Demystifying Power-of-Two Quantization: Benchmarking Inference on AVX and RVV

Saleh Jamali Golzar[1,*], Giuseppe Pagano[1] and Biagio Cosenza[1]

[1]University of Salerno, Italy

Abstract

Recent trends in deep learning models have been characterized by a rapid increase in the number of parameters to achieve higher performance across a wide range of tasks. However, this growth in model size has intensified computational demands, leading to increased power consumption and latency during inference, making model compression more important than ever. As a result, it has become extremely important to provide and use efficient compression techniques, such as quantization, on different target architectures and platforms. This work revisits power-of-two (PoT) quantization (specifically the MatMul kernel) for inference workloads, evaluating a wide range of configurations, including fixed- and floating-point PoT, and targeting AVX512 and RVV-1.0. Our work proposes techniques tailored to the underlying architecture, including two methods for unpacking 8- and 16-bit PoT-encoded data for AVX512, two packing configurations for RVV-1.0, as well as a novel, lightweight solution for handling signed infinity and NaN floating-point values properly. Experimental results for floating-point PoT quantization for MatMul workloads show speedups of up to 3.67.

Keywords

AVX512, Deep Neural Networks, Quantization, RVV

1. Introduction

The rise of deep learning is marked by an incredible increase in the number of trainable parameters in models. Rapid scaling of these models in size allowed for their groundbreaking results, but also hindered their deployment in resource-constrained devices. Nowadays, with the growing disparity between processors' speed and memory bandwidth/latency, referred to as the *Memory Wall* [1], and the sheer growth of the AI workloads in size [2], minimizing memory footprints, memory bandwidth requirements, and data movements could be considered one of the most relevant forms of optimizations. Quantization, as one of the prominent methods for model compression, plays a vital role in a wide spectrum of applications ranging from large language models running on HPC machines to tiny models running on microcontrollers [3]. Bitwise shifts are commonly employed in integer arithmetic as a faster alternative to multiplication. This technique was recently used to implement a faster non-uniform quantization scheme [4]. *DenseShift* [5] is a seminal work in this field that introduces a novel power-of-two quantization scheme based on reparameterization. It enhances dynamic range and enables seamless integration with floating-point arithmetic. Unfortunately, *DenseShift* does not provide the code and only offers an evaluation limited to ARMv7 Neon. This paper presents an open-source implementation[1] and a detailed analysis of DenseShift's PoT quantization, targeting AVX and RVV ISAs and focusing on the MatMul inference kernel (in fact, the MatMul kernel accounts for the largest portion of the inference runtime: e.g., 77.85% [6] for DGCNN [7]). In summary, the contributions of this paper are:

❶ The first implementation of PoT-quantized MatMul inference kernel for **RISC-V RVV-1.0** , supporting 4-bit packed and 8-bit quantization, fixed and floating point, and different packing strategies;

❷ The first implementation of PoT-quantized MatMul inference kernel for **AVX512**, supporting 8-bit quantization, fixed and floating point with different unpacking strategies;

ITADATA-WS 2025: The 4th Italian Conference on Big Data and Data Science – Workshops, September 9–11, 2025, Turin, Italy

*Corresponding author.

sjamaligolzar@unisa.it (S. Jamali Golzar)

0009-0003-1905-1537 (S. Jamali Golzar); 0000-0002-9738-2150 (G. Pagano); 0000-0002-8869-6705 (B. Cosenza)

[1]Available at https://github.com/unisa-hpc/QPOT-VEC

❸ A novel approach to **handle infinity and *NaN* floating-point values** for PoT with an acceptable overhead;

❹ An experimental evaluation of the PoT implementations for RVV-1.0 on SpacemiT K1, for AVX512 on Intel Xeon 5218, Xeon 8260 and AMD Ryzen 9 7950X, all evaluated with autotuning, different input sizes and compilers.

2. Background on Quantization

Quantization reduces the precision of model parameters to improve latency and memory usage, trading off an acceptable accuracy drop. Depending on its scale and offset, it can be symmetric/asymmetric and uniform/non-uniform [8]. Post Training Quantization (PTQ) applies quantization after training with optional fine-tuning, while Quantization Aware Training (QAT) integrates it directly into training for better accuracy under the same constraints. The Power-of-Two quantization offers higher dynamic range and more efficient implementation compared to fixed-point counterparts. This work focuses on QAT PoT quantization.

2.1. Fixed-point PoT

DeepShift [4] offers two schemes to train PoT-quantized models, namely *DeepShift-Q* and *DeepShift-PS*, which use the same PoT encoding for the forward-pass but various approaches for the backward-pass. It uses a shift matrix $\tilde{P}$ along with a sign matrix $\tilde{S}$ to replace a floating-point weight matrix of W with $\mathrm{Flip}(2^{\tilde{P}}, \tilde{S})$ [4]. Furthermore, since the sign of the elements of $\tilde{P}$ only indicates left or right shifts, the function $\mathrm{Flip}(x, \tilde{s}) = x\,\mathrm{sign}(\tilde{s})$ handles the negation of x. This work maps full-precision values to $\{0\} \cup \{\pm 2^{p_1}\},\ p_1 \in \mathbb{W}$.

S^3 [9] aims to solve the gradient vanishing [10] and weight sign freezing problems when using re-parametrization to train models with discrete power-of-two weights. Similar to *DeepShift*, it maps full-precision values to $\{0\} \cup \{\pm 2^{p_2}\},\ p_2 \in \mathbb{W}$. The weights are re-parameterized by Eq. 1 where S_t is defined in Eq. 2, T is the target quantization bitwidth, and $\mathbb{1}(x) = +1$ if $(x \geq 0)$ else 0.

$$w_{\text{shift}} = \underbrace{\mathbb{1}(w_{\text{sparse}})}_{\text{zero}} * \underbrace{\left\{2\mathbb{1}(w_{\text{sign}}) - 1\right\}}_{\text{Sign}} \underbrace{2^{S_T}}_{\text{Scale}} \tag{1}$$

$$S_t = \mathbb{1}(w_t)(S_{t-1} + 1) \quad 1 \leq t \leq T, \quad S_0 = 0 \tag{2}$$

2.2. Floating-point PoT

DenseShift[5] on the other hand improves the accuracy by removing zero from the dynamic range, effectively mapping full-precision values to $\{\pm 2^{p_3}\}$ instead of $\{0\} \cup \{\pm 2^{p_4}\},\ p_3, p_4 \in \mathbb{W}$. Compared to S^3 [9], this work uses the same recursion formula for S_t, while using Eq. 3 for computing quantized weights. *DenseShift* [4] reports a 1.6x speedup on an ARM A57 Neon.

$$w_{\text{shift}} = \underbrace{\left\{2\mathbb{1}(w_{\text{sign}}) - 1\right\}}_{\text{Sign}} \underbrace{2^{S_T}}_{\text{Scale}} \tag{3}$$

For clarification, S_T for $T = 3$ is computed using the Eq. 2 recursively and simplified to Eq. 4. The range of S_3 is $\{0, 1, 2, 3\}$, leading to quantization levels of $\{\pm 1, \pm 2, \pm 4, \pm 8\}$ for w_{shift} (Eq. 5). Every floating point weight in the original model is replaced by four unique floating point trainable parameters: w_{sign} and w_1, w_2, w_3 for $T = 3$. As a comparison, for T=3, S^3 [9] yields a range of $\{0, \pm 1, \pm 2, \pm 4\}$.

