BISDU: A Bit-Serial Dot-Product Unit for Microcontrollers

DAVID METZ, VINEET KUMAR, and MAGNUS SJÄLANDER, Norwegian University of Science and Technology (NTNU), Norway

Low-precision quantized neural networks (QNNs) reduce the required memory space, bandwidth, and computational power, and hence are suitable for deployment in applications such as IoT edge devices. Mixed-precision QNNs, where weights commonly have lower precision than activations or different precision is used for different layers, can limit the accuracy loss caused by low-bit quantization, while still benefiting from reduced memory footprint and faster execution. Previous multiple-precision functional units supporting 8-bit, 4-bit, and 2-bit SIMD instructions have limitations, such as large area overhead, under-utilization of multipliers, and wasted memory space for low and mixed bit-width operations.

This article introduces BISDU, a bit-serial dot-product unit to support and accelerate execution of mixed-precision low-bit QNNs on resource-constrained microcontrollers. BISDU is a multiplier-less dot-product unit, with frugal hardware requirements (a population count unit and 2:1 multiplexers). The proposed bit-serial dot-product unit leverages the conventional logical operations of a microcontroller to perform multiplications, which enables efficient software implementations of binary (XNOR), ternary (XOR), and mixed-precision [W×A] (AND) dot-product operations.

The experimental results show that BISDU achieves competitive performance compared to two state-of-the-art units, XpulpNN and Dustin, when executing low-bit-width CNNs. We demonstrate the advantage that bit-serial execution provides by enabling trading accuracy against weight footprint and execution time. BISDU increases the area of the ALU by 68% and the ALU power consumption by 42% compared to a baseline 32-bit RISC-V (RV32IC) microcontroller core. In comparison, XpulpNN and Dustin increase the area by 6.9× and 11.1× and the power consumption by 3.8× and 5.97×, respectively. The bit-serial state-of-the-art, based on a conventional popcount instruction, increases the area by 42% and power by 32%, with BISDU providing a 37% speedup over it.

CCS Concepts: • **Computer systems organization** → *Neural networks*; *System on a chip*; *Reduced instruction set computing*; • **Computing methodologies → Neural networks;**

Additional Key Words and Phrases: Bit-serial, Dot-product, Microcontroller, Quantized neural networks, Low power, ISA extension

ACM Reference format:
David Metz, Vineet Kumar, and Magnus Själander. 2023. BISDU: A Bit-Serial Dot-Product Unit for Microcontrollers. *ACM Trans. Embedd. Comput. Syst.* 22, 5, Article 79 (September 2023), 22 pages.
https://doi.org/10.1145/3608447

***Please note this paper should have been included in the Special Issue on tinyML.
This work is supported in part by an ERCIM Postdoctoral fellowship awarded to Vineet Kumar having CONTRACT Nr 2019-28.
Authors' address: D. Metz, V. Kumar, and M. Själander, Norwegian University of Science and Technology (NTNU), Department of Computer Science, Trondheim 7491, Norway; emails: {David.C.Metz, Vineet.Kumar, Magnus.Sjalander}@ntnu.no.

1539-9087/2023/09-ART79 $15.00
https://doi.org/10.1145/3608447

1 INTRODUCTION

Quantized neural networks (**QNNs**) are neural networks with reduced bit-precision of weights and activations. The quantization reduces the model size, which enables faster execution, less power consumption, and lower memory requirements. Quantization has been particularly crucial for executing machine learning on low-cost embedded processors instead of high-end CPUs or GPUs. One of the main applications of QNNs lies in deployment of machine learning models at the edge, as reduced precision enables their execution on resource-constraint IoT devices and mobile devices [42]. In contrast to using a cloud, running machine-learning inference at the edge gives the benefits of low latency, data privacy, reliability, reduced network traffic, and reduced energy usage by not transmitting raw data from the device to the cloud. However, edge devices often have tight resource constraints in terms of compute, memory, and importantly power budget.

Highly quantized neural networks, which use sub-byte or low-bit precision for their parameters, are emerging as they have potential to further reduce the required memory space, bandwidth, and computational power.

Binary [4, 25, 27, 30, 31] and ternary [8, 23, 24] neural networks are the extreme cases of quantization and have demonstrated their use in applications, such as digit recognition [4, 15, 38], object detection and image classification tasks [7, 18, 21, 23, 24, 35, 50], and ECG signal classification [43, 46]. The downside of QNNs is a potential accuracy loss as they approach lower bit-widths of parameters. This can be overcome by using mixed quantization and quantization-aware training [4, 5, 10, 26, 28, 44, 47–49]. In the mixed quantization approach, the different layers of a neural network can have different bit-precision based on how sensitive the layer is with respect to quantization [5, 26, 44]. Another mixed quantization approach, is to use different precision for the weights and activations [4, 10, 47–49]. Many QNN use-cases prioritize a reduction in the precision of weights over activations, because weights directly increase the permanent memory requirements, whereas the impact of activations on memory requirement depends on the architecture of the neural network and its dataflow [36]. It has also been observed that activations are more sensitive than weights, thus requiring higher bit-widths [48].

One drawback of mixed-layer precision is the need for hardware support of multiple precisions (e.g., 2×2 and 3×3), instead of only supporting one single precision for all layers (e.g., 8×8, which is the smallest precision supported by most conventional **instruction set architectures** (**ISAs**)). Mixed weight and activation precision requires mixed-operand arithmetic units for efficient implementation, i.e., hardware that can support input operands of different bit widths (e.g., 2×1 for TAB [49] and 3×1 for FINN-R [4]). However, applying mixed quantization (both mixed-layer and mixed-operand precision) can limit the accuracy loss caused by low-bit quantization, while still benefiting from reduced memory footprint and faster execution. Examples of such mixed-quantization use-cases are shown in Figure 1, where weight precision is limited to up to three bits.

FPGAs have proven to be the most efficient computation engines for executing mixed-precision low-bit QNNs, due to their flexibility that makes it possible to adapt accelerators to the specific needs of each QNN [32]. However, embedded machine learning mostly relies on the use of **microcontrollers** (**MCUs**) for inference tasks, due to their low cost, low power, and software programmability. The **instruction set architectures** (**ISA**) of commercial state-of-the-art embedded MCUs do not commonly have hardware support for mixed-precision low-bit arithmetic operations. However, researchers have proposed extensions to existing ISAs to support low-precision operations. Garofalo et al. introduced XpulpNN [13], a multi-precision dot-product unit integrated into a RISC-V core that enhances the efficiency of QNN inference on microcontrollers. XpulpNN has support for 16-bit, 8-bit, 4-bit, and 2-bit **single instruction multiple data** (**SIMD**) instructions. The extended core achieves 5.3× and 8.9× speed-up when considering 4-bit and 2-bit operands respectively, compared to a baseline core that only supports 8-bit SIMD instructions [13].

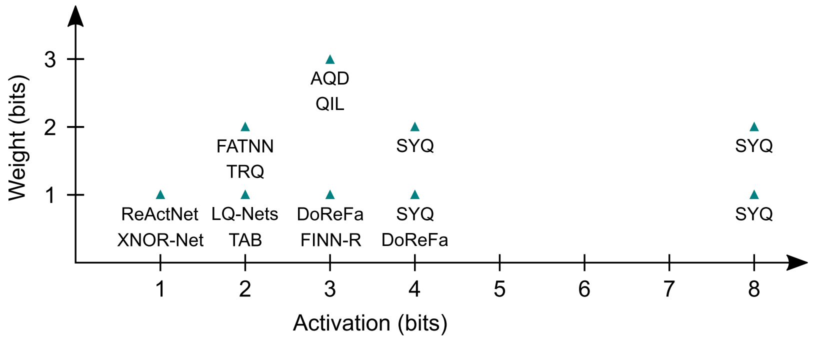


Fig. 1. Various bit-width requirements for QNNs (ReActNet [27], XNOR-Net [31], LQ-Nets [47], TAB [49], FATNN [8], TRQ [23], DoReFa-Net [48], FINN-R [4] AQD [7], QIL [17], and SYQ [10]) .

This kind of fixed-datapath precision-scalable units have limitations, such as under-utilization of multipliers for mixed-operand precision and large area overhead to support different bit-precision. For instance, multiplications of two mixed-operands of size 3-bit and 1-bit will have to use a 4×4-bit multiplier, which results in under-utilization of the hardware. The use of non-native operand-precision also either increases the memory footprint or requires special instructions for unpacking lower bit-width values from memory to the multiplier width. The state-of-the-art multi-precision dot-product units XpulpNN [13] and Dustin [11] are further discussed in Section 6.

Bit-serial operations have the advantage that in principle any precision can be supported with the trade-off that computations requiring higher precision incur a lower performance, since more bit-serial operations are required. Umuroglu and Jahre presented a bit-serial method for executing low-bit QNNs on an ARM Cortex-A57 [39]. They explain that integer matrix multiplication can be carried out by a weighted sum of binary matrix multiplications using bit-serial operations (Section 2.1). However, their work presents a pure software solution for implementing bit-serial operations based on existing ARM instructions. Adding hardware support (custom instruction) can further increase the speed of these operations and compete favorably with bit-parallel methods.

In this article, we propose BISDU a bit-serial dot-product unit and ISA extension, which enable efficient execution of bit-serial dot-products. The efficient execution of bit-serial dot-products enables efficient software implementations of low-precision and mixed-precision matrix multiplications (Section 3.4.2), convolutions (Section 3.4.3), and of complete **convolutional neural networks** (**CNNs**) (Section 5.3).

BISDU consists of only two new instruction types: (1) four variants of a binary dot-product instruction that takes a logical result and an accumulator as input and produces a partial dot-product as the result, and (2) a bit-packing instruction that takes bit-parallel values as input and produces bytes of a single precision as the result (Section 3.2). The hardware support consists of a multiplier-less dot-product unit with frugal hardware requirements and a multiplexer for bit-packing. The dot-product unit uses existing ALU resources (logical And, Xor, and Xnor, parallel adder, and negation), but requires a new population count (popcount) unit, if not already available in the ISA, and 2:1 multiplexers (Section 3.1).

