ULPPACK: FAST SUB-8-BIT MATRIX MULTIPLY ON COMMODITY SIMD HARDWARE

Jaeyeon Won [1] **Jeyeon Si** [2] **Sam Son** [2] **Tae Jun Ham** [2] **Jae W. Lee** [2]

ABSTRACT

Recent progress in quantization techniques has demonstrated the feasibility of sub-8-bit quantization with a negligible end-to-end accuracy drop. However, today's commodity hardware such as CPUs and GPUs is still suboptimal in executing these sub-8-bit quantized networks as its SIMD instructions only support the granularity of 8 bits or wider. This paper presents ULPPACK, a software technique to accelerate those **u**ltra **l**ow-**p**recision networks via effective operand **pack**ing. The key idea of ULPPACK is to pack multiple low-precision (<8 bits) operands densely into a single wide (≥16 bits) register and perform multiple narrow multiply-accumulate (MAC) operations with a single wide multiply. We introduce two effective packing schemes with different tradeoffs as well as optimizations to amortize the overhead of shifting and masking the output partial sum. Our evaluation of ULPPACK with a 512×512×512 GEMM kernel demonstrates substantial performance gains over state-of-the-art low-precision linear algebra libraries with a speedup of 2.1×, 1.8×, and 2.7× for 3-bit weights/activations (W3A3) over Google's GEMMLOWP, Facebook's QNNPACK, and an optimized bit-serial implementation, respectively. For end-to-end evaluation on PyTorch with seven 3-bit quantized convolutional neural networks (CNNs), ULPPACK achieves geomean speedups of 3.9× and 1.5× over the baseline 32-bit floating-point (FP32) and QNNPACK, respectively.

1 INTRODUCTION

Recently, deep neural networks (DNNs) have demonstrated excellent performance in various application domains. These complex models require a large amount of computation and thus are often deployed on data-parallel accelerators such as GPUs and TPUs (Jouppi et al., 2017). Deploying those increasingly complex DNNs on the mobile CPU is a challenging task as it has relatively low performance, small memory size, and limited power budget. To address this challenge, many proposals have emerged to reduce the computational cost of DNNs. Among them network quantization techniques (Zhou et al., 2016; Choi et al., 2018a; 2019; 2018b; Jung et al., 2019) have gained a lot of attraction to reduce the computational intensity and memory footprint with minimal degradation of the model accuracy.

Quantized DNNs represent weights and activations in a low-precision format such as an 8-bit integer. Recent studies have been actively exploring ways to quantize network models using sub-8-bit precision numbers while minimizing

Table 1. Top-1 accuracy change on ImageNet with Learned Step-size Quantization (LSQ) (Esser et al., 2020) over the baseline single-precision floating-point. WxAy represents that x bits are used for weights and y bits for activations.

MODEL	W8A8	W4A4	W3A3	W2A2
RESNET-18	+0.6%	+0.6%	-0.3%	-2.9%
RESNET-34	0%	0%	-0.7%	-2.5%
VGG-16	+0.1%	+0.6%	0.0%	-2.0%

accuracy degradation. For instance, LSQ (Esser et al., 2020), a state-of-the-art quantization technique, yields no accuracy drop for the ImageNet classification task on a 4-bit quantized network (Table 1). Furthermore, Figure 1 shows that the advancement of quantization techniques keeps reducing the accuracy loss caused by an aggressive sub-8-bit quantization. We observe that a 3-bit representation for both weights and activations (W3A3) preserves the accuracy while that of a 2-bit representation (W2A2) is continuously improving.

Network quantization enables mobile devices to significantly improve DNN inference throughput as well as energy efficiency, and hence is widely adopted in practice (Wu et al., 2019). Typically, CPUs utilize SIMD units to efficiently perform multiple operations in parallel. When 8-bit fixed-point numbers (integers) are used, SIMD instructions can effectively exploit byte-level data parallelism, which

[1]Massachusetts Institute of Technology, Massachusetts, USA [2]Neural Processing Research Center (NPRC), Seoul National University, Seoul, South Korea. Correspondence to: Jae W. Lee <jaewlee@snu.ac.kr>. This work was conducted while Jaeyeon Won was an undergraduate researcher at Seoul National University.

Proceedings of the 5th MLSys Conference, Santa Clara, CA, USA, 2022. Copyright 2022 by the author(s).

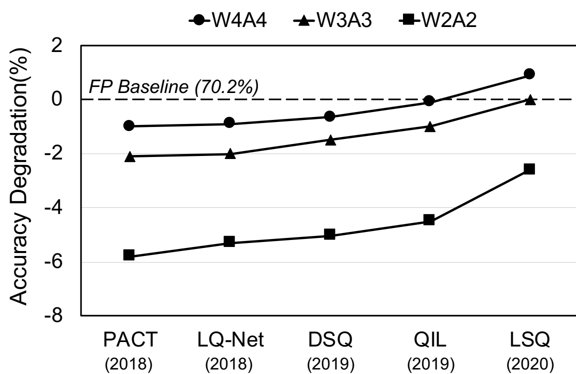


Figure 1. Recent advance in network quantizations (Esser et al., 2020; Gong et al., 2019; Jung et al., 2019; Choi et al., 2018a; Zhang et al., 2018) for ImageNet top-1 classification on ResNet18.

are well supported in mainstream ISAs. Popular DNN processing frameworks already integrate low-precision linear algebra libraries based on these instructions. For example, TensorFlow-Lite utilizes GEMMLOWP (Jacob et al., 2017), and PyTorch utilizes QNNPACK (Dukhan et al., 2018) and FBGEMM (Park et al., 2018).

However, these low-precision libraries are not effective when executing sub-8-bit quantized networks. It is because the underlying SIMD units support data parallel execution only for 8 bits or wider. One existing proposal to address this problem is to use *bit-serial* computation. However, according to our evaluation, even the state-of-the-art bit-serial linear algebra libraries (Umuroglu & Jahre, 2017; Cowan et al., 2020) have a competitive advantage only when the bit-width is extremely narrow (e.g., $\leq$2).

Thus, this paper proposes ULPPACK, a software technique to accelerate that ultra low-precision ($\leq$8 bits) network inference on commodity mobile devices. The key idea of ULPPACK is that we can compute a dot product of two low-precision vectors by carefully packing their elements into two wide ($\geq$16) registers and computing their product. To optimize packing, we present two packing schemes with different tradeoffs. We also introduce an optimized local accumulation scheme to effectively amortize the overhead of shifting and masking the output partial sum over multiple rounds of dot-product computation. Our evaluation of ULPPACK on a Raspberry Pi 4 device with a 512×512×512 GEMM (general matrix-multiply) kernel demonstrates substantial performance gains over those production-grade low-precision linear algebra libraries with a speedup of 2.1×, 1.8×, and 2.7× for 3-bit quantization (W3A3) over Google's GEMMLOWP, Facebook's QNNPACK, and the optimized bit-serial implementation, respectively. For end-to-end evaluation using PyTorch with seven 2-bit quantized (W2A2) convolutional neural networks (CNNs) ULPPACK achieves geomean speedups of 3.9× and 1.5× over the baseline single-precision floating-point (FP32) and QNNPACK, respectively.