$$S_3 = \mathbb{1}(w_3)\mathbb{1}(w_2)\mathbb{1}(w_1) + \mathbb{1}(w_3)\mathbb{1}(w_2) + \mathbb{1}(w_3) \tag{4}$$

$$S_3 \in \{0, 1, 2, 3\} \implies w_{\text{shift}} \in \{\pm 1, \pm 2, \pm 4, \pm 8\} \tag{5}$$

DenseShift requires only two bits to store the **exponents** of the quantized levels ($\{\pm1, \pm2, \pm4, \pm8\}$), along with one bit for their signs, adding up to three bits ($T = 3$). It exploits the floating-point format to replace multiplications of floating point numbers by PoT values, with unsigned integer additions (Fig. 1). **Our work investigates the conventional fixed-point PoT briefly, along with a deeper study of the floating-point PoT.** Building on top of *DenseShift*, we explore how the sign and exponent bits can be efficiently encoded and packed for modern processors with SIMD capabilities, targeting various values for T. In addition, we introduce strategies to correctly handle special floating-point values like *NaN* and $\pm\infty$.

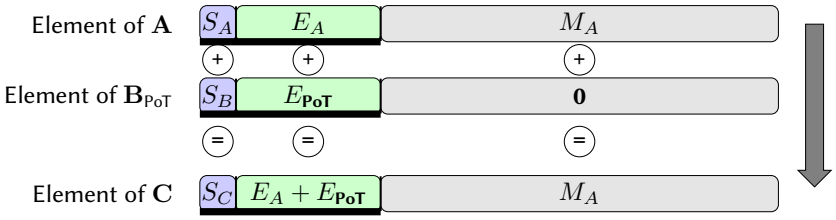

Figure 1: Replacing a multiplication of a `Float32` by a PoT value with an unsigned addition.

2.3. Other Work

ShiftAddViT [11] and *P*^{2}*-ViT* [12] target PoT quantization for Vision Transformers. Similar studies [13, 14, 15, 16, 17, 18, 19] propose specialized RISC-V extensions, FPGA-, and ASIC-based designs related to PoT and ternary models. Other works, such as *SIMDE* [20] aim for portability of SIMD kernels across different ISAs. Unfortunately, the support for the needed AVX512 extensions for our use case was missing in *SIMDE*. In addition, *APoT* [21] and the related literature [22] pursue a different approach for implementing PoT quantization.

3. Experimental Methodology

This section provides details on ❶ handwritten vectorized kernels for RVV-1.0 and AVX512, ❷ handwritten vectorized floating-point PoT kernels for AVX512 and RVV-1.0, ❸ the same kernels with extra logic to handle $\pm\infty$ and *NaN* properly, and ❹ handwritten vectorized fixed-point PoT kernels for AVX512 and RVV-1.0.

3.1. Notation

We define the target operation as $\boldsymbol{C} = \boldsymbol{AB}$ where $\boldsymbol{A}$ is the non-trainable input tensor and $\boldsymbol{B}$ is the trainable tensor, known at compile-time. Using PoT quantization, $\boldsymbol{C} = \boldsymbol{A} \bigoplus \boldsymbol{B}_{\text{PoT}}$ is used to replace multiplication with an addition along with some extra operations, depending on whether a fixed-point or a floating-point PoT is used. We use a Python-like approach for loops and arrays, meaning that the index range "$0 : 2 : 8$" (first, step, last) maps to the set $\{0, 2, 4, 6\}$. However, we use address offsets (as in C++) for vectorized loads and stores. To refer to a PoT quantization configuration, we use `T1:T2:E:P` where `T1` and `T2` refer to $\boldsymbol{A}$ and $\boldsymbol{B}_{\text{PoT}}$. The number of bits required to store the signed exponent of each element of $\boldsymbol{B}$ in $\boldsymbol{B}_{\text{PoT}}$ is denoted by `E`, while `P` refers to the number of PoT-encoded elements of $\boldsymbol{B}$, packed into a word of $\boldsymbol{B}_{\text{PoT}}$. An example would be `F32:U8:E5:P1`. Also, to save space, **FPoT** and **FXPoT** are used to refer to floating-point and fixed-point PoT quantization.

3.2. Quantization Methodology

We have limited our work only to **MatMul inference** workload with a known second operand at compile-time. This is critical, as the encoding operation on the raw elements of tensor $\mathbf{B}$ affects runtime performance unless the tensor data is known at compile time, in which case the encoding can be

performed ahead of model inference at no additional cost. Throughout this work, we use the proposed PoT scheme in *DenseShift* ($\{\pm 2^p\}$) for the floating point PoT and **assume the quantized PoT weight tensor is always the second operand.**

For FXPoT, considering $s \mathrel{+}= a * b$ operation in a MatMul kernel, variables s, a, and b are **signed integers** and $p = \log_2(|b|)$. Therefore, $a * b$ can be replaced by $\text{sign}(b) \cdot (a \ll p)$. This approach replaces an integer multiplication with a variable shift (opposed to immediate) and some extra logic to handle the negation, if b is negative.

For FPoT, in floating-point multiplication of $s \mathrel{+}= a * b$, using *DenseShift*'s floating-point methodology, we replace $a * b$ with a direct unsigned integer addition of the sign and exponent fields of a, with a PoT quantized value [5], assuming that the value is not represented in two's complement and its sign and positive exponent field align with those of a (Fig. 1).

3.3. Encoding

Table 1 lists the encoding configurations used in this work. Tensors $\boldsymbol{A}$ and $\boldsymbol{C}$ are row-major and tensor $\boldsymbol{B}_{\text{PoT}}$ is column-major ($\boldsymbol{C} = \boldsymbol{A}\boldsymbol{B}_{\text{PoT}}$).

3.4. Baselines

This paper aims to investigate how PoT influences MatMul inference. Therefore, we use a naive MatMul implementation (no tiling, single-thread). The code consists of independent executables for each configuration of PoT kernels, each with three baselines of scalar, scalar-autovectorized, and intrinsic-based. For example, Alg. 1 shows the baseline `Float32` MatMul kernel with RVV-1.0. Note that AVX512 does not have an FMA instruction for integer types, so the non-fused multiplication and addition instructions are used.

Table 1
The encoding schemes used for B_{PoT}.

Kernel	Packing	Encoding
AVX512 Unpack1	F32:U8:E5:P1	`(sign<<7)|(exponent&0x1F)`
AVX512 Unpack2	F32:U16:E8:P1	`(sign<<15)|((exponent&0xFF)<<7)`
RVV-1.0 Unpack1	F32:U8:E5:P1	`(sign<<7)|(exponent&0x1F)`
RVV-1.0 Unpack1	F32:U8:E3:P2	`word=(((sign_1<<3)|(exp_1&0x07))<<4)|` `((sign_0<<3)|(exp_0&0x07))`

Algorithm 1 The baseline RVV-1.0 `F32:F32`.