Though bit-serial computation is not a new idea, an efficient implementation of a bit-serial dot-product unit for microcontroller devices is presented for the first time in this work. BISDU provides the following contributions:

— BISDU has frugal hardware requirements, consisting of a popcount unit (if not already available in the ISA) and multiplexers, and requires only five instruction variants adhering to standard RISC-V semantics.

ALGORITHM 1: Bit-serial dot-product for vectors of two's complement integers

```
Input: l-bit vector L, r-bit vector R, vector length N
Output: ACC = L·R
ACC = 0
for $i = 0 \dots l - 1$ do
    for $j = 0 \dots r - 1$ do
        sgnL (i == l - 1? - 1 : 1)
        sgnR (j == r - 1? - 1 : 1)
        sign = sgnL · sgnR
        weight = sign · $2^{i+j}$
        # Binary dot-product between $L^{[i]}$ and $R^{[j]}$
        # $L_n^{[i]}$ refers to $i^{th}$ bit position of element n
        for $n = 1 \dots N$ do
            ACC = ACC + weight × ($L_n^{[i]} \cdot R_n^{[j]}$)
        end
    end
end
```

— BISDU efficiently supports popular low- and mixed-precision quantization, such as binary, ternary, and mixed precision (Weight × Activation), which are used in numerous QNNs as depicted in Figure 1.
— BISDU is state-less, meaning that each operation is completed in a single cycle, which simplifies the handling of hazards, interrupts, and context switches.

We evaluate BISDU by integrating it in a baseline 32-bit TinyRocket [3], a RISC-V (RV32) core available in the open-source Chipyard framework [1], and compare it against state-of-the-art works by executing matrix multiplications, convolutions, and CNNs. The results (Section 5) show that BISDU achieves competitive performance compared to the state-of-the-art XpulpNN [13] and Dustin [11] at a significantly lower area overhead and performs favorably when compared with a conventional popcount instruction, the bit-serial state-of-the-art.

2 BACKGROUND

This section provides the necessary information for understanding how the bit-serial dot-product works and how CNNs can be executed using bit-serial matrix multiplications.

2.1 Bit-Serial Dot-Product

The dot-product of two vectors is the sum of products of corresponding elements of the vectors. The bit-serial method of computing the dot-product of two vectors is described in Algorithm 1, which is derived from the bit-serial matrix multiplication algorithm proposed by Umuroglu and Jahre [39]. Let L and R be two vectors consisting of N elements each and ACC be their dot-product. The algorithm computes the dot-product by summing up weighted dot-products of binary vectors, and is able to handle any bit-precision, but will require more binary dot-products for higher bit-precision. If two vectors contain l- and r-bit numbers, the algorithm has to perform $l \times r$ binary dot-products to compute the full dot-product. Since the number of binary dot-products scales linearly with the number of bits of either operand, the number of binary dot-products scales quadratically when the precision of both operands are increased. This makes the method attractive when one, or both, of the operands use few bits.

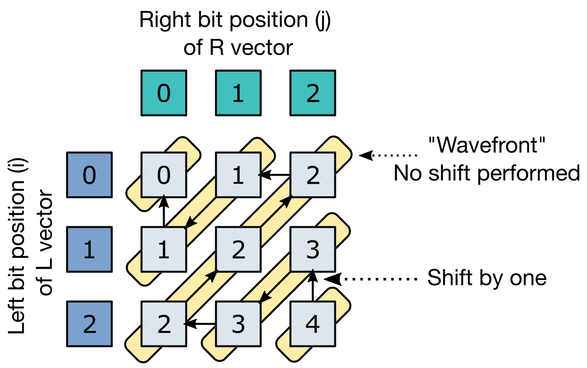


Fig. 2. Bit-position traversal order for a 3 × 3-bit accumulation.

Computing the dot-product of two binary vectors instead of two integer vectors has the benefit that a multiplier is not required; the multiplication is simply done by a logical AND operation (or XNOR in the case of binarized networks or XOR in the case of ternary networks) and summation is done by a popcount operation.

The binary dot-product, after multiplying it by a weight, is added to the accumulator, *ACC*, as shown in Algorithm 1. This weighted dot-product would normally be computed in hardware by left shifting the dot-product by a shift amount equal to the sum of bit positions, that is, *i+j*. However, the sum of weighted dot-products can be computed by instead starting with the dot-product of the most significant bit position and adding it to *ACC*. The accumulator is then left-shifted by one when crossing to a less significant bit-position [37], as illustrated in Figure 2.

Computations with equal weighting, i.e., the same required shift, are grouped together, and the group is called a wavefront. The binary dot-product is computed for each sum in a wavefront and added to *ACC*. No shifting is done while moving within the same wavefront, as shown in the figure. While moving from one wavefront to another, the previous *ACC* is shifted by one-bit position and added to the current contribution. The optional negation is still applied to the current contribution, when needed for bit-position combinations that yield a negative result, i.e., when the most significant bit of either but not both operands is part of the dot-product.

2.2 Convolutional Neural Networks (CNNs)

CNNs are a commonly used neural network type, which employ convolution as the main computation; specifically, two-dimensional convolution over a three-dimensional input tensor. The dimensions of the input are height, width, and channel. A convolution is performed by calculating the dot-product of the input and by applying a kernel as a stencil at different points in the height and width dimension. As this is done for multiple different kernels, the output matrix has a channel dimension equal to the number of kernels used.

When multiple different kernels are used, it can be beneficial to copy the elements involved in a convolution into a matrix and the kernels into a second matrix, an operation referred to as *im2col*. This way, an optimized matrix multiplication routine can be used, that benefits from being able to access weights linearly. This is beneficial because multiple kernels are applied to the same data, and this way the more complicated accesses for convolution have to only be performed once [19].

In order to reduce the height and width, pooling layers are used, which collect values in a step size of N across the width and height dimension and calculate a function such as max, min, or average over them. This returns one value for every step and is repeated across each channel, resulting in an N times reduction in size in the height and width dimension [33].

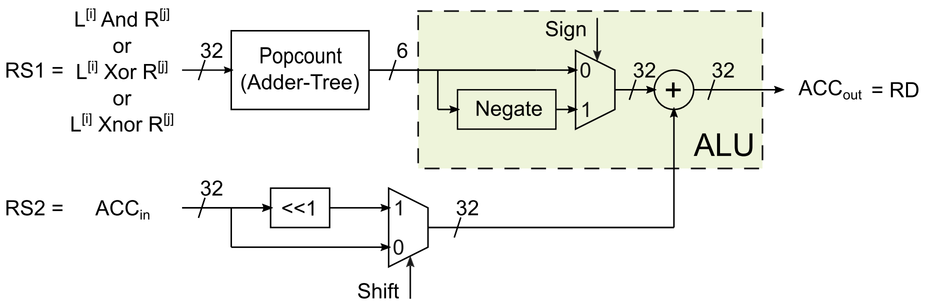


Fig. 3. The figure shows the proposed bit-serial dot-product unit. The highlighted area in green shows the logic already present in an ALU for performing additions and subtractions. The additional logic consist of a population count (Popcount) unit, which counts the number of "1"s in its input operand. The input operand is the result of a previously performed logical operation (And, Xor, or Xnor) that has been stored in the register referenced by RS1. The second operand, RS2, contains the accumulated value (or zero for the first dot-product computation) and is either used as is, or right shifted by one if starting the computation of a new wavefront, see Figure 2. The resulting value of the addition, represents the accumulated value of the dot-product operation and is written back to the register referenced by RD. Since both RS2 and RD represent the accumulated value, both are commonly referencing the same register.

In QNNs, the output of convolution is of higher bit-width than the input and hence, a quantization step is necessary to reduce the width for the next layer. This can be done through scaling or thresholding, which stores thresholds and counts how many of them are smaller than a value. The advantage of this approach is that thresholding can at the same time serve as an activation function, introducing a nonlinearity into the network. Thresholding with an n-bit output requires $2^n - 1$ thresholds, and there can be separate thresholds for each channel [40].

The last layers of CNNs are commonly fully-connected layers, which are implemented using matrix-vector multiplication [19].

3 THE PROPOSED BISDU

BISDU provides support for bit-serial dot-product and bit-packing instructions that operate on the integer registers of a microcontroller. Each instruction takes two input registers (RS1 and RS2) and produces one output register (RD). The new instructions are stateless, meaning there is no additional architectural state that otherwise could complicate context switching, interrupt routines, hazards, and so on. The following subsections present the proposed bit-serial dot-product unit, bit-packing support, instruction formats, and software support.

3.1 Bit-Serial Dot-Product Unit

The dot-product unit (see Figure 3) computes the dot-product of two binary vectors of size 32. The dot-product of two arbitrary long vectors is computed by executing the unit multiple times.

The input is the logical And of binary vectors, $L^{[i]}$ and $R^{[j]}$, the logical Xor, for ternary networks or the logical Xnor for the special case of binarized neural networks where Xnor is used, since −1 is encoded as 0 and 1 as 1. The binary vectors are obtained by grouping the ith and jth bits of each element in vector L and R, respectively (see Section 3.2). The previous output (ACC_{out}) is applied as input (ACC_{in}) and then either left-shifted by 1-bit position or not shifted, depending on whether the wavefront changes (see Figure 2).

BISDU is intended to be tightly integrated with the existing **arithmetic logic unit** (**ALU**) of a microcontroller. The hardware used for a dot-product operation includes a population count

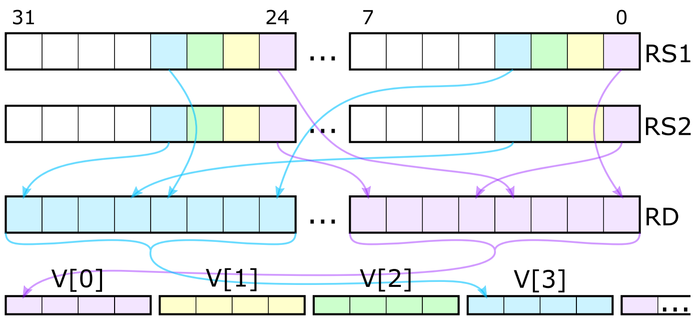


Fig. 4. Bit-packing custom instruction and memory layout for 4-bit precision, with destinations for the bits of the first eight elements indicated.

