BISMO: A Scalable Bit-Serial Matrix Multiplication Overlay for Reconfigurable Computing

Yaman Umuroglu[†]
yamanu@xilinx.com
Xilinx Research Labs
Dublin, Ireland

Lahiru Rasnayake and Magnus Själander
[firstname.lastname]@ntnu.no
Department of Computer Science
Norwegian University of Science and Technology (NTNU)
Trondheim, Norway

***Abstract*—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. We present BISMO, a vectorized bit-serial matrix multiplication overlay for reconfigurable computing. BISMO utilizes the excellent binary-operation performance of FPGAs to offer a matrix multiplication performance that scales with required precision and parallelism. We characterize the resource usage and performance of BISMO across a range of parameters to build a hardware cost model, and demonstrate a peak performance of 6.5 TOPS on the Xilinx PYNQ-Z1 board.**

I. 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, and multimedia domain, can use reduced precision and still produce adequate results. This property has been leveraged in approximate computing [1] and quantized neural networks (QNNs) [2], [3], to improve performance and energy efficiency and to reduce area by tailoring computations to the required precision. This precision may vary between different phases of the application. As an example, Park et al. [3] achieve the best performance-accuracy tradeoff for QNNs by using fewer bits for the intermediate layers.

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 [4]. The amount of computations 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 [5] where the integer matrix multiplication is expressed as a weighted sum of binary matrix multiplications (Section II). The bit-serial alternative provides the possibility to use an efficient binary matrix multiplication accelerator to compute matrix multiplications of any precision.

We present a scalable bit-serial matrix multiplication overlay called BISMO that can be efficiently instantiated on an FPGA. The core of BISMO is a software-programmable weighted binary matrix multiplication engine and associated hardware for fetching data and storing back the result (Section III-A). The hardware architecture is design-time configurable and we provide a cost model for estimating the resource usage for a given set of parameters (Section III-B). BISMO's software programmability makes it possible to operate on any matrix size and any fixed-point or integer precision (Section III-C).

We evaluate BISMO on the Xilinx PYNQ-Z1 board and show *i)* that the number of look-up-tables (LUTs) scales linearly with the number of parallel binary dot-product operations and *ii)* an average 94% accuracy for the proposed cost model (Section IV-A). We also show that the performance scales linearly with allocated resources and that the runtime scales better than expected when increasing the dot-product precision. BISMO achieves a peak performance of 6.5 binary TOPS and an energy efficiency of up to 1.4 binary TOPS/W (Section IV-B), which is best-in-class with only a dedicated ASIC accelerator showing better performance (Section V). BISMO is open-sourced at https://git.io/fWb0m.

[†] Corresponding author. Work performed while author was at NTNU.

Algorithm 1 Bit-serial matrix multiplication on signed 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 $j \leftarrow 0 \ldots p-1$ do
 5:       $\mathrm{sgnL} \leftarrow (i == l-1\ ?\ -1 : 1)$
 6:       $\mathrm{sgnR} \leftarrow (j == p-1\ ?\ -1 : 1)$
 7:       $\mathrm{weight} = \mathrm{sgnL} \cdot \mathrm{sgnR} \cdot 2^{i+j}$
 8:       # Binary matrix multiplication between $L^{[i]}$ and $R^{[j]}$
 9:       for $r \leftarrow 1 \ldots m$ do
10:          for $c \leftarrow 1 \ldots n$ do
11:             for $d \leftarrow 1 \ldots k$ do
12:                $P_{rc} = P_{rc} + \mathrm{weight} \cdot (L^{[i]}_{rd} \cdot R^{[j]}_{dc})$
```

II. Bit-Serial Matrix Multiplication

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.

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 [5]. 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) consists of 2-bit unsigned integer numbers. By expressing L and R as weighted sums of binary matrices, the matrix product ($L \cdot R$) 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.