1: **for** $j \leftarrow 0 : N$ **do** UF0
2: **for** $i \leftarrow 0 : N$ **do** UF1
3: $v_s \leftarrow 0,\ vl \leftarrow 0$ ▷ vfmv_v_f_f32m1 (for v_s)
4: **for** $k \leftarrow 0 : vl : N$ **do** UF2
5: $vl \leftarrow$ vsetvl_e32m1$(N - k)$
6: $v_a \leftarrow \boldsymbol{A}[j,k],\ v_b \leftarrow \boldsymbol{B}[k,i]$ ▷ vle32_v_f32m1
7: $v_s \leftarrow v_s + v_a * v_b$ ▷ vfmacc_vv_f32m1
8: $v_r \leftarrow$ reduce(v_s) ▷ vfmv_f_s_f32m1_f32(vfredosum_vs_f32m4_f32m1())
9: $\boldsymbol{C}[j,i] \leftarrow v_r$

3.5. Autotuning

We used brute-force autotuning implemented in Bash, sweeping all the possible combinations for the defined tunable parameters. A shared combination of unrolling factors for all kernels in a benchmark executable was used to reduce the autotuning time. Each PoT kernel benchmark has one executable. The tuning objective is to minimize the runtime of the PoT kernels. The unroling factors assigned to each loop (if any) are shown with the blue markings on the algorithms.

4. FPoT-quantized MatMul on AVX512

Tensor $\boldsymbol{B}_{\text{PoT}}$ can have more than one word per element and has to be unpacked so each word aligns with its relevant `Float32` fields of tensor $\boldsymbol{A}$. We introduce two methods for unpacking. The first unpacking logic, which is shared by the two RVV-1.0 kernels and one of the AVX512 kernels, and will from now on be referred to as **Unpacking1**, consists of unpacking 8-bit elements into 32-bit elements.

The second unpacking method, referred to as **Unpacking2**, is implemented using the `permutexvar` intrinsic to place an element of value zero to the left of every 16-bit element of the packed vector, while unpacking to 32-bit vector registers. This is done by providing `0xFF` as the indices for `permutexvar`.

The first AVX kernel (`F32:U8:E5:P1`) unpacks the elements with Unpacking1, creating a mask using the sign bits to remove the sign bit itself (`mask_sub`) to perform multiplication via addition. Finally, the sign bits are flipped selectively by `mask_xor` (Alg. 2). **The second** AVX kernel with Unpacking2 (`F32:U16:E8:P1`) uses 16-bit words to simplify the runtime operations, using `permutexvar` to unpack the elements by inserting zeros to align the fields against `Float32` (Alg. 3).

Algorithm 2 AVX512 FPoT Unpacking1 `F32:U8:E5:P1`

1: $n_d \leftarrow$ 0x1F ▷ m5_set1_epi32
2: $s_r \leftarrow$ 0x20 ▷ m5_set1_epi32
3: $s_v \leftarrow$ 0x80000000 ▷ m5_set1_epi32
4: **for** $j \leftarrow 0 : N$ **do** UF0
5: **for** $i \leftarrow 0 : N$ **do** UF1
6: $v_s \leftarrow 0$ ▷ m5_setzero_ps
7: **for** $k \leftarrow 0 : 64 : N$ **do** UF2
8: $v_b \leftarrow \boldsymbol{B}[k, i]$ ▷ m5_load_si512
9: **for** $t \leftarrow 0 : 4$ **do**
10: $v_a \leftarrow \boldsymbol{A}[j, k + 16t]$ ▷ m5_load_ps
11: $b_t \leftarrow v_b[16t]$ ▷ m5_cvtepi8_epi32(m5_extracti64x2_epi64())
12: $v_{a_{\text{int}}} \leftarrow \text{cast}(v_a)$ ▷ m5_castps_si512
13: $n_m \leftarrow \text{cmp}(b_t, n_d)$ ▷ m5_cmpgt_epi32_mask
14: $b_t \leftarrow \text{mask\_sub}(b_t, n_m, b_t, n_d)$ ▷ m5_mask_sub_epi32
15: $b_t \leftarrow b_t \ll 23$ ▷ m5_slli_epi32
16: $v_{sum} \leftarrow v_{a_{\text{int}}} + b_t$ ▷ m5_add_epi32
17: $v_{sum} \leftarrow \text{mask\_xor}(v_s, n_m, v_s, s_v)$ ▷ m5_mask_xor_epi32
18: $v_f \leftarrow \text{cast}(v_{sum})$ ▷ m5_castsi512_ps
19: $v_s \leftarrow v_s + v_f$ ▷ m5_add_ps
20: $v_r \leftarrow \text{reduce}(v_s)$ ▷ m5_reduce_add_ps
21: $\boldsymbol{C}[j, i] \leftarrow v_r$

5. FPoT-quantized MatMul on RVV

For RVV, Vector Length Specific (VLS) is a programming style with vectors whose size is known beforehand and always static, allowing for greater optimization opportunities at the cost of less portability. In a Vector Length Agnostic (VLA) code, the size of the vectors depends on the target machine and the code has to query for it. This leads to better portability across different machines. As for Vector Length Multiplier (LMUL), for a value of greater than one, the vector registers are combined to form a vector register group. If LMUL is less than one, only fractions of the vector registers are utilized

Algorithm 3 AVX512 FPoT Unpacking2 `F32:U16:E8:P1`

$a_{\text{indices}} \leftarrow \{0\text{xFF}, 0, 0\text{xFF}, 1, 0\text{xFF}, 2, \ldots, 0\text{xFF}, 15\}$ ▷ alignas(64)
$v_{\text{indices}} \leftarrow a_{\text{indices}}$ ▷ m5_load_si512
for $j \leftarrow 0 : N$ **do** UF0
  for $i \leftarrow 0 : N$ **do** UF1
    $v_s \leftarrow 0$ ▷ m5_setzero_ps
    for $k \leftarrow 0 : 16 : N$ **do** UF2
      $v_a \leftarrow \boldsymbol{A}[j, k]$ ▷ m5_load_ps
      $v_{a_{\text{int}}} \leftarrow \text{cast}(v_a)$ ▷ m5_castps_si512
      $v_b \leftarrow \boldsymbol{B}[k, i]$ ▷ m2_load_si256
      $v_{b_{512}} \leftarrow \text{cast}(v_b)$ ▷ m5_castsi256_si512
      $v_{b_{512,\text{ unpacked}}} \leftarrow \text{permute}(v_{\text{indices}}, v_{b512})$ ▷ m5_permutexvar_epi16
      $v_{sum} \leftarrow v_{a_{\text{int}}} + v_{b_{512,\text{ unpacked}}}$ ▷ m5_add_epi16
      $v_f \leftarrow \text{cast}(v_{sum})$ ▷ m5_castsi512_ps
      $v_s \leftarrow v_s + v_f$ ▷ m5_add_ps
    sum $\leftarrow$ reduce(v_s) ▷ m5_reduce_add_ps
    $\boldsymbol{C}[jN + i] \leftarrow$ sum

(LMUL $\in \{\frac{1}{8}, \frac{1}{4}, \frac{1}{2}, 1, 2, 4, 8\}$). If set to *dynamic*, the compiler will adapt the `LMUL` for the code, unless a specific `LMUL` is enforced by the code. Both `F32:U8:E5:P1` (Alg. 5) and `F32:U8:E3:P2` (Alg. 4) with Unpacking1 are VLA and implemented via `riscv_vwcvtu` intrinsic. They both create a mask for the negative elements (`cmp`) to remove the sign bits (`mask_sub`) and XOR the sign bits (`mask_xor`) after multiplication by addition (Alg. 5). In the other kernel shown in Alg. 4, after the initial unpacking, lower and upper nibbles of the 8-bit elements of the vectors are concatenated and reordered via the use of `concat` and `gather`. Then, a similar logic to that of `E5:P1` is utilized.