(Popcount) unit, a negation unit, a 32-bit adder, 1-bit shifter (implemented as just wires), and two 32-bit 2:1 multiplexers as shown in Figure 3. The adder and negation units are not realized separately, as they are commonly available operations performed by the ALU, as shown in the figure. However, additional 2:1 multiplexers are inferred when BISDU is integrated with the ALU. The 6-bit output of the Popcount adder-tree is zero-extended to 32 bits to match the width of the ALU. If the ALU, into which BISDU is integrated, already has a Popcount unit, then the hardware overhead is reduced to a handful of multiplexers and additional control signals.

3.2 Support for Bit-Packing

To fully utilize memory bandwidth, the matrices should be stored in memory in bit-serial fashion [37]. The data layout in memory for bit-serial operations is quite different from bit-parallel operations. In bit-serial operations, the bits are processed bit-position wise, for example, the bits of vector-elements belonging to bit position zero are processed in parallel, then the bits belonging to bit position one are processed, and so on. Therefore, the bits belonging to the same bit position have to be arranged together in memory. First is the least significant bit (bit position zero) of all elements in a matrix stored sequentially in memory, followed by all the bits from bit position one, and so on. This can be done in software using existing instructions. However, this conversion from bit-parallel to bit-serial is a costly operation if done using just logical and shift instructions, and hence, to accelerate the vector bit-packing process, a custom instruction is introduced.

The bit-packing instruction takes eight byte-sized elements of a vector stored in two registers (RS1 and RS2), and stores the rearranged bits in an output register (RD), as shown in Figure 4. Only the first (bits 7–0) and the last (bits 31–24) bytes are shown for each register. Since there is only one destination register for two source registers, it is only possible to process half of the bits in the source registers. Put differently, the bit-packing instruction is only able to compact up to 4-bits from each byte in the sources registers. The four bit positions are shown by different colors in Figure 4. The bit-packing instruction takes the least significant bit from all eight input bytes (four from RS1 and four from RS2) and places them in the least significant byte of RD (shown in purple). All the bits from bit position one are placed in byte one, bit position two in byte two, and bit position three in the most significant byte of RD (shown in blue).

The packed bits in RD can then be stored to their destinations in memory using byte-sized stores, that is, the memory is used to construct longer bit vectors of size 32 or more. Figure 4 illustrates this as a vector with 32 bits in each element that stores one bit position. A byte-size store is used to write the first eight bits of RD to the first byte of V[0], RD is then right-shifted by eight upon which a new byte-sized write is performed to the first byte of V[1], and so on. The bin-packing instruction would have to be repeated four times to fill all four bytes in each vector element. When

31 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	fn3	rd	opcode	instruction
0000000			111		1011011	dot.n.u
0000001			111		1011011	dot.n.s
0000010			111		1011011	dot.s.u
0000011			111		1011011	dot.s.s
0000100			111		1011011	pack

Fig. 5. Custom instruction encoding used by BISDU.

packing elements with more than four bits of precision, the instruction has to be used twice, with the inputs (RS1 and RS2) shifted to the right by four before the second invocation.

3.3 Instruction Formats

We use RISC-V as an example ISA for how the BISDU instructions can be added to an existing ISA. Since all our custom instructions use two source and one destination registers, they all use the R-type encoding. We use the `custom-2` opcode reserved for custom extensions in the RV32 and RV64 ISA [45]. We use the `funct7` field to specify five different operations: four are used for the dot-product and one for the bit-packing operation. The four variants of the dot-product (`dot.S.N`) are needed to specify if the input accumulator should be shifted by one (`S` : [`s`=shift by one] or [`n`=no shift]) and if the popcount result should be negated due to including a signed bit (`N` : [`s`=signed (negate)] or [`u`=unsigned]), see Line 6 in Algorithm 1. We further follow the convention introduced by ROCC [3] to indicate the registers used via the `funct3` field. The encoding scheme is shown in Figure 5.

3.4 Software Support

BISDU's native format is 32-bit words consisting of the same bit position of 32 consecutive bit-parallel elements, e.g., the least significant bits of 32 consecutive bit-parallel elements are stored as one 32-bit data word. We use the bit-packing instruction to convert any bit-parallel data to packed 32-bit data words. If multiple bits of precision are used, the bits for each bundle of 32 elements are stored consecutively in ascending order from lowest to highest bit, as shown at the bottom of Figure 4.

3.4.1 Dot-Product Computation. We define dot-product functions for different combinations of precision. An example function written using the BISDU custom instructions is shown in Listing 1. The function calculates the dot-product of two signed vectors (pointed to by `L` and `R`) with each vector consisting of 32 elements stored in a packed serial format. The first 32 bits (index zero) of L contain the lowest bit of each of its 32 elements, the second 32 bits (index one) contain the second bit, and the third 32 bits (index two) contains the highest (sign) bit. R behaves accordingly, but with only 2-bit precision. `SIGN0` and `SIGN1` are used to select between no negation and negation of the popcount value, respectively (Figure 3). Similarly, `SHIFT0` and `SHIFT1` select between no shift and shift of `ACC` by 1-bit position. Each `BISDU_DOTP` macro invocation generates a custom BISDU instruction using the support for custom instructions in RISC-V inline assembly in gcc. The output of each dot-product is accumulated in `ACC`.

A special case is ternary arithmetic. We use the representation and method described in TAB [49] to encode and compute ternary arithmetic. The encoding used is 2-bit two's complement, with the encoding for −2, i.e., 10, being unused. The result of the multiplication of two ternary numbers is always a ternary number, i.e., −1, 0 or 1. This method exploits the fact that only −1 and 1 change

```c
inline int32_t dotp_3x2(
  uint32_t *L,
  uint32_t *R)
{
  int32_t ACC = 0;
  uint32_t input = 0;
  input = L[2] & R[1];
  BISDU_DOTP(ACC, input, SHIFT0|SIGN0);
  input = L[2] & R[0];
  BISDU_DOTP(ACC, input, SHIFT1|SIGN1); // Change wave front
  input = L[1] & R[1];
  BISDU_DOTP(ACC, input, SHIFT0|SIGN1);
  input = L[1] & R[0];
  BISDU_DOTP(ACC, input, SHIFT1|SIGN0); // Change wave front
  input = L[0] & R[1];
  BISDU_DOTP(ACC, input, SHIFT0|SIGN1);
  input = L[0] & R[0];
  BISDU_DOTP(ACC, input, SHIFT1|SIGN0); // Change wave front
  return  ACC;
}
```

Listing 1. A *3-bit signed* × *2-bit signed* bit-serial dot-product function that processes 32 elements per invocation. L contains 32 elements with 3-bit precision stored in serial fashion, and R contains 32 elements with 2-bit precision stored in serial fashion (similar to what is shown at the bottom of Figure 4).

```c
inline int32_t dotp_TxT(
  uint32_t *L,
  uint32_t *R)
{
  int32_t ACC = 0;
  uint32_t any_ones = L[0] & R[0];
  uint32_t negative = L[1] ^ R[1];
  uint32_t minus_ones = negative & any_ones;
  BISDU_DOTP(ACC, minus_ones, SHIFT0|SIGN1);
  // ACC = 0 - Popcount(minus_ones)
  BISDU_DOTP(ACC, any_ones, SHIFT1|SIGN0);
  // ACC = 2 * ACC + Popcount(any_ones)
  // ACC = Popcount(any_ones) - 2 * Popcount(minus_ones)
  return  ACC;
}
```

Listing 2. A *ternary* × *ternary* bit-serial dot-product function that processes 32 elements per invocation. The ternary values are encoded bit-serial in 2-bit two's complement representation.

the result of an addition, while 0 does not. As a result, only two popcounts instead of four are required. This is further optimized by counting *any_ones*, i.e., -1 or 1 and *minus_ones*, because these can be generated using only three bit-wise operations. To compensate for using *any_ones* instead of a count of $+1$, *minus_ones* has to be subtracted twice. A function implementing this approach is shown in Listing 2.

TAB uses two separate popcount accumulators for *any_ones* and *minus_ones*, and only combines them in the end after multiplying one of them by two. Since BISDU incorporates a shift of the accumulator, as well as compute and negate of the popcount being added, we can directly

```
void matmul_3sx2s(
  uint32_t *pa,
  uint32_t *pb,
  int32_t *restrict c,
  int l,
  int m,
  int n)
{
  int A_BITS = 3;
  int B_BITS = 2;
  for(int i=0; i<l; i++){
    for(int j=0; j<n; j++){
      int32_t tmp = 0;
      for(int k=0; k<(m/32); k++){
        tmp += dotp_3x2(
            &pa[(i*(m/32)+k)*A_BITS],
            &pb[(j*(m/32)+k)*B_BITS]);
      }
      c[i * n + j] = tmp;
    }
  }
}
```

Listing 3. A bit-serial matrix multiplication function using the macro from Listing 1. pa contains the packed bits of matrix A of size $l \times m$, pb contains the packed bits of the transposed matrix B of size $n \times m$. The shared dimension m has to be divisible by 32.

perform the combination at no cost, saving registers that can be used for tiling. A normal 2-bit by 2-bit bit-serial computation function requires four loads, four bit-wise operations and four BISDU dot-product instructions for a total of 12 instructions. The improved ternary computation still requires four loads, but only three bit-wise operations and two BISDU dot-product instructions, for a total of nine instructions. This reduction of 25% is significant, since the dot-product forms the computational core of matrix multiplications and thus also convolutions.

3.4.2 Matrix Multiplication. The dot-product macros can be used as a building block for matrix multiplication, as shown in Listing 3. To improve performance and reduce the number of required loads, the matrix multiplication can be unrolled across the outer two loops. This is preferable over unrolling the innermost loop, since this way data loaded into registers can be reused, reducing the number of required load operations. The limit to this form of unrolling is the number of available registers, since the number of required registers varies based on the bit-precision, with higher precision requiring more registers.

3.4.3 Convolution. The matrix multiplication shown in Listing 3 can either be used directly, or as a building block for convolution. For convolution, we follow the general approach described by CMSIS-NN [19] and PULP-NN [12]. The latter library is used by XpulpNN and Dustin.