In summary, this paper makes the following contributions:

- We identify opportunities on commodity hardware for a single-wide ($\geq$8b) multiply to compute a dot product of two vectors with multiple narrow ($<$8b) elements.
- To capitalize on these opportunities, we propose ULPPACK, an efficient implementation of sub-8-bit GEMM (General Matrix Multiplication) computation, by leveraging efficient packing and local accumulation optimizations to maximize data parallelism while minimizing the overhead of shifting and masking operations.
- We evaluate ULPPACK on both ARM and Intel CPUs using GEMM kernels as well as end-to-end DNN models running on PyTorch to substantially outperform the production-grade low-precision GEMM libraries as well as state-of-the-art bit-serial kernels.

2 Background and Motivation

Neural Network Quantization. Quantization has attracted great interests for its effectiveness in reducing both computational intensity and memory size. To preserve accuracy even with ultra low-precision ($<$8b) quantization, novel network structures and training methods have been proposed. PROFIT (Park & Yoo, 2020) introduces training strategies for MobileNet V1/2 to achieve $<$0.5% accuracy loss with 4-bit weights and activations (W4A4). LSQ (Esser et al., 2020), DSQ (Gong et al., 2019) and QIL (Jung et al., 2019) propose to learn quantization parameters to preserve the end-to-end accuracy of a quantized model. HAQ (Wang et al., 2019) and HAWQ (Dong et al., 2019) present mixed-precision models by quantizing each layer with different bit-width. To bound the activation value range after ReLU, PACT (Choi et al., 2019; 2018a) introduces a new activation function that clamps the value of activation. All these schemes attempt to quantize DNNs using sub-8-bit integers to make them lightweight. To fully harness the performance potential of those ultra low-precision models, it is necessary to provide an efficient implementation of sub-8-bit integer operations. Several schemes (Ullrich et al., 2017; Zhang et al., 2018) exploit non-uniform quantization, where quantized weights are mapped to an irregular sequence of values. While potentially providing some accuracy gains, these approaches are computationally inefficient as they often require floating-point operations, extra memory accesses, and so on, to make them not suitable for resource-constrained mobile devices.

Low-Precision Linear Algebra Kernels. Low-precision

	W1	W2	W3	W4	W5	W6	W7
A2	0.52x	0.28x	0.19x	0.15x	0.12x	0.09x	0.08x
A1	1.0x	0.54x	0.37x	0.29x	0.23x	0.2x	0.17x

Figure 2. Performance drop of the bit-serial computation in comparison to W1A1 over different WxAy configurations.

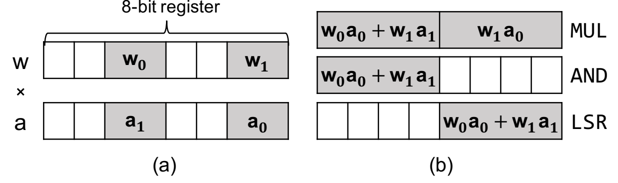


Figure 3. Example dot product by multiplying two packed registers

linear algebra kernels extend the existing wider bit-width linear algebra kernels to maximize the throughput of computing on low-precision operands. The use of lower-precision operands improves performance by enabling i) caches to fit more data and ii) lower-precision SIMD instructions to be utilized (e.g., `vmlaq_u8()` in ARMv8 ISA) to process more elements in parallel than higher-precision instructions (e.g., `vmlaq_u32()` in ARMv8 ISA). Google's GEMMLOWP and Facebook's QNNPACK represent the state-of-the-art implementing such kernels. These kernels are very effective in improving the performance of DNN inference. According to our measurements, 8-bit quantization (W8A8) using QNNPACK achieves more than 3× end-to-end speedups over the FP32 baseline on PyTorch running on a 64-bit ARM Cortex-A72 CPU (Raspberry Pi 4). However, more aggressive sub-8-bit quantization yields no further performance gains as the commodity CPU only support SIMD of 8-bit or wider. In this case, these low-precision kernels simply zero-extend those sub-8-bit operands to make them byte-aligned and treat them like 8-bit operands.

Bit-serial Computation. Bit-serial computation provides a potential solution to support efficient sub-8-bit data parallel computation. When computing a product of two operands, the bit-serial computation scheme processes each bit of the operand in series while processing multiple pairs of operands in parallel. Theoretically, it can achieve a speedup inversely proportional to the operand's bit-width.

$$\begin{aligned} & w_0[x-1:0] \cdot a_0[y-1:0] \\ &= \sum_{i=0}^{x-1}\sum_{j=0}^{y-1} popcount(w_0[i] \,\&\, a_0[j]) \cdot 2^{i+j} \end{aligned} \tag{1}$$

Equation (1) shows how the bit-serial scheme computes a product of x-bit weight (w_0) and y-bit activation (a_0). Here, $w_0[i]$ ($a_0[j]$) refers to the i-th (j-th) bit of the weight (activation) value. Equation 1 shows that the amount of computation scales linearly with x and y, which are the bit-widths of weights and activations, respectively. However, constructing K-dimensional bit-sliced binary vectors of $w_{0...(K-1)}[i]$ and $a_{0...(K-1)}[j]$ takes a considerable amount of time and the *popcount* operation also has a limited throughput. Thus, the bit-serial computation gives rapidly diminishing returns on commodity CPUs as the bit-width increases. (In Section 5.1 we demonstrate that the bit-serial computation outperforms ULPPACK only for a small number of 1- or 2-bit configurations.) Figure 2 shows the performance drop of WxAy bit-serial computation over W1A1 for the multiplication of two matrices, each having 512×512 dimensions. As shown, the performance of bit-serial rapidly degrades as bit-width increases. Bit-serial computation only improves performance over existing low-precision libraries when both activations and weights utilize very narrow bit-width.

3 ULPPACK

3.1 Overview

The key idea of ULPPACK is to compute a low-precision dot product by packing multiple sub-byte elements into a wider register and performing a regular multiply using the two packed registers. Figure 3(a) illustrates this by packing two 2-bit weights (w_0 and w_1) and two 2-bit activations (a_0 and a_1) into two 8-bit registers. (The remaining bits are set to zero.) Then an regular 8-bit multiply yields the result shown in Figure 3(b). Formally, the result can be expressed as follows:

$$\begin{aligned} & (w_0 \cdot 2^4 + w_1) \cdot (a_1 \cdot 2^4 + a_0) \\ &= w_0a_1 \cdot 2^8 + (w_0a_0 + w_1a_1) \cdot 2^4 + w_1a_0 \end{aligned} \tag{2}$$

We observe that the coefficient of 2^4 is the dot product ($w \cdot a$) of the weight vector (w=(w_1, w_0)) and the activation vector (a=(a_1, a_0)). Thus, it is possible to compute the dot product of two low-precision (2-bit) vectors by performing a single, higher-precision (8-bit) multiply and then masking and shifting the coefficient at the right position. In a RISC ISA, such as ARMv8, this process can be implemented by only three instructions: i) `MUL` for performing an integer multiply, ii) `AND` for masking out the irrelevant portion of the result, iii) `LSR` for performing a right shift for alignment. In what follows, we generalize this packing procedure for a given set of weight/activation bit-widths and derive the conditions that need to be satisfied to prevent potential overflow.