$$L = \begin{bmatrix}2 & 0\\1 & 3\end{bmatrix} = 2^1 L^{[1]} + 2^0 L^{[0]} = 2^1 \begin{bmatrix}1 & 0\\0 & 1\end{bmatrix} + 2^0 \begin{bmatrix}0 & 0\\1 & 1\end{bmatrix}$$

$$R = \begin{bmatrix}0 & 1\\1 & 2\end{bmatrix} = 2^1 R^{[1]} + 2^0 R^{[0]} = 2^1 \begin{bmatrix}0 & 0\\0 & 1\end{bmatrix} + 2^0 \begin{bmatrix}0 & 1\\1 & 0\end{bmatrix}$$

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

Fig. 1. Example of a bit-serial matrix multiplication on unsigned integers with the two first for-loops unrolled (Algorithm 1: for-loop on line 4 and 5 unrolled and weight on line 8 always positive).

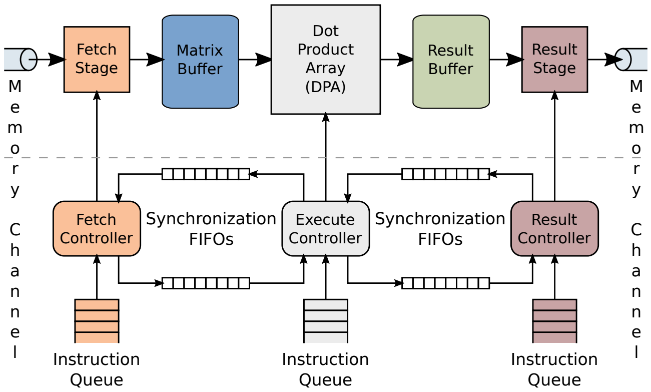

Fig. 2. Overview of BISMO's' hardware architecture.

III. 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 as follows:

Precision-scalable. By expressing an integer or fixed-point matrix multiplication as a weighted sum of binary matrix multiplications (Section II), 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-scalable. 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 III-A. We also provide a cost model to estimate the resource usage for a given set of parameters as described in Section III-B.

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

A. Hardware Architecture

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.

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 I.

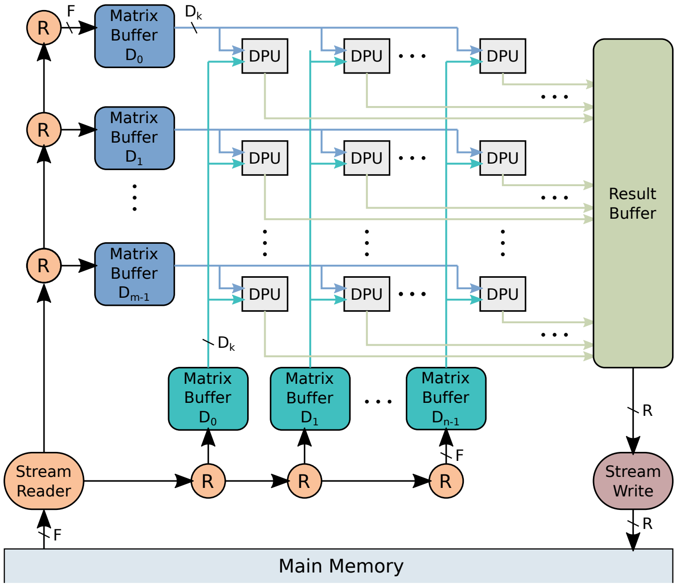

Fig. 3. Key components of the BISMO datapath.

TABLE I
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

1) 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 network as there is no backpressure. The fetch stage can be scaled at design time to match the memory read bandwidth (F) of a particular platform.

2) 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 is fed with the same data broadcasted by the left-hand-side matrix buffer. Similarly, the DPUs on the same column is fed with the same data broadcasted by the right-hand-side matrix buffer (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.

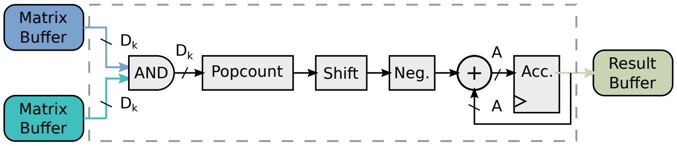

Fig. 4. The BISMO dot product unit (DPU).

The DPU pipeline 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 13 in Algorithm 1. The single-bit multiplications are performed by a multi-bit logic AND operation 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 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 [5], [6] to avoid overflow.