Activations are stored in **height-width-channel** (**HWC**) order. As 2D convolutions move in the height and width dimension, this enables copying all channel data of a pixel if it is used for a convolution in a linear copy operation. We implement two different versions of convolution. The first version is used when the input channel is a multiple of 32. This is commonly the case for the inner layers of CNNs. In this case, the activations are stored packed. The packing can be done by the preceding layer, which is commonly a convolution or max pooling layer, at low cost, due to the packing instruction. The second version first copies the bits to a buffer, pads them, and then packs that buffer. Kernels are always stored bit-packed, but have to be padded if their number of elements is not a multiple of 32.

The implemented convolution is either performed in *full* across all outputs (same height and width for output and input) with the borders being padded with zeros during the *im2col* operation, or only for *valid* outputs where the kernel completely overlaps the activations, leading to a smaller output width and height.

4 METHODOLOGY

We evaluate BISDU using Chipyard [1], an open-source SoC design framework developed at UC Berkeley. This framework provides the Rocket core [3], which is a configurable in-order RISC-V core. We use the 32-bit TinyRocket configuration of Rocket as baseline. The core configuration consists of a basic five-stage pipeline without virtual-memory or floating-point support and uses a tightly coupled 512 KiB SRAM scratchpad. To reduce the core size further, support for debug, atomic instructions, breakpoints, and physical memory protection are disabled.

We have extended the baseline TinyRocket core with common solutions such as SIMD and popcount support, as well as the state-of-the-art solutions, XpulpNN [13], and Dustin [11]. We evaluate the following core configurations:

- **RV32IC** is the baseline TinyRocket configuration that uses 8-bit integers for all operations. Since we target resource-constrained IoT devices, we opt to not include a hardware multiplier.
- **RV32IMC** is the RV32IC baseline, with support for the RISC-V M standard extension for integer multiplication and division [45].
- **SIMD** is the RV32IC baseline extended with support for performing four 8-bit operations in parallel. It is emulated using Dustin instructions.
- **Popcount** is the RV32IC baseline extended with a population count operation, which is used for performing bit-serial matrix multiplications in software as described by Umuroglu and Jahre [39].
- **XpulpNN** is the RV32IC baseline extended with our implementation of XpulpNN [13], which supports 2, 4, and 8-bit SIMD operations but requires that both operands are of equal precision.
- **Dustin** is the RV32IC baseline extended with our implementation of Dustin [11], which, in addition to 2, 4, and 8-bit SIMD operations, is also able to perform mixed-precision operations, e.g., 2×8-bit operations.
- **BISDU** is the RV32IC baseline extended with the proposed dot-product unit as described in Section 3.

We implemented support for bit-packing and dot-products, but none of the other proposed XpulpNN instructions, because we wanted to directly compare the performance of different types of dot-product instructions on a level playing field. Rocket's register file supports two register reads, and one register write per cycle. This is in contrast to RI5CY, the basis of XpulpNN and Dustin, which supports three reads and one write. As a consequence, the destination register can not be read when implemented in Rocket, requiring additional instructions for accumulation. As this limitation is shared across all units (BISDU could perform the AND operation as part of the same instruction if three registers could be read), this comparison remains fair and arguably more appropriate for small embedded cores. The register file configuration also implies that address incrementing loads can not be supported. A further difference is that mixed-precision for Dustin is implemented based on different instructions instead of an embedded controller. This increases the number of opcodes needed but simplifies the software, as compared to the original work [11].

We consider this a fair comparison, since the area cost of the embedded controller is saved and the increase in decode complexity seems to be small, as the increase in core area is dominated by the ALU. Furthermore, our comparisons focus on the size of the ALU instead of the whole core. Other, non-ISA-related optimizations Dustin introduces, such as VLEM mode, are not considered [11].

We use Cadence Genus 19.15 to synthesize the TinyRocket core without scratchpads and frontend, targeting a 28 nm FDSOI process to evaluate area and power of the implemented units. The ALUs were marked to not be ungrouped, and switching power was evaluated using vectorless power estimation. We use CACTI 7.0 [22] to estimate the leakage power and access power of the SRAM scratchpad.

5 EVALUATION

We evaluate BISDU by an in-depth analysis of low-bit and mixed-precision matrix multiplication for both statically and dynamically known matrix sizes. We discuss the performance for a single convolutional layer, how it depends on the precision of the activations and weights, and how the precision affects the required memory bandwidth. We end the evaluation by presenting the results of several CNNs and compare BISDU against the other solutions in terms of performance, area, power, and energy.

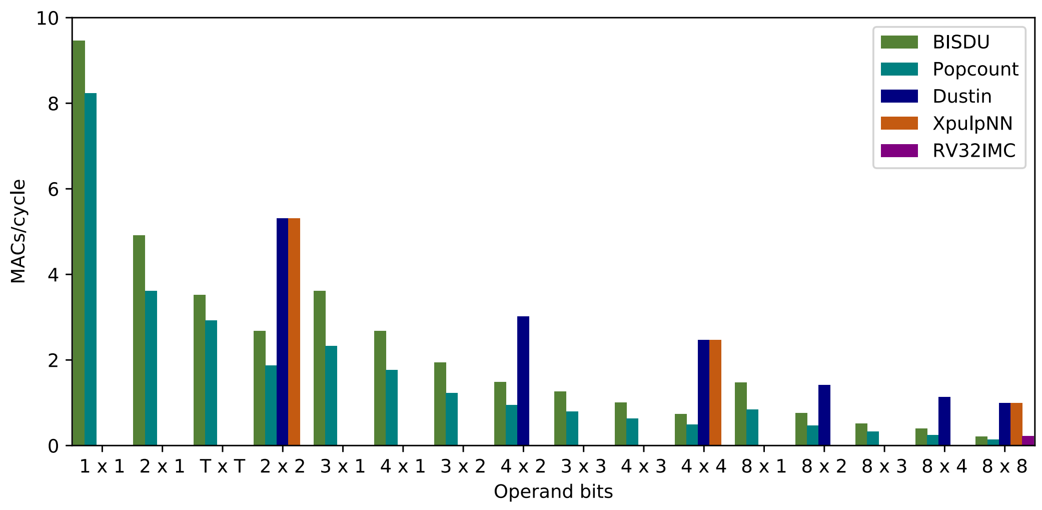
5.1 Matrix Multiplication

The performance of the different solutions are compared using tiled matrix-multiplication with both inputs already packed and the weight matrix in transposed form. Tiling is applied by calculating multiple results together, to reduce the number of loads needed. Different tiling configurations (1×1, 1×2, 1×4, 2×2, 2×4, and 4×4) were created using inlining and the best performing configurations have been used for the presented results. This automatically selects the best configuration, in terms of register pressure, for a given combination of precisions.

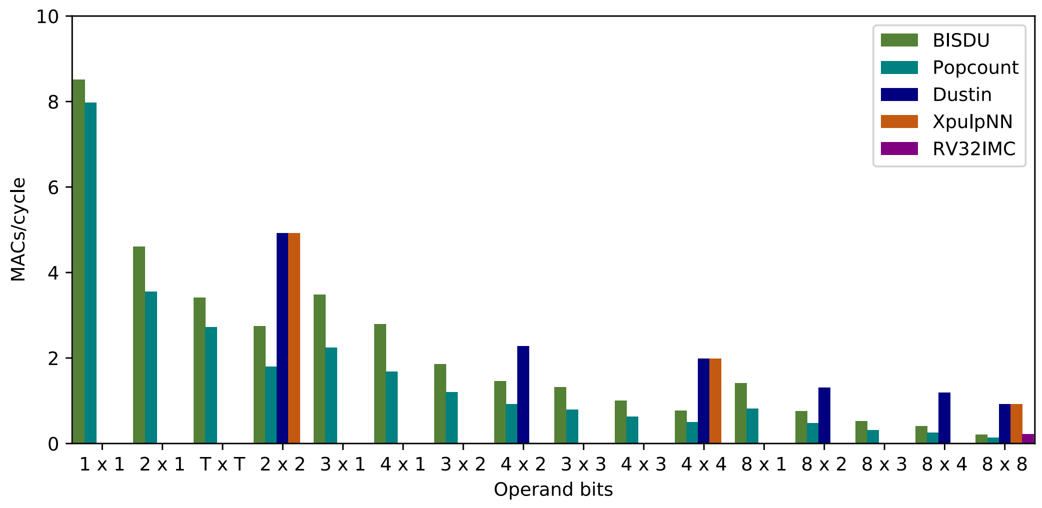
The performance results in terms of **multiply and accumulates** (**MACs**) per cycle are shown in Figure 6. We evaluate square matrices of two different sizes, 64 and 128. These sizes are known at compile time, enabling compiler optimisations. The different bit-precision combinations in the charts are denoted along the X-axis, and each combination represents the size ($A \times W$) of the activations (A) and weights (W). The charts show only natively supported combinations populated for each method, and are arranged so that for unpopulated values, the nearest combination to the right would be used. For instance, for the 4×1 combination, XpulpNN and Dustin will use 4×4 and 4×2, respectively.

The performance is low for the RV32IMC baseline, as it does not have support for SIMD instructions. The RV32IMC performance is shown for an 8×8 combination, but will remain the same for all other combinations. The figure also shows that BISDU achieves better performance for low-bit precision, and the performance degrades towards 8-bit precision, especially if the precision for both weight and activation is increased. For example, the performance for 8×1 is still acceptable, whereas for 8×8 the performance is slightly worse than the baseline. Compared to XpulpNN [13] and Dustin [11], BISDU is almost 2× faster for 1×1 precision and slightly faster for the 3×1 combination as well. However, the performance reduces for all other combinations, including a marginal drop for some combinations, such as 2×1, 4×1, and 8×1. $\mathrm{T} \times \mathrm{T}$ uses a ternary format for both weights and activations, as described in Section 3.4.1, leading to a speedup over conventional 2×2 arithmetic.