Algorithm 4 RVV-1.0 FPoT `F32:U8:E3:P2`.

vlmax $\leftarrow$ vsetvlmax_e32m8()
elements $\leftarrow$ vsetvlmax_e8m2()
$a_{\text{indices}} \leftarrow \{0\}$
for $i \leftarrow 0 :$ elements **do**
  if $i \mod 2$ **then**
    $a_{\text{indices}}[i] \leftarrow i/2$
$v_{\text{index}} \leftarrow a_{\text{indices}}$ ▷ vle8_v_u8m2
for $j \leftarrow 0 : N$ **do** UF0
  for $i \leftarrow 0 : N$ **do** UF1
    $v_s \leftarrow 0$ ▷ vfmv_v_f_f32m8
    $vl \leftarrow 0$
    for $k \leftarrow 0 : vl : N$ **do** UF2
      $vl \leftarrow$ vsetvl_e32m8$(N - k)$
      $v_a \leftarrow \boldsymbol{A}[j, k]$ ▷ vle32_v_f32m8
      $v_b \leftarrow \boldsymbol{B}[k, i]$ ▷ vle32_v_u8m1
      $v_{lb} \leftarrow$ lowerHalf(v_b) ▷ vand_vx_u8m1
      $v_{ub} \leftarrow$ upperHalf(v_b) ▷ vand_vx_u8m1
      $v_{cjb} \leftarrow$ concat(v_{lb}, v_{ub}) ▷ vset_v_u8m1_u8m2
      $v_{cjb} \leftarrow$ gather$(v_{cjb}, v_{\text{index}})$ ▷ vrgather_vv_u8m2
      $n_m \leftarrow$ cmp$(v_{cjb}, 0\text{b}11111)$ ▷ vmsgtu_vx_u8m1_b8
      $v_{cjb}b \leftarrow$ mask_sub$(n_m, v_{cjb}, v_{cjb}, 0\text{b}100000)$ ▷ vsub_vx_u8m1_tumu
      $v_{cjb} \leftarrow$ expand(v_{cjb}) ▷ vwcvtu_x_x_v_u32m8(vwcvtu_x_x_v_u16m4())
      $v_{cjb} \leftarrow v_{cjb} \ll 23$ ▷ vsll_vx_u32m8
      $v_{a_{\text{uint}}} \leftarrow$ rintrp(v_a) ▷ vreinterpret_v_f32m8_u32m8
      $v_{a_{\text{uint}}} \leftarrow v_{a_{\text{uint}}} + v_{cjb}$ ▷ vadd_vv_u32m8
      $v_a \leftarrow$ rintrp$(v_{a_{\text{uint}}})$ ▷ vreinterpret_v_u32m8_f32m8
      $v_s \leftarrow v_s + v_a$ ▷ vfadd_vv_f32m8
    $v_r \leftarrow$ reduce(v_s) ▷ vfmv_f_s_f32m1_f32(vfredosum_vs_f32m4_f32m1())
    $\boldsymbol{C}[j, i] \leftarrow v_r$

Algorithm 5 RVV-1.0 FPoT `F32:U8:E5:P1`.

```
1:  n_d ← 0x1F, s_r ← 0x20                                   ▷ m5_set1_epi32
2:  s_v ← 0x80000000                                         ▷ m5_set1_epi32
3:  for j ← 0 : N do UF0
4:    for i ← 0 : N do UF1
5:      v_s ← 0, vl ← 0                                      ▷ vfmv_v_f_f32m1 (for v_s)
6:      for k ← 0 : vl : N do UF2
7:        vl ← vsetvl_e32m4(N − k)
8:        v_a ← A[j, k]                                      ▷ vle32_v_f32m4
9:        v_b ← B[k, i]                                      ▷ vle32_v_u8m1
10:       n_m ← cmp(v_b, n_d)                                ▷ vmsgtu_vx_u8m1_b8
11:       v_b ← mask_sub(n_m, v_b, v_b, s_r)                 ▷ vsub_vx_u8m1_tumu
12:       v_b ← expand(v_b)                                  ▷ vwcvtu_x_x_v_u32m4(vwcvtu_x_x_v_u16m2())
13:       v_b ← v_b ≪ 23                                     ▷ vsll_vx_u32m4
14:       v_a_uint ← rintrp(v_a)                             ▷ vreinterpret_v_f32m4_u32m4
15:       v_a_uint ← v_a_uint + v_b                          ▷ vadd_vv_u32m4
16:       v_a_uint ← mask_xor(n_m, v_a_uint, s_v)            ▷ vxor_vx_u32m4
17:       v_a ← rintrp(v_a_uint)                             ▷ vreinterpret_v_u32m4_f32m4
18:       v_s ← v_s + v_a                                    ▷ vfadd_vv_f32m4
19:     v_r ← reduce(v_s)                                    ▷ vfmv_f_s_f32m1_f32(vfredosum_vs_f32m4_f32m1())
20:     C[j, i] ← v_r
```

6. Handling $\pm\infty$ and NaN

Since FPoT replaces a multiplication of two floating point values with an integer addition, the edge cases defined in IEEE754 are no longer handled by the hardware. Therefore, the exponent bits after performing addition can overflow and flip the sign bit of the output. *DenseShift* [5] is also prone to this issue. A full software solution for this could render the FPoT speedup pointless, but we could still implement extra logic to manage just the $\pm\infty$ and *NaN* elements of A. These values ($\pm\infty$ and *NaN*) both have the exact exponents of `0xFF`. These values ($\pm\infty$ and *NaN*) both have exponent bits set to `0xFF`. Only the non-trainable tensor ($\boldsymbol{A}$) can contain such values, as the trainable tensor is restricted to valid PoT quantized levels in DenseShift, which do not include zero or special IEEE754 representations. As shown in Fig. 2, since $\infty \cdot x = \infty$ if x is not a ∞, *NaN*, or zero, as long as the FPoT exponents are not added to the bad elements of tensor $\boldsymbol{A}$, the accumulated results in `Float32` will be correct. Later, the hardware would handle the edge cases for any pairs of $\pm\infty$ or *NaN* values in `Float32`. As shown in Alg. 6, the mask inf_m is used to perform masked addition by excluding the bad elements from the addition with the FPoT exponents. The extra instructions needed to perform the mask addition come with an acceptable overhead, discussed in Sec. 8.3.

