses, the performance of the real DDR4-2400 memory subsystem is below the two simulated speed grades DDR4-1600 and DDR4-3200, not in between as expected.

When switching to the optimized mapping (row OPT 1 in Table 4), the results are not as expected. Although the bandwidth utilization is the same for row-wise and column-wise accesses, it is much lower than the simulation results, and even lower than for the row-major order. The reason for this behavior lies in the internal architecture of the memory controller. It is composed of four so-called *group FSMs*, one per DRAM bank group [28]. They queue up incoming transactions to the four banks within the respective bank group and issue precharge, activate and read/write commands while observing the timing constraints between them. Within a group FSM, requests are not reordered. Only the command outputs of the group FSMs can be reordered in the following unit. In the case that a row miss occurs on one of the four banks within a bank group, all subsequent requests to other banks of the same bank group are postponed until the precharge, activate and read/write command is completed. It is therefore better to use only one of the four banks in each bank group at a time with this controller. For the same reason, the RCB INTLV and RoBaCoBg* mappings slightly outperformed the RCB mapping as was shown in the previous paragraph. The memory controller inside DRAMSys uses a separate FSM for each bank, i.e. in total 16 bank FSMs for DDR4. This allows commands to be reordered on bank granularity, which also explains why the simulation results for column-wise accesses on DDR4-3200 are better than those achieved on the real DDR4-2400 subsystem although the speed grade is higher. In order to circumvent the limitations of the memory controller by AMD/Xilinx,

a slightly adjusted optimized mapping is used. Instead of distributing the traffic across all banks as a first step, it is only distributed across all bank groups. A different bank within the bank group is only accessed when a row miss occurs, because the latency for accessing a different row in the same bank is longer than the latency for accessing a different row in a different, inactive bank. The dimensions of a basic block remain unchanged. Using the adjusted mapping, the results in row OPT 2 are achieved. With a minimum bandwidth utilization across all four access patterns of 82.75 % with and 33.72 % without optimized mapping, it is increased by 2.45 × on the real DDR4 subsystem. These results also come much closer to the simulation results, but are again lower due to the limitations of the memory controller.

### 2) HBM2

The 32 HBM2 channels are so-called pseudo channels, which means that two adjacent channels share the command/address bus, but execute commands independently and have separate data buses. These data buses are 64-bit wide and operate at a maximum speed of 1800 MT/s. Internally, a pseudo channel is composed of 16 banks distributed over 4 bank groups, each with 16,384 rows and 32 columns per row. Each pseudo channel can be accessed through its own AXI3 slave interface, which has 256-bit-wide data channels and a maximum operating frequency of 450 MHz. As the burst length of HBM2 pseudo channels is only four, one AXI beat again translates into a complete HBM2 burst with a proper alignment. The HBM2 controller is a hard IP core with 32 individual channel controllers and an additional $32 \times 32$ crossbar switch. This switch allows to access each pseudo channel from each AXI3 slave interface. Section VI-B will provide more details on the crossbar and its limitations. One limitation of the HBM2 controller that is already relevant to this section is the minimum AXI burst length. Bursts of length one are supported, but they limit the bandwidth utilization to 50 % even for sequential accesses [29], [30]. In order to fully utilize the bandwidth, AXI bursts with a length of two or higher must be issued. This doubles the minimum access granularity to 512 bits, the same value as for DDR4. For simplicity, we design all our components for an AXI burst length of one and set the width of their AXI data channels to 512 bits. Before accessing the AXI3 slave ports of the HBM2 controller, a data width converter reduces the data channel width to 256 bits and increases the burst length to two. For benchmarking, the traffic generator is operated at only half the frequency of the AXI3 slave ports, i.e. 225 MHz. Due to the wider data channel, this frequency is sufficient to fully utilize the DRAM bandwidth. An AXI clock converter is placed between traffic generator and data width converter to increase the frequency to 450 MHz. The right-side length of the triangle is set to only 2048 instead of 4096 because the HBM2-based interleaver in Section VI-B will use four channels for reading and four channels for writing in parallel ($4 \cdot \frac{2048 \cdot 2049}{2} \approx \frac{4096 \cdot 4097}{2}$). Table 5 provides the measurement results, which will be explained next.

**TABLE 5. Measured HBM2 bandwidth utilization on Alveo U280.**

| Address Mapping | Read | | Write | |
|---|---|---|---|---|
| | Row-Wise | Col.-Wise | Row-Wise | Col.-Wise |
| RBC | 65.78 % | 66.01 % | 66.02 % | 64.47 % |
| RBC INTLV | 97.24 % | 43.35 % | 95.96 % | 33.10 % |
| OPT 1 | 45.42 % | 45.94 % | 35.34 % | 35.77 % |
| OPT 2 | 87.46 % | 87.39 % | 83.48 % | 83.42 % |
| OPT 3 | 92.27 % | 92.13 % | 89.03 % | 88.98 % |

