Optimizing Bit-Serial Matrix Multiplication for Reconfigurable Computing

YAMAN UMUROGLU, Xilinx Research Labs, Ireland
DAVIDE CONFICCONI, Xilinx Research Labs, Ireland and Politecnico di Milano, Italy
LAHIRU RASNAYAKE, Norwegian University of Science and Technology, Norway
THOMAS B. PREUSSER, Accemic Technologies GmbH, Germany
MAGNUS SJÄLANDER, Uppsala University, Sweden and Norwegian University of Science and Technology, Norway

Matrix-matrix multiplication is a key computational kernel for numerous applications in science and engineering, with ample parallelism and data locality that lends itself well to high-performance implementations. Many matrix multiplication-dependent applications can use reduced-precision integer or fixed-point representations to increase their performance and energy efficiency while still offering adequate quality of results. However, precision requirements may vary between different application phases or depend on input data, rendering constant-precision solutions ineffective. BISMO, a vectorized bit-serial matrix multiplication overlay for reconfigurable computing, previously utilized the excellent binary-operation performance of FPGAs to offer a matrix multiplication performance that scales with required precision and parallelism. We show how BISMO can be scaled up on Xilinx FPGAs using an arithmetic architecture that better utilizes 6-input LUTs. The improved BISMO achieves a peak performance of 15.4 binary TOPS on the Ultra96 board with a Xilinx UltraScale+ MPSoC.

CCS Concepts: • **Computer systems organization** → *Pipeline computing*; • **Hardware** → **Hardware accelerators**;

Additional Key Words and Phrases: Bit serial, Matrix multiplication, Overlay, FPGA

ACM Reference Format:
Yaman Umuroglu, Davide Conficconi, Lahiru Rasnayake, Thomas B. Preusser, and Magnus Själander. 2019. Optimizing Bit-Serial Matrix Multiplication for Reconfigurable Computing. *ACM Trans. Reconfig. Technol. Syst.* 1, 1, Article 1 (May 2019), 24 pages. https://doi.org/10.1145/3337929

1 INTRODUCTION

Using constant precision for all operations is the predominant practice when designing digital systems, since logical and arithmetic operations, registers, memories, and interconnects can be designed to accommodate one specific precision. Their main disadvantage is the associated overhead in storing, communicating, and performing operations with full precision when an application only requires a fraction of the supported precision. Numerous applications, in the engineering, scientific,

Authors' addresses: Yaman Umuroglu, Xilinx Research Labs, Dublin, Ireland, yamanu@xilinx.com; Davide Conficconi, Xilinx Research Labs, Dublin, Ireland, Politecnico di Milano, Milano, Italy, davidec@xilinx.com; Lahiru Rasnayake, Norwegian University of Science and Technology, Trondheim, Norway, lahiru.rasnayake@ntnu.no; Thomas B. Preusser, Accemic Technologies GmbH, Dresden, Germany, thomas.preusser@utexas.edu; Magnus Själander, Uppsala University, Uppsala, Sweden, Norwegian University of Science and Technology, Trondheim, Norway, magnus.sjalander@ntnu.no.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2019 Association for Computing Machinery.
1936-7406/2019/5-ART1 $15.00
https://doi.org/10.1145/3337929

and multimedia domain, can use reduced precision and still produce adequate results. This property has been leveraged in approximate computing [11] and quantized neural networks (QNNs) [5, 15], to improve performance and energy efficiency and to reduce area by tailoring computations to the required precision. The required precision may vary between different phases of the application. As an example, Park et al. [15] achieve the best performance-accuracy tradeoff for QNNs by using fewer bits for the intermediate layers, and Wang et al. [20] use a reinforcement learning approach to discover efficient QNNs with different per-layer quantization.

Matrix-matrix multiplication is a commonly used computational kernel and represents one of the seven Berkeley dwarfs, which are important computational constructs for engineering and scientific computing [1]. The amount of computation required for matrix multiplications makes it highly beneficial to adapt the operational precision to an application's requirements. FPGAs are a good fit for low-precision operations and for instantiating efficient matrix multiplication accelerators with a specific precision. However, fixed-precision accelerators are not suitable for applications with variable precision as they either require multiple instances of the same accelerator, each with a different precision, or require dynamic reconfiguration with associated overhead and system complexity.

A promising alternative to fixed-precision accelerators is to use bit-serial computations [18] where the integer matrix multiplication is expressed as a weighted sum of binary matrix multiplications (Section 2). The bit-serial alternative provides the possibility to use one efficient binary matrix multiplication accelerator to compute matrix multiplications of any precision.

Towards this end, a bit-serial matrix multiplication overlay called BISMO was presented by Umuroglu et al. [19]. BISMO consists of a software-programmable weighted binary matrix multiplication engine and associated hardware for fetching data and storing back the result (Section 3.1). The hardware architecture is design-time configurable and comes with a cost model for estimating the resource usage for a given set of parameters (Section 3.4). BISMO's software programmability enables it to operate on any matrix size and at any fixed-point or integer precision (Section 3.5).

This article proposes several improvements to the original BISMO [19]. We present a new and highly LUT-efficient compressor architecture for performing the core And-popcount operation for bit-serial (Section 3.2.2). The DPU architecture has been further improved by eliminating the need for a barrel shifter. This is achieved by organizing the bit-serial matrix multiplications into wavefronts starting with the highest weighted matrix multiplication being performed followed by consecutively less weighted matrix multiplications (Section 3.2.3). The new wavefront schedule requires only a fixed left shift of 1-bit instead of a variable shift-amount. The new DPU architecture is compared against our previously proposed DPU architecture [19] (Section 4.1.1). To address data layout conversion challenges for bit-serial, we introduce a new parallel-to-serial (P2S) accelerator that takes a conventional bit-parallel matrix and produces the equivalent bit-serial matrices Section 3.3, and evaluate its resource cost and performance (Section 4.2.5). We also present an updated BISMO cost model (Section 3.4) that has been validated on an Ultra96 FPGA board (Section 4.1.4).

The most recent BISMO prototype achieves a top performance of 15.4 binary TOPS at 2.1 TOPS/W power efficiency when implemented on an Ultra96 board (Section 4.2). A scalability evaluation shows that the BISMO dot product array (DPA) is capable of achieving a peak performance of at least 783 binary TOPS on a Xilinx Virtex UltraScale+ VU9P (Section 4.1.7).

2 BACKGROUND: BIT-SERIAL

Fixed-precision operations have to be designed to accommodate the largest supported precision, which causes overheads in cases where the required precision of an application varies throughout its execution or when the precision depends on its input data. In contrast, bit-serial operations are inherently frugal since they only compute as many bits as specified by the precision of the operands. However, their serial nature causes high latencies and potentially poor performance. In this section,

Algorithm 1 Bit-serial matrix multiplication on signed two's complement integers.

1: **Input:** $M \times K$ l-bit matrix L, $K \times N$ r-bit matrix R
2: **Output:** $P = L \cdot R$
3: **for** $i \leftarrow 0 \ldots l-1$ **do**
4: **for** $j \leftarrow 0 \ldots r-1$ **do**
5: sgnL $\leftarrow (i == l-1\ ?\ -1 : 1)$
6: sgnR $\leftarrow (j == r-1\ ?\ -1 : 1)$
7: weight = sgnL $\cdot$ sgnR $\cdot\ 2^{i+j}$
8: *# Binary matrix multiplication between* $L^{[i]}$ *and* $R^{[j]}$
9: *#* $L^{[i]}_{mk}$ *refers to* i^{th} *bit position of element at row* m *and column* k *of matrix* L
10: **for** $m \leftarrow 1 \ldots M$ **do**
11: **for** $n \leftarrow 1 \ldots N$ **do**
12: **for** $k \leftarrow 1 \ldots K$ **do**
13: $P_{mn} = P_{mn} + \text{weight} \cdot (L^{[i]}_{mk} \cdot R^{[j]}_{kn})$

we will describe how bit-serial matrix multiplication works on an algorithmic level, and briefly cover the data layout implications for bit-serial matrix multiplication for implementation purposes.

2.1 Bit-Serial Matrix Multiplication

Matrix multiplication is a suitable kernel for taking advantage of the frugality of bit-serial operations while overcoming the high-latency by performing many bit-serial operations in parallel. Umuroglu and Jahre showed that by expressing a matrix multiplication as a weighted sum of binary matrix multiplications (Algorithm 1) it is possible to efficiently compute matrix multiplications of variable precision using the logical AND and population count (popcount) instructions available in most modern processors [18]. In addition, the algorithm works for both integer as well as fixed point number representations, where the new fixed point location is given by the product of the input matrices' scaling factors.

Fig. 1 illustrates Algorithm 1 for the example where the two input-matrices (L and R) consist of 2-bit unsigned integer numbers. By expressing L and R as weighted sums of binary matrices, the matrix product ($P = L \cdot R$) can be expressed as a weighted sum of products between binary matrices. The matrix multiplication can thus be expressed as a large number of binary operations that can be performed in parallel.

$$
\begin{aligned}
L &= \begin{bmatrix} 2 & 0 \\ 1 & 3 \end{bmatrix} = 2^1 L^{[1]} + 2^0 L^{[0]} = 2^1 \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} + 2^0 \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix} \\
R &= \begin{bmatrix} 0 & 1 \\ 1 & 2 \end{bmatrix} = 2^1 R^{[1]} + 2^0 R^{[0]} = 2^1 \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} + 2^0 \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \\
P &= L \cdot R = (2^1 L^{[1]} + 2^0 L^{[0]}) \cdot (2^1 R^{[1]} + 2^0 R^{[0]}) \\
&= 2^2 L^{[1]} \cdot R^{[1]} + 2^1 L^{[1]} \cdot R^{[0]} + 2^1 L^{[0]} \cdot R^{[1]} + 2^0 L^{[0]} \cdot R^{[0]}
\end{aligned}
$$

Fig. 1. Example of a bit-serial matrix multiplication on unsigned integers (Algorithm 1: for-loop on line 3 and 4 unrolled and weight on line 7 always positive).