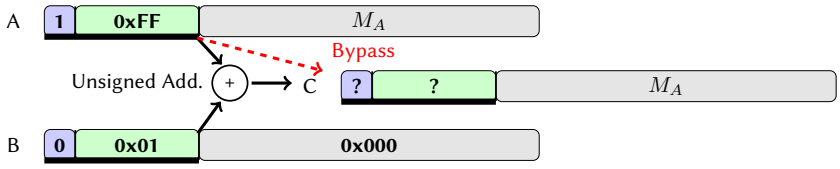

Figure 2: Bypassing the unsigned addition to avoid overflow of the exponent field into the sign bit, when a non-trainable in $\boldsymbol{A}$ is *NaN* or $\pm\infty$.

7. FXPoT

To clarify the nature of speedup, if any, for the FXPoT kernels compared to normal fixed-point MatMul, we developed two benchmarks for AVX512 and RVV. The idea is to see if a performance gain could be achieved by using shift instructions to replace integer multiplications **in a loop**. Therefore, we excluded negation logic for handling negative numbers from the code and opted for `I32:I32` data types for the baseline kernel. For AVX512, the multiplication intrinsic instances (`mullo_epi32`)

Algorithm 6 AVX512 FPoT Unpacking1 `F32:U8:E5:P1` with $\pm\infty$ and *NaN* handling.

function `not_NaN_infinity`(x)
  $m_{\text{mantissa}} \leftarrow$ `0x100FFFFFF` ▷ m5_set1_ps
  $\text{inf}_v \leftarrow \infty$ ▷ m5_set1_ps
  $x \leftarrow (\neg x)$ & m_{mantissa} ▷ m5_andnot_ps
  $mask \leftarrow \text{cmp}(x, \text{inf}_v)$ ▷ m5_cmp_ps_mask
  return mask
$n_d \leftarrow$ `0x1F`, $s_r \leftarrow$ `0x20` ▷ m5_set1_epi32
$s_v \leftarrow$ `0x80000000` ▷ m5_set1_epi32
for $j \leftarrow 0 : N$ **do** UF0
  for $i \leftarrow 0 : N$ **do** UF1
    $v_s \leftarrow 0$ ▷ m5_setzero_ps
    for $k \leftarrow 0 : 64 : N$ **do** UF2
      $v_b \leftarrow \boldsymbol{B}[k, i]$ ▷ m5_load_si512
      for $t \leftarrow 0 : 4$ **do**
        $v_a \leftarrow \boldsymbol{A}[j, k + 16t]$ ▷ m5_load_ps
        $b_t \leftarrow v_b[16t]$ ▷ m5_cvtepi8_epi32(m5_extracti64x2_epi64())
        $\text{inf}_m \leftarrow$ `not_NaN_infinity`(v_a)
        $v_{a_{\text{int}}} \leftarrow \text{cast}(v_a)$ ▷ m5_castps_si512
        $n_m \leftarrow \text{cmp}(b_t, n_d)$ ▷ m5_cmpgt_epi32_mask
        $b_t \leftarrow \text{mask\_sub}(b_t, n_m, b_t, s_r)$ ▷ m5_mask_sub_epi32
        $b_t \leftarrow b_t \ll 23$ ▷ m5_slli_epi32
        $v_{sum} \leftarrow \text{mask\_add}(v_{a_{\text{int}}}, \text{inf}_m, v_{a_{\text{int}}}, b_t)$ ▷ m5_mask_add_epi32
        $v_{sum} \leftarrow \text{mask\_xor}(v_s, n_m, v_s, s_v)$ ▷ m5_mask_xor_epi32
        $v_f \leftarrow \text{cast}(v_{sum})$ ▷ m5_castsi512_ps
        $v_s \leftarrow v_s + v_f$ ▷ m5_add_ps
    $v_r \leftarrow \text{reduce}(v_s)$ ▷ m5_reduce_add_ps
    $\boldsymbol{C}[j, i] \leftarrow v_r$

are replaced by bitwise left-shifts. The elements of $\boldsymbol{A}$ and $\boldsymbol{B}$ are loaded with `load_si512`, left-shifted with `sllv_epi32`, and accumulated by `add_epi32`. The resulted vector is then reduced to a scalar using `reduce_add_ps` and written to the output tensor $\boldsymbol{C}$. The same approach is used for the RVV kernel using `sll_vv_i32m4`, `vadd_vv_i32m4`, and `vredsum_vs_i32m4_i32m1`. Note that the FXPoT kernels do not need more than 5 bits to store the exponent bits, since the selected base data type is 32 bits, which means the PoT type is `I32:U32:E5:P1`.

8. Experimental Evaluation

Table 2 lists the machines used for running the benchmarks. To compute the speedups, we compared the runtime of our handwritten SIMD PoT-based MatMul kernels against the handwritten SIMD multiplication-based MatMul baselines. GCC 13.3.0, 14.2.0, LLVM 17.0.6, and 18.1.8 were used for compiling the benchmarks. The flags used for each one are listed in Table 3. We did not use GCC 13.3.0 for RISC-V since it does not provide full support for VLA/VLS. The F, BW, DQ, and VL extensions of AVX512 are used.

Table 2
The list of the used machines.

Name	Fab.	ISA	f_{Base}	f_{Max}	Memory	Cache
Xeon 5218	14 nm	AVX512	2.30G	3.9G	DDR4 2666	L1d: 32KB/C L2: 1MB/C L3: 22MB
Xeon 8260	14 nm	AVX512	2.40G	3.9G	DDR4 2933	L1d: 32KB/C L2: 1MB/C L3: 35.75MB
Ryzen 9 7950X	5/6 nm	AVX512	4.5G	5.7G	DDR5 4800	L1d: 32KB/C L2: 1MB/C L3: 64MB
SpacemiT-K1	12 nm	RVV-1.0	614M	1.6G	LPDDR4 2133	L1d: 32KB/C L2: 512KB L3: -

Table 3
The used compiler flags for SIMD baseline and PoT kernels.

AVX512	**GCC**	-O3 -march=native -ffast-math -fno-tree-vectorize -fno-tree-slp-vectorize
	LLVM	-O3 -march=native -ffast-math -fno-vectorize
RVV-1.0	**GCC**	-O3 -march=rv64gcv_zvl256b -ffast-math -mrvv-max-lmul=dynamic -fno-tree-vectorize -fno-tree-slp-vectorize
	LLVM	-O3 -march=rv64gcv_zvl256b -ffast-math -fno-vectorize