The HBM2 controller supports five different mappings from physical addresses to DRAM addresses and also allows to specify completely custom mappings where each bit can be assigned individually. However, only for the two default mappings RBC and RBC INTLV the AXI reordering core stays active [29]. If required, this limitation can be circumvented by selecting one of the two default mappings and implementing a different mapping outside of the controller on the FPGA. Using the RBC (RoBgBaCo) mapping without optimizations, the bandwidth utilization for all four access patterns is at around 65 %, which is much higher than the results achieved with DDR4. While this seems surprising at first glance, the explanation for this behavior is straightforward. Due to the limitations of the memory controller, each access on the AXI3 slave interface has a burst length of two. The second beat has the same address as the first, except for the least significant bit. As this bit is mapped to a column bit using the RBC mapping, it always results in a row hit. For the row-wise accesses, the row-hit ratio is even higher, but the bank parallelism is low. Column-wise accesses only have a row-hit ratio of 50 %, but they maximize the bank parallelism because also the upper address bits experience high toggling rates. These two factors roughly balance each other out as the measurements show. With the RBC INTLV mapping, the results are more comparable to the RCB results of DDR4. This mapping places one of the two bank group bits as least significant bit. As a result, the bank parallelism increases for the row-wise accesses, which results in a bandwidth utilization of over 95 %. For the column-wise accesses however, the bandwidth utilization drops to 43 % in the read case and 33 % in the write case because the second beat is no longer a row hit.

The results of the optimized mapping (row OPT 1 in Table 5) are again worse than expected. Although not explicitly stated in the datasheet [29], the HBM2 controller of AMD/Xilinx also seems to have no individual FSMs for each bank, but only for each bank group just like the DDR4 controller. When the adjusted optimized mapping is used (row OPT 2 in Table 5), which distributes the traffic only across the bank groups, the bandwidth utilization is comparable to the results achieved with DDR4. In the case of HBM2, this mapping only needs to toggle the upper bank group bit after each access. The lower bank group bit is toggled implicitly due to the AXI burst length of two

in combination with the RBC INTLV mapping inside the controller. Still, there is one more optimization that can be applied specifically for the HBM2 controller. As we found out during our experiments, the controller automatically precharges a row after accessing its column with the highest index (all column bits are one). While this behavior is beneficial for sequential accesses, it is counterproductive when using the optimized mapping. Whenever accessing the lower tip of the triangle, which is transformed to the bottom right corner of the rectangle (see Fig. 10), the basic blocks are traversed in reverse order, i.e. the column indices are decremented and not incremented. This means that the column with the highest index is accessed first and followed by more row hits afterwards. In this case, the automatic precharge should be avoided. To circumvent this behavior, we invert the column bits whenever the basic blocks are traversed in reverse order, so the column indices are again incremented. Since the basic blocks do not overlap when mirroring the lower tip of the triangle in the empty space of the rectangle both for a right side length of 2048 and 4096, this does not lead to duplicate addresses.[2] Additionally, the bank-dependent shift introduced in Section IV-F is disabled to avoid duplicate addresses. Using this final optimization, the worst case bandwidth utilization increases to 88.98 % (row OPT 3 in Table 5), which is an improvement of 1.38 × over the row-major order with RBC mapping. HBM2 achieves a higher bandwidth utilization than DDR4 because it uses per-bank refresh instead of all-bank refresh, and because the memory controller is more advanced. However, the improvement is much smaller because HBM2 already performs well without the optimized mapping due to the increased AXI burst length.

## VI. HARDWARE IMPLEMENTATION OF DRAM-BASED INTERLEAVER

This section finally demonstrates the hardware implementation of a complete DRAM-based right triangle interleaver on the Alveo U280. In a first step, the interleaver is implemented with the two DDR4 channels. This implementation is also used to analyze the memory and FPGA resource overhead for the optimized mapping. In a second step, the interleaver is implemented with eight of the HBM2 pseudo channels as they offer a higher maximum bandwidth.

### A. TWO DDR4 CHANNELS

The first part of this section explains the interleaver architecture. This is followed by an analysis of the achievable interleaver throughput, FPGA resource utilization and memory overhead for the optimized mapping.

#### 1) INTERLEAVER ARCHITECTURE

For the interleaver hardware implementation, we assume that a continuous input data stream is received and, after an initial delay, a continuous output data stream should be provided. Both data streams are 512 bits wide and have an equal frequency. For demonstration, they are generated by a *traffic producer* and a *traffic consumer*, which are connected to the interleaver subsystem through an AXI4-Stream (AXIS) master and an AXI4-Stream slave interface, respectively. In the final design, these interfaces will be connected to other components of the en-/decoder like the SRAM interleaver or to board I/O. Fig. 13 shows a block diagram of the subsystem. As the achieved DRAM bandwidth utilization is always below 100 % (see Section V-B), the input and output data streams have to operate at a lower frequency than the AXI interfaces of the memory controllers. Additionally, the fluctuations in DRAM bandwidth utilization caused by row misses and especially refreshes must be balanced. Therefore, *dual clock FIFOs* are placed after the traffic producer and in front of the traffic consumer. The main component of the interleaver is the so-called *address generator*, which is placed between the FIFOs and memory controllers. It has four interfaces, namely an AXI4-Stream slave interface, an AXI4-Stream master interface, and two AXI4 master interfaces. Internally, the address generator forwards the data received on the AXI4-Stream slave interface to the write data channel of one of the two AXI4 master interfaces and the data received on the read data channel of the other AXI4 master interface to the AXI4-Stream master interface. Furthermore, it calculates the write and read addresses based on the selected mapping (row-major or optimized, row-wise read and column-wise write or vice versa) and issues them to the correct address channels of the AXI4 master interfaces. Once the addresses and write data for a full interleaving window have been issued and the write responses and read data received, the DRAM channels are swapped and the process starts again. In the first iteration after reset, no data is read because the memory is still empty.