3) 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 computations and data transfers. The result stage can be scaled at design time to match the memory write bandwidth (R) of a particular platform.

B. Cost Model

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

1) LUT cost: We propose the following equations to model the LUT usage of a BISMO instance:

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

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

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

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

2) BRAM cost: Assuming dual-port $36 \cdot 1024$-bit Xilinx BRAMs, we model the BRAM usage as follows:

$$\mathrm{BRAM_{total}} = \mathrm{BRAM_{base}} + \mathrm{BRAM_{array}} \tag{2a}$$

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

In Equation 2a, $\mathrm{BRAM_{base}}$ refers to the BRAMs used for DPA-size independent infrastructure, such as DMA buffers and instruction queues. $\mathrm{BRAM_{array}}$ 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.

C. *Programming* BISMO

BISMO provides programmability through the use of instructions that control each of the pipeline stages. 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.

1) Instructions: There are three types of instructions per pipeline stage in BISMO, namely `Wait`, `Signal` and `Run`. Table II provides a summary of these instructions with the usage described as follows:

a) 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.

TABLE II
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
	Weight
	Accumulator reset
`RunResult`	Result base address in main memory
	Address offset

b) 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 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, the weight controlling the shift amount and if the dot product should be negated (line 8 in Algorithm 1), and the possibility to reset the accumulators before performing any computations.

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.

2) Instruction Scheduling: The BISMO instructions enable the possibility to tailor the computation to the input matrix characteristics, e.g., by taking their dimensions into account.

Fig. 5 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

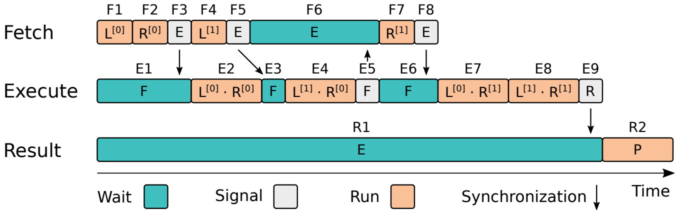

Fig. 5. Timeline of the example schedule shown in Table III.

TABLE III
INITIALIZED INSTRUCTION QUEUES FOR THE EXAMPLE SHOWN IN FIG. 1

Fetch	Execute	Result
F1 `Run` L$^{[0]}$	E1 `Wait` Fetch	R1 `Wait` Execute
F2 `Run` R$^{[0]}$	E2 `Run` P =+ L$^{[0]}$ · R$^{[0]}$	R2 `Run` P
F3 `Signal` Execute	E3 `Wait` Fetch	
F4 `Run` L$^{[1]}$	E4 `Run` P =+ L$^{[2]}$· R$^{[0]}$	
F5 `Signal` Execute	E5 `Signal` Fetch	
F6 `Wait` Execute	E6 `Wait` Fetch	
F7 `Run` R$^{[1]}$	E7 `Run` P =+ L$^{[0]}$· R$^{[1]}$	
F8 `Signal` Execute	E8 `Run` P =+ L$^{[1]}$· R$^{[1]}$	
	E9 `Signal` Result	

three of the four binary matrices ($L^{[1]}$, $L^{[0]}$, $R^{[1]}$, and $R^{[0]}$) fit at the same time in the matrix buffers to make the schedule slightly more interesting. The corresponding instructions for each pipeline stage can be seen in Table III, with P denoting the matrix that accumulates the result of these operations.

The fetch stage begins by fetching $L^{[0]}$ and $R^{[0]}$ (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 $L^{[0]}$ and $R^{[0]}$, the fetch stage continues fetching $L^{[1]}$, 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 $L^{[1]}$ resides in the matrix buffers (E3). The execute stage continues by executing $L^{[1]} \cdot R^{[0]}$ (E4) while the fetch stage has to wait since all the buffer space is occupied (F6). When the execute stage finishes the matrix multiplication, it signals the fetch stage (E5). Since $R^{[0]}$ is no longer needed, the fetch stage fetches $R^{[1]}$ (F7) enabling the execute stage to finish the remaining matrix multiplications (E7 and E8). 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. 5 causes the fetch stage and execute stage to stall (F6 and E6) since there is not enough space to fetch $R^{[1]}$ before $L^{[1]} \cdot R^{[0]}$ 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.