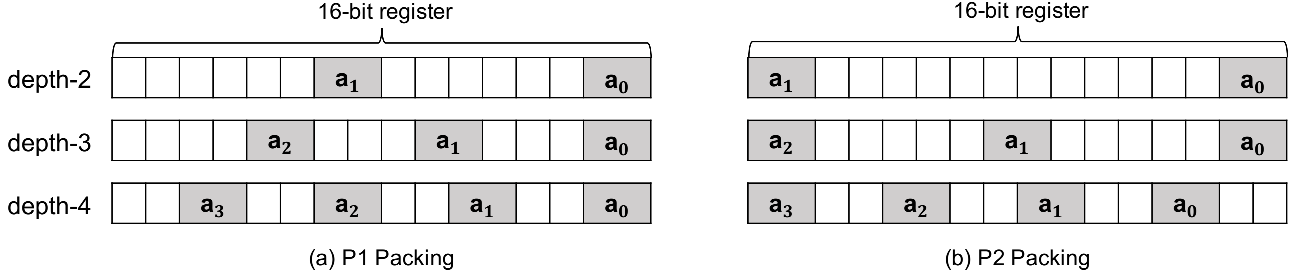


Figure 4. Packing d 2-bit integers (depth d=2, 3, 4) on a 16-bit wide register.

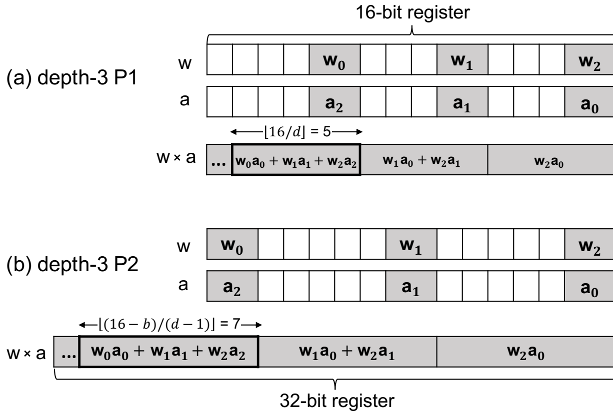


Figure 5. Multiplication between two W2A2 vectors with P1 and P2 depth-3 packing.

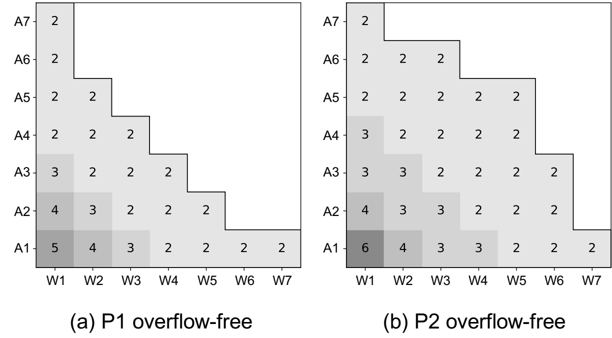


Figure 6. Overflow-free WxAy configuration for P1 and P2. The number at each configuration is the maximum depth-d. The white region outside the gray triangular region is the "overflow" region where the maximum packing depth is 1.

3.2 Packing Schemes

In this section, we introduce two packing schemes with different tradeoffs so that a network model can choose a better scheme according to its bit-widths of weights and activations (WxAy). Throughout the section, we assume 16-bit wide registers to pack multiple lower-precision operands as they work best in our setup. However, note that there is no limitation in applying both packing schemes to any other register widths, such as 8 bits and 32 bits. Finally, we define the *packing depth* as the number of low-precision operands packed in a single wide register; the packing depth of d (*depth-d*) means that d low-precision operands are packed together into a single register.

Packing Scheme 1 (P1). Assuming a 16-bit wide register, the P1 scheme divides the wide register into d uniform intervals whose width is $\lfloor 16/d \rfloor$ and allocates each of the d low-precision operands to the LSBs (Least Significant Bits) of each distinct interval. Figure 4(a) illustrates the P1 packing scheme with 2-bit low-precision elements with varying depths (d) of 2 through 4. Figure 5(a) shows the result of multiplying two 2-bit vectors of depth-3 (labeled w and a). This multiply produces a 3-dimensional dot-product, $w_0a_0 + w_1a_1 + w_2a_2$, at the bit position 10 through 14 (where the 10th bit is the LSB) as identified by the box in bold.

To avoid any overflow, the result of the dot product must be contained within the 5-bit field (note that $\lfloor 16/d \rfloor$=5 in this example). To generalize this, the dot-product result must be less than $2^{\lfloor 16/d \rfloor}$ when packing d elements into a 16-bit register. For the configuration of WxAy, where weights and activations are represented using x and y bits, respectively, the value of the d-dimensional dot-product is upper-bounded by $d \cdot (2^x - 1) \cdot (2^y - 1)$. Thus, to guarantee overflow freedom, the following condition must be satisfied: $d \cdot (2^x - 1) \cdot (2^y - 1) < 2^{\lfloor 16/d \rfloor}$. Figure 6(a) shows the maximum packing depth d for each of WxAy configurations without violating this condition for overflow freedom. For example, we can safely pack up to four operands into a single 16-bit register at both W2A1 and W1A2 configurations. However, we cannot safely pack multiple 4-bit operands at the W4A4 configuration due to overflows (i.e., maximum depth d is 1). To increase the coverage of multi-operand packing, we introduce the second packing scheme.

Packing Scheme 2 (P2). The second packing scheme enables us to increase the maximum packing depth d (i.e., the maximum number of packed operands) for a given WxAy

Algorithm 1 Naïve Dot-Product

Input: depth-d packed vector w, a and its size K
Output: K-dimensional dot-product $w \cdot a$

$sum = 0$
for $i = 0$ **to** $\frac{K}{d}$ **do**
  $sum = sum + (((w_i \cdot a_i) \ \&\ mask) \gg shift)$
end for
return sum

Algorithm 2 Optimized Dot-Product

Input: depth-d packed vector w, a of size K, and $iter$ of local accumulation
Output: K-dimensional dot-product $w \cdot a$

$sum = 0$
for $i = 0$ **to** $\frac{K}{d \cdot iter}$ **do**
  $local = 0$
  for $j = i \cdot iter$ **to** $(i+1) \cdot iter$ **do**
    $local = local + w_j \cdot a_j$ *// multiply-add*
  end for
  $local = local \ \&\ mask$
  $sum = sum + (local \gg shift)$ *// shift-add*
end for
return sum

configuration. The P2 packing scheme first reserves the b MSBs (Most Significant Bit) for one low-precision operand, where $b = max(x, y)$, and divides the remaining (16-b) bits uniformly into $(d-1)$ intervals. Figure 4(b) illustrates the P2 packing d 2-bit operands of (depth-d=2, 3, 4) into a 16-bit register. For example, unlike the P1 scheme, the P2 scheme of depth-2 places a_0 and a_1 at both ends of the register to have a 14-bit interval instead of 8-bit of P1. Figure 5(b) shows the result of computing a dot-product using P2, producing a 3-dimensional dot-product. To generalize P2 packing scheme, each of the $(d-1)$ low-precision operands are allocated with its uniform interval. Thus, the P2 packing scheme of depth-d maintains a $\lfloor(16-b)/(d-1)\rfloor$-bit interval between a pair of adjacent packed operands.

The use of a wider interval in the P2 scheme has two implications. First, as in Figure 6(b), it relaxes the overflow-free condition compared to the P1 scheme. The condition for overflow freedom is now $d \cdot (2^x - 1) \cdot (2^y - 1) < 2^{\lfloor(16-b)/(d-1)\rfloor}$. The right-hand side of this inequality is always greater than or equal to the corresponding term in the P1 scheme ($2^{\lfloor 16/d \rfloor}$). This allows us to accommodate larger values of x and y without causing an overflow. Second, in Figure 5(b), the P2 scheme places the resulting dot-product value at the bit position of $(16-b)$ through $(16-b+\lfloor(16-b)/(d-1)\rfloor)$, which does not fit in the 16-bit output register even if WxAy satisfies the inequality for overflow freedom. Therefore, it requires to use a wider output register, such as 32-bit register. This increases the register pressure, hence potentially degrading performance, in spite of an increased degree of data parallelism.