2.2 Bit-Serial Data Layout

From an implementation point of view, it is important to match the data delivered by the memory system of an accelerator and what the algorithm implemented by the accelerator expects. Typically, the memory system will deliver a number of bits grouped together in response to a request. If the order of bits provided by the memory is substantially different from the order in which the accelerator expects them, the memory bandwidth will be underutilized. For bit-serial matrix multiplication, the data layout requirements are substantially different than bit-parallel matrix multiplication. A bit-parallel layout, where all bit positions of an element are consecutive, is well-matched with bit-parallel matrix multiplication, which makes use of all bit positions at once. In contrast, bit-serial works on a single bit position at a time, but the same bit position for neighboring elements can be processed together. If the input matrices are provided in bit-parallel format, they should first be converted into a bit-serial layout to ensure performance.

In this work, we assume the **[bits][rows][columns]** data layout for bit-serial matrices, as was also assumed in prior work [4, 18, 19]. Section 3.3 provides an example of this data layout in context of a parallel-to-serial accelerator for BISMO.

3 THE BIT-SERIAL MATRIX MULTIPLICATION OVERLAY

BISMO consists of a hardware part and a software part. The hardware part is composed of a scalable bit-serial matrix multiplication datapath and associated memory and control logic. The software part generates instructions for the hardware for a given matrix size and precision. The key features offered by this hardware-software design are the following:

Precision Scalability. By expressing an integer or fixed-point matrix multiplication as a weighted sum of binary matrix multiplications (Section 2), the same hardware can be utilized for a range of different precisions. Lower-precision matrix multiplications are finished quickly, while higher-precision requires more clock cycles.

Hardware Scalability. Our overlay generator can scale the memory and compute resource utilization to match system-level requirements. This is achieved by controlling the parameters described in Section 3.1. The dot product unit (DPU) is BISMO's core processing element, which performs a multiply and accumulate between two weighted binary vectors. We present a new DPU datapath and an efficient FPGA compressor (Section 3.2) that improves resource utilization and DPU scalability. A parallel-to-serial (P2S) accelerator is described in Section 3.3, which takes bit-parallel matrices and transforms them into the required bit-serial data format. We also provide a cost model to estimate the resource usage for a given set of parameters as described in Section 3.4.

Software Programmability. Our hardware architecture is software-programmable at the granularity of instructions as described in Section 3.5. This offers several advantages such as the ability to tailor block sizes and dynamically skip bit positions for sparse or approximate computing.

3.1 Hardware Architecture Overview

Fig. 2 provides an overview of the BISMO hardware. The architecture is organized into three pipeline stages: *fetch*, *execute*, and *result*. Each stage communicates data to the next stage via shared on-chip memory buffers. Inter-stage synchronization is achieved by blocking reads and writes to synchronization FIFOs. All stage operations, including datapath control and synchronization, are controlled by instructions, which are fetched from instruction queues and executed in order. In addition to these stages, there is a Parallel-to-Serial (P2S) component for data layout conversion (Section 3.3), which is incorporated into BISMO as an optional, standalone accelerator.

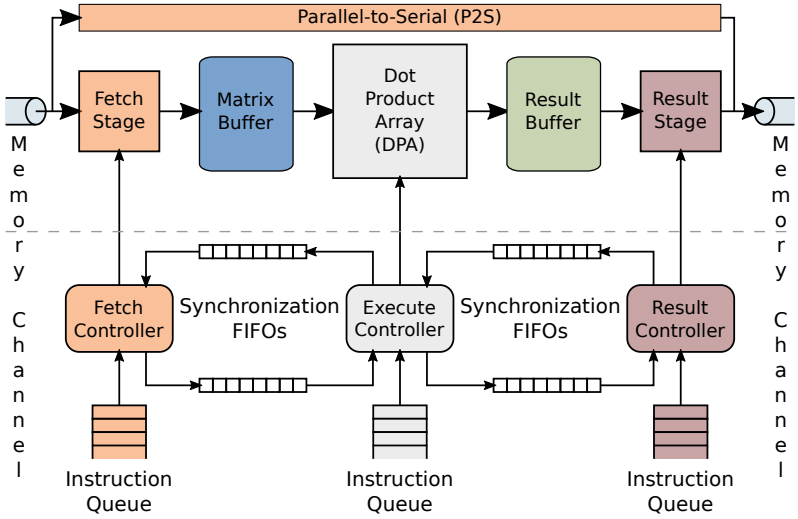

Fig. 2. Overview of BISMO's' hardware architecture.

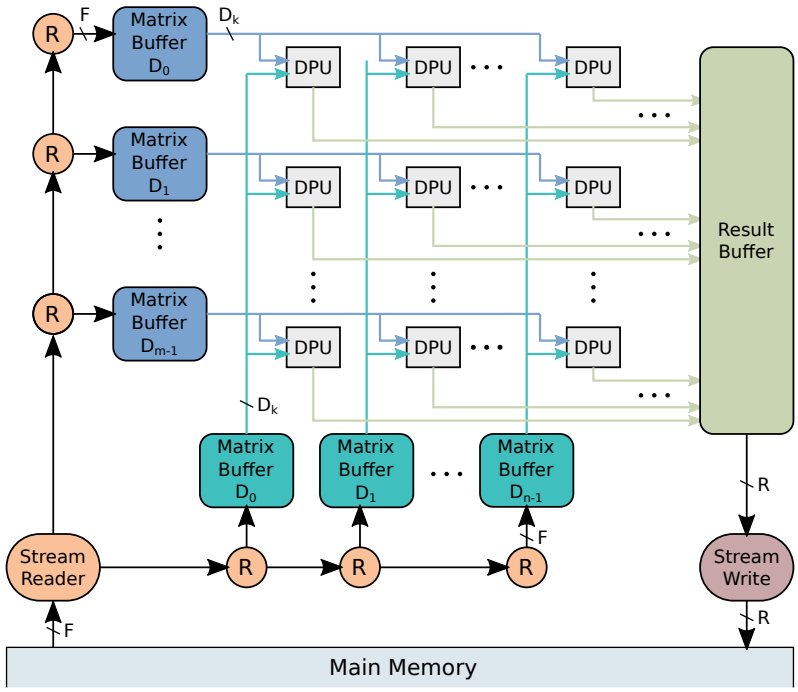

Fig. 3. Key components of the BISMO datapath.

The core of the hardware architecture is the bit-serial matrix-matrix multiplication datapath illustrated in Fig. 3. Accelerator performance and resource usage can be controlled by the parameters specified in Table 1.

Table 1. Key BISMO hardware parameters.

Symbol	Description
D_m, D_n	Number of DPUs in the DPA
D_k	DPU input bit width (popcount width)
B_m, B_n	Depth of input matrix buffers
B_r	Depth of result matrix buffer
A	Accumulator bitwidth
F	Main memory read channel bit width
R	Main memory write channel bit width
M	Maximum bit-parallel bitwidth for P2S

The Fetch Stage is responsible for reading matrix data from main memory and populating the matrix buffers with data. Internally, the fetch stage contains a simple DMA engine and route generator called a *StreamReader*, as well as a linear array interconnect. The StreamReader sends read requests to main memory and determines where read responses are to be written, as specified by fetch instructions. The read data and its destination form a packet that is carried through the interconnect to the appropriate matrix buffer. The interconnect is bandwidth-matched to the main-memory read channel to avoid any bottlenecks and ensure efficient use of off-chip bandwidth. The synchronization with the execute stage is ensured prior to fetching data, which greatly simplifies the design of the interconnect as there is no back pressure. The fetch stage can be scaled at design time to match the memory read bandwidth (F) of a particular platform.

The Execute Stage is responsible for performing the matrix multiplication on the data present in the matrix buffers. The core of the stage consists of an array of dot product units (DPUs), where each DPU is fed with a design-time configurable number of bits (D_k) from the left-hand-side and right-hand-side matrix buffers. The DPUs on the same row of the data processing array are fed with the same data broadcasted by the left-hand-side matrix buffers. Similarly, the DPUs on the same column are fed with the same data broadcasted by the right-hand-side matrix buffers (Fig. 3). A single software controllable sequence generator is responsible for reading out the appropriate data from the matrix buffers. The same generated sequence is used for both the left- and right-hand-side matrix buffers but with different offsets. The execute stage can easily be scaled at design time by configuring the number of rows (D_M) and columns (D_N) of DPUs. Part of the contribution of this work is a version of the DPU that is optimized for Xilinx FPGAs. Both the original BISMO DPU and the improved version are described in further detail in Section 3.2.

The Result Stage is responsible for writing the results generated by the execute stage to main memory. The stage consists of a *StreamWriter*, which contains a downsizer (wide-in-narrow-out) to resize the array of results into the appropriate width needed by the memory channel and a DMA engine with striding support to carry out the actual memory write operations. The striding is needed to produce the result matrix one tile at a time. When the execute stage has produced a new set of results, the accumulated dot-products are written to the result buffer, from which the result stage writes them to main memory. This enables the two stages to work independently and to overlap computation and data transfer. The result stage can be scaled at design time to match the memory write bandwidth (R) of a particular platform.

Parallel-to-Serial (P2S) is an optional component that converts bit-parallel matrices commonly used for CPUs into bit-serial ones as required by BISMO. The P2S does not communicate with

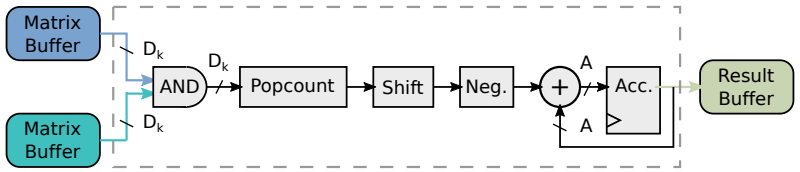

Fig. 4. The original BISMO dot product unit (DPU).

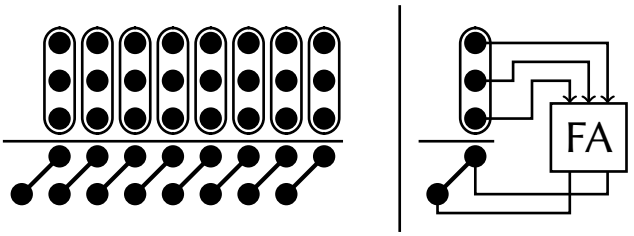

Fig. 5. Example matrix compression using carry-save-addition

the regular BISMO stages and is invoked as a separate, standalone accelerator. Its architecture is further described in Section 3.3.