IV. EVALUATION

We implement the BISMO parametrizable hardware generator in Chisel [7] and use Xilinx Vivado 2017.4 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 PYNQ-Z1 board, which has a Xilinx Z7020 FPGA with 53,200 LUTs, 140 BRAMs, and 3.2 GB/s of DRAM bandwidth.

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.

A. 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. All data in this section is obtained by using out-of-context synthesis for the Z7020, with a target clock period of 1 ns to prioritize timing optimizations.

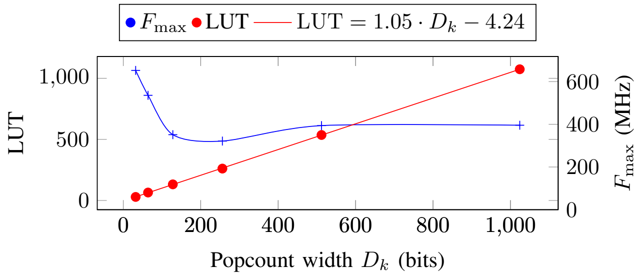

Fig. 6. Popcount unit LUT usage characterization.

1) Popcount: Fig. 6 plots the LUT usage and maximum frequency ($F_{\max}$) versus input bitwidth for the popcount unit. We observe that the least squares regression line is a good fit for the LUT usage, indicating a resource cost of approximately one LUT per input bit. This may be further improved by incorporating better compressor synthesis techniques, as proposed by Preußer [8]. The reported maximum clock frequency $F_{\max}$ is between 320 and 650 MHz for all tested popcount widths.

2) Dot Product Unit: In addition to the popcount unit cost, the DPU cost includes the AND operation, barrel shifter, negator, and accumulator. We expect that the resource cost of the three latter gets amortized for larger values of D_k as their size grows proportional to $\log_2 D_k$. Fig. 7 shows the LUT usage as well as the LUT cost per binary operation. We observe that the cost per operation starts at 2.8 LUTs for $D_k = 32$ and decreases to 1.07 LUTs for $D_k = 1024$. The parameters α_{DPU} and β_{DPU} of the BISMO cost model (Section III-B1) are 2.04 and 109.41, respectively. For the tested bitwidths, the reported maximum frequency ($F_{\max}$) is between 300 and 350 MHz.

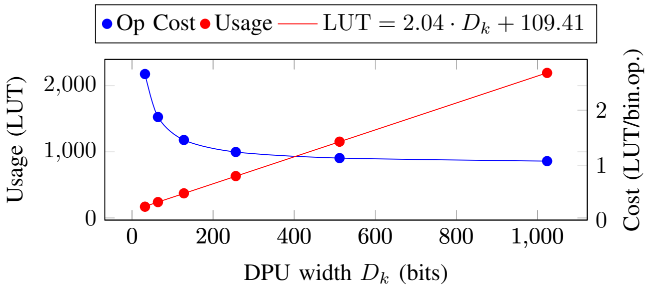

Fig. 7. DPU LUT usage and efficiency characterization.

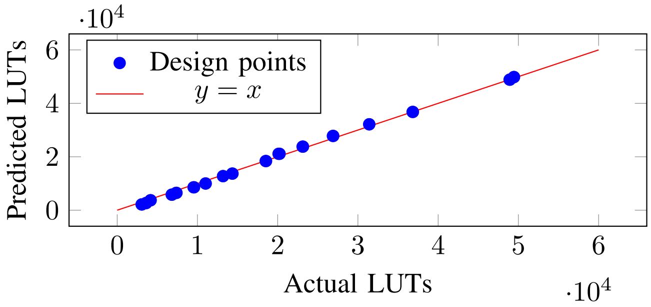

Fig. 8. Predicted vs actual LUT usage.

3) 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 $\text{LUT}_{\text{res}} = 87.3 + 32.8 = 120.1$. The DMA engine currently limits $F_{\max}$ to 200 MHz, and may be pipelined to further increase $F_{\max}$ for the entire accelerator.