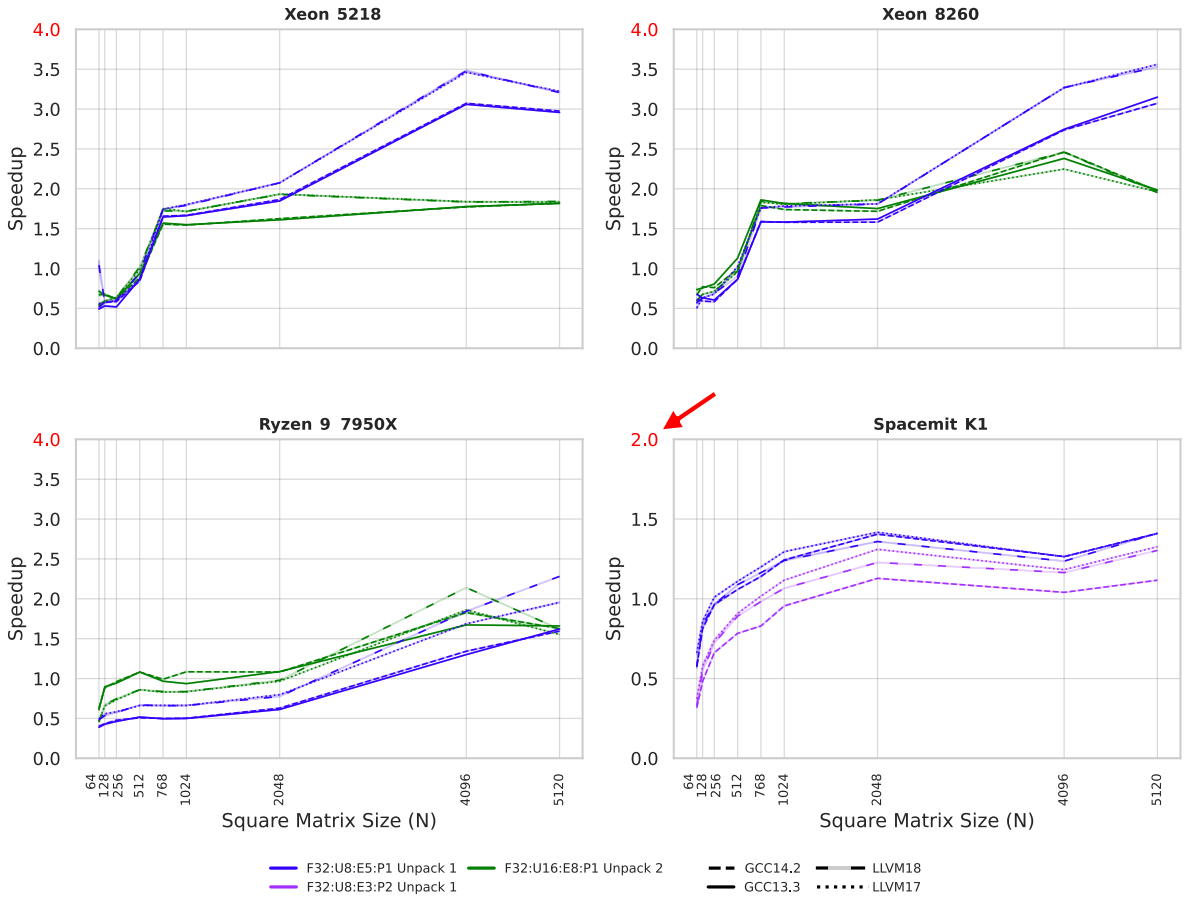

Figure 3: The speedups achieved by using different configurations of FPoT.

Autotuning For RISC-V, we used dynamic `LMUL` and let the autotuner script find the best between `VLA` and `VLS`. Note that `VLS`-build is selected by adding GCC flag `-mrvv-vector-bits=zvl` and LLVM flag `-mrvv-vec tor-bits=256`. For GCC, `VLS` build flags work in conjunction with the `-march`. To manage slow tuning, we only considered tuning the unrolling factor for the inner-most loop with a value from $\{1, 2, 4\}$. We kept SMT (i.e., Intel's hyperthreading) active and the rest of the cores idling. The CPU affinity was enforced with `taskset`. The `SIMD` kernels' measurements are repeated 75 times.

Tensor Shapes We considered square matrices ($N \times N$) that satisfy the constraints of all our baseline and PoT kernels. For the AVX512 kernels, the most restrictive constraint is for N to be a multiple of 64. For the RVV kernels, the VLA design imposes no restrictions on N, while `E3:P2` encoding requires N to be a multiple of 2. Therefore, we considered $N \in \{64, 128, 256, 512, 768, 1024, 2048, 4096, 5120\}$ to cover a wide range of shapes while keeping the runtime of the experiments manageable. Handling the cases for non-square matrices and the ones that do not satisfy the constraints of our kernels are straightforward, but left for future work, as our main goal is to prove the effectiveness of the core idea behind FPoT and FXPoT in a fair setting.

8.1. AVX512 Results

The results for Alg. 2 and 3 for various square matrix sizes (N) are shown in Fig. 3. The same data for $N = 5K$ and LLVM 18 is shown in Fig. 4. Unpacking1 is 1.13x (geomean) faster than Unpacking2 with LLVM compilers. The profiling data on Xeon 5218 with LLVM 18 using Intel Advisor and VTune indicates that the baselines are L2-bound and perform worse for $N = 3K$ as opposed to $N = 1K$, while

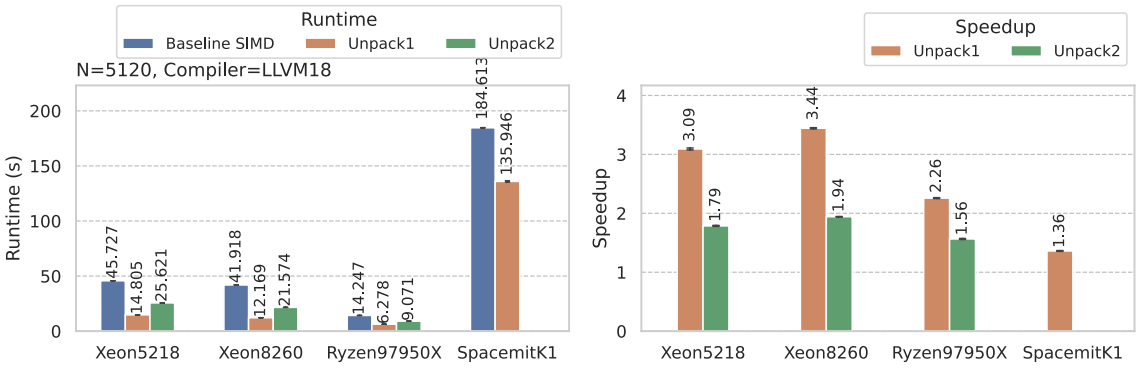

Figure 4: The FPoT speedups with LLVM 18, for $N = 5K$.

the FPoT kernels sustain the same performance thanks to the increased arithmetic intensity of FPoT by the spared loads. This leads to the steep increase of speedup in $1024 \leq N < 4096$. For $N \leq 768$, the tensors can fit entirely in the caches, diminishing the effectiveness of FPoT kernels over the baselines. Because Unpacking2-based kernels use `F32:U16` rather than `F32:U8` as in Unpacking1, the increase in arithmetic intensity is smaller, resulting in performance that relies more heavily on the L3 cache. This is evident in the green lines in Fig. 3; performance correlates with L3 cache size (Ryzen 9 > Xeon 8260 > Xeon 5218) and the kernels that use Unpack2 achieve higher speedup than the ones using Unpack1, on devices with larger L3 caches. Unpack1 (blue lines) achieves better speedups on devices with smaller L3 caches. Furthermore, a better performance is observed for Unpacking2-based FPoT on Xeon 8260 compared to Xeon 5218. The official specifications indicate that Xeon 5218 has one FMA unit, while Xeon 8260 has two, which may partly explain the steeper speedup increase on Xeon 5218. The documentation of Ryzen 9 does not specify the number of available FMA units. Unlike the Intel CPUs, Unpacking2-based FPoT is 1.72x faster on Ryzen 9 than Unpacking1 with GCC compilers. As in Table 2, Ryzen 9 has faster memory. Therefore, the spared load instructions by PoT are less effective compared to Intel devices. LLVM 17 and 18 perform best, with geomean speedup of 1.28 on Intel and 0.85 on AMD. The maximum speedup for Xeon 5218, Xeon 8260, and Ryzen 9 are 3.67, 3.60, and 2.28, respectively.