#### 2) INTERLEAVER THROUGHPUT

The maximum interleaver throughput can be determined based on the measurements of Table 4. Without the optimized mapping, the maximum achievable bandwidth utilization across all four access patterns is 33.72 % with the RCB mapping. This corresponds to a maximum operating frequency for traffic producer and consumer of 101.16 MHz since the AXI4 slave interface of the DDR4 controllers are operated at 300 MHz. The closest frequency that can be selected on the FPGA is 100 MHz. This results in a maximum throughput without optimized mapping of 47.68 Gbit/s. As explained in Section II-D, each binary symbol is accompanied by two bits of soft information or two placeholder bits. A DRAM burst of 512 bits then fits 170 binary symbols, so the interleaver can only handle a maximum net data rate of 15.83 Gbit/s. With the optimized mapping, the maximum achievable bandwidth utilization increases to 82.75 % and the maximum operating frequency for traffic producer and consumer to 248.3 MHz. On the FPGA, we can select a frequency of 245 MHz, leading to a maximum throughput of 125.4 Gbit/s. However,

[2] If they overlapped, the number of basic blocks in the row-wise access direction could be increased by one to ensure unique addresses. This would only result in a slightly higher memory overhead.

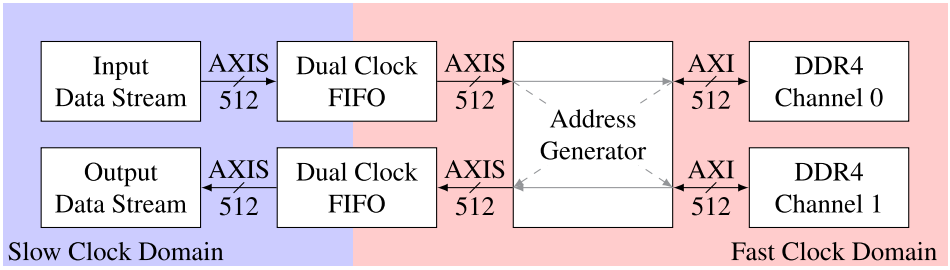


**FIGURE 13.** Block diagram of interleaver with two DDR4 channels.

**TABLE 6.** FPGA resource utilizations for DDR4-based interleaver.

| Unit | LUTs | Registers | CARRY8 | BRAM |
|---|---|---|---|---|
| Address Generator | 420 | 288 | 42 | 0 |
| | 617 | 259 | 34 | 0 |
| Dual Clock FIFOs | 295 | 366 | 0 | 114 |
| | 297 | 366 | 0 | 114 |
| DDR4 Controllers | 34,991 | 38,492 | 112 | 51 |
| | 35,028 | 38,513 | 112 | 51 |
| Complete Interleaver Design | 36,638 | 42,063 | 154 | 165 |
| | 36,876 | 42,055 | 146 | 165 |
| Alveo U280 | 1,303,680 | 2,607,360 | 162,960 | 2,016 |

the maximum net data rate is still only 41.65 Gbit/s, which is less than half of the required 100 Gbit/s. To verify the results, the complete interleaver design is implemented on the Alveo U280. As expected, without the optimized mapping, it can process traffic at 100 MHz without stalling, while at 105 MHz, it stalls regularly. The same results are observed with the optimized mapping at 245 MHz and 250 MHz, respectively.

### 3) FPGA RESOURCE UTILIZATION

Next, the resource overhead for the optimized mapping is analyzed and set into relation to the FPGA resource utilization of the complete interleaver design and the available resources of the Alveo U280. To do this, the complete interleaver design is implemented on the FPGA once without and once with the optimized mapping enabled in the address generator. For a fair comparison, the traffic producer and consumer are operated at 245 MHz in both cases. Furthermore, the depth of the FIFOs is set to 4096 in both cases. This is the minimum depth required at this frequency with the optimized mapping so that the traffic producer and consumer are not stalled. Table 6 shows the resulting resource utilizations, with the lower rows again representing the optimized mapping. The address generator implementing the optimized mapping has an overhead of almost 50 % in lookup tables (LUTs), while the number of registers and CARRY8 circuits slightly decreases. While this seems a lot at first sight, the resource utilization of the address generator is only a fraction of the complete interleaver design. The majority of resources is required to build the two DDR4 controllers, which are soft IP cores. However, even the complete interleaver design requires only a fraction of the available resources on the Alveo U280. All in all, the overhead of the optimized mapping is completely negligible.