4) Cost model validation: We generated 34 different BISMO designs ranging from (D_m=2, D_k=64, D_n=2) to (D_m=8, D_k=256, D_n=8) in size to validate the cost models described in Section III-B. The BRAM predictions were 100% accurate for this particular range of designs. Fig. 8 shows the LUT usage from synthesis results versus the prediction from the cost model. The model's prediction is 93.8% accurate on average. Fig. 9 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 overestimated by the model, likely due to the effect of additional synthesis optimizations applied by Vivado for small designs.

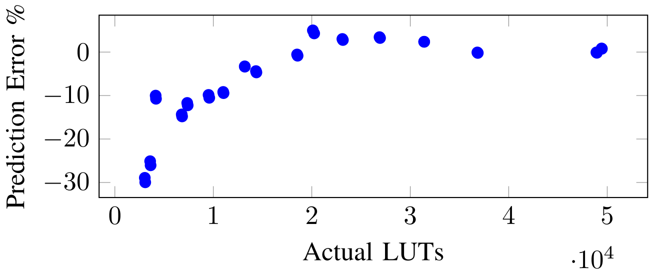

Fig. 9. LUT cost model prediction error with design size.

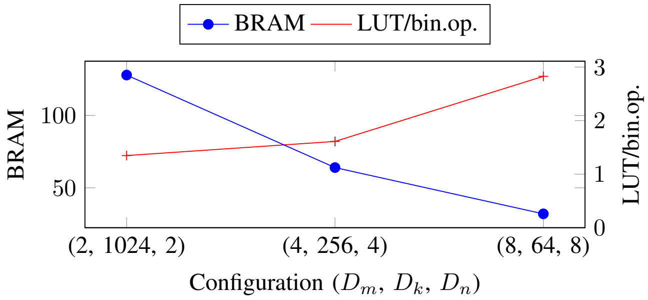

Fig. 10. LUT vs BRAM tradeoffs for 1.6 binary TOPS at 200 MHz.

5) LUT-BRAM Tradeoffs: Fig. 10 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.

6) Hardware Cost of Flexible Precision: When 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

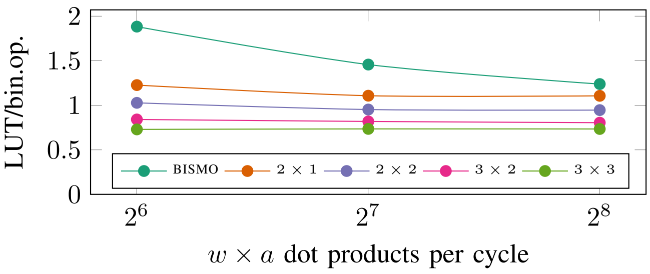

Fig. 11. Comparing the LUT/bin.op. cost of bit-serial and bit-parallel DPUs.

TABLE IV

BISMO INSTANCES FOR RUNTIME MEASUREMENTS.

#	D_m	D_k	D_n	LUT	BRAM	GOPS
1	8	64	8	19545 (37%)	121 (86%)	1638.4
2	8	128	8	27740 (52%)	129 (92%)	3276.8
3	8	256	8	45573 (86%)	129 (92%)	6553.6
4	4	256	4	13352 (25%)	129 (92%)	1638.4
5	8	256	4	24202 (45%)	129 (92%)	3276.8
6	4	512	4	21755 (41%)	129 (92%)	3276.8

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

no shifter and negator. This bit-parallel DPU performs the equivalent of $2 \cdot w \cdot a \cdot D_k$ binary operations per cycle. Fig. 11 compares the LUT cost for binary operation equivalents between the BISMO DPU and several bit-parallel variants. We first observe that the LUTs per operation decreases with higher bit-parallel precision, from 1.1 for 2×1 down to 0.73 for 3×3. Beyond 3×3 bits we did not observe further lowering of the LUT cost. 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 0.5 LUT per operation. We note that this is not a fully fair comparison since 1) the BISMO hardware supports significantly larger precisions, and 2) our implementation is not fully optimized down to the LUT level.