3.3 Computing Dot-Products

A single wide multiply with packed operands of depth-d in Section 3.2 computes a d-dimensional (i.e., d=2,3,4) dot-product. However, considering the shape of the layer in DNN, weight and activation vectors have a much greater dimension K such that $K \gg d$. In this section, we discuss how we can efficiently compute a K-dimensional dot-product by utilizing multiple iterations of computing d-dimensional dot-product.

Naïve Method. Algorithm 1 shows a basic method of computing a K-dimensional dot-product by accumulating the result of a d-dimensional dot-products in K/d iterations. Although conceptually simple, this method requires many extra bookkeeping instructions such as `AND`, `LSR`, and `ADD` for every iteration in addition to a wide multiply `MUL`. Thus, Algorithm 1 requires $4 \cdot K/d$ instructions to compute a K-dimensional dot-product. Even if we assume the existence of fused shift-and-add instructions, which are common in mainstream ISAs, it still requires $3 \cdot K/d$ arithmetic instructions in total to compute the K-dimensional dot-product.

Optimized Method with Local Accumulation. Alternatively, it is possible to accumulate d-dimensional dot-products in place without extracting the partial sum at every iteration using masking (`AND`) and shifting (`LSR`) instructions. We refer to this technique as *local accumulation*. For example, in Figure 5(b), the output value of a single 3-dimensional W2A2 dot-product (i.e., $w_0a_0 + w_1a_1 + w_2a_2$) is upper-bounded by $3 \cdot (2^2-1) \cdot (2^2-1) = 27$. Since a 7-bit interval is allocated to hold the output, the maximum value that can be held by the output field without causing an overflow is $2^7 - 1 = 127$. Thus, we can accumulate the d-dimensional dot-products up to four times ($= \lfloor 127/27 \rfloor$) by reusing the same output register without an overflow. With this local accumulation, we can amortize the overhead of those bookkeeping instructions such as `AND` and `LSR`, as well as `ADD` by leveraging fused-multiply-add (`FMA`) instructions.

Algorithm 2 describes a procedure of this optimized method with local accumulation. Recall that the naïve method cannot utilize `FMA` instructions, and requires $3 \cdot K/d$ instructions to compute a K-dimensional dot-product. In contrast, the optimized algorithm requires only K/d `FMA`s for accumulation plus $2 \cdot K/(d \cdot iter)$ bookkeeping instructions for extracting those partial sums, where $iter$ is the number of iterations over which we accumulate the d-dimensional dot-products without extraction (i.e., 4 in our previous example). Thus, the total number of instructions gets reduced

Algorithm 3 ULPPACK with ARM Neon intrinsic

```
1: #define depth 2
2: #define iter 4
3: #define mask vmovq_n_u16(0xFF00)
4:
5: int dp(uint16_t* w, uint16_t* a, int K) {
6:   uint16x8_t sum = vmovq_n_u16(0);
7:   for (int i=0; i<K; i+=8*depth*iter) {
8:     uint16x8_t local = vmovq_n_u16(0);
9:     for (int j=0; j<iter; j++) {
10:      uint16x8_t vecw = vld1q_u16(w);
11:      uint16x8_t veca = vld1q_u16(a);
12:      local = vmlaq_u16(local,vecw,veca);
13:      w += 8; a += 8;
14:    }
15:    local = vandq_u16(local, mask);
16:    sum = vsraq_n_u16(sum, local, 8);
17:  }
18:  return vaddvq_u16(sum); // reduced sum
19: }
```

substantially compared to the naïve method.

Consideration for Choosing d and $iter$. The width of the output field also becomes a limiting factor when we perform local accumulation. The P1 packing scheme uses $\lfloor 16/d \rfloor$ bits for accumulating the partial sums, and the P2 packing scheme $\lfloor (16-b)/(d-1) \rfloor$ bits. This can limit the number of iterations that can be run without extracting the accumulated value (i.e., *iter* in Algorithm 2). To avoid an overflow with local accumulation, the following condition must be satisfied: $iter \le \lfloor \frac{2^{\lfloor 16/d \rfloor}-1}{d\cdot(2^x-1)\cdot(2^y-1)} \rfloor$ for P1, and $iter \le \lfloor \frac{2^{\lfloor (16-b)/(d-1) \rfloor}-1}{d\cdot(2^x-1)\cdot(2^y-1)} \rfloor$ for P2. This complicates the problem of choosing the packing depth d, as just selecting the maximum d to maximize data parallelism is no longer the most efficient setting. The choice of a larger d leaves less space for larger *iter* while the choice of a smaller d enables the choice of relatively larger *iter*. We explore the best choice of d and $iter$ for a given WxAy in Section 4.2.

3.4 Implementation on ARMv8 ISA

Implementation. We modify and extend the matrix multiplication kernel of QNNPACK (Dukhan et al., 2018) to implement ULPPACK. Algorithm 3 shows a simplified C implementation of dot-product in ULPPACK on ARMv8 ISA for the configuration of $d = 2, iter = 4$. Before the `dp` function is called, the two input vectors (`w` and `a`) are properly packed with two low-precision operands. Using 128-bit Neon SIMD registers, a total of eight 16-bit lanes in `uint16x8_t` will be processed in parallel. A single SIMD lane accumulates d-dimensional dot-product for *iter* times (Line 8-14), extracts the output partial sum (Line 15-16), and repeat this process for `K/(8*depth*iter)` times to obtain output partial sums across the eight SIMD lanes. Finally, a reduction across the lanes is performed (Line 18) to compute the final outcome. In practice, we unroll the loop j (Line 9-14) and carefully optimize data movement for the kernel with cache blocking.

Table 2. Microarchitectural specifications of Raspberry Pi 3B+/4

	Raspberry Pi 3B+	**Raspberry Pi 4**
CPU	ARM Cortex-A53 4 cores, 1.4GHz	ARM Cortex-A72 4 cores, 1.5GHz
Feature	Dual Issue In-order Execution Pipeline Depth 8	Dual Issue Out-of-order Execution Pipeline Depth 15
Cache	L1 16KB / L2 512KB	L1 32KB / L2 1MB
Memory	1GB LPDDR2 SDRAM	8GB LPDDR4 SDRAM
ISA	ARM v8.0-A (32 / 64bit)	

Register Width. In this work we mainly utilize 16-bit registers for ULPPACK. However, ULPPACK is generally applicable to any register width. Using 32-bit registers is unlikely to yield higher speedups since it reduces the number of SIMD lanes by half. On the other hand, utilizing an even narrower register (e.g., 8-bit in Figure 3) significantly limits the range of x and y in the WxAy configuration and thus only suitable for extremely low-precision networks (e.g., $x + y \le 4$).

Application to Other Architectures. We have prototyped the GEMM kernel of ULPPACK on ARMv8 architecture as it is the most popular mobile ISA for running neural network inference. However, ULPPACK can easily be ported to any other architectures that provide an unsigned integer multiply instruction. Furthermore, higher performance is expected for the ISAs that support integer fused multiply-add (FMA) instructions. Such potential target platforms include ARM Cortex-M class microcontrollers, Intel CPUs, and NVIDIA GPUs. We demonstrate that ULPPACK also performs well on Intel CPU in Section 5.3. In contrast, bit-serial computation (Section 2) requires hardware support for `POPCOUNT` instruction, and thus performs poorly on a platform not supporting it such as ARM Cortex-M.