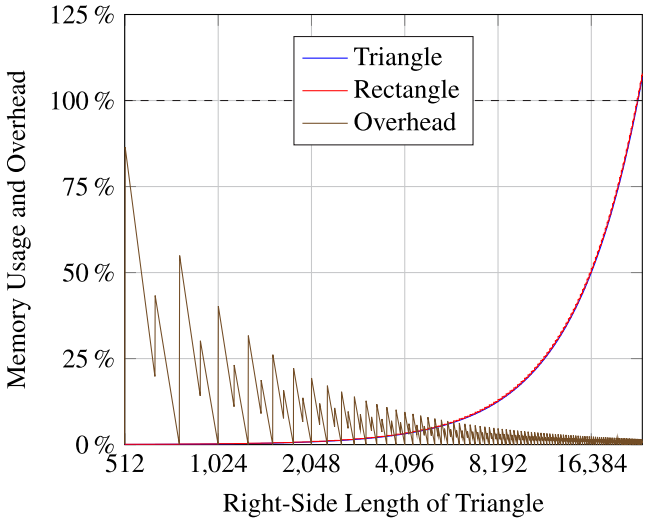


**FIGURE 14.** Memory usage and overhead of the optimized mapping for different right-side lengths on 16 GiB DDR4 channel.

### 4) MEMORY OVERHEAD

Next, the memory overhead resulting from underfull basic blocks in the optimized mapping is analyzed. The DDR4 channels on the Alveo U280 have 16 banks, each with 131,072 rows and 128 columns per row. As the number of columns is not square, the highest column address bit is treated as a row address bit, i.e. $n_{bank} = 16$, $n_{col} = 64$ and $n_{row} = 262{,}144$. The overhead can be calculated as $\frac{n_{rect}}{n_{tri}} - 100\,\%$ using the equations (2), (3), (12), (14), (15) and (16) of Section IV-E. For a right-side length of $s_{tri} = 4096$, the isosceles triangle has $n_{tri} = 8{,}390{,}656$ elements and the rectangle of the optimized mapping has $n_{rect} = 8{,}912{,}896$ elements. This corresponds to an overhead of 6.22 %. In both cases, the memory usage is very low: 3.13 % for the default mapping and 3.32 % for the optimized mapping. Fig. 14 shows the memory usage and overhead as a function of the right-side length for a better understanding of the relation. For the shortest right-side length of $s_{tri} = 512$, the overhead is as high as 86 % because the side length

of a basic block is $s_{block} = 128$. However, this is not a concern as the memory usage is below 0.1 %. With increasing dimensions, the memory usage also increases, while the overhead decreases. The sawtooth shape is caused by the ceilings in (15) and (16). The maximum right-side length that can fit onto the 16 GiB DDR4 channels without optimized mapping is $s_{tri} = 23{,}169$, while with optimized mapping, it is $s_{tri} = 23{,}040$, i.e. only 0.56 % smaller.

### B. EIGHT HBM2 CHANNELS

As the throughput of the two DDR4 channels on the Alveo U280 is insufficient to meet the data rate requirement of 100 Gbit/s, in this section, we use the HBM2 DRAM to implement the interleaver. It offers a much higher theoretical peak bandwidth of 3.7 Tbit/s compared to 307.2 Gbit/s of the two DDR4 channels. The peak bandwidth of each of the 32 pseudo channels is 115.2 Gbit/s, with a capacity of 256 MiB. Due to the added soft information, a 512-bit DRAM burst fits 170 binary symbols, requiring a concurrent read and write throughput of 301.2 Gbit/s. Based on the measurement results for the optimized mapping in Table 5, six HBM2 pseudo channels (three for reading and three for writing) should provide sufficient bandwidth and capacity. However, as this is not a power of two, the addressing becomes more complex. For this reason, we choose eight pseudo channels (four for reading and four for writing). The following sections will first explain the extended interleaver architecture. Afterwards, the achievable throughput and FPGA resource utilization will be analyzed.