B. Runtime Performance

In this section, we assess the runtime performance and energy efficiency achievable by BISMO instances running on the PYNQ-Z1. We assume that the input matrices are stored in DRAM using a bit-packed data layout [5], 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 IV to evaluate how the overlay size interacts with workload size.

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 the matrices have already been fetched into on-chip memory and disregard the cost of result writing. Fig. 12 plots the achieved performance for different number of columns as a percentage of observed peak performance. 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, instance #3 reaches 64% efficiency, while instance #1 achieves 89%. The inefficiency for narrow matrices is due to the lack of work to fill the DPA pipeline, e.g., the DPA pipeline may be 10-deep but each dot product is finished in 6 cycles. This can be remedied by issuing more work to the DPA without waiting for the previous execution to finish, or by decreasing the DPA pipeline depth. Wide matrices achieve close to 100% of the peak performance for all instances.

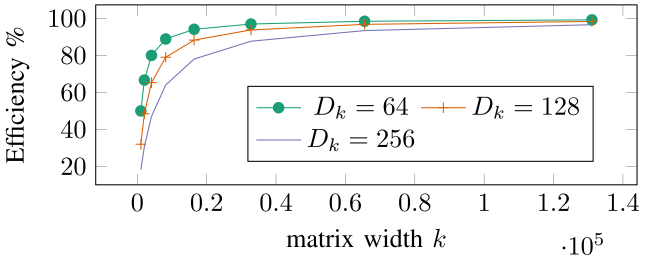

Fig. 12. Execute stage efficiency depending on D_k and matrix width k.

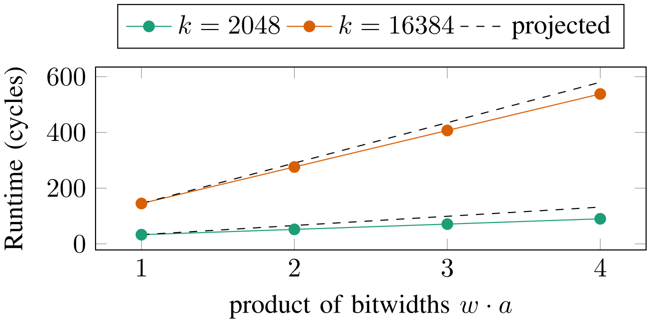

Fig. 13. Runtime with increasing precision on instance #2.

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. 13 plots the performance for $8 \times 2048 \times 8$ and $8 \times 16384 \times 8$ with increasing w, a on instance #2. 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. 12).

3) Stage Overlap: We now quantify the performance gain by overlapping the fetch, execute and result stages for larger matrix multiplications. We create an instruction sequence to run a $256 \times 4096 \times 256$ binary matrix multiplication on instance #1, similar to the example in Section III-C2. The input matrices here are twice the size of the on-chip memory. By overlapping the operation of different stages, the multiplication finishes in 121133 cycles, achieving a speedup of 2.2× compared to the 266510 cycles when the stages are executing without overlap.

TABLE V

POWER CONSUMPTION DATA FROM BISMO INSTANCES ON PYNQ-Z1.

Configuration	Power (W)				Binary	Binary
(Instance, F_{clk})	Idle	Exec	F & R	Full	GOPS	GOPS/W
(#1, 200 MHz)	2.53	+0.33	+1.09	4.07	1638	402.16
(#2, 100 MHz)	2.10	+0.19	+0.87	3.11	1638	527.51
(#3, 50 MHz)	1.76	+0.30	+0.63	2.53	1638	646.39
(#4, 200 MHz)	2.53	+0.34	+1.09	3.86	1638	424.98
(#5, 100 MHz)	2.05	+0.24	+0.92	3.06	1638	536.02
(#3, 200 MHz)	2.87	+0.71	+1.19	4.64	6554	1413.39

TABLE VI
COMPARING BISMO TO RECENT WORK.