3.2 The Dot Product Unit

The dot product unit (DPU) forms the core of the BISMO execute stage. Each DPU performs a bit-serial dot-product operation between two weighted binary vectors. Here, we start by describing the DPU of the original BISMO [19] and its shortcomings in terms of how it maps to FPGAs (Section 3.2.1). Afterwards, we discuss a new DPU implementation with an FPGA-optimized compressor (Section 3.2.2) and an improved datapath (Section 3.2.3).

3.2.1 Original BISMO DPU. The original DPU [19] can be seen in Fig. 4. The DPU computes a partial result of the dot product between a row and column of two bit-matrices, line 12 in Algorithm 1. The single-bit multiplications are performed by bitwise logic AND operations and the summation is a simple population count (popcount) of the result. The weight in Algorithm 1 is implemented by a left-shift unit and an optional negation, which are controllable by software. The partial results are accumulated and stored in a register (Acc.) of width A, which is typically 32 bits to avoid overflows [17, 18]. The shortcomings of this DPU architecture are twofold:

(1) The binary multiply and accumulate operation is implemented as a bitwise AND followed by a popcount unit built as a tree of 6:3 popcount operators and adders. Especially with large D_k, the popcount unit can require a large number of LUTs and many stages to pipeline the adder tree.
(2) The number of positions to left-shift the AND-popcount result is supplied dynamically, which requires an expensive barrel shifter.

3.2.2 Efficient AND-Popcount for Xilinx FPGAs. The input to a popcount operation is a column of equally weighted bits, which are to be summed up. While exhibiting an extreme aspect ratio of $D_k \times 1$, the input still forms a bit heap, which can be reduced by standard matrix compression techniques. Step by step, reshaped matrices are derived. They increase in width introducing more and more higher weight bits but decrease in height while always maintaining the numeric sum of the matrix rows. Only the final summation into a single row representing the conventional binary result

requires an addition with a critical carry propagation. All preceding compression steps can rely on parallel counters with bounded critical path lengths that are independent from the matrix width.

For the general idea of carry-free bit heap compression, refer to Fig. 5. It shows a carry-save addition using regular full adders operating in parallel to reshape a three-row input matrix into a two-row output matrix with the same arithmetic sum. The customary representation as a dot diagram abstracts the individual input and output bits into plain dots. The numeric weight of each bit is determined by its column, just as in the binary number system. In fact, when reading each row as a binary number, the compression maintains the invariant that the sum of the three input numbers equals the sum of the two output numbers. The structural implementation of the compression is implied by encircling the inputs to and connecting the outputs of each bit counter, i.e., simple full adders in this case. Note that the carry outputs of these full adders move up by a column as their numeric weights are two times higher than that of the associated sum bits. Also, notice that the combinational delay of this carry-save compression is a single full adder irrespective of the actual width of the matrices.

More sophisticated parallel counters have been proposed and implemented specifically targeting an efficient mapping to FPGA devices [7, 8, 14, 16]. We leverage the open-source set of parallel counters and the associated generic compressor implementation for Xilinx FPGAs proposed by Preußer [16]. It produces solutions optimized for our target FPGA architectures and integrates easily into a regular synthesis flow. Its efficacious greedy scheduling of parallel counters avoids optimization efforts that would be intolerable within a design cycle.

The parallel counters used by the chosen generic compressor implementation are mapped explicitly to concrete physical device primitives of the targeted Xilinx devices. While this approach certainly enables highly optimized implementations, its high degree of specialization also implies an inflexible operator interface. It practically leaves no opportunities for the synthesis tool to optimize the implementation within the context of the surrounding logic. In our particular case, we actually need a *fused* And-popcount operator. Optimizing the popcount alone isolates trivial 2-input And gates at its inputs. These are, in the end, greatly underutilizing the functional capabilities of the 6-input LUTs found on modern FPGA devices.

In order to eliminate this interfacing inefficiency, we designed a physically fused operator implementation by preceding the generic compressor with an equally rigorously optimized pre-compression. Instead of computing individual bit products, they are combined into groups of three whose computations are absorbed into the equivalent of a full-adder compression. Note that all these groups can be pre-compressed independently and in parallel. The computation implementing this functionality is depicted in Fig. 6. It can be mapped directly to two 6-input LUTs. It is worth noting that this pre-compression favorably changes the geometry of the bit heap input to the generic compressor. Instead of feeding a $D_k \times 1$ matrix, the pre-compression already reduces this height to $\lceil D_k/3 \rceil$ while spreading the input across two columns.

The structure of the complete summation process of a 32-bit popcount operation is illustrated in Fig. 7. Following the convention to encircle the inputs and to connect the outputs of a counter primitive, it shows the summation structure generated by the algorithm proposed by Preußer [16]. It comprises two compression steps with parallel bit counters and a final carry-propagating ternary addition. Identifying the bit counters by the individual heights of their input columns from left to right, a pair of $(5, 2)$-counters and a (6)-counter accomplish the first parallel compression step. This is only followed by one other small compression through a single $(5, 2)$-counter prior to the carry-propagating summation. In this particular case, only the third column of the compression result is high enough to introduce the second, only locally forwarded carry signal that is typical for a ternary addition. The carry propagation chain is terminated by a final half adder.

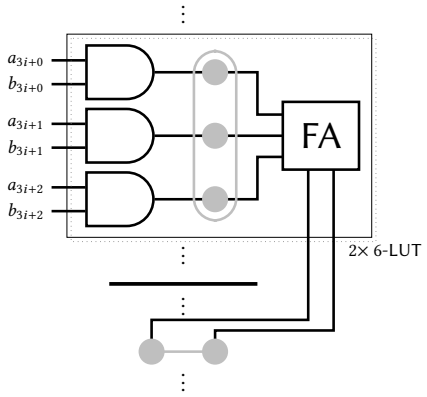

Fig. 6. Fused pre-compression of three bit products

Table 2. Bit counter statistics across operator sizes

N	(2,5:*]	(6:3]	(3:1]	Slice
32	3	1		
64	4	3	2	1
128	8	7	5	3
256	17	13	3	9
512	38	25	4	19
1024	79	51	6	38

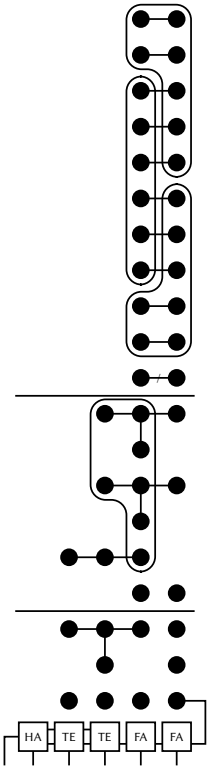

Fig. 7. Continued compression of 32-bit popcount

The first compression step is dominated by $(5, 2)$-counters for all larger operator sizes. The pre-compressed two-column input is too narrow for slice-based counters leaving the $(5, 2)$-counters as the most economic choice. The bits their application leaves, predominantly in the second column, are mostly handled by (6)-counters just as shown in the 32-bit example. Full adders would take care of fewer leftover bits. As larger operator implementations reach wider intermediate bit heap geometries before the final addition, they will also utilize slice-based counters in these later compression steps. These counters leverage the carry chain to combine four LUTs of a slice to obtain counter primitives optimized for the target device architecture . An overview of the use of the different counters in our designs is given by Tab. 2.

We employ a fully pipelined compressor with register stages separating all compression steps to optimize the operating frequency of the AND-popcount reduction. It is worth to mention that we had to replace the trivial behavioral register description by an explicit instantiation of FDRE register primitives in order to avoid excessively growing synthesis times for larger operators. It appeared that the Xilinx Vivado synthesis engine [21] had a hard time or was trying too hard to optimize the many interfaces between behavioral code and the netlists of primitives generated for the compression steps.

3.2.3 Improved DPU. The barrel shifter in the original BISMO DPU is needed to account for the differences in weight between the accumulator and the contribution. This difference depends on the order in which the bits of the L and R matrices are traversed. First, we note that the loop nest in Algorithm 1 is affine, and the L and R bit positions (variables i and j) can be traversed in any order

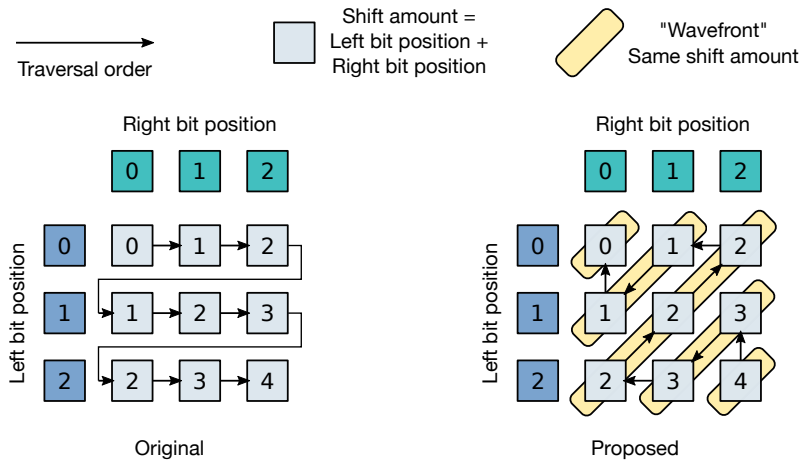

Fig. 8. Original and proposed bit position traversal order.

as long as the correct weight is applied. Based on this observation, we propose to traverse the bit positions as shown in Fig. 8. Here, the sum of L and R bit positions constitute *wavefronts*, where each wavefront has a left-shift value that is one less than the previous one. Using this schedule, instead of left-shifting the current contribution by a variable amount with a barrel shifter, we can left-shift the previous accumulator by either one position (if changing wavefronts) or use it as-is, before summing the accumulator and the contribution. The optional negation is still applied to the current contribution, when needed for bit position combinations that yield a negative result. Combined with the new AND-popcount unit (Section 3.2.2), this yields the improved DPU design illustrated in Fig. 9, where the barrel shifter is replaced with a constant one-left-shift and a multiplexer. Barring accumulator overflows, our new DPU is able to handle any input precision, whereas the original DPU was limited by the maximum left-shift supported by the barrel shifter.