#### 1) INTERLEAVER ARCHITECTURE

In the case that more than two DRAM channels are used, the interleaver architecture of Fig. 13 needs to be extended by additional units, which is shown in Fig. 15. As explained in Section IV-G, there are two different approaches to handle multiple channels. One is to treat them like a single channel with a wider data bus, while the other is to treat them like a single channel with more banks. To keep the access granularity as small as possible, the second approach is implemented. In order to provide all four write channels with data simultaneously and accept the data of all four read channels simultaneously, the input data stream and output data stream would need to be operated at a frequency of at least $\frac{100\,\text{Gbit/s}}{170\,\text{bit}} = 588.2\,\text{MHz}$. As it is challenging to operate the FPGA at this frequency, we directly scale up the widths of the input and output data stream from 512 bits to 2048 bits. This way, the interleaver receives and provides four DRAM bursts (i.e. one per channel) in parallel in each clock cycle. The layout of Fig. 5 ensures that the traffic is equally distributed across all channels both when accessing the triangle row-wise and column-wise. However, this also means that there is no fixed assignment between the four portions of the wider data streams and specific DRAM channels. As an example, in Fig. 15, the lowest 512 bits of the input data stream are not always mapped to DRAM channel 0 or 1, but it can also be mapped to all other DRAM channels. This depends on the current position within the triangle and on which channels are currently writing. In the dual channel case, the complete routing was done inside the address generator (see dashed arrows in Fig. 13). If more channels are used, the interleaver requires two additional routing units, the so-called *data stream splitter* and *data stream merger*. Just like the address generator, these units internally also track the current position within the triangle. Depending on the position, they route each 512-bit portion of the wider data stream to one of the 512-bit-wide ports or the other way round as shown by the dashed arrows. Unlike the address generator, which only requires 2-to-1 multiplexers, the number of inputs of the multiplexers in the data stream splitter and data stream merger depends on the number of concurrent write and read channels. Fig. 15 shows that in our case the units use 4-to-1 multiplexers. As this leads to complex routing, they are placed in front of the dual clock FIFOs to operate them at the lowest possible frequency. In theory, the complete routing could also be performed within the $32 \times 32$ crossbar switch of the HBM2 controller. This hard IP is able to route each AXI3 slave port to each HBM2 pseudo channel. However in our experiments, the bandwidth utilization dropped below 20 % when transactions were issued to multiple AXI3 slave ports at the same time although all transactions targeted a different pseudo channel. For this reason, the routing is done externally and the crossbar switch is disabled. This means that each AXI3 slave port can only access one specific pseudo channel. Before accessing the AXI3 slave ports, the data width is reduced from 512 bits to 256 bits by the data width converters (DWC).

#### 2) INTERLEAVER THROUGHPUT

For the HBM2-based interleaver, the throughput can be determined based on the measurements of Table 5. When using the optimized mapping, the maximum achievable bandwidth utilization across all four access patterns is 88.98 %. Since the traffic producer was only operated at 225 MHz, the corresponding maximum operating frequency for the input and output data streams is 200.2 MHz. On the device, we can select a frequency of 192.86 MHz, which means that the maximum achievable throughput of the interleaver is 394.98 Gbit/s. This corresponds to a net data rate of 131.1 Gbit/s without soft information. If the target data rate of only 100 Gbit/s should be achieved, it is sufficient to operate the input and output data stream at 150 MHz. For both frequencies, we were able to implement the complete design on the FPGA without timing violations and run it stall free. The unoptimized mapping cannot be directly scaled up to more than two channels without increasing the minimum access granularity or also using a special mapping for the channels. Therefore, it was not implemented using HBM2, but based on the measurements in Table 5, the maximum achievable net data rate would only be 98.64 Gbit/s, so eight HBM2 channels would not be sufficient to meet the target. In theory, the channel count could be increased further, since the Alveo U280 offers a

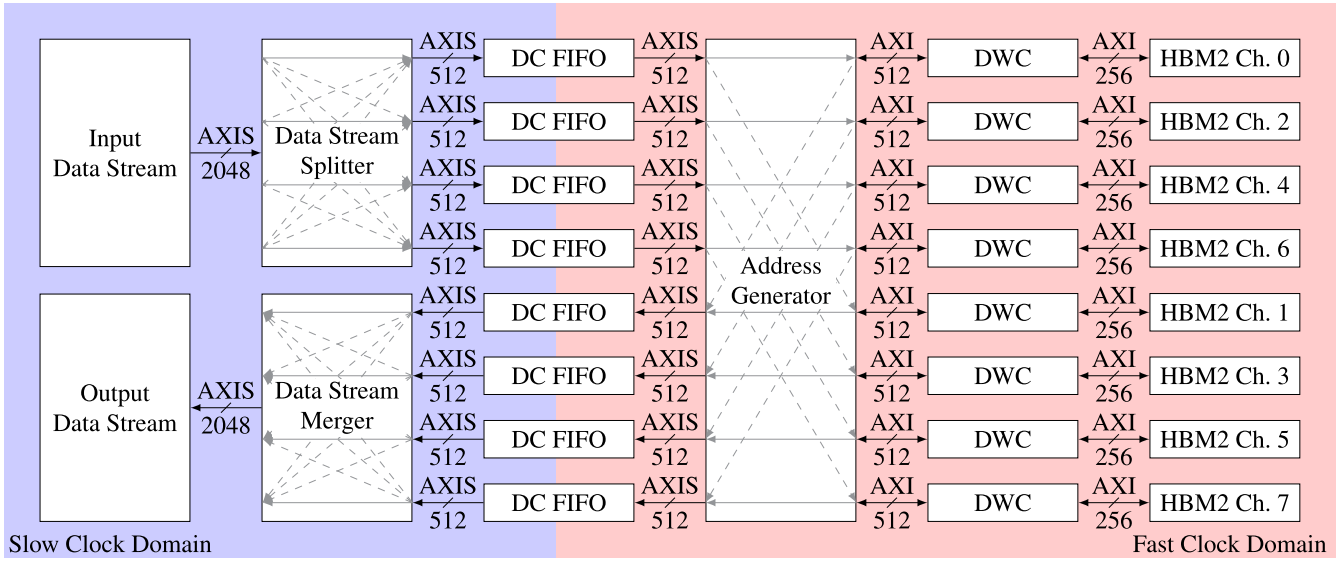


**FIGURE 15.** Block diagram of interleaver with eight HBM2 channels.

total of 32 HBM2 pseudo channels. However, even with eight channels, routing becomes challenging, as will be demonstrated in the next section.