4 METHODOLOGY

4.1 Experimental Setup

Table 2 summarizes the specifications of the two hardware platforms used for evaluation. Raspberry Pi 4 is our default platform and Raspberry Pi 3B+ is also used for sensitivity study when evaluating GEMM kernels (Section 5.1). We run Ubuntu 64-bit 20.04 LTS OS on these platforms and GNU gcc/g++ version 9.3.0 with "-O3 -march=native" flag for compilation. We use both general matrix-multiply (GEMM)

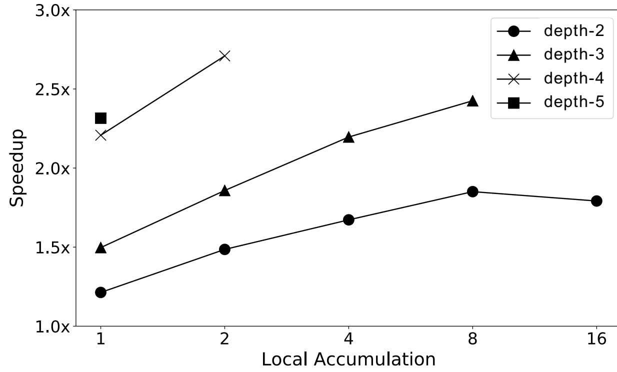


Figure 7. Speedup of P1-packed ULPPACK over GEMMLOWP for different depth and degree of local accumulation on W1A1.

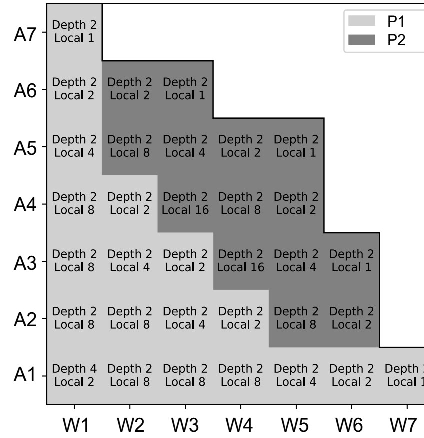


Figure 8. Optimal implementation among trade-offs while light-gray packed by P1 and dark-gray by P2. The white region outside the gray triangular region is where ULPPACK does not guarantee overflow freedom.

kernels with various dimensions and end-to-end networks, which run on PyTorch taken from the master branch (Commit de8c888). We compare ULPPACK with the following three existing systems representing the state-of-the-art in ultra low-precision inference.

- **GEMMLOWP** has been integrated into TensorFlow-Lite for quantized operation. The core of its kernels is carefully hand-written in assembly.[1]
- **QNNPACK** has been integrated into PyTorch for ARM devices, and also hand-written in assembly.[2]
- **Bit-serial** speeds up low-precision arithmetic, whose performance is inversely proportional to the bit-width. We use two state-of-the-art implementations (Umuroglu & Jahre, 2017; Cowan et al., 2020) and take the best of the two for every configuration.[3] [4]
- **ULPPACK** extends QNNPACK to implement our techniques, and is integrated into PyTorch for end-to-end evaluation.

4.2 Configuring ULPPACK

Section 3.2 introduces two packing schemes for ULPPACK, and Section 3.3 discusses the choice of both packing depth (d) and the degree of local accumulation ($iter$) can have a significant impact on performance. We first explore the impact of the choice of d and $iter$ on matrix multiplication performance. Figure 7 shows the normalized speedup of the P1-packed ULPPACK over GEMMLOWP across the varying depth and degree of local accumulation for the W1A1 case. The figure shows that increasing $iter$ gives diminishing returns, which can even lead to performance degradation due to the increased code size after the loop unrolling. While a larger $iter$ further amortizes the performance degradation from extra instructions for extraction (i.e., `AND`, `LSR`, `ADD`), once it reaches a certain level (say, 8), the performance overhead of those bookkeeping instructions is already very low. Furthermore, a large $iter$ limits the size of depth d. In the end, the optimal configuration achieving the best speedup turns out to be $d = 4, iter = 2$.

We repeat similar experiments for all combinations of WxAy and report the best configuration for each combination of (x, y) on Figure 8. Overall, the P1 packing scheme performs better than the P2 packing scheme for most configurations where the P1 packing scheme can be applied. This is because the P2 packing scheme utilizes a wider output register, which leads to an increased register pressure and potential extra memory accesses. Still, for configurations where P1 is not supported due to its tighter overflow-free condition, P2 can be utilized. Another thing to note is that most configurations prefer $d = 2$. This is because a choice of larger d leaves no opportunity for local accumulations, and thus such configurations' performance suffers from overheads incurred by extra instructions for extracting the dot-products.

5 RESULTS

5.1 Matrix Multiplication Kernels

We compare the performance of the four schemes using GEMM kernels first. Our measured execution time for ULPPACK includes the overhead of packing activations at runtime but not weights which can be preprocessed offline before an inference task begins.

[1] https://github.com/google/gemmlowp
[2] https://github.com/pytorch/QNNPACK
[3] https://github.com/cowanmeg/cgo-artifact-2020
[4] https://github.com/maltanar/gemmbitserial

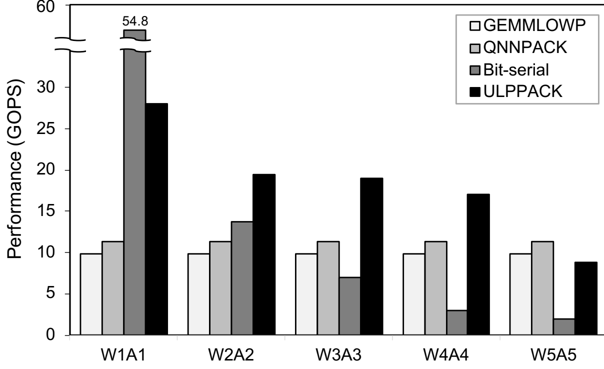
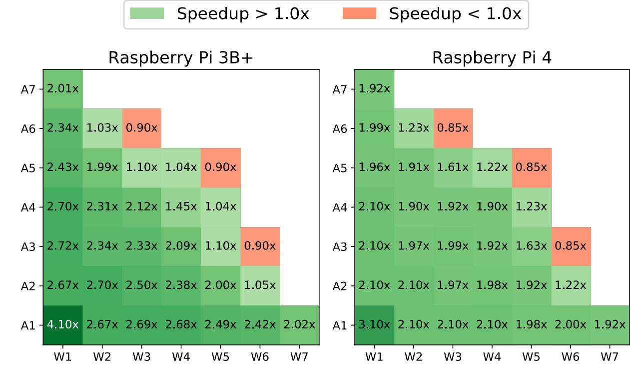


Figure 9. Throughput of 512×512×512 matrix multiplication across varying bit-widths.