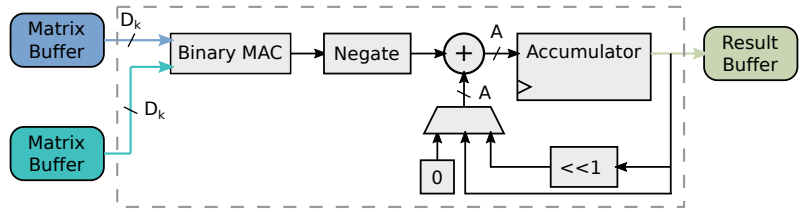

Fig. 9. The optimized dot product unit (DPU).

3.3 Bit-Parallel to Bit-Serial Matrix Transformation

As described in Section 2.2, BISMO assumes that the input matrices are present in main memory, using a bit-serial data layout. However, due to the bit-parallel nature of the arithmetic in general-purpose CPUs, matrices are almost always stored using a bit-parallel data layout in practice. Furthermore, as CPUs typically offer 8-bits as the smallest native data type, matrices that require fewer bits are also stored using 8-bit data types. Conversion from bit-parallel to bit-serial can be a costly operation, whose cost must be taken into account as part of the accelerator performance.

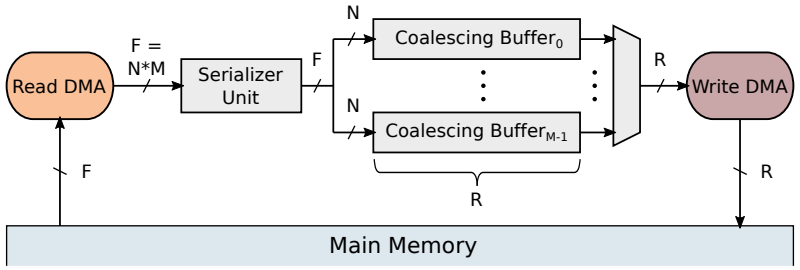

Fig. 10. High-level view of the P2S accelerator.

To address this problem for BISMO, we enhance it with a stand-alone parallel-to-serial (P2S) accelerator. The accelerator, illustrated in Fig. 10, is a data-layout transformer with run-time configurable precision. The P2S retrieves a bit-parallel matrix (left-hand side in Fig. 11), transforms it into a bit-serial matrix (right-hand side), and writes it back to main memory. The P2S *read DMA* sequentially fetches the column elements constituting a row of the bit-parallel matrix from main memory and feeds these into the *serializer unit*. The individual bits of each column element are split up across as many *coalescing buffers* as the bit-precision of the parallel matrix. This is repeated for all the rows of the input matrix. The bit precision and number of rows and columns of the parallel matrix are runtime configurable. The total number of coalescing buffers defines the maximum supported precision M of the bit-parallel input matrix and is specified at synthesis time. The coalescing buffer size is given by the bitwidth of the write bus, which is also specified at synthesis. We assume that the bitwidths of the read and write buses are given by the BISMO parameters F and R specified in Table 1.

Fig. 11 illustrates an example of a 4-bit parallel matrix of size 2x64 where each column element has been padded to eight bits and 64-bit read and write data buses are used. Eight column elements (total 64 bits) of the bit-parallel matrix are fetched on each memory access. The four most significant bits of each 8-bit column element are padding and are discarded (i.e., the actual precision is specified to be four bits by the P2S instruction at runtime). The remaining four bits are split across four different coalescing buffers, one for each bit weight (B_0-B_3). The column index within the row dictates the bit position written in the coalescing buffers. As shown in the example, the final column (C_{63}) of the second row (R_1) is written to the last bit position (c_{63}) of the coalescing buffers that are allocated for the row (B_0-B_3). If the row of the bit-parallel matrix contains more columns than bit positions in the coalescing buffer, then the P2S kernel stalls to write back the coalescing buffers to main memory before continuing the transformation of the remaining columns. The allocated coalescing buffers are also written back to main memory when a new row is encountered in the bit-parallel matrix (e.g., R_1).

To simplify the implementation, the number of columns of the bit-parallel matrix has to be a multiple of the coalescing buffer bit-width (R). This requires some input matrices to be padded but greatly simplifies the write back of the bit-serial matrix. This ensures that the coalescing buffers are completely filled and can be written back to memory without requiring the data to be realigned. The binary matrices are stored consecutively, i.e., all the rows of binary matrix B_0 are stored together which are then followed by B_1 and so on. This requires the coalescing buffers to be written back in a strided fashion with B_0-R_0 being written together with B_1-R_0 and a stride equal to the size of a complete binary matrix.

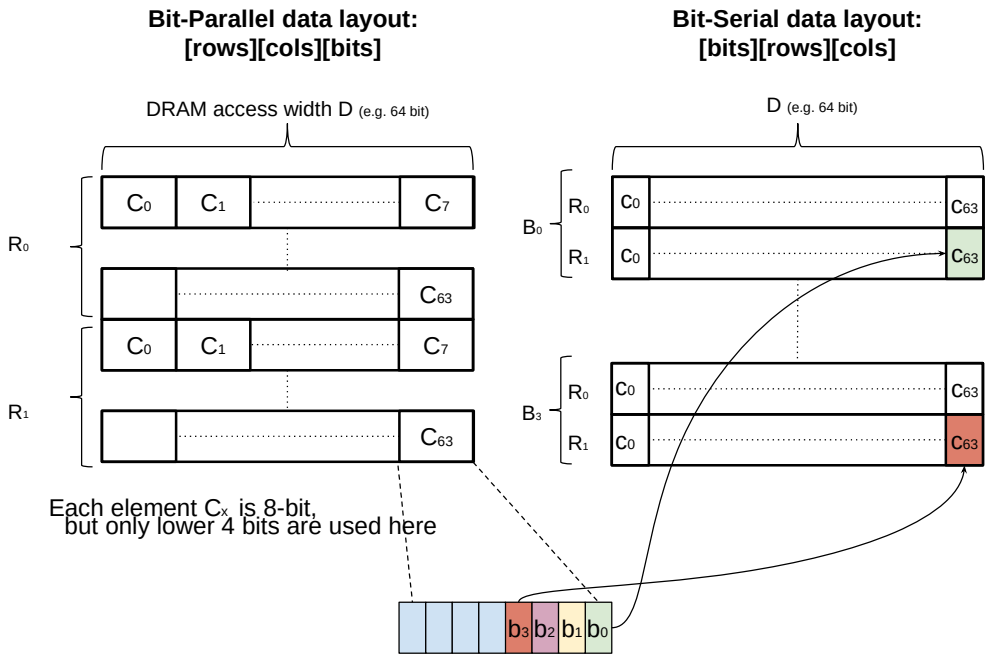

Fig. 11. An example of a memory layout for an example matrix of 2x64 of 4 bits.

3.4 Cost Model

For any parametrizable overlay architecture, it is beneficial to provide a model of how the FPGA resource usage relates to its configuration parameters. This enables a quick performance estimation when scaling to other devices.

3.4.1 LUT cost. We propose the following equations to model the LUT usage of a BISMO instance:

$$\text{LUT}_{\text{total}} = \text{LUT}_{\text{base}} + \text{LUT}_{\text{array}} \tag{1a}$$

$$\text{LUT}_{\text{array}} = D_m \cdot D_n \cdot (\text{LUT}_{\text{DPU}} + \text{LUT}_{\text{res}}) \tag{1b}$$

$$\text{LUT}_{\text{DPU}} = \alpha_{\text{DPU}} \cdot D_k + \beta_{\text{DPU}} \tag{1c}$$

Equation 1a breaks the total cost into LUT_{base}, which covers the DPA size-independent LUT usage such as the DMA engines, P2S and other fixed platform infrastructure, and $\text{LUT}_{\text{array}}$ which covers the DPA size-dependent part. In turn, Equation 1b further breaks down $\text{LUT}_{\text{array}}$ into LUT cost for the DPU and for result generation, multiplied by the array size. Finally, we model LUT_{DPU} as a linear function of the popcount width D_k in Equation 1c, and LUT_{res} as a constant. The constants $\alpha_{\text{DPU}}, \beta_{\text{DPU}}, \text{LUT}_{\text{base}}$ and LUT_{res} are determined empirically in Section 4.1.

3.4.2 BRAM cost. Assuming dual-port 36×1024-bit Xilinx BRAMs, we model their usage as:

$$\text{BRAM}_{\text{total}} = \text{BRAM}_{\text{base}} + \text{BRAM}_{\text{array}} \tag{2a}$$

$$\text{BRAM}_{\text{array}} = \left\lceil \frac{D_k}{32} \right\rceil \cdot \left(D_m \cdot \left\lceil \frac{B_m}{1024} \right\rceil + D_n \cdot \left\lceil \frac{B_n}{1024} \right\rceil \right) \tag{2b}$$

Table 3. BISMO's Instruction Summary

Instruction type	Fields
`Wait` & `Signal`	Associated FIFO: Fetch stage: Execute Execute stage: Fetch or Result Result stage: Execute
`RunFetch`	Source (main memory) parameters: Base address Block size (bytes) Block offset (bytes) Number of blocks to fetch Destination (matrix buffer) parameters: Matrix buffer offset Starting matrix buffer Range of matrix buffers Consecutive words per matrix buffer
`RunExecute`	Matrix buffer offset Dot product length Negate contribution mode Accumulator shift mode
`RunResult`	Result base address in main memory Address offset
`RunP2S`	Bit-parallel base address in main memory Bit-serial base address in main memory Number of rows and columns Actual precision

In Equation 2a, $\text{BRAM}_{\text{base}}$ refers to the BRAMs used for DPA-size independent infrastructure, such as DMA buffers and instruction queues. $\text{BRAM}_{\text{array}}$ is the cost for the input matrix buffers. We use 32 of the native 36-bit width due to constraints from the fetch stage, since DRAM buses are typically power-of-two-wide and we require BRAM read/write widths to be an integer multiple of each other. We assume that the result matrix buffer consists of small LUTRAM buffers, and cover their cost in Equation 1b.

3.5 Programming BISMO

BISMO provides programmability through the use of instructions that control each of the pipeline stages and the P2S. Taking into account the dimensions of the input matrices and the data layout in memory, it is possible for a programmer to perform scheduling in various ways. The capabilities facilitated by these instructions and their usage are illustrated in this section.