The results reveal that BISDU's performance is promising for all combinations up to four bits, but the performance degrades significantly for higher bit-precision. The figure shows that BISDU achieves better performance for mixed-precision cases, such as 3×1, 4×1, 3×2, and 4×2. This is not surprising, since the number of required bit-serial operations scale with the product of the input-operand precisions when performing multiplications. Thus, keeping the precision for one (or



(a) Size 64

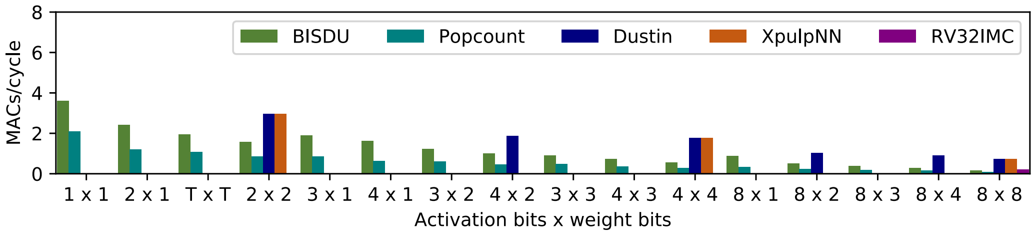


(b) Size 128

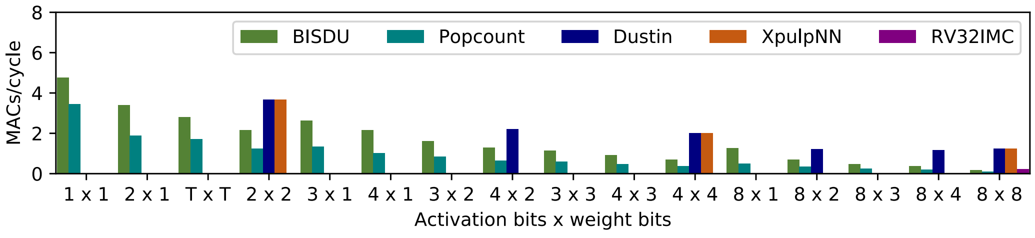
Fig. 6. Throughput for matrix multiplication of square matrices (only natively supported results are shown).

both) of the operands low, reduces the total number of operations. Luckily, there exist numerous QNN models where weights use 1 or 2 bits and activation bit-widths range from 2 to 8 bits, as shown in Figure 1. Furthermore, unless bit-serial operations are used, the data management of a none-power-of-two precision becomes a significant burden, since memory systems commonly only support 8, 16, 32, or 64 bit operations. Thus, bit-widths of 3, 5, 6, and 7 are unlikely to be supported by bit-parallel hardware. Compared to just the 8×8 Dustin results, which represent conventional 8-bit SIMD instructions, BISDU achieves a speedup of between 9.45× and 0.83× for operands up to 4 bits, with only 4×4 being a slowdown.

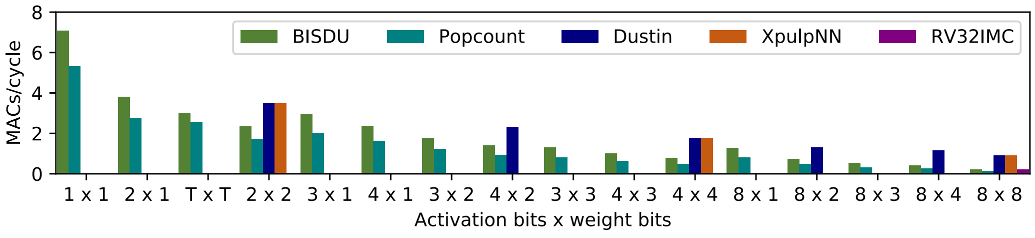
The performance of XpulpNN and Dustin is equal for the combinations that are supported by both, and favors Dustin where its mixed-precision support is applicable. BISDU experiences greater



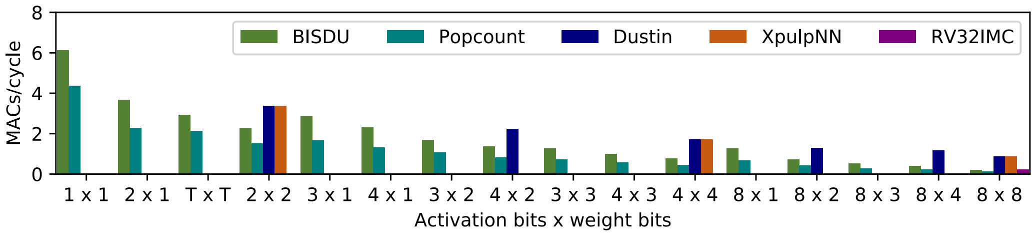
(a) Full convolution with dynamic activation packing, input size 32×32×3 and filter size 5×5×3×32



(b) Valid convolution with dynamic activation packing, input size 32×32×3 and filter size 3×3×3×64



(c) Full convolution with packed activations, input size 32×32×32 and filter size 5×5×32×32



(d) Valid convolution with packed activations, input size 12×12×128 and filter size 3×3×128×128

Fig. 7. Throughput for convolution. Kernel weights are packed and stride is 1.

register pressure compared to XpulpNN and Dustin, because multiple bit vectors have to be kept in registers while they are combined, leading to less aggressive tiling being used in some combinations. This results in more loads being required for these cases.

5.2 Convolution

The performance of BISDU when used for convolutions is shown in Figure 7. We evaluated the first layer and the biggest inner layer of two different networks, CNV from the QONNX model

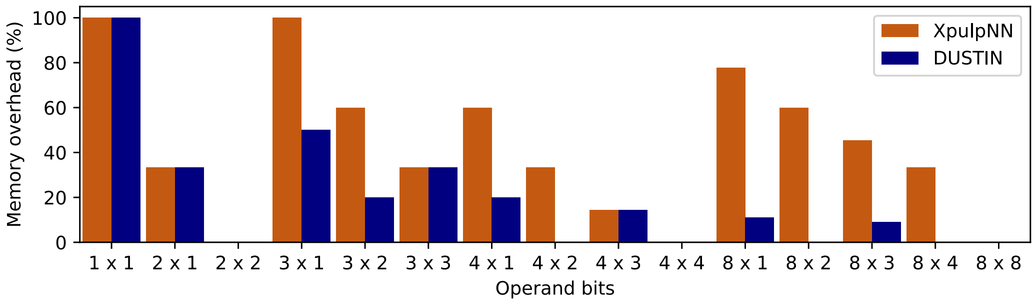


Fig. 8. Memory footprint overhead compared to BISDU.

zoo [29] and the network described in CMSIS-NN [19]. CNV uses valid mode convolution and targets low bit-precision, while the CMSIS-NN network uses full convolution and targets higher bit-precision. The convolutions are performed using the approaches described in Section 3.4. Results are reported in MACs/cycle to enable easy estimations of performance for similar layers and enable direct comparisons between the convolution types, as well as to matrix multiplication.

The first layer of CMSIS-NN has an input size of $32 \times 32 \times 3$ and a filter size of $5 \times 5 \times 3 \times 32$ for a total number of, 2 457 600 MACs. Since the kernel size of $3 \times 3 \times 3$ is not a multiple of 32, dynamic padding and packing is used. Results for the convolution are shown in Figure 7(a). The first layer of CNV has an input size of $32 \times 32 \times 3$ and a filter size of $3 \times 3 \times 3 \times 64$ for a total number of, 1 555 200 MACs and also uses dynamic packing. Results for the convolution are shown in Figure 7(b). These scenarios require a lot of copy, pad, and pack operations, leading to a comparatively low number of MACs/cycle. Comparing BISDU and Popcount demonstrates the importance of packing instructions for this type of layer. The larger vector size of 32 elements leads to more padding being required for BISDU and Popcount when compared to Dustin and XpulpNN. For Dustin or XpulpNN matrices have to be padded to a multiple of 16, 8 or 4, depending on if the smallest operand size is 2, 4, or 8-bit. For the first layer of CMSIS-NN each kernel of 75 elements needs to be padded to 96 elements for BISDU and Popcount and either 76 or 80 elements for Dustin and XpulpNN. For the first layer of CNV each kernel of 27 elements needs to be padded to 32 elements for BISDU and Popcount and either 28 or 32 elements for Dustin and XpulpNN. This leads to higher performance of the CNV layer for BISDU and Popcount, since less padding is required compared to CMSIS-NN.

The dominant inner layer of CMSIS-NN has an input size of $32 \times 32 \times 32$ and a filter size of $5 \times 5 \times 32 \times 32$ for a total number of, 26 214 400 MACs. Since the kernel size of $5 \times 5 \times 32$ is a multiple of 32, efficient packing can be used without need of padding. Results for the convolution are shown in Figure 7(c). The dominant inner layer of CNV has an input size of $12 \times 12 \times 128$ and a filter size of $3 \times 3 \times 128 \times 128$ for a total number of, 14 745 600 MACs. This convolution uses packed activations and results for the convolution are shown in Figure 7(d). Theses scenario shows that under suitable conditions, BISDU's matrix multiplication performance translates to efficient convolution implementations, that retain a large fraction of the performance seen during matrix multiplications.