### 3) FPGA RESOURCE UTILIZATION

As mentioned in the previous section, the interleaver design is implemented on the Alveo U280 with two different frequencies for the slow clock domain: 150 MHz and 192.86 MHz. The fast clock domain is operated at 450 MHz in both cases, which is the maximum frequency for the AXI3 slave interfaces. Although it would also be sufficient to operate the address generator at only 225 MHz and move to 450 MHz between the address generator and the data width converters, this is not done because we were not able to meet the timing requirements after adding a third clock domain. When the slow clock domain is operated at 150 MHz, the required FIFO depth is 512, while at 192.86 MHz, it increases to 1024. This is only one eighth or one fourth of the required FIFO depth of the DDR4-based interleaver (see Section VI-A3) because HBM2 uses per-bank refresh instead of all-bank refresh, which causes smaller bandwidth fluctuations. We also found that, if the column bits are not inverted when the basic blocks are traversed in reverse order (see Section V-B2), there are much larger fluctuations in bandwidth utilization. In this case, FIFOs with a depth of 8,192 at 150 MHz and 16,384 at 192.86 MHz are required. These FIFOs only fit on the FPGA if UltraRAM is used instead of block RAM or if they are split across multiple super logic regions.[3] However, we were not able to meet the timing requirements with the larger FIFOs. The special channel layout (0-2-4-6-1-3-5-7) shown in Fig. 15 is used to reduce the routing complexity because the I/O ports of neighboring channel IDs are also placed next to each other on the FPGA. In order to meet the timing requirements, it was necessary to place register slices between the data stream splitter/merger and FIFOs and between the FIFOs and the address generator (not shown in Fig. 15). By choosing the so-called *auto-pipelined* mode, the place and route tool automatically inserts additional pipeline stages based on the setup timing slack [31]. Additionally, the implementation strategy `Congestion_SpreadLogic_high` was selected because of the complex routing. Table 7 shows the resource utilizations for the individual units and the complete interleaver design. The upper rows are for 150 MHz and the lower rows for 192.86 MHz. The address generator requires ∼4 × more registers and ∼3 × more

**TABLE 7.** FPGA resource utilizations for HBM2-based interleaver.

| Unit | LUTs | Registers | CARRY8 | BRAM |
|---|---|---|---|---|
| Address Generator | 1,839 | 1,069 | 116 | 0 |
| | 1,839 | 1,072 | 116 | 0 |
| Register Slices | 10,696 | 14,508 | 0 | 0 |
| FIFO – AG | 10,631 | 15,349 | 0 | 0 |
| Data Stream Splitter | 8,274 | 11,690 | 0 | 0 |
| | 8,276 | 15,250 | 0 | 0 |
| Data Stream Merger | 8,275 | 12,444 | 0 | 0 |
| | 8,278 | 16,623 | 0 | 0 |
| Data Width Converters | 2,270 | 4,392 | 0 | 0 |
| | 2,310 | 4,392 | 0 | 0 |
| Dual Clock FIFOs | 804 | 1,173 | 0 | 60 |
| | 922 | 1,309 | 0 | 116 |
| HBM2 Controller | 496 | 430 | 10 | 2 |
| | 495 | 430 | 10 | 2 |
| Complete Interleaver Design | 32,584 | 45,837 | 126 | 62 |
| | 32,712 | 54,556 | 126 | 118 |
| Alveo U280 | 1,303,680 | 2,607,360 | 162,960 | 2,016 |

[3]The device is composed of three so-called *super logic regions* (SLRs), each containing approximately one third of the resources. They are single device die slices connected through a silicon interposer. The complete HBM2 controller is located on SLR0 [27].

LUTs than the one used for DDR4 (see Table 6). While the number of registers directly scales with the increase in channels, less LUTs are used because the bank-dependent shift is disabled. The majority of the resources is required for the complex routing and the resulting register slices. In the report, the register slices between FIFOs and address generator (AG) can be found in the second row, while the register slices between data stream splitter/merger and FIFOs are included in the resources of the data stream splitter/merger. The numbers also show that ∼20 % more registers are used by the place and route tool to meet the timing requirements at the higher frequency. In contrast to the DDR4 controller, the HBM2 controller requires only some FPGA resources for initialization, with the rest being hard IP. For this reason, the FPGA resource utilization of both interleaver designs is similar, even though the HBM2 design uses four times more channels and additional units.