3.5.1 Instructions. There are three types of instructions per pipeline stage in BISMO, namely `Wait`, `Signal` and `Run`. The P2S is treated as a separate accelerator synchronized at a coarser level, and only has `RunP2S`. Table 3 provides a summary of these instructions with the usage described as follows:

The Synchronization Instructions are used for synchronization between two different pipeline stages. The `Signal` instruction issues a token to the associated synchronization FIFO, while the `Wait` instruction blocks on the associated synchronization FIFO until it receives a token. For both the fetch and result stage, the only associated synchronization FIFO is their respective FIFO for the

execute stage. The execute stage has consequently two associated FIFOs for synchronization with either the fetch or the result stage. The tokens do not convey any information and a programmer is free to decide what each synchronization represents, e.g., that a particular matrix buffer is now full or empty. We note that the P2S is treated as a separate accelerator synchronized at a coarser level, and cannot be the source or destination for any synchronization instructions.

The Run Instructions are used to carry out the particular function of a pipeline stage.

The `RunFetch` instruction specifies from where in main memory to read data and the destination matrix buffers to store the read data. The parameters with regard to main memory are: *i)* the base address from where the fetch should begin, *ii)* the size of the contiguous block to be fetched, *iii)* the offset between such blocks (providing strided accesses), and *(iv)* the number of blocks to be fetched. The parameters with regard to matrix buffers are: *i)* the buffer offset at which to start writing data, *ii)* the matrix buffer to begin writing to (all buffers are enumerated from zero to $D_m \cdot D_n - 1$), *iii)* the range of matrix buffers to be written (number of consecutive buffers), and *iv)* the number of consecutive words to be written in each matrix buffer before switching to the next. These set of parameters enable consecutive data blocks to be placed in one matrix buffer before moving to the next or to place the blocks in a cyclic fashion across a range of buffers.

The `RunExecute` instruction specifies the matrix buffer offset from where to begin reading data, how many buffer addresses will be read, whether to negate the current contribution, and whether to accumulate with a zero, the accumulator register, or the accumulator register left-shifted by one.

The `RunResult` instruction specifies the base address of the result matrix stored in main memory and an offset to which the current results are to be written.

Finally, the `RunP2S` instruction specifies a source matrix expressed by its base address in main memory, spatial dimensions (rows and columns) and actual bit-parallel precision, i.e., how many bits starting from the least significant bit should be converted, as well as, a main memory address where the resulting bit-serial matrices are to be written.

3.5.2 Instruction Scheduling. Using conventional block matrix multiplication algorithms that were previously applied to FPGA matrix multiplication accelerators [10], BISMO can process matrices of any dimension. Fig. 12 shows one possible schedule for the matrix multiplication example in Fig. 1. Here, the DPA is assumed to be as large as the input matrices for simplicity. The computation would otherwise have to be divided into separate tiles resulting in many more instructions. Furthermore, it is assumed that only three of the four binary matrices ($L^{[1]}$, $L^{[0]}$, $R^{[1]}$, and $R^{[0]}$) fit in the matrix buffers at the same time to demonstrate the off-chip tiling capabilities. The P2S is not part of the example as it is assumed that the input matrix has already been converted to the bit-serial layout. The corresponding instructions for each pipeline stage can be seen in Table 4, with P denoting the matrix that accumulates the result of these operations.

The fetch stage begins by fetching $L^{[1]}$ and $R^{[1]}$ (instruction F1 and F2) and then signals the execute stage (F3) that it can perform the first binary-matrix multiplication (E2). While the execute stage computes the dot product between $L^{[1]}$ and $R^{[1]}$, the fetch stage continues fetching $L^{[0]}$, effectively achieving an overlap between data fetch and execution (F4 and E2 performed in parallel). Once the execute stage finishes the first binary-matrix multiplication, it receives the signal from the fetch stage (F5) that $L^{[0]}$ resides in the matrix buffers (E3). The execute stage continues by executing $L^{[0]} \cdot R^{[1]}$ (E4) while the fetch stage has to wait since all the buffer space is occupied (F6). Note that E4 is part of a new wavefront, and the previous accumulator is left-shifted by one by setting the appropriate accumulator mode to account for this. When the execute stage finishes the matrix multiplication, it signals the fetch stage (E5). Since $R^{[1]}$ is no longer needed, the fetch stage fetches $R^{[0]}$ (F7) enabling the execute stage to finish the remaining matrix multiplications (E7 and E8). As this is the next step in the wavefront, this requires the accumulator to be shifted again.

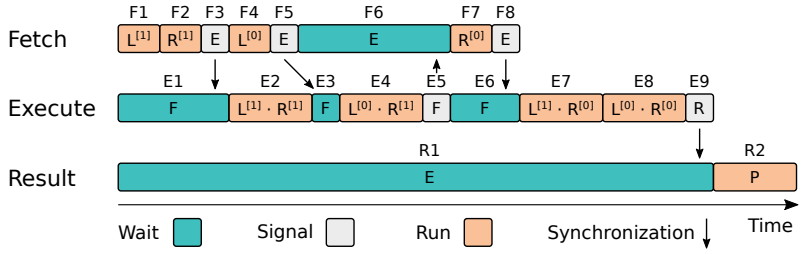

Fig. 12. Timeline of the example schedule shown in Table 4.

Table 4. Initialized Instruction Queues for the Example Shown in Fig. 1

Fetch	Execute	Result
F1 Run $L^{[1]}$	E1 `Wait` Fetch	R1 `Wait` Execute
F2 Run $R^{[1]}$	E2 Run P = P + $L^{[1]} \cdot R^{[1]}$	R2 Run P
F3 `Signal` Execute	E3 `Wait` Fetch	
F4 Run $L^{[0]}$	E4 Run P = (P«1) + $L^{[0]} \cdot R^{[1]}$	
F5 `Signal` Execute	E5 `Signal` Fetch	
F6 `Wait` Execute	E6 `Wait` Fetch	
F7 Run $R^{[0]}$	E7 Run P = P + $L^{[1]} \cdot R^{[0]}$	
F8 `Signal` Execute	E8 Run P = (P«1) + $L^{[0]} \cdot R^{[0]}$	
	E9 `Signal` Result	

Once the execute stage has finished all binary matrix multiplications, it signals the results stage (E9) which writes the result P to main memory (R2).

The schedule in Fig. 12 causes the fetch stage and execute stage to stall (F6 and E6) since there is not enough space to fetch $R^{[0]}$ before $L^{[0]} \cdot R^{[1]}$ has been computed. An alternative schedule could be to split the binary matrices into tiles enabling greater flexibility in what data to bring into the matrix buffers and the possibility of overlapping fetch and execute.

4 EVALUATION

We implement the improved BISMO parametrizable hardware generator in Chisel [6] and VHDL, and use Xilinx Vivado 2017.4 [21] for synthesis, placement, and routing. We add registers to critical paths on the pipeline and enable register retiming instead of manual floorplanning and timing optimizations to achieve higher clock frequencies. We target the Ultra96 board [2], which has a Xilinx ZU3EG MPSoC [24] containing an FPGA with 71k LUTs and 214 BRAMs, and a quad-core ARM Cortex-A53 CPU. The accelerator is connected to a 64-bit wide AXI high-performance port, provisioning it with 4.8 GB/s of DRAM bandwidth when running at 300 MHz. The BISMO software stack and runtime are coded in C++, and executes on a single ARM core. We use Ubuntu 18.04 provided with the PYNQ platform [22] for Ultra96 as the operating system, and the PYNQ PMBUS interface for power measurements.

As binary operations are the building block for bit-serial computations, we use them as the common denominator for performance measurements. We treat And and popcount as analogues to multiplication and addition when counting binary operations, i.e., a binary dot product between two N-element binary vectors is counted as $2N$ binary operations.

4.1 Synthesis Results and Resource Cost

We start by presenting synthesis results across a range of parameters for different components of the BISMO architecture. Our aim is to explore the resource cost of scaling performance along different axes of parallelism and building up a hardware cost model in the process. Unless otherwise stated, all data in this section is obtained by using out-of-context synthesis for the ZU3EG FPGA, with a target clock period of 2 ns to prioritize timing optimizations.

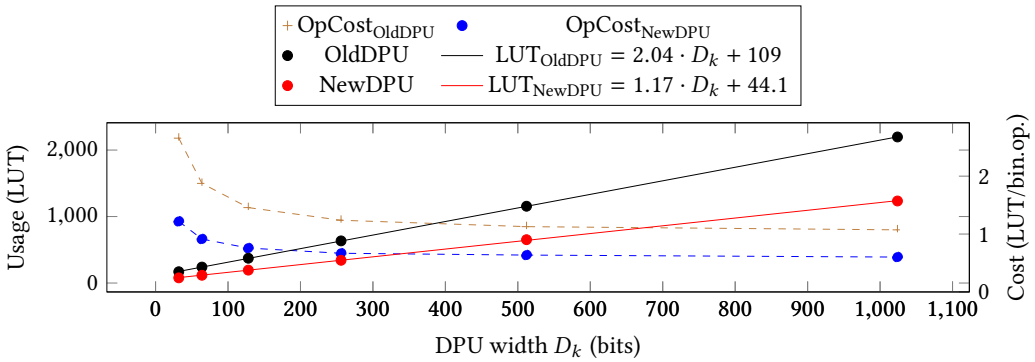

Fig. 13. DPU LUT usage and efficiency characterization.

4.1.1 Dot Product Unit. We start by characterizing the resource cost of the DPU, which constitutes the core computational unit of our overlay. Fig. 13 plots the LUT usage as well as the LUT cost per binary operation of both the original and the improved DPUs. Similar to the original BISMO, the improved DPU resource cost includes the components whose sizes is constant and does not scale with D_k, such as the accumulator and mode multiplexer. We expect that their resource cost gets amortized for larger values of D_k, making up a smaller proportion of the total DPU. The dashed lines in Fig. 13 plot the LUT cost per binary operation. We observe that the cost per binary operation for the improved DPU starts at 1.2 LUTs for $D_k = 32$, decreasing to 0.6 LUTs for $D_k = 1024$. Compared to the original BISMO with 2.6 LUTs for $D_k = 32$ and 1.07 LUTs for $D_k = 1024$, this constitutes an improvement of 1.8×. Using linear regression on this data, the parameters α_{DPU} and β_{DPU} of the BISMO cost model (Section 3.4.1) are 1.17 and 44.1, respectively. We note that the additive constant β_{DPU} for the improved DPU is 44.1 compared to 109 for the original DPU, decreasing the per-DPU overhead by 60% due to the removal of the expensive barrel shifter. For the improved DPU, the reported maximum frequency (F_{max}) is between 600 and 719 MHz.