BISDU has zero memory overhead for storing the weights and activations of any precision, as numbers can always be stored with the minimum number of bits required, whereas XpulpNN and Dustin incur memory overheads ranging from 0 to 100% for different precisions, as shown in Figure 8. Assuming a QNN that has 1-bit weights and 4-bit activations, they will be stored in memory as 4-bit numbers in case of XpulpNN, therefore, resulting in three bits of overhead for every five bits (1-bit weight + 4-bits activation). This equates to 60% memory overhead. Similarly,

Table 1. Cycle Count when Running a CIFAR-10 CNN

	Popcount			BISDU				Dustin		XpulpNN		SIMD
	T × T	T × 1	1 × 1	3 × 1	T × T	T × 1	1 × 1	4 × 2	T × T	4 × 4	T × T	8 × 8
Reported Accuracy	89.03%	87.80%	84.22%		89.03%	87.80%	84.22%		89.03%		89.03%	79.9%
First Layer	5420980	3091908	3035509	1328912	2295224	1328912	1328912	1277548	1277548	2142837	2142790	3377830
Convolution	100%	57%	56%	24.5%	42.3%	24.5%	24.5%	23.6%	23.6%	39.5%	39.5%	62.3%
Packed	22509575	19726591	10295604	18956053	18967486	15728941	8273258	24308752	16276999	41063176	16277138	10831633
Convolution	100%	87.6%	45.7%	84.2%	84.3%	69.9%	36.8%	108%	72.3%	182.4%	72.3%	48.1%
Fully	166997	145527	76418	148691	141221	118485	66528	210912	154922	304759	154920	19320
Connected	100%	87.1%	45.8%	89%	84.6%	71%	39.8%	126.3%	92.8%	182.5%	92.8%	11.6%
Max Pooling	395040	395184	395090	395184	395184	395135	395136	395089	395132	395088	395180	283559
	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	71.8%
Thresholding &	2881251	2638175	1547517	2571458	1050531	1050531	1163738	3592712	968597	3585466	969462	736135
Packing	100%	91.6%	53.7%	89.2%	36.5%	36.5%	40.4%	124.7%	33.6%	124.4%	33.6%	25.5%
Total (cycles)	31373843	25997385	15350138	23400298	22849646	18622004	11227572	29785013	19073198	47491326	19939490	15248477
Total (seconds)	0.027	0.023	0.013	0.020	0.020	0.016	0.010	0.026	0.017	0.041	0.017	0.013
	100%	82.9%	48.9%	74.6%	72.8%	59.4%	35.8%	94.9%	60.8%	151.4%	63.6%	48.6%

Dustin will give rise to 20% of memory overhead for the same example, as it will store the weights and activations as 2-bit and 4-bit numbers. This overhead is especially significant for weights, that need to be stored during the whole run-time of an application. The only scenario in which BISDU incurs memory overhead is when the shared dimension of two matrices is not a multiple of 32, and they have to be padded, for example for layer one of CNV and CMSIS-NN.

5.3 Convolutional Neural Networks (CNNs)

We evaluated QNN performance using variations of CNV, a network from the QONNX model zoo [29]. This network targets the CIFAR-10 dataset and is available in $T \times T$, $T \times 1$, and 1×1 bit configurations for activations and weights respectively, where T stands for ternary represented using 2 bits. The configurations have reported accuracy of 89.03%, 87.80%, and 84.22%, respectively. The advantage of the 1-bit weight configurations is that they require only half as much memory to store weights. The first layer uses 8-bit activations in all configurations, making it both a mixed-operand-precision and mixed-layer-precision network.

We also report results for an 8-bit quantized network for the same data-set described in CMSIS-NN as a comparison point suitable for 8-bit SIMD. This network has a reported accuracy of 79.9% [19]. One potential reason for the lower accuracy is that the network is much smaller in terms of number of weights, but it has comparable storage requirements and execution time due to using 8-bit precision.

The results are shown in Table 1 and Figure 9. The table shows number of clock cycles taken for executing different CNN layer types by different core configurations, as well as total execution time in terms of cycles and seconds at a frequency of 1.15 GHz. The table also indicates the percentage of cycles taken by each configuration with respect to the $T \times T$ configuration run using Popcount. Among all convolutional layers, Layer 1 uses dynamic packing, while others use packed inputs achieved by packing as a part of the previous thresholding layer.

For the complete network, $T \times T$ XpulpNN is 15% and Dustin 20% faster than BISDU, which is largely caused by the convolution layers. The first layer uses 8-bit activations and ternary weights, while all other convolutional layers use packed convolution with ternary activations and weights, and show a lower slowdown over XpulpNN and Dustin for BISDU. Dustin and XpulpNN perform largely comparable, since they only differ in the behavior of the first layer. Dustin uses 2-bit weights for the first layer, while XpulpNN has to use 8-bit weights, leading to higher storage requirements. The use of 2-bit and 8-bit weights also affects the amount of tiling possible and padding required, with both favoring XpulpNN slightly. As a result, the performance benefit of mixed-operand support is lower for the first layer. Further discussion of the first layer can be found in Section 5.2.

The flexibility of BISDU enables trading of accuracy against run-time and memory requirements for weights. This enables the use of $T \times 1$ or 1×1 configurations, leading to both lower cycle

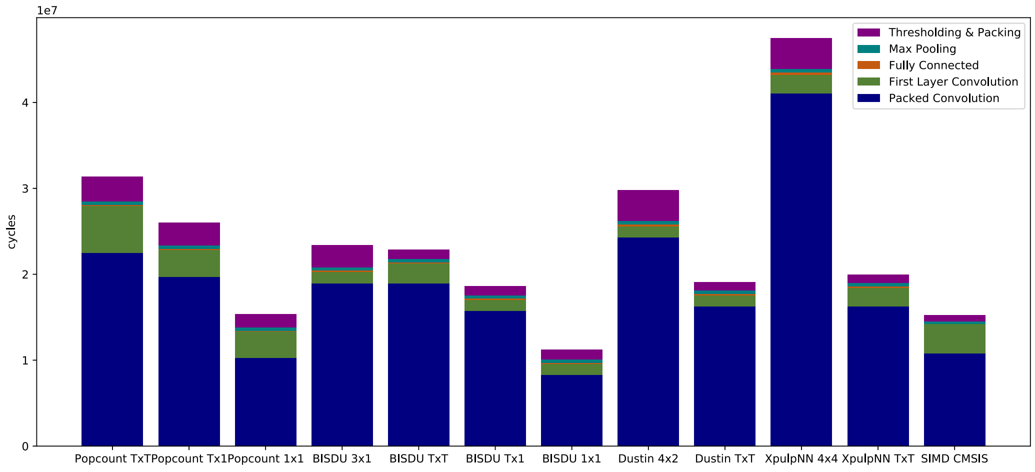


Fig. 9. Cycle count when running different CIFAR-10 CNNs.

Table 2. Area and Power consumption of ALU and Core Using a 28 nm FDSOI Process Targeting 1.15 GHz

	RV32IC	Popcount		XpulpNN [13]		Dustin [11]		BISDU	
ALU Power (μW)	1 159	1 518	1.31×	4 403	3.8×	6 923	5.97×	1 643	1.42×
ALU Area (μm^2)	813	1 154	1.42×	5 649	6.9×	9 031	11.1×	1 366	1.68×
Core Power (μW)	15 991	16 530	103%	19 768	124%	22 073	138%	16 712	105%
Core Area (μm^2)	22 117	22 837	103%	27 528	124%	30 893	140%	23 159	105%

counts and weight storage requirements, as compared to Dustin and XpulpNN. Since Dustin and XpulpNN do not natively support 1-bit precision, they would see no performance gain and would only sacrifice the result accuracy.

Compared to the Popcount $T \times T$ configuration, BISDU is overall 37% faster. However, on Layer 1, BISDU is 136% faster, largely due to the high cost of dynamic packing without dedicated instructions in Popcount. BISDU's packing instructions lead to a 174% speedup over Popcount in thresholding and packing layers.

We also present performance results for BISDU in a 3×1, XpulpNN in a 4×4 and Dustin in a 4×2 configuration, with the latter two being the minimum required to represent 3×1 on XpulpNN and Dustin. We present these results even though we do not have accuracy numbers for them, to illustrate a use case where a higher result accuracy is required, but weight storage is limited. It is likely that 3×1 would increase the accuracy over 2×1, without increasing the weight footprint. However, this is not a valid option for Dustin and XpulpNN, since they are unable to use 1-bit weights.

5.4 Area, Power, and Energy

Table 2 shows the results after synthesizing the RV32IC baseline, Popcount, XpulpNN, Dustin, and BISDU with a target frequency of 1.15 GHz, the highest frequency achievable by all designs. We show the area and power for the whole core, excluding memory for instructions and data. However, since core area and power are highly dependent on the specific core and its configuration, and since the ALU is the most impacted part of the core, we also show the area and power for only the ALU.

The estimated core power consumption for the RV32IC baseline is 15 991 μW, as shown in the table. This estimate is of limited reliability, since it was calculated using vectorless activation propagation, which can give imprecise results for sequential designs. Nevertheless, XpulpNN and

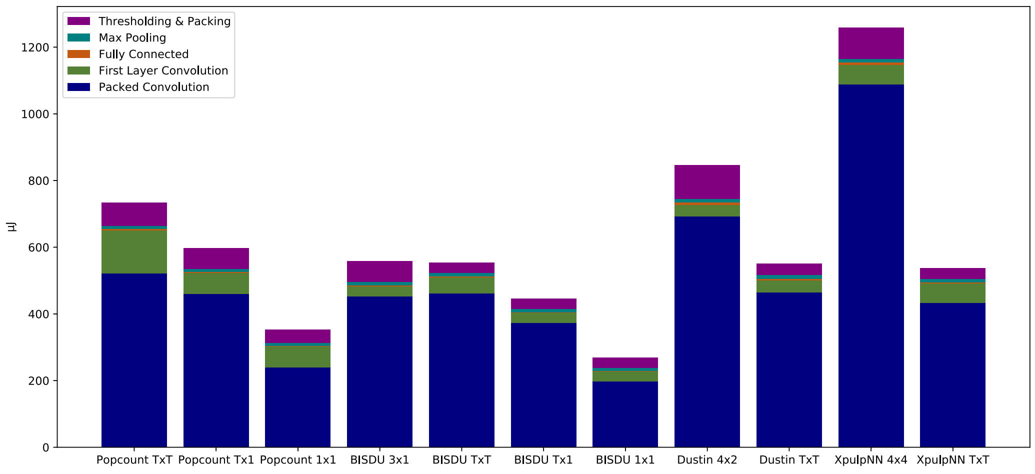


Fig. 10. Energy consumed by the core including 512 KB of memory when executing the CNNs described in Section 5.3.

Dustin increase the core power consumption by 24% and 28%, respectively, compared to the RV32IC baseline. The results show that Dustin's support for mixed-precision comes at a significant cost compared to XpulpNN in terms of power. In contrast, BISDU increases the core power consumption by only 5%. The core area of the RV32IC baseline is 0.022 mm^2 and XpulpNN and Dustin increase the area by 24% and 40%, respectively, while BISDU only increases the area by 5%.