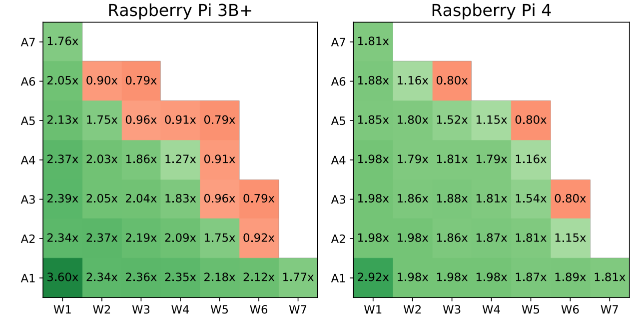
Overall Performance. Figure 9 shows the performance in Giga-Operations Per Second (GOPS) of the four systems executing a multiply of two 512 $\times$ 512 matrices with varying WxAy configurations. As expected, the performance of both GEMMLOWP and QNNPACK remains same even if weights and activations use sub-8-bit representations. They simply zero-extend low-precision numbers, and internally treat them as 8-bit. On the other hand, bit-serial computation shows significant performance improvements on extremely low-precision cases such as W1A1. However, their performance drops quickly as the bit-width of weights and activations increases. Specifically, its performance degrades linearly to an increase in $x \cdot y$. Finally, ULPPACK exhibits substantial speedups over the other three systems over a wide range of configurations except for W5A5.

Pairwise Comparison. To demonstrate the effectiveness of ULPPACK, we report the normalized speedups of ULPPACK over each of the three existing schemes with an extensive set of WxAy configurations on two different architectures (i.e., Raspberry Pi 3B+ with Cortex-A53 in-order CPU and Raspberry Pi 4 with Cortex-A72 out-of-order CPU as summarized in Table 2). A matrix multiplication of a 512 $\times$ 2048 matrix and a 2048 $\times$ 512 matrix is performed. We increase the matrix dimension to give an advantage to the bit-serial scheme while the throughput of other three schemes are not much affected. Figure 10 reports the result for this experiment. Each tile represents a specific WxAy configuration, and the green colored cell indicates that ULPPACK performs better. Further, the number of the cell represents the speedup of the ULPPACK over the baseline system being compared.

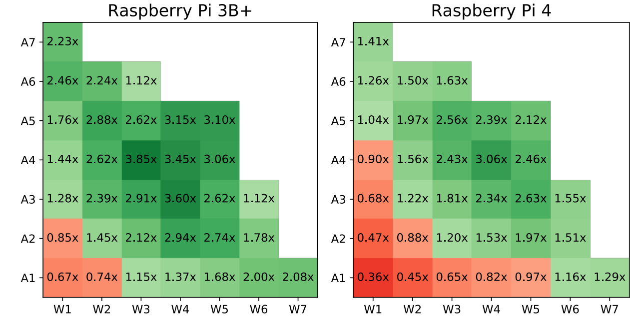
As shown in Figure 10(a) and (b), ULPPACK outperforms existing low-precision libraries in most configurations. It is not performant for W3A6, W5A5, and W6A3 because local accumulation is not possible in those configurations. Comparing the two platforms (Raspberry Pi 3B+ and 4),



(a) Speedups over GEMMLOWP



(b) Speedups over QNNPACK



(c) Speedups over Bit-serial

Figure 10. Normalized speedups of ULPPACK on all WxAy configurations for which ULPPACK is applicable. ULPPACK performs better in green cells. The number in the cell reports the speedup. The omitted region shows the configurations where ULPPACK cannot be used due to overflow.

GEMMLOWP and QNNPACK benefit more from out-of-order processor than ULPPACK does because they are implemented in hand-written assembly which extremely well exploits the instruction level parallelism in the limited size of the reorder buffer. However, we find that ULPPACK performs better at some configurations, such as W3A5, also benefiting from the processor's powerful microarchitecture.

In Figure 10(c), Bit-serial is the most effective in cases with the extremely low weight and activation bit-widths. One thing to note is that ULPPACK performs the best in regions

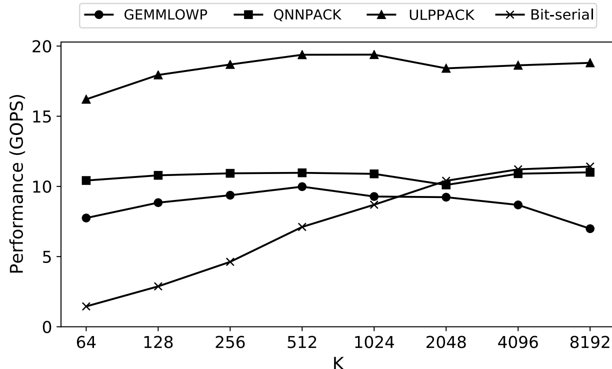


Figure 11. Sensitivity to matrix dimension K while fixing $M = N = 512$ in W3A3.

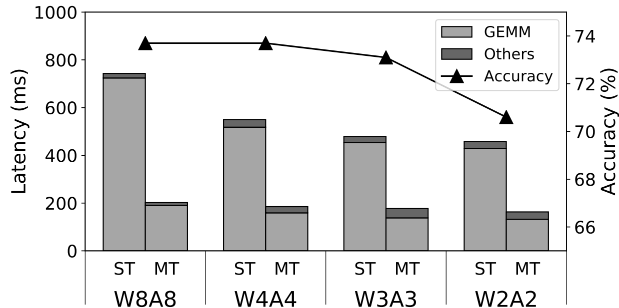


Figure 12. Runtime breakdown and accuracy of ResNet34 across varying WxAy configurations. ST and MT stand for Single-Threaded and Multi-Threaded execution, respectively. For accuracy we report the Top-1 accuracy from (Jung et al., 2019).

Table 3. End-to-end performance (images/sec) of various image classification models on PyTorch. Speedup/FP32 indicates a geomean speedup over FP32, and Speedup/W8A8 a geomean speedup over QNNPACK (W8A8). We used a batch size of 1.

Model	FP32	W8A8	W4A4	W3A3	W2A2
ResNet18	1.7	2.6	3.4	3.9	4.0
ResNet34	0.9	1.3	1.8	2.1	2.2
ResNet50	0.7	1.2	1.6	1.8	1.9
InceptionV3	1.0	1.6	2.2	2.5	2.6
GoogleNet	1.8	2.8	3.5	3.9	4.0
ShuffleNetV2	2.2	14.3	14.9	15.9	16.4
MobileNetV2	0.8	11.1	12.8	13.7	14.3
Speedup/FP32	1.0×	2.7×	3.3×	3.7×	3.9×
Speedup/W8A8	-	1.0×	1.2×	1.4×	1.5×

where the accuracy degradation is minimal (e.g., W3A3 and W4A4 as shown in Table 1). Bit-serial also benefits from the more powerful processor having the better `POPCOUNT` instruction throughput in multiple parallel execution pipelines.

Sensitivity to Matrix Size. The dimension of the matrix affects the efficiency of the cache blocking in matrix multiplication. Specifically, when multiplying a matrix whose size is $M \times K$ and the other matrix whose size is $K \times N$, the size of K is mostly relevant to the performance since it determines the vector dimension of the dot product operation. Figure 11 illustrates the performance of each method on W3A3 setting for varying Ks, where M and N are fixed to 512. Most schemes are not very sensitive to the choice of K; however, the bit-serial scheme tends to get better performance with the larger K since bit-sliced vector allows tighter data layout which leads better data reuse on larger K. Still, its performance plateaus at $K = 4096$, leaving a significant gap to ULPPACK.