4.1.2 Fetch and Result Stage. We evaluate the cost of the fetch and result stages for a single 64-bit memory channel on the PYNQ-Z1, with F=R=64, A=32, and B_r=2. The fetch stage includes a DMA engine and the interconnect to move data into matrix buffers. We observe that the LUT cost of the fetch stage is approximated well by $1.89 \cdot (D_m + D_n) + 463$. We do not include the $1.89 \cdot (D_m + D_n)$ component in the cost model since it is small even for large DPAs. The result stage includes a DMA engine, result matrix buffers, and a downsizer (parallel-to-serial unit), which are all implemented using LUTs. The result buffer requires approximately $87.3 \cdot D_m \cdot D_n$ LUTs, while the DMA engine and the downsizer need $32.8 \cdot D_m \cdot D_n + 255$ LUTs. Completing the cost model, the fetch and result stages contribute $463 + 255 = 718$ LUTs to LUT_{base}, which may increase with more advanced DMA engines, and the LUT cost per DPU associated with the result stage is $\text{LUT}_{\text{res}} = 87.3 + 32.8 = 120.1$.

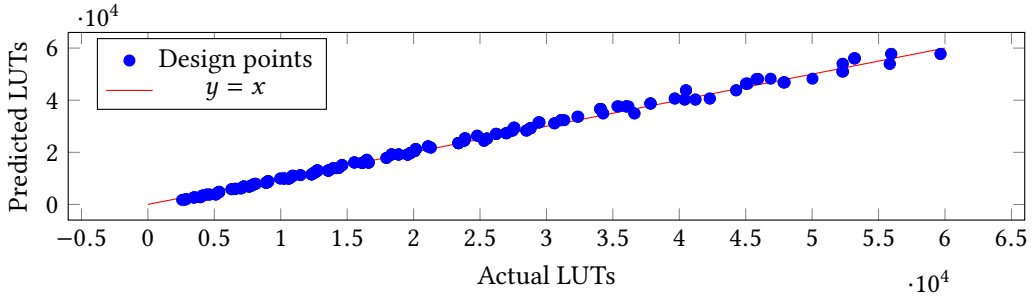

Fig. 14. Predicted vs actual LUT usage.

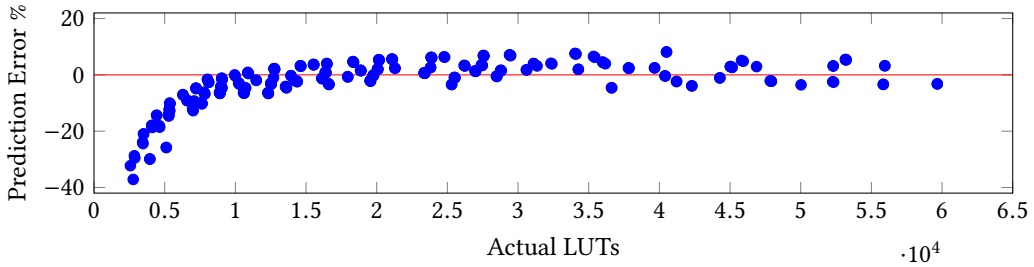

Fig. 15. LUT cost model prediction error with design size.

4.1.3 Parallel-to-Serial Accelerator Resource Cost. To evaluate the cost of the hardware-accelerated data-layout conversion, we evaluate the P2S with $M = 8$ since 8-bit is the smallest natively supported bit-parallel datatype for most CPUs. For $F = R = 64$ the P2S contributes 929 LUTs to LUT_{base}. Currently the majority of these LUTs are used for multiplexing between the coalescing buffers when writing their contents to DRAM. As the access pattern to the coalescing buffers is quite regular, a more optimized interconnect can be deployed here to further reduce the LUT cost.

4.1.4 Cost model validation. We generated 295 different BISMO designs ranging from (D_m=2, D_k=64, D_n=2) to (D_m=12, D_k=256, D_n=10) in size to validate the cost models described in Section 3.4. The BRAM predictions were 100% accurate for this particular range of designs. Fig. 14 shows the LUT usage from synthesis results versus the prediction from the cost model. The model's prediction is 97.8% accurate on average across the tested sizes. Fig. 15 shows how the prediction error is affected by the size of the design. We observe that large designs are accurately predicted, while smaller designs tend to be underestimated by the model.

4.1.5 LUT-BRAM Tradeoffs. Fig. 16 shows three BISMO instances with the same performance and buffer depth but different overlay dimensions (D_m, D_k, D_n) and plot the number of BRAMs used and the LUT cost per binary operation. We observe a tradeoff between BRAM and LUT cost by scaling different parameters. We see that larger D_k results in lower LUT cost, but requires more BRAMs to deliver the bandwidth. Conversely, smaller D_k needs fewer BRAMs, but has larger LUT cost. We note that the DPA dimensions should be matched to the workload dimensions for higher efficiency, e.g., $D_n > 1$ is wasteful for matrix-vector multiplication, but LUT and BRAM budget may impose additional constraints.

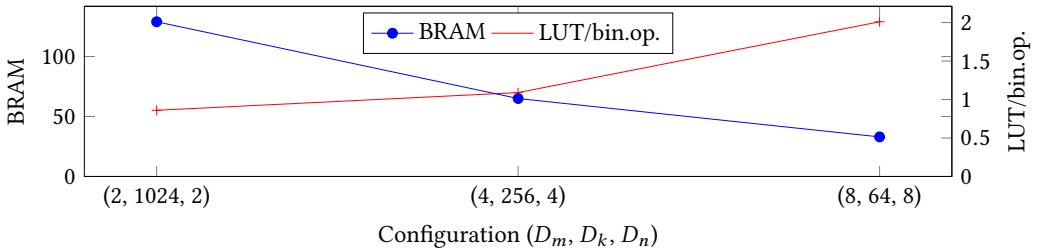

Fig. 16. LUT vs BRAM tradeoffs for 2.4 binary TOPS at 300 MHz.

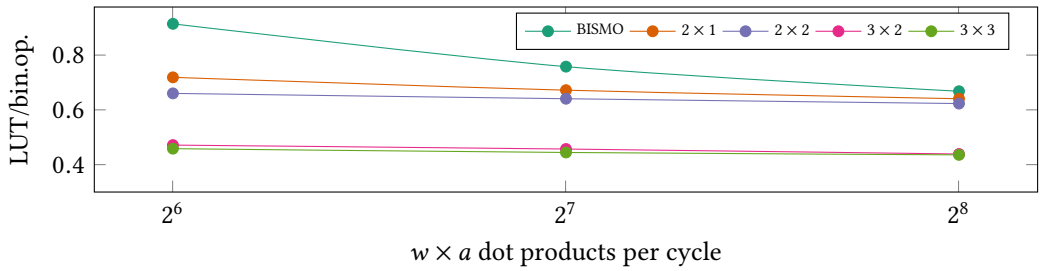

Fig. 17. Comparing the LUT/bin.op. cost of bit-serial and bit-parallel DPUs.

4.1.6 Hardware Cost of Flexible Precision. When the required precision is known beforehand, a matrix multiplier that uses fixed-precision bit-parallel arithmetic is the commonly used alternative, though bit-serial could still be used. To quantify the overhead associated with bit-serial for those cases, we implemented a version of the DPU with $w \times a$-bit multipliers instead of AND, an adder tree instead of popcount, and no shifter and negator. This bit-parallel DPU performs the equivalent of $2 \cdot w \cdot a \cdot D_k$ binary operations per cycle using the same compressor generator as the BISMO DPU, as explained in Section 3.2.2. Fig. 17 compares the LUT cost for binary operation equivalents between the BISMO DPU and several bit-parallel variants. We first observe that given the same number of bit-parallel operations ($w \cdot a$), the LUTs per binary operation decreases with higher bit-parallel precision from 0.72 for 2×1 down to 0.46 for 3×3 when performing 2^6 bit-parallel operations. As expected, bit-parallel DPUs have lower cost per bit operation compared to bit-serial as they do not suffer from the shifter/negator overhead. For larger dot product sizes, the overhead is amortized and the worst-case gap between BISMO and 3×3 closes down to 0.23 LUT per binary operation. We note that this is not a fully fair comparison since BISMO hardware supports significantly larger precisions compared to the fixed-precision operators here. We also expect this data to be useful for designers who would like to build *digit-serial* architectures, where the building block can be, e.g., 2×2-bit matrix multipliers instead of binary.

4.1.7 Scaling the BISMO DPA to Larger Sizes. BISMO scales performance by using a broadcast-style array of DPUs. Traditionally, semibroadcast or systolic arrays are preferred over broadcast for VLSI designs due to the high fan-out requirements of broadcast interconnects [9, 25]. However, modern FPGAs have massive on-chip routing bandwidth and a large number of flip flops for register duplication that can alleviate these concerns. To investigate how the BISMO DPA scales to larger sizes on Xilinx FPGAs, we ran a number of experiments targeting the Xilinx Virtex UltraScale+ VU9P [23] with out-of-context synthesis for large BISMO DPAs. Note that this assumes a LUT-bound design, i.e., we do not consider the matrix buffer BRAMs necessary to feed the array, only the DPA

Table 5. Large DPA synthesis results targeting Xilinx Virtex UltraScale+ VU9P.

D_m	D_k	D_n	LUT	FF	$F_{\max}$ (MHz)	Bin. TOPS
50	64	50	337,500	555,000	523.56	167.54
16	1024	16	313,856	720,640	543.77	285.09
32	1024	16	627,712	1,441,280	532.77	558.64
32	1024	24	941,568	2,161,920	498.01	783.30

Table 6. Improved BISMO instances for runtime measurements on the Ultra96.