A more interesting comparison is to look at the impact on the ALU, since the core constitutes a large "constant" fraction that is not affected by either BISDU, XpulpNN, or Dustin. This constant fraction will depend on the particular core and its configuration that the different techniques are integrated into. The power estimation for the ALU is also more realistic, since it is a pure combinatorial circuit. The ALU power of the RV32IC baseline is only 1 159 μW (Table 2). XpulpNN increases the power by 3.8× while Dustin increase it by 5.97×. In comparison, BISDU increases the power by only 42% (i.e., 1.42×). The difference in area is even more dramatic, with XpulpNN and Dustin being 6.9× and 11.1× larger than the RV32IC baseline ALU, while BISDU only increases the ALU area by 68% (i.e., 1.68×).

Figure 10 shows the energy usage of the whole core, including the SRAM scratchpad energy calculated using load and store count, when running the CNNs from Section 5.3. BISDU (554μJ), Dustin (550μJ) and XpulpNN (538μJ) have comparable energy usage that is significantly below Popcount (734μJ) for $T \times T$. BISDU and Popcount provide the option of significantly lowering the energy usage at the cost of accuracy by using reduced precision for the weights, with BISDU more than halving its energy usage to 269μJ for 1×1. Not yet taken into consideration here is the fact that the reduced number of weights could lead to a smaller memory being viable, leading to a further reduction in energy usage, as well as cost savings.

Another factor to consider is power consumed when not performing CNN calculations. The large increase in ALU area and power consumption for Dustin and XpulpNN could have significant impacts on standby power and power consumed during other tasks. While power gating the unit might help mitigate this, it would further increase area and design complexity.

6 RELATED WORK

In recent works [11, 13], 4-bit and 2-bit SIMD instructions have been added to the RISC-V ISA to support execution of low-bit QNNs on microcontrollers. These works extend the ISA of the

32-bit RI5CY core [14], a state-of-the-art open-source RISC-V core. The baseline dot-product unit available in the RI5CY core consists of two sets of multipliers, supporting 16-bit and 8-bit vector operands. The XpulpNN ISA extension [13] adds support for 4-bit and 2-bit vector operands. The added multipliers compute the dot-product of two vectors, each having either eight 4-bit elements or sixteen 2-bit elements, and the result is accumulated in a 32-bit register using an adder tree. XpulpNN has the limitation that it does not support mixed-precision operations, where operands have different bit-widths. BISDU is able to support any combination of bit-precision, while fully utilizing hardware resources and with no memory and bandwidth overhead.

Dustin [11] introduces a RISC-V ISA extension for mixed-precision and heavily quantized SIMD instructions from 16 down to 2 bits. This work presents a mixed-precision dot-product unit that can compute the dot-product of two vectors with different bit-widths as operands. For instance, 4-bit operands can be multiplied with 2-bit operands while the numbers are stored in memory in 4-bit and 2-bit packed formats. However, Dustin incurs a hardware overhead and requires an additional mixed-precision controller to *unpack the lower-bit operand to match its precision to the precision of the higher-bit operand*. Moreover, this controller introduces state in Dustin's custom instructions for performing dot-product operations. BISDU supports all forms of precision (not only those that are a power of two) and has a completely state-less design, which simplifies exception and interrupt handling as well as context switches.

Umuroglu and Jahre presented a bit-serial method for executing low-bit QNN on microcontrollers [39]. They explain that integer matrix multiplication can be carried out by a weighted sum of binary matrix multiplications using bit-serial operations (Section 2.1). Cowan et al. [9] extended the TVM framework to automatically generate high-performance kernels for QNNs using existing popcount instructions in the Arm NEON instruction set [2]. They target application class processors with vector extensions employing parallelism and cache-aware memory layout optimization, among other optimizations. Their focus is on maximizing performance using existing hardware and ISA extensions rather than evaluating the cost of different extensions, and the class of hardware targeted is different. BISDU is a hardware extension that provides efficient conversion of bit-parallel to bit-serial data and efficient execution of bit-serial matrix multiplication and targets lower power scenarios.

Several standalone bit-serial accelerators have been proposed [6, 16, 20, 32, 34, 41] with the goal of efficient execution of precision variable neural networks. All these works are proposed as standalone accelerators and are not tightly integrated into the datapath of a microcontroller. As separate accelerators, they require more hardware than is commonly warranted for resource constrained microcontrollers. BISDU is a light-weight extension for existing ISAs, which enables efficient execution of precision variable neural networks on microcontrollers at a low hardware cost (i.e., low silicon area increase).

7 CONCLUSION

This article has presented BISDU, a bit-serial dot-product unit that supports efficient processing of mixed-precision and low-bit QNNs. BISDU is a simple and hardware-efficient dot-product unit that is based on computing the dot-product of two binary vectors instead of integer vectors, which eliminates the need for multipliers. BISDU can be used for fast processing of binarized and ternary neural networks, which are gaining popularity for possible deployment on IoT edge devices. These edge devices have tight constraints on hardware resources and power budget, yet require good performance. This work demonstrates that BISDU can be easily integrated in low-cost and low-speed embedded microcontrollers, providing a low-cost solution for performing machine learning inference at the edge.

REFERENCES

[1] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, Paul Rigge, Colin Schmidt, John Wright, Jerry Zhao, Yakun Sophia Shao, Krste Asanović, and Borivoje Nikololić. 2020. Chipyard: Integrated design, simulation, and implementation framework for custom SoCs. *IEEE Micro* 40, 4 (2020), 10–21. DOI: https://doi.org/10.1109/MM.2020.2996616

[2] Arm. 2013. *NEON Programmer's Guide.* Technical Report. 411 pages.

[3] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. 2016. *The Rocket Chip Generator.* Technical Report. Retrieved from http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[4] Michaela Blott, Thomas B. Preußer, Nicholas J. Fraser, Giulio Gambardella, Kenneth O'brien, Yaman Umuroglu, Miriam Leeser, and Kees Vissers. 2018. FINN-R: An end-to-end deep-learning framework for fast exploration of quantized neural networks. *ACM Transactions on Reconfigurable Technology and Systems* 11, 3 (2018), 1–23. DOI: https://doi.org/10.1145/3242897

[5] Alessandro Capotondi, Manuele Rusci, Marco Fariselli, and Luca Benini. 2020. CMix-NN: Mixed low-precision CNN library for memory-constrained edge devices. *Proceedings of the IEEE Transactions on Circuits and Systems II* 67, 5 (2020), 871–875. DOI: https://doi.org/10.1109/TCSII.2020.2983648

[6] Maurizio Capra, Francesco Conti, and Maurizio Martina. 2021. A multi-precision bit-serial hardware accelerator IP for deep learning enabled internet-of-things. In *Proceedings of the IEEE International Midwest Symposium on Circuits and Systems.* 192–197. DOI: https://doi.org/10.1109/MWSCAS47672.2021.9531722

[7] Peng Chen, Jing Liu, Bohan Zhuang, Mingkui Tan, and Chunhua Shen. 2021. AQD: Towards accurate quantized object detection. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition.* IEEE Computer Society, 104–113. DOI: https://doi.org/10.1109/CVPR46437.2021.00017

[8] Peng Chen, Bohan Zhuang, and Chunhua Shen. 2021. FATNN: Fast and accurate ternary neural networks. In *Proceedings of the IEEE/CFV International Conference on Computer Vision.* IEEE Computer Society, 5199–5208. DOI: https://doi.org/10.1109/ICCV48922.2021.00517

[9] Meghan Cowan, Thierry Moreau, Tianqi Chen, James Bornholt, and Luis Ceze. 2020. Automatic generation of high-performance quantized machine learning kernels. In *Proceedings of the International Symposium on Code Generation and Optimization.* Association for Computing Machinery, 305–316. DOI: https://doi.org/10.1145/3368826.3377912

[10] Julian Faraone, Nicholas Fraser, Michaela Blott, and Philip H. W. Leong. 2018. SYQ: Learning symmetric quantization for efficient deep neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition.* 4300–4309. DOI: https://doi.org/10.1109/CVPR.2018.00452

[11] Angelo Garofalo, Gianmarco Ottavi, Alfio di Mauro, Francesco Conti, Giuseppe Tagliavini, Luca Benini, and Davide Rossi. 2021. A 1.15 TOPS/W, 16-cores parallel ultra-low power cluster with 2b-to-32b fully flexible bit-precision and vector lockstep execution mode. In *Proceedings of the IEEE European Solid State Circuits Conference.* 267–270. DOI: https://doi.org/10.1109/ESSCIRC53450.2021.9567767

[12] Angelo Garofalo, Manuele Rusci, Francesco Conti, Davide Rossi, and Luca Benini. 2020. PULP-NN: Accelerating quantized neural networks on parallel ultra-low-power RISC-V processors. *RSTA* 378, 2164 (2020), 20190155. DOI: https://doi.org/10.1098/rsta.2019.0155

[13] Angelo Garofalo, Giuseppe Tagliavini, Francesco Conti, Luca Benini, and Davide Rossi. 2021. XpulpNN: Enabling energy efficient and flexible inference of quantized neural networks on RISC-V based IoT end nodes. *IEEE Transactions on Emerging Topics in Computing* 9, 3 (2021), 1489–1505. DOI: https://doi.org/10.1109/TETC.2021.3072337

[14] Michael Gautschi, Pasquale Davide Schiavone, Andreas Traber, Igor Loi, Antonio Pullini, Davide Rossi, Eric Flamand, Frank K. Gürkaynak, and Luca Benini. 2017. Near-threshold RISC-V core with DSP extensions for scalable IoT endpoint devices. *TVLSI* 25, 10 (2017), 2700–2713. DOI: https://doi.org/10.1109/TVLSI.2017.2654506

[15] Mohammad Ghasemzadeh, Mohammad Samragh, and Farinaz Koushanfar. 2018. ReBNet: Residual binarized neural network. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines.* IEEE Computer Society, 57–64. DOI: https://doi.org/10.1109/FCCM.2018.00018

[16] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor M. Aamodt, and Andreas Moshovos. 2016. Stripes: Bit-serial deep neural network computing. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture.* 1–12. DOI: https://doi.org/10.1109/MICRO.2016.7783722

[17] Sangil Jung, Changyong Son, Seohyung Lee, Jinwoo Son, Jae-Joon Han, Youngjun Kwak, Sung Ju Hwang, and Changkyu Choi. 2019. Learning to quantize deep networks by optimizing quantization intervals with task loss. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition.* IEEE Computer Society, 4345–4354. DOI: https://doi.org/10.1109/CVPR.2019.00448