5.2 End-to-end Network Performance

Table 3 reports the end-to-end neural network inference throughput with ULPPACK for seven convolutional neural networks (CNNs) across varying weight and activation bit-widths. Following the conventions in previous works on quantization schemes (Choi et al., 2018a; Jung et al., 2019), we did not apply the quantization for the first and the last layer. Also, like a previous work (Jain et al., 2020), we measure the numbers in a single-thread environment. In the table, FP32 baseline throughput represents the baseline PyTorch performance, and W8A8 throughput represents the performance of PyTorch with the QNNPACK backend. The next three following columns (W4A4, W3A3, W2A2) represent the ULPPACK throughput. The results show that low-precision models (e.g., W2A2) running on ULPPACK report substantial speedups over both the FP32 baseline and QNNPACK (W8A8). ULPPACK achieves higher speedups over both the baseline (FP32) and QNNPACK (W8A8) on all models. Specifically, ResNet, InceptionV3, and GoogleNet achieved more speedup than others as they heavily utilize GEMM in their end-to-end inference. Finally, the relatively lower speedup from W3A3 to W2A2 is mostly because ULPPACK utilizes the same depth (d) for both configurations, albeit with different degrees of local accumulation ($iter$) (see Figure 8).

Figure 12 shows runtime breakdown for ResNet34 on ULPPACK. The figure shows that the matrix multiplication kernel accounts for most of the inference time. One thing to note is that the portion of time spent on packing and other types of layers become relatively larger as the bit-width decreases. Still, even in the W2A2 case, this portion is limited to about 15%. The figure also shows the model accuracy across varying bit-widths. This particular model can maintain the original accuracy at W4A4. This indicates that ULPPACK can achieve a 2.0× speedup over FP32 and a 1.4× speedup over W8A8 without accuracy loss.

5.3 Results on x86 Architecture

To demonstrate the performance portability of ULPPACK across different ISAs, we have ported ULPPACK to the

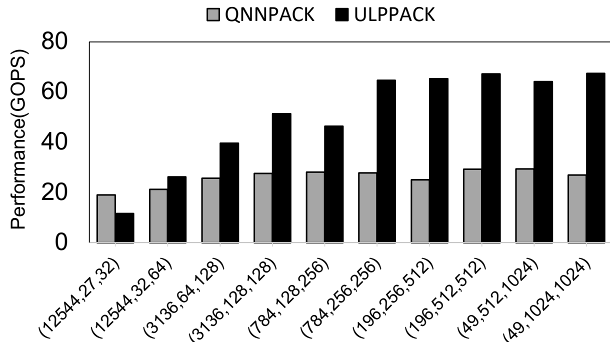


Figure 13. Throughout of each layer in MobileNetV1 at W2A2 on Intel CPU. The shape of each layer is characterized by a tuple of (M, K, N). The geomean speedup of ULPPACK is 1.77×.

x86 architecture. We have conducted experiments on a 6-core Intel i9-8950HK CPU with a 12MB L3 cache and 32GB memory running at 2.90GHz. Figure 13 shows the throughput of each layer in MobileNetV1. The x axis shows each layer characterized by a 3-tuple of (M, K, N), where $(M, K) \times (K, N)$ GEMM is performed. Except for the first layer where the overhead of bookkeeping instructions is higher than the benefit of multi-packed dot-product due to small K, ULPPACK outperforms the QNNPACK on the Intel CPU as well. Overall, ULPPACK achieves a 1.77× geomean speedup over QNNPACK.

6 Related Work

Software Support for Quantized Networks. In order to accelerate quantized networks on mobile devices, most DNN frameworks have integrated low-precision SIMD libraries (Jacob et al., 2017; Dukhan et al., 2018) targeting specific ARM backends. Similar effort has been made for deep learning compilers. QNN (Jain et al., 2020) proposes a novel graph-level optimizer for TVM to generate efficient INT8 kernels for various hardware backends. For sub-8-bit quantized networks, bit-serial computation has been explored to flexibly support multiple narrow bit-widths. Umuroglu and Jahre (Umuroglu & Jahre, 2017) suggest that matrix multiplication can be efficiently implemented via bit-serial computation on mobile devices. Cowan et al. (Cowan et al., 2020) generate an efficient bit-serial kernel using a *sketching* (Solar-Lezama et al., 2005) synthesis tool. However, their automatically generated kernels can cover only a limited range of WxAy's (e.g. $x + y \leq 4$). The same group of researchers (Fromm et al., 2020) find that the performance gain of the bit-serial convolutional layers makes the other layers, such as batch normalization, the new bottleneck. Thus, they introduce an effective quantization scheme for those layers to address the issue. Binarized Neural Network(BNN) is an extreme case of quantization, which can defined as W1A1 quantization. In fact, previous works for BNN, such as Larq (Bannink et al., 2021), utilizes the bit-serial scheme (explained in Section 2). The bit-serial scheme is indeed better than ULPPACK on W1A1; however, the potential issue with such extremely low-bitwidth quantization is that its accuracy degradation is often be substantial (Bethge et al., 2019).

Hardware Support for Quantized Networks. There are a number of specialized hardware accelerators proposed to adaptively support a range of precisions. Some of those architectures (Judd et al., 2016; Delmas et al., 2017; Sharify et al., 2018; Eckert et al., 2018) are based on bit-serial computation, targeting ASIC implementation. Other proposals do not exploit bit-serial computation to provide flexible precision. For example, Bitfusion (Sharma et al., 2018) arranges an array of 2-bit multipliers inside, and then dynamically constructs any-precision multipliers by composing multiple 2-bit multipliers. Bitfusion demonstrates substantial performance gains compared to Stripes (Judd et al., 2016), a bit-serial based accelerator. For FPGA, Xilinx DSP48E2 (Fu et al., 2017) optimizes low-precision operations by concurrently running two INT8 MACs while sharing the same weights in a single DSP slice. For CPU, TF-Net (Yu et al., 2019) also proposes the idea of multi-operand packing, but their technique requires an ISA extension and cannot be readily applied to the commodity hardware. Specifically, they propose a new instruction called `MSA` which computes `MUL-LSR-ADD` sequence at once. They also suggest a limited version of the P1 packing scheme in ULPPACK, in which `MSA` can only support WxAy configurations with $x + y \leq 6$. Unlike TF-Net, ULPPACK can support even wider WxAy configurations without requiring any hardware extension.

7 Conclusion

We propose ULPPACK, a software-only technique to accelerate ultra low-precision networks on mobile devices by packing multiple elements into a single wide register. Performing a single wide multiply utilizes multiple narrow multiply-accumulate (MAC) operations at once to compute a dot-product. We introduce two effective packing schemes with different trade-offs and analyze the overflow-free condition for each. Calculating the dot-product with local accumulation further optimizes ULPPACK by amortizing the overhead of extracting partial output. Our evaluation shows that ULPPACK beats the existing approaches on most of WxAy configurations for GEMM computation. Furthermore, for 2-bit quantized (W2A2) networks, PyTorch with ULPPACK achieves geomean speedups of 3.9× and 1.5× over the baseline FP32 model and QNNPACK (W8A8) model, respectively.

ACKNOWLEDGMENTS

This work was supported by Samsung Advanced Institute of Technology (SAIT) and Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (2021-0-00105, Development of Model Compression Framework for Scalable On-device AI Computing on Edge Applications).

REFERENCES

Bannink, T., Hillier, A., Geiger, L., de Bruin, T., Overweel, L., Neeven, J., and Helwegen, K. Larq compute engine: Design, benchmark and deploy state-of-the-art binarized neural networks. *Proceedings of Machine Learning and Systems*, 3, 2021.