#	D_m	D_k	D_n	LUT	BRAM	$F_{\max}$ (MHz)	GOPS
1	4	256	4	12,657 (18%)	65 (31%)	313.19	2,565.6
2	8	256	4	19,613 (28%)	97 (46%)	323.31	5,297.1
3	8	256	8	33,418 (48%)	129 (61%)	309.89	10,154.3
4	10	128	10	34,252 (49%)	81 (38%)	306.84	7,855.2
5	12	256	6	36,879 (53%)	145 (68%)	302.39	11,147.3
6	12	128	12	46,847 (67%)	97 (46%)	281.85	10,390.1
7	10	256	10	50,734 (72%)	161 (76%)	311.53	15,950.2

$F = R = 64$ and $F_{\text{clk}} = 300$ MHz.

itself. The results are summarized in Table 5. We observe that these large designs can still manage a respectable 500 MHz clock without any manual floorplanning. The largest synthesized design uses approximately 80% of the LUTs on this device, achieving 783 binary TOPS at maximum frequency.

4.2 Runtime Performance

In this section, we assess the runtime performance and energy efficiency achievable by the improved BISMO instances running on the Ultra96. We assume that the input matrices are stored in DRAM using a bit-packed data layout (Section 2.2) and that one matrix is transposed. We create matrix-multiplication workloads with different dimensions and bitwidths, manually build the corresponding instruction sequences, and run the workloads on the enumerated BISMO instances listed in Table 6 to evaluate how the overlay size interacts with workload size. We also reproduce the original BISMO results in Table 7 to demonstrate the improvements of the new BISMO. For instance, the $8 \times 256 \times 8$ instance is 27% smaller for the improved BISMO, and the design can be clocked 1.5× faster compared to the original. The resource improvement is due to the improved DPU design as the LUTs themselves are very similar between the two devices, while the clock improvement mainly comes from the process node improvement (16 vs 28 nm).

4.2.1 Peak Binary Compute. We start by measuring the maximum achievable binary matrix-multiply performance dictated purely by the execute stage. For this experiment, we assume that the matrices have already been fetched into on-chip memory and disregard the cost of result writing. Fig. 18 plots the achieved performance for different number of matrix columns (K) as a percentage of observed peak performance for different popcount widths (D_k). We observe that the efficiency increases with more columns, and that instances with larger D_k require wider matrices than smaller D_k ones to be efficient. As an example, for a matrix with 8192 columns (dotted line in Fig. 18), the instance with $D_k = 256$ reaches 68% efficiency, while $D_k = 128$ achieves 82%. Wide matrices achieve close to 100% of the peak performance for all instances. The inefficiency for narrow matrices is

Table 7. Original BISMO instances for runtime measurements on the PYNQ-Z1 [19].

D_m	D_k	D_n	LUT	BRAM	GOPS
8	64	8	19,545 (37%)	121 (86%)	1,638.4
8	128	8	27,740 (52%)	129 (92%)	3,276.8
8	256	8	45,573 (86%)	129 (92%)	6,553.6
4	256	4	13,352 (25%)	129 (92%)	1,638.4
8	256	4	24,202 (45%)	129 (92%)	3,276.8
4	512	4	21,755 (41%)	129 (92%)	3,276.8

$F = R = 64$ and $F_{\text{clk}} = 200$ MHz.

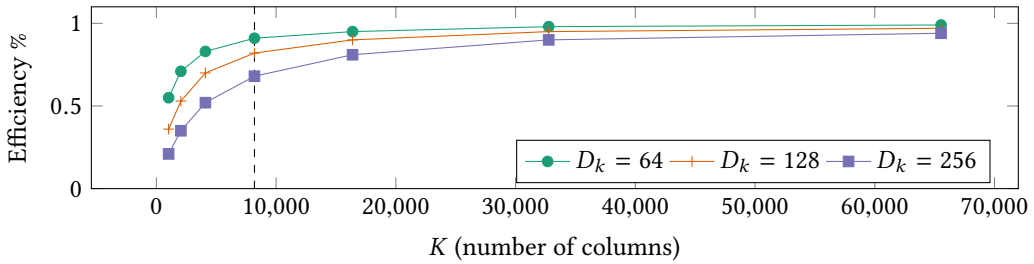

Fig. 18. Execute stage efficiency depending on D_k and number of columns K.

due to the lack of work to fill the compressor pipeline. For example, assume that a $D_k = 1024$ compressor pipeline has 10 stages, and is processing a dot product with $K = 6144$. This workload is fed to the compressor within 6 clock cycles, after which the execute stage controller must wait for the operation to complete to synchronize with the result stage, thus creating bubbles in the pipeline. This can be remedied by decreasing the DPA pipeline depth. As the improved BISMO DPU has fewer compressor stages compared to the original BISMO [19], we observe up to 10% relative improvement for the same D_k and matrix size.

4.2.2 Peak Bit-Serial Compute. Per Algorithm 1, if the runtime of a binary (1×1) matrix multiplication of a given size is t, we expect the runtime of a $w \times a$-bit matrix multiplication of the same size to be $w \cdot a \cdot t$. Fig. 19 plots the performance for matrices of size $10 \times 2048 \times 10$ and $10 \times 16384 \times 10$ with increasing w, a on instance #4. We observe slightly better performance than the projected $w \cdot a \cdot t$ since multiple dot products are accumulated together for the multi-bit case, behaving like a longer dot product and increasing the execute-stage efficiency (Fig. 18).

4.2.3 Stage Overlap. We now quantify the performance gain by overlapping the fetch, execute, and result stages for larger matrix multiplications. Using the block matrix multiplication algorithm from Matam et al. [10] we create an instruction sequence to run a $256 \times 4096 \times 256$ binary matrix multiplication on an $8 \times 64 \times 8$ instance. The input matrices here are twice the size of the on-chip memory, similar to the example in Section 3.5.2. By overlapping the operation of different stages, the multiplication finishes in 121,133 cycles, achieving a speedup of 2.2× compared to the 266,510 cycles when the stages are executing without overlap.

4.2.4 Power Consumption. We use the PMBUS interface on the Ultra96 to measure the total board power while running one or more stages in a loop to measure the power efficiency of BISMO. We

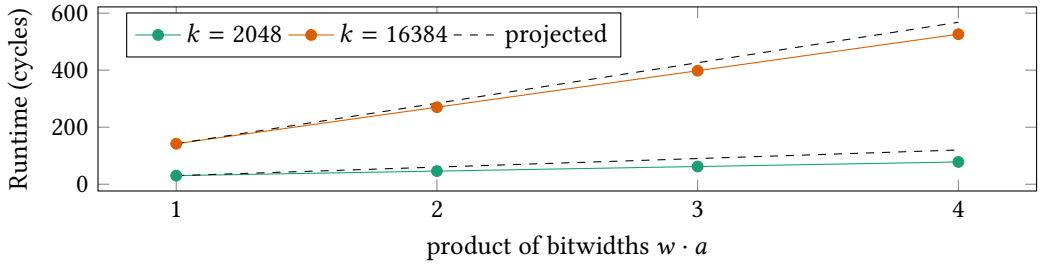

Fig. 19. Runtime with increasing precision on instance #4.

Table 8. Power consumption data from improved BISMO on the Ultra96.

Configuration (Instance, F_{clk})	Power (W) Idle	Exec	F & R	Full	Binary GOPS	Binary GOPS/W
(10x256x10, 50 MHz)	5.10	+0.01	+0.26	5.39	2,560.00	475.13
(4x256x4, 300 MHz)	5.39	+0.09	+0.30	5.76	2,457.60	426.67
(8x256x8, 300 MHz)	6.17	+0.17	+0.41	6.65	9,830.40	1,478.70
(10x256x10, 300 MHz)	6.76	+0.23	+0.36	7.20	15,360.00	2,133.33

Table 9. Power consumption data from the original BISMO instances on PYNQ-Z1 [19].

Configuration (Instance, F_{clk})	Power (W) Idle	Exec	F & R	Full	Binary GOPS	Binary GOPS/W
(8x64x8, 200 MHz)	2.53	+0.33	+1.09	4.07	1,638.00	402.16
(8x128x8, 100 MHz)	2.10	+0.19	+0.87	3.11	1,638.00	527.51
(8x256x8, 50 MHz)	1.76	+0.30	+0.63	2.53	1,638.00	646.39
(4x256x4, 200 MHz)	2.53	+0.34	+1.09	3.86	1,638.00	424.98
(8x256x4, 100 MHz)	2.05	+0.24	+0.92	3.06	1,638.00	536.02
(4x512x4, 200 MHz)	2.87	+0.71	+1.19	4.64	6,554.00	1,413.39

turn off the wireless interfaces on the Ultra96 to obtain better idle power readings. Table 8 lists the power consumption of four instances. In the top part of the table, we compare three different design points with similar performance, while the bottom part are top-performance designs. We list four power readings: the idle power with no stages running, the increment from idle with only the execute stage running, the increment with only the fetch and result stages running, and the full power with all stages running.

Overall, the idle power on the Ultra96 constitutes more than 90% of the full power consumption. We find that on average the execute stage contributes 1% of the full power consumption, while the fetch and result stages contribute 5%. For the cases with similar performance, we see that a large but slow-clocked design achieves 1.1× better power efficiency than a small but fast-clocked design, similar to what is reported for FINN [17]. We also include the original BISMO power consumption data in Table 9 for comparison. Although the Ultra96 has higher idle power consumption compared to the PYNQ-Z1, we see that the improved BISMO has 1.5× better power efficiency compared to the

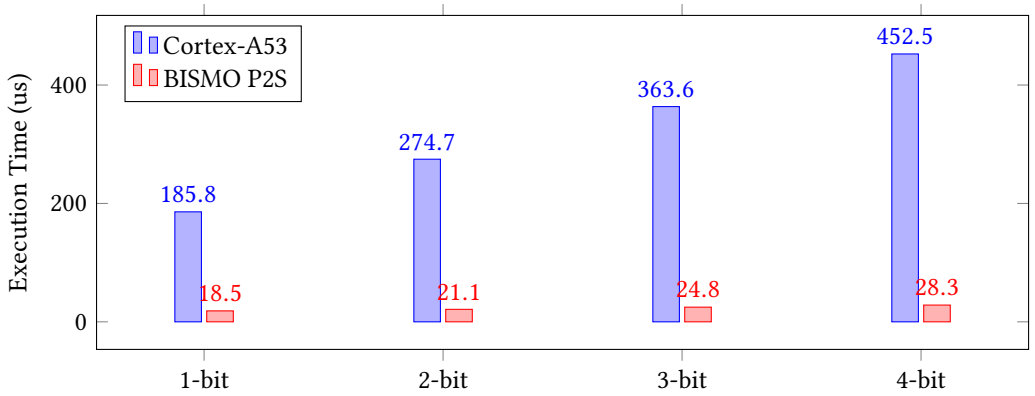

Fig. 20. Parallel-to-serial runtime for 20x1280 matrices of varying precision.

original BISMO for the top-performing designs. This can be attributed to a combination of process scaling (16 vs 28 nm) and the more LUT-efficient design in the improved BISMO.