[18] Jaeha Kung, David Zhang, Gooitzen van der Wal, Sek Chai, and Saibal Mukhopadhyay. 2018. Efficient object detection using embedded binarized neural networks. *Journal of Signal Processing Systems* 90, 6 (2018), 877–890. DOI: https://doi.org/10.1007/s11265-017-1255-5

[19] Liangzhen Lai, Naveen Suda, and Vikas Chandra. 2018. CMSIS-NN: Efficient neural network kernels for arm cortex-M CPUs. arXiv:1801.06601. Retrieved from https://arxiv.org/abs/1801.06601

[20] Jinmook Lee, Changhyeon Kim, Sanghoon Kang, Dongjoo Shin, Sangyeob Kim, and Hoi-Jun Yoo. 2019. UNPU: An energy-efficient deep neural network accelerator with fully variable weight bit precision. 54, 1 (2019), 173–185. DOI: https://doi.org/10.1109/JSSC.2018.2865489

[21] Rundong Li, Yan Wang, Feng Liang, Hongwei Qin, Junjie Yan, and Rui Fan. 2019. Fully quantized network for object detection. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. IEEE Computer Society, 2805–2814. DOI: https://doi.org/10.1109/CVPR.2019.00292

[22] Sheng Li, Ke Chen, Jung Ho Ahn, Jay B. Brockman, and Norman P. Jouppi. 2011. CACTI-P: Architecture-level modeling for SRAM-based structures with advanced leakage reduction techniques. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*. 694–701. DOI: https://doi.org/10.1109/ICCAD.2011.6105405

[23] Yue Li, Wenrui Ding, Chunlei Liu, Baochang Zhang, and Guodong Guo. 2021. TRQ: Ternary neural networks with residual quantization. *Proceedings of the AAAI Conference on Artificial Intelligence* 35, 10 (2021), 8538–8546. DOI: https://doi.org/10.1609/aaai.v35i10.17036

[24] Yuhang Li, Xin Dong, Sai Qian Zhang, Haoli Bai, Yuanpeng Chen, and Wei Wang. 2020. RTN: Reparameterized ternary network. *Proceedings of the AAAI Conference on Artificial Intelligence* 34, 04 (2020), 4780–4787. DOI: https://doi.org/10.1609/aaai.v34i04.5912

[25] Xiaofan Lin, Cong Zhao, and Wei Pan. 2017. Towards accurate binary convolutional neural network. In *Proceedings of the Conference on Advances in Neural Information Processing Systems*. 9.

[26] Hongyang Liu, Sara Elkerdawy, Nilanjan Ray, and Mostafa Elhoushi. 2021. Layer importance estimation with imprinting for neural network quantization. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*. IEEE Computer Society, 2408–2417. DOI: https://doi.org/10.1109/CVPRW53098.2021.00273

[27] Zechun Liu, Zhiqiang Shen, Marios Savvides, and Kwang-Ting Cheng. 2020. ReActNet: Towards precise binary neural network with generalized activation functions. In *Proceedings of the European Conference on Computer Vision*. Springer International Publishing, 143–159. DOI: https://doi.org/10.1007/978-3-030-58568-6_9

[28] Yuriy Mishchenko, Yusuf Goren, Ming Sun, Chris Beauchene, Spyros Matsoukas, Oleg Rybakov, and Shiv Naga Prasad Vitaladevuni. 2019. Low-bit quantization and quantization-aware training for small-footprint keyword spotting. In *Proceedings of the IEEE International Conference On Machine Learning And Applications*. 706–711. DOI: https://doi.org/10.1109/ICMLA.2019.00127

[29] Alessandro Pappalardo, Yaman Umuroglu, Michaela Blott, Jovan Mitrevski, Ben Hawks, Nhan Tran, Vladimir Loncar, Sioni Summers, Hendrik Borras, Jules Muhizi, Matthew Trahms, Shih-Chieh Hsu, Scott Hauck, and Javier Duarte. 2022. QONNX: Representing arbitrary-precision quantized neural networks. arXiv:2206.07527. Retrieved from https://arxiv.org/abs/2206.07527

[30] Haotong Qin, Ruihao Gong, Xianglong Liu, Mingzhu Shen, Ziran Wei, Fengwei Yu, and Jingkuan Song. 2020. Forward and backward information retention for accurate binary neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. IEEE Computer Society, 2247–2256. DOI: https://doi.org/10.1109/CVPR42600.2020.00232

[31] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. 2016. XNOR-Net: ImageNet classification using binary convolutional neural networks. In *Proceedings of the European Conference on Computer Vision*. Springer International Publishing, 525–542. DOI: https://doi.org/10.1007/978-3-319-46493-0_32

[32] Sungju Ryu, Hyungjun Kim, Wooseok Yi, Eunhwan Kim, Yulhwa Kim, Taesu Kim, and Jae-Joon Kim. 2022. BitBlade: Energy-efficient variable bit-precision hardware accelerator for quantized neural networks. *IEEE Journal of Solid-State Circuits* 57, 6 (2022), 1924–1935. DOI: https://doi.org/10.1109/JSSC.2022.3141050

[33] Dominik Scherer, Andreas Müller, and Sven Behnke. 2010. Evaluation of pooling operations in convolutional architectures for object recognition. In *Proceedings of the International Conference on Artificial Neural Networks*. Springer, 92–101. DOI: https://doi.org/10.1007/978-3-642-15825-4_10

[34] Sayeh Sharify, Alberto Delmas Lascorz, Kevin Siu, Patrick Judd, and Andreas Moshovos. 2018. Loom: Exploiting weight and activation precisions to accelerate convolutional neural networks. In *Proceedings of the ACM/IEEE Design Automation Conference*. 1–6. DOI: https://doi.org/10.1109/DAC.2018.8465915

[35] Siyang Sun, Yingjie Yin, Xingang Wang, De Xu, Wenqi Wu, and Qingyi Gu. 2018. Fast object detection based on binary deep convolution neural networks. *CAAI Transactions on Intelligence Technology* 3, 4 (2018), 191–197. DOI: https://doi.org/10.1049/trit.2018.1026

[36] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. 2017. Efficient processing of deep neural networks: A tutorial and survey. *Proc. IEEE* 105, 12 (2017), 2295–2329. DOI: https://doi.org/10.1109/JPROC.2017.2761740

[37] Yaman Umuroglu, Davide Conficconi, Lahiru Rasnayake, Thomas B. Preusser, and Magnus Själander. 2019. Optimizing bit-serial matrix multiplication for reconfigurable computing. *ACM Transactions on Reconfigurable Technology and Systems* 12, 3 (2019), 15:1–15:24. DOI: https://doi.org/10.1145/3337929

[38] 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*. Association for Computing Machinery, 65–74. DOI: https://doi.org/10.1145/3020078.3021744

[39] Yaman Umuroglu and Magnus Jahre. 2017. Towards efficient quantized neural network inference on mobile devices: Work-in-progress. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. Association for Computing Machinery, 1–2. DOI: https://doi.org/10.1145/3125501.3125528

[40] Yaman Umuroglu and Magnus Jahre. 2018. Streamlined deployment for quantized neural networks. arXiv:1709.04060. Retrieved from https://arxiv.org/abs/1709.04060

[41] Yaman Umuroglu, Lahiru Rasnayake, and Magnus Själander. 2018. BISMO: A scalable bit-serial matrix multiplication overlay for reconfigurable computing. In *Proceedings of the Conference on Field Programmable Logic and Applications*. 307–3077. DOI: https://doi.org/10.1109/FPL.2018.00059

[42] Vaibhav Verma, Tommy Tracy II, and Mircea R. Stan. 2022. EXTREM-EDGE–EXtensions To RISC-V for energy-efficient ML inference at the EDGE of IoT. *Sustainable Computing: Informatics and Systems* 35 (2022), 100742. DOI: https://doi.org/10.1016/j.suscom.2022.100742

[43] Ao Wang, Wenxing Xu, Hanshi Sun, Ninghao Pu, Zijin Liu, and Hao Liu. 2022. Arrhythmia classifier using binarized convolutional neural network for resource-constrained devices. In *Proceedings of the International Conference on Communications, Information System and Computer Engineering*. IEEE Computer Society, 213–220. DOI: https://doi.org/10.1109/CISCE55963.2022.9851002

[44] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. 2019. HAQ: Hardware-aware automated quantization with mixed precision. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 8604–8612. DOI: https://doi.org/10.1109/CVPR.2019.00881

[45] Andrew Waterman and Krste Asanovic. 2019. The RISC-V Instruction Set Manual. (2019). Retrieved 17 June 2022 from https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf

[46] Qing Wu, Yangfan Sun, Hui Yan, and Xundong Wu. 2020. ECG signal classification with binarized convolutional neural network. *Computers in Biology and Medicine* 121 (2020), 103800. DOI: https://doi.org/10.1016/j.compbiomed.2020.103800

[47] Dongqing Zhang, Jiaolong Yang, Dongqiangzi Ye, and Gang Hua. 2018. LQ-Nets: Learned quantization for highly accurate and compact deep neural networks. In *Proceedings of the European Conference on Computer Vision*. Springer International Publishing, 373–390. Retrieved from http://link.springer.com/10.1007/978-3-030-01237-3_23

[48] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. 2018. DoReFa-Net: Training low bitwidth convolutional neural networks with low bitwidth gradients. (Feb. 2018). arXiv:1606.06160. Retrieved from https://arxiv.org/abs/1606.06160

[49] Shien Zhu, Luan H. K. Duong, and Weichen Liu. 2022. TAB: Unified and optimized ternary, binary and mixed-precision neural network inference on the edge. *ACM Transactions on Embedded Computing Systems* 21, 5 (2022), 26. DOI: https://doi.org/10.1145/3508390

[50] Bohan Zhuang, Chunhua Shen, Mingkui Tan, Lingqiao Liu, and Ian Reid. 2019. Structured binary neural networks for accurate image classification and semantic segmentation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. IEEE Computer Society, 413–422. DOI: https://doi.org/10.1109/CVPR.2019.00050

Received 31 October 2022; revised 12 May 2023; accepted 28 May 2023