## VII. CONCLUSION AND FUTURE WORK

In this paper, we have shown how large right triangle interleavers for free-space optical communication can be implemented with DRAM. By introducing an optimized mapping from the two-dimensional index space to linear DRAM addresses, we were able to achieve a large increase in DRAM bandwidth utilization both in simulations and on real hardware. At the same time, we have demonstrated that the overhead of FPGA resources required to implement the optimized mapping is negligible compared to the resource utilization of the complete interleaver design. The overhead in memory usage can be neglected as well. Finally, we have implemented the complete interleaver design on the AMD Alveo U280 accelerator card and achieved a maximum data rate of 131.1 Gbit/s, fulfilling the requirement of at least 100 Gbit/s. For future work, we will integrate the DRAM-based interleaver and deinterleaver into the complete communication chain.

## REFERENCES

[1] C. Daehnick, I. Klinghoffer, B. Maritz, and B. Wiseman. (May 2020). *Large LEO Satellite Constellations: Will It Be Different This Time?*. McKinsey & Company. [Online]. Available: https://www.mckinsey.com/industries/aerospace-and-defense/our-insights/large-leo-satellite-constellations-will-it-be-different-this-time

[2] A. Orlova, R. Nogueira, and P. Chimenti, "The present and future of the space sector: A business ecosystem approach," *Space Policy*, vol. 52, May 2020, Art. no. 101374. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0265964620300163

[3] D. Kolev and M. Toyoshima, "Satellite-to-ground optical communications using small optical transponder (SOTA)–received-power fluctuations," *Opt. Exp.*, vol. 25, no. 23, p. 28319, Nov. 2017. [Online]. Available: https://opg.optica.org/oe/abstract.cfm?URI=oe-25-23-28319

[4] O. Griebel, A. Sauter, U. Wasenmüller, L. Steiner, J. Poliak, B. Matuz, and N. Wehn, "Physical layer forward error correction for free-space optical links," *Proc. SPIE*, vol. 12877, pp. 134–145, Mar. 2024, doi: 10.1117/12.3001394.

[5] L. Steiner, T. Lehnigk-Emden, M. Fehrenz, and N. Wehn, "A mapping of triangular block interleavers to DRAM for optical satellite communication," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2024, pp. 1–2.

[6] *5G; NR; Multiplexing and Channel Coding*, Standard TS 138 212, European Telecommunications Standards Institute (ETSI), Jul. 2025. [Online]. Available: https://www.etsi.org/deliver/etsits/

[7] J. L. Lakshmi and J. Jayakumari, "A reduced complexity rate-matching and channel interleaver/de-interleaver for 5G NR," *Eng. Res. Exp.*, vol. 6, no. 2, Apr. 2024, Art. no. 025301, doi: 10.1088/2631-8695/ad37f1.

[8] X. Xiong, Y. Dai, Z. Hu, K. Huo, Y. Bai, H. Li, and D. Liu, "Hardware sharing for channel interleavers in 5G NR standard," *Secur. Commun. Netw.*, vol. 2021, pp. 1–13, Jan. 2021. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1155/2021/8872140

[9] V. Bioglio, C. Condo, and I. Land, "Design of polar codes in 5G new radio," *IEEE Commun. Surveys Tuts.*, vol. 23, no. 1, pp. 29–40, 1st Quart., 2021.

[10] J. Zhou, Y. Dou, Y. Lei, and Y. Dong, "Window memory accesses method in alternate row/column matrix access systems," in *Proc. 2nd Int. Conf. Comput. Eng. Technol.*, vol. 3, Apr. 2010, pp. V3-201–V3-205.

[11] L. Guo, Y. Tang, Y. Dou, Y. Lei, M. Ma, and J. Zhou, "Window memory layout scheme for alternate row-wise/column-wise matrix access," *IEICE Trans. Inf. Syst.*, vol. 96, no. 12, pp. 2765–2775, 2013.

[12] M. Garrido and P. Pirsch, "Continuous-flow matrix transposition using memories," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 67, no. 9, pp. 3035–3046, Sep. 2020.

[13] B. Akin, F. Franchetti, and J. C. Hoe, "Data reorganization in memory using 3D-stacked DRAM," in *Proc. ACM/IEEE 42nd Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2015, pp. 131–143.

[14] H. Kim and I.-C. Park, "Array address translation for SDRAM-based video processing applications," *Proc. SPIE*, vol. 4067, pp. 922–931, May 2000.

[15] T. Baozhao, L. Dong, and H. Chengde, "Two-dimensional image processing without transpose," in *Proc. 7th Int. Conf. Signal Process.*, Aug. 2004, pp. 523–526.

[16] Y. Dou, J. Zhou, Y. Lei, and X. Zhou, "FPGA SAR processor with window memory accesses," in *Proc. IEEE Int. Conf. Appl.-Specific Syst., Archit. Processors (ASAP)*, Jul. 2007, pp. 95–100.

[17] B. Akin, P. A. Milder, F. Franchetti, and J. C. Hoe, "Memory bandwidth efficient two-dimensional fast Fourier transform algorithm and implementation for large problem sizes," in *Proc. IEEE 20th Int. Symp. Field-Program. Custom Comput. Mach.*, Apr. 2012, pp. 188–191.