4.2.5 Parallel-to-Serial Accelerator Performance. To quantify the performance gains from the P2S accelerator, we compare the execution time for data layout transformation between the accelerator (with the parameters in Section 4.1.3) and a CPU version on the Ultra96. For the CPU version, we use the open-source implementation from [18]. This is a single-thread implementation that uses 32-bit multiplication with a specifically crafted constant to pack bit positions from multiple 8-bit words, originally proposed by Mula [12]. We report the average of 30 runs to account for caching effects. As the Ultra96 ZU3EG possesses a 64-bit quad-core CPU, we optimistically divide the CPU execution time by eight to allow for future multithreading and wider datapath optimizations.

Fig. 20 plots the execution time for both methods for a 20x1280 matrix of varying precision stored using 8-bit elements. On average, the P2S accelerator is 13.8× faster than the CPU implementation. The CPU is limited by its ability to perform fine-grained (bit-level) data movement between registers, while the FPGA is well-suited to this task. Especially for larger matrices where data layout conversion can become costly, the P2S accelerator can contribute significantly to overall bit-serial matrix multiplication peformance.

5 RELATED WORK

Table 10 compares BISMO against several recently-proposed implementations for low-precision matrix multiplication, using peak binary performance and performance per watt as metrics. The top part of the table includes DRAM power, while the bottom part only considers on-chip compute and memory power. The improved BISMO presented in this work achieves a peak energy efficiency of 2.13 binary TOPS/W, which is an improvement of 1.5× compared to the original BISMO [19]. The peak performance of improved BISMO is also 2.3× that of the original, owing to a combination of the improved DPU design and newer FPGA. To our knowledge, BISMO is the first FPGA implementation for bit-serial matrix multiplication, but comparable related work on binarized neural networks by Umuroglu et al. [17] and low-precision matrix multiplication by Moss et al. [3] report respectively 5.2× and 2.5× lower power efficiency than ours. Although the GPU binary matrix multiplication kernels proposed by Pedersoli et al. [4] achieve an impressive 90 TOPS for large binary matrices, their work does not report power measurements. Assuming a power consumption of 120 W for the GTX 960, BISMO achieves 2.8× better power efficiency in comparison. On CPUs, the single-threaded implementation by Umuroglu and Jahre [18] performed far worse than BISMO, and is

Table 10. Comparing BISMO to recent work.

Work	Platform	Type	Precision	Binary GOPS	GOPS/W	
Improved BISMO	ZU3EG on Ultra96	FPGA	bit-serial	15,360	2,133.33	incl. DRAM
Original BISMO [19]	Z7020 on PYNQ-Z1	FPGA	bit-serial	6,554	1,413.40	
FINN [17]	Z7045 on ZC706	FPGA	binary	11,613	407.50	
Moss et al. [3]	GX1150 on HARPv2	FPGA	reconfigurable	41	849.38	
Umuroglu et al. [18]†	Cortex-A57 on Jetson TX1	CPU	bit-serial	92	18.80	
Pedersoli et al. [4]†	GTX 960	GPU	limited bit-serial	90,909	757.60	
Judd et al. [13]†	ASIC	ASIC	limited bit-serial	128,450	4,253.30	
Improved BISMO	ZU3EG on Ultra96	FPGA	bit-serial	15,360	2,245.61	excl. DRAM
Original BISMO [19]	Z7020 on PYNQ-Z1	FPGA	bit-serial	6,554	1,889.70	
FINN [17]	Z7045 on ZC706	FPGA	binary	11613	992.50	
Umuroglu et al. [18]†	Cortex-A57 on Jetson TX1	CPU	bit-serial	92	43.80	
Umuroglu et al. [18]†	i7-4790	CPU	bit-serial	355	12.20	

† *indicates our experiments from released code or projections based on paper.*

still outperformed by more than an order of magnitude even when assuming 4× performance improvement with multi-core parallelization. Finally, Stripes by Judd et al. [13] outperforms ours by 2.0× due to the performance and energy efficiency of an ASIC implementation.

6 CONCLUSION

We have presented an improved version of BISMO, a bit-serial matrix multiplication overlay that can scale its precision to match an application's computational requirements and its hardware to match available system resources. A new architecture and an FPGA specific compressor implementation for the dot product unit (DPU) are shown to reduce the LUT cost per binary operation by 1.8× compared to the original BISMO. The new design achieves a peak performance of 15.4 binary TOPS with an energy efficiency of 2.1 TOPS/W on an Ultra96 board, an improvement of 2.3× and 1.5×, respectively. Synthesis results targeting a Xilinx Virtex UltraScale+ VU9P show that the core dot product array (DPA) can achieve a peak performance of 783 binary TOPS at 500 MHz and a LUT utilization of 80%.

ACKNOWLEDGMENTS

This work was funded by Vetenskapsrådet project 2015-05159. The computations were performed on resources provided by NTNU through the EPIC cluster.

REFERENCES

[1] Krste Asanović, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. 2006. *The Landscape of Parallel Computing Research: A View from Berkeley.* Technical Report UCB/EECS-2006-183. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html

[2] AVNET. 2018. ULTRA96. http://www.ultra96.org/sites/default/files/product_briefs/5354-pb-ultra96-v3b.pdf. Accessed on: 2018-12-12.

[3] D. J. Moss et al. 2018. A Customizable Matrix Multiplication Framework for the Intel HARPv2 Xeon+ FPGA Platform: A Deep Learning Case Study. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 107–116.

[4] F. Pedersoli et al. 2018. Espresso: Efficient Forward Propagation for BCNNs. In *Proceedings of the International Conference on Learning Representations*.

[5] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Quantized neural networks: Training neural networks with low precision weights and activations. *arXiv preprint arXiv:1609.07061* (2016).

[6] J. Bachrach et al. 2012. Chisel: Constructing Hardware in a Scala Embedded Language. In *Proceedings of the ACM/IEEE Design Automation Conference*. ACM, 1216–1225.
[7] M. Kumm and J. Kappauf. 2018. Advanced Compressor Tree Synthesis for FPGAs. *IEEE Trans. Comput.* PP, 99 (2018). https://doi.org/10.1109/TC.2018.2795611
[8] Martin Kumm and Peter Zipf. 2014. Pipelined Compressor Tree Optimization Using Integer Linear Programming. In *Proceedings of the Conference on Field Programmable Logic and Applications*. IEEE, IEEE, 1–8. https://doi.org/10.1109/FPL.2014.6927468
[9] Hsiang-Tsung Kung. 1982. Why systolic architectures? *IEEE computer* 15, 1 (1982), 37–46.
[10] Kiran Kumar Matam and Viktor K Prasanna. 2013. Energy-efficient large-scale matrix multiplication on FPGAs. In *2013 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*. IEEE, 1–8.
[11] Sparsh Mittal. 2016. A survey of techniques for approximate computing. *Comput. Surveys* 48, 4 (2016), 62.
[12] Wojchech Mula. 2018. Scalar version of SSE move mask instruction. http://0x80.pl/articles/scalar-sse-movmask.html.
[13] P. Judd et al. 2016. Stripes: Bit-serial deep neural network computing. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*. IEEE, 1–12.
[14] Hadi Parandeh-Afshar, Arkosnato Neogy, Philip Brisk, and Paolo Ienne. 2011. Compressor Tree Synthesis on Commercial High-Performance FPGAs. *ACM Transactions on Reconfigurable Technology and Systems* 4, 4 (Dec. 2011), 39:1–39:19. https://doi.org/10.1145/2068716.2068725
[15] Eunhyeok Park, Junwhan Ahn, and Sungjoo Yoo. 2017. Weighted-Entropy-Based Quantization for Deep Neural Networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 5456–5464.
[16] Thomas B. Preußer. 2017. Generic and Universal Parallel Matrix Summation with a Flexible Compression Goal for Xilinx FPGAs. In *Proceedings of the Conference on Field Programmable Logic and Applications*. IEEE, 1–7. https://doi.org/10.23919/FPL.2017.8056834
[17] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. 2017. FINN: A Framework for Fast, Scalable Binarized Neural Network Inference. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, New York, NY, USA, 65–74. https://doi.org/10.1145/3020078.3021744
[18] Yaman Umuroglu and Magnus Jahre. 2017. Streamlined Deployment for Quantized Neural Networks. *arXiv preprint arXiv:1709.04060* (2017).
[19] Y. Umuroglu, L. Rasnayake, and M. Själander. 2018. BISMO: A Scalable Bit-Serial Matrix Multiplication Overlay for Reconfigurable Computing. In *Proceedings of the Conference on Field Programmable Logic and Applications*.
[20] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. 2018. HAQ: Hardware-Aware Automated Quantization. *arXiv preprint arXiv:1811.08886* (2018).
[21] Xilinx 2017. *Vivado Design Suite User Guide - Release Notes, Installation, and Licensing* (UG973 (v2017.4) ed.). Xilinx.
[22] Xilinx 2018. *Python productivity for Zynq (Pynq) Documentation* (release 2.2 ed.). Xilinx.
[23] Xilinx. 2018. UltraScale Architecture and Product Data Sheet: Overview. https://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf. Accessed on: 2018-12-12.
[24] Xilinx. 2018. Zynq UltraScale+ MPSoC Data Sheet: Overview. https://www.xilinx.com/support/documentation/data_sheets/ds891-zynq-ultrascale-plus-overview.pdf. Accessed on: 2018-12-12.
[25] Mehdi R Zargham. 1996. *Computer architecture: single and parallel systems*. Prentice-Hall, Inc.