8.2. RVV-1.0 Results

The results for Alg. 1, 5, and 4 for different N values on Banana Pi F3 with SpacemiT-K1 SoC are shown in Fig. 3. The results for $N = 5K$ with LLVM 18 are demonstrated in Fig. 4. The geomean speedup of FPoT `E3:P2` over `E5:P1` is 0.72 for GCC, due to the overhead of unpacking. The maximum speedup on SpacemiT-K1 is 1.45x with LLVM 17, followed by 1.43x with GCC 14.2 and 1.42x with LLVM 18. The geomean speedup of LLVM 17 and 18 is 1.01.

8.3. AVX512 $\pm\infty$ and NaN Handling Results

In this section, we experimentally evaluated the infinity and *NaN* handling algorithm, as described in Sec. 6 and Alg. 6. We built the kernel with LLVM 18 and ran it on Xeon 5218. A slightly lower speedup is observed (Fig. 5) compared to the kernels that do not account for the extra $\pm\infty$ and *NaN* handling (Fig. 3). The maximum speedup for this kernel is 2.78, while the kernel described in Alg. 2 achieves 3.67.

8.4. FXPoT Results

We used LLVM 18 on Xeon 5218 and Banana Pi F3 (SpacemiT-K1) with autotuning. Fig. 6 provides the results for these kernels. As can be seen, FXPoT provides no meaningful and dependable speedup over the baseline. Since both FXPoT and its baseline are using 32-bit data type (`I32` and `U32`) without saving any load instructions, the transient gain for $N \leq 512$ on Xeon 5218 represents architectural properties such as the available units for shifting and multiplication.

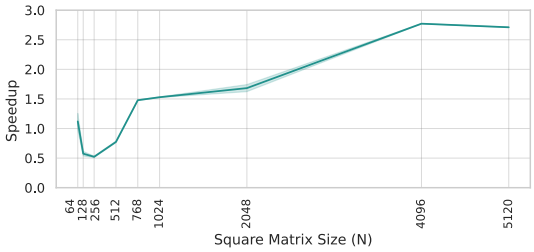

Figure 5: The speedups for FPoT kernels with Unpacking1 on AVX512 with LLVM 18 with $\pm\infty$ and *NaN* handling on Xeon 5218.

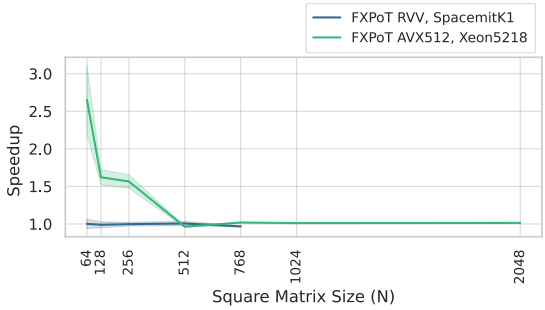

Figure 6: The FXPoT speedups on AVX512 (Xeon 5218) and RVV-1.0 (SpacemiT-K1) with LLVM 18.

8.5. Autovectorization

To measure how well different compilers were able to auto-vectorize the scalar baseline kernels, we define $\text{Speedup}_{ss} = \frac{t_{SNA}}{t_{SAV}}$. Note that SAV and SNA refer to scalar kernels with and without auto-vectorization. We also provide $\text{Speedup}_{vs} = \frac{t_{SNA}}{t_{SIMD}}$ to compare FPoT kernels against scalar non-autovectorized baselines. SIMD refers to the FPoT handwritten SIMD kernels, all shown in Fig. 7, for the matrix size of $5K$ and LLVM 18. The compiler flags used for these settings are similar to the ones listed in Table 3, but without flags to disable autovectorization. The SNA and SAV kernels' measurements are repeated 7 times. Note that Speedup_{vs} is a good approximate indicator of the upper bound for Speedup_{ss}. **For AVX512**, GCC achieves the highest geomean speedup_{ss}, with 7.34 on Intel and 14.75 on AMD. For speedup_{vs}, GCC also leads with 9.0 (Intel) and 11.81 (AMD). **For RVV**, the best-performing compiler is GCC with a geomean speedup_{ss} of 3.66.

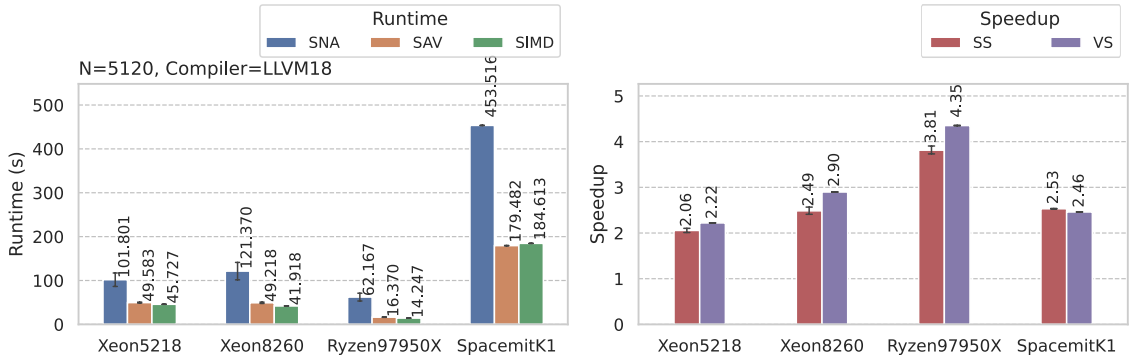

Figure 7: The runtimes and speedups of scalar autovectorized and handwritten SIMD baseline MatMul kernels against the scalar non-autovectorized MatMul baseline (SS and VS, respectively) with LLVM 18, for $N = 5K$.

9. Discussion

Based on our results, FPoT provided a maximum speedup of 3.67. Using the geomean speedup with the best compiler, implementing FPoT with `permutexvar` yielded a 1.16x speedup on Ryzen 9, but resulted in a 0.87x slowdown on the Intel CPUs (Fig. 3). Furthermore, on SpacemiT-K1, FPoT achieved a geomean speedup of 1.39x without packing multiple elements into a byte (Fig. 3). Finally, LLVM compilers yielded better speedup overall for AVX512 and RVV (Fig. 3). While *DenseShift* [5] uses `F16:F16` *GEMM* kernels from *NCNN* [23], we opted for `float32` handwritten baselines, as the `AVX512-FP16` ISA extension is only available since *Saphire Rapids* on Intel, even though the SpacemiT-K1 has support

for `Float16`. A direct comparison with *DenseShift* would not be fair, considering the different CPU architectures, ISA, data types, workload, and missing details on the encoding scheme used in *DenseShift*. Unfortanely, we did not have access to an ARM A57 SoC to reproduce the results from *DenseShift* with our implementation. We hope our work would encourage future research in this direction, providing detailed information on the encoding scheme and the baselines used for comparison. Nevertheless, we observed a similar speedup of **1.45x on SpacemiT-K1 with RVV-1.0** compared to a speedup of **1.6x on ARM A57 Neon** [5] (we report the results published in the paper because the artifacts are not available). To the best of our knowledge, no other work provides results for handwritten AVX512 and RVV-1.0-based FPoT quantized MatMul with various configurations.