Bethge, J., Yang, H., Bornstein, M., and Meinel, C. Binarydensenet: Developing an architecture for binary neural networks. In *2019 ICCV Workshop*, pp. 1951–1960, 2019. doi: 10.1109/ICCVW.2019.00244.

Choi, J., Wang, Z., Venkataramani, S., Chuang, P. I., Srinivasan, V., and Gopalakrishnan, K. PACT: Parameterized clipping activation for quantized neural networks. *CoRR*, abs/1805.06085, 2018a. URL http://arxiv.org/abs/1805.06085.

Choi, J., Venkataramani, S., Srinivasan, V., Gopalakrishnan, K., Wang, Z., and Chuang, P. Accurate and efficient 2-bit quantized neural networks. In *Proceedings of the 2nd SysML Conference*, 2019.

Choi, Y., El-Khamy, M., and Lee, J. Learning low precision deep neural networks through regularization. *CoRR*, abs/1809.00095, 2018b. URL http://arxiv.org/abs/1809.00095.

Cowan, M., Moreau, T., Chen, T., Bornholt, J., and Ceze, L. Automatic generation of high-performance quantized machine learning kernels. In *CGO '20: 18th ACM/IEEE International Symposium on Code Generation and Optimization*, 2020.

Delmas, A., Sharify, S., Judd, P., and Moshovos, A. Tartan: Accelerating fully-connected and convolutional layers in deep learning networks by exploiting numerical precision variability. *arXiv preprint arXiv:1707.09068*, 2017.

Dong, Z., Yao, Z., Gholami, A., Mahoney, M. W., and Keutzer, K. HAWQ: hessian aware quantization of neural networks with mixed-precision. In *2019 IEEE/CVF International Conference on Computer Vision, ICCV*, 2019.

Dukhan, M., Wu, Y., and Lu, H. Qnnpack: Open source library for optimized mobile deep learning, 2018. URL https://engineering.fb.com/ml-applications/qnnpack/.

Eckert, C., Wang, X., Wang, J., Subramaniyan, A., Iyer, R., Sylvester, D., Blaaauw, D., and Das, R. Neural cache: Bit-serial in-cache acceleration of deep neural networks. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018.

Esser, S. K., McKinstry, J. L., Bablani, D., Appuswamy, R., and Modha, D. S. Learned step size quantization. In *8th International Conference on Learning Representations, ICLR 2020*, 2020.

Fromm, J., Cowan, M., Philipose, M., Ceze, L., and Patel, S. N. Riptide: Fast end-to-end binarized neural networks. In *Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020*, 2020.

Fu, Y., Wu, E., Santhaseelan, V., Denolf, K., Khan, K., and Kathail, V. Embedded vision with int8 optimization on xilinx devices. *White Paper WP490*, 2017.

Gong, R., Liu, X., Jiang, S., Li, T., Hu, P., Lin, J., Yu, F., and Yan, J. Differentiable soft quantization: Bridging full-precision and low-bit neural networks. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2019.

Jacob, B., Warden, P., and Guney, M. E. GEMMLOWP: a small self-contained low-precision gemm library, 2017. URL https://github.com/google/gemmlowp.

Jain, A., Bhattacharya, S., Masuda, M., Sharma, V., and Wang, Y. Efficient execution of quantized deep learning models: A compiler approach. *arXiv preprint arXiv:2006.10226*, 2020.

Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., Boyle, R., Cantin, P., Chao, C., Clark, C., Coriell, J., Daley, M., Dau, M., Dean, J., Gelb, B., Ghaemmaghami, T. V., Gottipati, R., Gulland, W., Hagmann, R., Ho, C. R., Hogberg, D., Hu, J., Hundt, R., Hurt, D., Ibarz, J., Jaffey, A., Jaworski, A., Kaplan, A., Khaitan, H., Killebrew, D., Koch, A., Kumar, N., Lacy, S., Laudon, J., Law, J., Le, D., Leary, C., Liu, Z., Lucke, K., Lundin, A., MacKean, G., Maggiore, A., Mahony, M., Miller, K., Nagarajan, R., Narayanaswami, R., Ni, R., Nix, K., Norrie, T., Omernick, M., Penukonda, N., Phelps, A., Ross, J., Ross, M., Salek, A., Samadiani, E., Severn, C., Sizikov, G., Snelham, M., Souter, J., Steinberg, D., Swing, A., Tan, M., Thorson, G., Tian, B., Toma, H., Tuttle, E., Vasudevan, V., Walter, R., Wang, W., Wilcox, E., and Yoon, D. H. In-datacenter performance analysis of a tensor processing unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017.

Judd, P., Albericio, J., Hetherington, T., Aamodt, T. M., and Moshovos, A. Stripes: Bit-serial deep neural network

computing. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

Jung, S., Son, C., Lee, S., Son, J., Han, J., Kwak, Y., Hwang, S. J., and Choi, C. Learning to quantize deep networks by optimizing quantization intervals with task loss. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*, 2019.

Park, E. and Yoo, S. Profit: A novel training method for sub-4-bit mobilenet models. In *Proceedings of the European conference on computer vision (ECCV)*, 2020.

Park, J., Khudia, D., and Huang, J. Fbgemm, 2018. URL https://github.com/pytorch/FBGEMM.

Sharify, S., Lascorz, A. D., Siu, K., Judd, P., and Moshovos, A. Loom: Exploiting weight and activation precisions to accelerate convolutional neural networks. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, 2018.

Sharma, H., Park, J., Suda, N., Lai, L., Chau, B., Chandra, V., and Esmaeilzadeh, H. Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018.

Solar-Lezama, A., Rabbah, R., Bodík, R., and Ebcioğlu, K. Programming by sketching for bit-streaming programs. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, 2005.

Ullrich, K., Meeds, E., and Welling, M. Soft weight-sharing for neural network compression. In *ICLR (Poster)*, 2017.

Umuroglu, Y. and Jahre, M. Towards efficient quantized neural network inference on mobile devices: Work-in-progress. In *Proceedings of the 2017 International Conference on Compilers, Architectures and Synthesis for Embedded Systems Companion*, CASES '17, 2017.

Wang, K., Liu, Z., Lin, Y., Lin, J., and Han, S. Haq: Hardware-aware automated quantization with mixed precision. *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

Wu, C., Brooks, D., Chen, K., Chen, D., Choudhury, S., Dukhan, M., Hazelwood, K., Isaac, E., Jia, Y., Jia, B., Leyvand, T., Lu, H., Lu, Y., Qiao, L., Reagen, B., Spisak, J., Sun, F., Tulloch, A., Vajda, P., Wang, X., Wang, Y., Wasti, B., Wu, Y., Xian, R., Yoo, S., and Zhang, P. Machine learning at facebook: Understanding inference at the edge. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019.

Yu, J., Lukefahr, A., Das, R., and Mahlke, S. Tf-net: Deploying sub-byte deep neural networks on microcontrollers. *ACM Trans. Embed. Comput. Syst.*, 2019.

Zhang, D., Yang, J., Ye, D., and Hua, G. Lq-nets: Learned quantization for highly accurate and compact deep neural networks. In *Proceedings of the European conference on computer vision (ECCV)*, pp. 365–382, 2018.

Zhou, S., Ni, Z., Zhou, X., Wen, H., Wu, Y., and Zou, Y. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *CoRR*, abs/1606.06160, 2016.