[18] S. Langemeyer, P. Pirsch, and H. Blume, "Using SDRAM memories for high-performance accesses to two-dimensional matrices without transpose," *Int. J. Parallel Program.*, vol. 41, no. 2, pp. 331–354, Apr. 2013.

[19] R. Chen and V. K. Prasanna, "DRAM row activation energy optimization for stride memory access on FPGA-based systems, "DRAM row activation energy optimization for stride memory access on FPGA-based systems," in *Applied Reconfigurable Computing*, K. Sano, D. Soudris, M. Hübner, and P. C. Diniz, Eds., Cham, Switzerland: Springer, 2015, pp. 349–356.

[20] R. Chen, S. G. Singapura, and V. K. Prasanna, "Optimal dynamic data layouts for 2D FFT on 3D memory integrated FPGA," in *Parallel Computing Technologies*, V. Malyshkin, Ed., Cham, Switzerland: Springer, 2015, pp. 338–348.

[21] M. Jung, D. M. Mathew, C. Weis, N. Wehn, I. Heinrich, M. V. Natale, and S. O. Krumke, "ConGen: An application specific DRAM memory controller generator," in *Proc. 2nd Int. Symp. Memory Syst.*, Oct. 2016, pp. 257–267, doi: 10.1145/2989081.2989131.

[22] M. V. Natale, M. Jung, K. Kraft, F. Lauer, J. Feldmann, C. Sudarshan, C. Weis, S. Krumke, and N. Wehn, "Efficient generation of application specific memory controllers," in *Proc. Int. Symp. Memory Syst.*, Sep. 2020, pp. 233–247, doi: 10.1145/3422575.3422796.

[23] X. Li, Z. Yuan, Y. Guan, G. Sun, T. Zhang, R. Wei, and D. Niu, "Flatfish: A reinforcement learning approach for application-aware address mapping," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 41, no. 11, pp. 4758–4770, Nov. 2022.

[24] V. Seshadri, T. Mullins, A. Boroumand, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Gather-scatter DRAM: In-DRAM address translation to improve the spatial locality of non-unit strided accesses," in *Proc. 48th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Dec. 2015, pp. 267–280.

[25] D. R. Arrieta, S. Almonacil, J.-M. Conan, L. Paillier, E. Dutisseuil, S. Bigo, J. Renaudier, and R. Boddeda, "Proof-of-concept real-time implementation of interleavers for optical satellite links," *J. Lightw. Technol.*, vol. 41, no. 12, pp. 3932–3942, Jun. 15, 2023.

[26] L. Steiner, M. Jung, F. S. Prado, K. Bykov, and N. Wehn, "DRAMSys4.0: A fast and cycle-accurate SystemC/TLM-based DRAM simulator," in *Embedded Computer Systems: Architectures, Modeling, and Simulation*. Cham, Switzerland: Springer, 2020, pp. 110–126.
[27] Alveo U280. (2023). *Data Center Accelerator Card Data Sheet (DS963)*. [Online]. Available: https://docs.amd.com/r/en-U.S./ds963-u280
[28] UltraScale. (2022). *Architecture-Based FPGAs Memory IP*. [Online]. Available: https://docs.amd.com/v/u/en-U.S./pg150-ultrascale-memory-ip
[29] AXI. (2024). *High Bandwidth Memory Controller*. [Online]. Available: https://docs.amd.com/r/en-U.S./pg276-axi-hbm
[30] P. Holzinger, D. Reiser, T. Hahn, and M. Reichenbach, "Fast HBM access with FPGAs: Analysis, architectures, and applications," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPSW)*, Jun. 2021, pp. 152–159.
[31] AXI4. (2023). *Stream Infrastructure IP Suite*. [Online]. Available: https://docs.amd.com/r/en-US/pg085-axi4stream-infrastructure

**LUKAS STEINER** (Graduate Student Member, IEEE) received the B.Sc. and M.Sc. degrees in electrical and computer engineering from the University of Kaiserslautern-Landau (RPTU), Germany, where he is currently pursuing the Ph.D. degree with the Microelectronic Systems Design Research Group. His research interests include the performance, power, and safety modeling of advanced DRAM architectures and the optimization of DRAM subsystems.

**UWE WASENMÜLLER** (Member, IEEE) received the Diploma degree in mathematics from the Technical University of Darmstadt, Germany. After working in industry and technology transfer, he joined the Microelectronic Systems Design Research Group. His research interests include the implementation of baseband signal processing units, especially synchronization units in mobile and satellite communications.

**NORBERT WEHN** (Senior Member, IEEE) holds the Chair for Microelectronic Systems Design, University of Kaiserslautern-Landau (RPTU), Germany. He has more than 450 publications in various fields of microelectronic systems design and holds 21 patents. His research interests include very large-scale integration (VLSI) architectures for mobile communication, forward error correction techniques, low-power techniques, advanced system-on-chip (SoC), memory architectures, reliability issues in SoC, the Internet of Things (IoT), and hardware accelerators for machine learning.