10. Conclusion

This work focuses on AVX512 and RVV-1.0 implementations of floating-point PoT quantized MatMul inference kernels, initially proposed by DenseShift. On AVX512, we proposed two ways to implement unpacking for floating-point PoT MatMul with one element per word. Furthermore, for RVV-1.0, we explored the implementation of floating-point PoT MatMul inference with one and two elements per word. We proposed and evaluated a novel approach to handle $\pm\infty$ and *NaN* floating-point values. Overall, the experimental results indicated that floating-point PoT quantization for MatMul inference yields speedups up to 3.67x on Xeon 5218 AVX512 and 1.45x on SpacemiT-K1 RVV-1.0 in the single-thread configuration.

Declaration on Generative AI

The author(s) have not employed any Generative AI tools.

References

[1] W. A. Wulf, S. A. McKee, Hitting the memory wall: implications of the obvious, SIGARCH Comput. Archit. News 23 (1995) 20–24. doi:10.1145/216585.216588.

[2] J. Hoffmann, S. Borgeaud, A. Mensch, E. Buchatskaya, T. Cai, E. Rutherford, D. d. L. Casas, L. A. Hendricks, J. Welbl, A. Clark, et al., Training compute-optimal large language models, arXiv preprint arXiv:2203.15556 (2022).

[3] E. Frantar, S. Ashkboos, T. Hoefler, D. Alistarh, GPTQ: Accurate post-training quantization for generative pre-trained transformers, arXiv preprint arXiv:2210.17323 (2022).

[4] M. Elhoushi, Z. Chen, F. Shafiq, Y. H. Tian, J. Y. Li, DeepShift: Towards multiplication-less neural networks, in: IEEE Conference on Computer Vision and Pattern Recognition Workshops, CVPR Workshops 2021, virtual, June 19-25, 2021, Computer Vision Foundation / IEEE, 2021, pp. 2359–2368. doi:10.1109/CVPRW53098.2021.00268.

[5] X. Li, B. Liu, R. H. Yang, V. Courville, C. Xing, V. P. Nia, DenseShift: Towards accurate and efficient low-bit power-of-two quantization, in: IEEE/CVF Int. Conf. on Computer Vision, ICCV, IEEE, 2023, pp. 16964–16974. doi:10.1109/ICCV51070.2023.01560.

[6] S. Jamali Golzar, G. Karimian, M. Shoaran, M. Fattahi Sani, DGCNN on FPGA: acceleration of the point cloud classifier using FPGAS, Circuits, Systems, and Signal Processing 42 (2023) 748–779.

[7] Y. Wang, Y. Sun, Z. Liu, S. E. Sarma, M. M. Bronstein, J. M. Solomon, Dynamic graph cnn for learning on point clouds, ACM Transactions on Graphics (tog) 38 (2019) 1–12.

[8] M. Nagel, M. Fournarakis, R. A. Amjad, Y. Bondarenko, M. van Baalen, T. Blankevoort, A white paper on neural network quantization, ArXiv abs/2106.08295 (2021).

[9] X. Li, B. Liu, Y. Yu, W. Liu, C. Xu, V. P. Nia, S3: sign-sparse-shift reparametrization for effective training of low-bit shift networks, in: Proceedings of the 35th Int. Conf. on Neural Information Processing Systems, NIPS, 2021.

[10] S. Hochreiter, The vanishing gradient problem during learning recurrent neural nets and problem solutions, Int. J. of Uncertainty, Fuzziness and Knowledge-Based Systems 6 (1998) 107–116.
[11] H. You, H. Shi, Y. Guo, Y. Lin, ShiftAddViT: Mixture of multiplication primitives towards efficient vision transformer, in: Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems, NeurIPS, 2023.
[12] H. Shi, X. Cheng, W. Mao, Z. Wang, P2-ViT: Power-of-two post-training quantization and acceleration for fully quantized vision transformer, IEEE Transactions on Very Large Scale Integration (VLSI) Systems 32 (2024) 1704–1717. doi:10.1109/TVLSI.2024.3422684.
[13] D. A. Gudovskiy, L. Rigazio, ShiftCNN: Generalized low-precision architecture for inference of convolutional neural networks, arXiv preprint arXiv:1706.02393 (2017).
[14] R. Saha, J. Haris, J. Cano, Accelerating pot quantization on edge devices, in: 2024 31st IEEE Int. Conf. on Electronics, Circuits and Systems (ICECS), IEEE, 2024, pp. 1–4.
[15] T. Dupuis, Y. Fournier, M. AskariHemmat, N. El Zarif, F. Leduc-Primeau, J. P. David, Y. Savaria, Sparq: A custom risc-v vector processor for efficient sub-byte quantized inference, in: 21st IEEE Interregional NEWCAS Conf. (NEWCAS), IEEE, 2023, pp. 1–5.
[16] G. Rutishauser, J. Mihali, M. Scherer, L. Bonini, xTern: Energy-efficient ternary neural network inference on risc-v-based edge systems, in: IEEE 35th Int. Conf. on Application-specific Systems, Architectures and Processors (ASAP), IEEE, 2024, pp. 206–213.
[17] S. Kalapothas, M. Galetakis, G. Flamis, F. Plessas, P. Kitsos, A survey on risc-v-based machine learning ecosystem, Information 14 (2023) 64.
[18] X. Geng, S. Liu, J. Jiang, K. Jiang, H. Jiang, Compact powers-of-two: An efficient non-uniform quantization for deep neural networks, in: Design, Automation & Test in Europe Conference & Exhibition (DATE), 2024, pp. 1–6. doi:10.23919/DATE58400.2024.10546652.
[19] T. Xia, B. Zhao, J. Ma, G. Fu, W. Zhao, N. Zheng, P. Ren, An energy-and-area-efficient cnn accelerator for universal powers-of-two quantization, IEEE Transactions on Circuits and Systems I: Regular Papers 70 (2023) 1242–1255. doi:10.1109/TCSI.2022.3227608.
[20] J.-H. Li, J.-K. Lin, Y.-C. Su, C.-W. Chu, L.-T. Kuok, H.-M. Lai, C.-L. Lee, J.-K. Lee, SIMD Everywhere optimization from arm neon to risc-v vector extensions, arXiv preprint arXiv:2309.16509 (2023).
[21] Y. Li, X. Dong, W. Wang, Additive powers-of-two quantization: An efficient non-uniform discretization for neural networks, in: 8th Int. Conf. on Learning Representations, ICLR, 2020.
[22] Y. M. Kim, K. Han, W.-K. Lee, H. J. Chang, S. O. Hwang, Non-zero grid for accurate 2-bit additive power-of-two cnn quantization, IEEE Access 11 (2023) 32051–32060. doi:10.1109/ACCESS.2023.3259959.
[23] H. Ni, The NCNN contributors, NCNN, 2017. URL: https://github.com/Tencent/ncnn.