Work	Platform	Type	Precision	Binary GOPS	GOPS/W	
BISMO	Z7020 on PYNQ-Z1	FPGA	bit-serial	6554	1413.40	incl. DRAM
FINN [6]	Z7045 on ZC706	FPGA	binary	11613	407.50	
Moss et al. [9]	GX1150 on HARPv2	FPGA	reconfigurable	41	849.38	
Umuroglu et al. [5]†	Cortex-A57 on Jetson TX1	CPU	bit-serial	92	18.80	
Pedersoli et al. [10]†	GTX 960	GPU	limited bit-serial	90909	757.60	
Judd et al. [11]†	ASIC	ASIC	limited bit-serial	128450	4253.30	
BISMO	Z7020 on PYNQ-Z1	FPGA	bit-serial	6554	1889.70	excl. DRAM
FINN [6]	Z7045 on ZC706	FPGA	binary	11613	992.50	
Umuroglu et al. [5]†	Cortex-A57 on Jetson TX1	CPU	bit-serial	92	43.80	
Umuroglu et al. [5]†	i7-4790	CPU	bit-serial	355	12.20	

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

4) Power Consumption: BISMO's power efficiency is measured using a PYNQ-Z1 board powered over a USB port with a power meter attached while running one or more stages in a loop. Table V lists five instances where the frequency (F_{clk}) is adjusted to achieve the same peak binary performance (GOPS) followed by the top-performing BISMO instance. 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. We find that on average the execute stage contributes 9.7% of the full power consumption, while the fetch and result stages contribute 27.2% and the idle power constitutes 65.6%. For the cases with constant performance, we see that a large but slow-clocked design achieves 1.5× better power efficiency than a small but fast-clocked design, similar to what is reported in FINN [6]. The majority of this increase in power efficiency can be attributed to lower idle power due to a slower clock.

V. Related Work

Table VI 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. 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. [6] and low-precision matrix multiplication by Moss et al. [9] report 3.5× and 1.6×, respectively, lower power efficiency than ours. Although the GPU binary matrix multiplication kernels proposed by Pedersoli et al. [10] achieve an impressive 90 TOPS for large matrices, their work does not report power measurements. Assuming a power consumption of 120 W for the GTX 960, BISMO achieves 1.9× better power efficiency in comparison. On CPUs, the single-threaded implementation by Umuroglu and Jahre [5] performed far worse than BISMO, and is still outperformed by more than an order of magnitude even assuming 4× performance improvement with multi-core parallelization. Finally, Stripes by Judd et al. [11] outperforms ours by 3× due to the performance and efficiency of an ASIC implementation.

VI. Conclusion

We have presented 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. The proposed cost model accurately predicts FPGA resource utilization and enables quick performance estimations. BISMO is software programmable, providing the possibility to adapt its execution to the dimension and precision of any input matrix. Our evaluation indicates that BISMO achieves a peak performance of 6.5 TOPS with an energy efficiency of up to 1.4 TOPS/W on a PYNQ-Z1 board.

References

[1] S. Mittal, "A survey of techniques for approximate computing," *ACM Computing Surveys*, vol. 48, no. 4, 2016.

[2] I. Hubara et al., "Quantized neural networks: Training neural networks with low precision weights and activations," *arXiv preprint arXiv:1609.07061*, 2016.

[3] E. Park et al., "Weighted-entropy-based quantization for deep neural networks," in *CVPR*, pp. 5456–5464, 2017.

[4] K. Asanović et al., "The landscape of parallel computing research: A view from Berkeley," Tech. Rep. UCB/EECS-2006-183, Dec 2006.

[5] Y. Umuroglu and M. Jahre, "Streamlined deployment for quantized neural networks," *arXiv preprint arXiv:1709.04060*, 2017.

[6] Y. Umuroglu et al., "FINN: A framework for fast, scalable binarized neural network inference," in *FPGA*, 2017.

[7] J. Bachrach et al., "Chisel: Constructing hardware in a Scala embedded language," in *DAC*, 2012.

[8] T. B. Preußer, "Generic and universal parallel matrix summation with a flexible compression goal for Xilinx FPGAs," in *FPL*, 2017.

[9] D. J. Moss et al., "A customizable matrix multiplication framework for the Intel HARPv2 Xeon+ FPGA platform: A deep learning case study," in *FPGA*, 2018.

[10] F. Pedersoli et al., "Espresso: Efficient forward propagation for BCNNs," in *ICLR*, 2018.

[11] P. Judd et al., "Stripes: Bit-serial deep neural network computing," in *MICRO*, 2016.