Extremely Low-bit Convolution Optimization for Quantized Neural Network on Modern Computer Architectures

Qingchang Han[1,2*] and Yongmin Hu[1*], Fengwei Yu[2], Hailong Yang[1†], Bing Liu[2], Peng Hu[1,2], Ruihao Gong[1,2], Yanfei Wang[2], Rui Wang[1], Zhongzhi Luan[1], Depei Qian[1]
Beihang University[1]
SenseTime Research[2]
{qingchanghan,varinic,hailong.yang,wangrui,07680,depeiq}@buaa.edu.cn,gongruihao@nlsde.buaa.edu.cn
{yufengwei,liubing,hupeng,wangyanfei}@sensetime.com

ABSTRACT

With the continuous demand for higher accuracy of deep neural networks, the model size has increased significantly. Quantization is one of the most widely used model compression methods, which can effectively reduce the model size without severe accuracy loss. Modern processors such as ARM CPU and NVIDIA GPU have already provided the support of low-bit arithmetic instructions. However, there lack efficient and practical optimizations for convolution computation towards extremely low-bit on ARM CPU (e.g., 2∼8-bit) and NVIDIA GPU (e.g., 4-bit and 8-bit). This paper explores the performance optimization methods of extremely low-bit convolution on diverse architectures. On ARM CPU, we propose two instruction schemes for 2∼3-bit and 4∼8-bit convolution with corresponding register allocation methods. In addition, we re-design the GEMM computation with data padding and packing optimizations. We also implement winograd algorithm for convolution with some specific bit width (e.g., 4∼6-bit) to achieve higher performance. On NVIDIA GPU, we propose a data partition mechanism and multi-level memory access optimizations, to better adapt the computation to GPU thread and memory hierarchy. We also propose quantization fusion to eliminate unnecessary data access. The experiment results demonstrate our implementations achieve better performance of extremely low-bit convolution compared to the state-of-the-art frameworks and libraries such as ncnn and cuDNN. To the best of our knowledge, this is the first work that provides efficient implementations of extremely low-bit convolutions covering 2∼8-bit on ARM CPU and 4-bit/8-bit on NVIDIA GPU.

CCS CONCEPTS

• **Computer systems organization → Parallel architectures**;
• **Computing methodologies → Artificial intelligence**.

*The first and second author contributed equally. †Corresponding author.

KEYWORDS

Extremely Low-bit Convolution, Quantized Neural Network, Computation Optimization, ARM CPU, NVIDIA GPU

ACM Reference Format:
Qingchang Han[1,2*] and Yongmin Hu[1*], Fengwei Yu[2], Hailong Yang[1†], Bing Liu[2], Peng Hu[1,2], Ruihao Gong[1,2], Yanfei Wang[2], Rui Wang[1], Zhongzhi Luan[1], Depei Qian[1]. 2020. Extremely Low-bit Convolution Optimization for Quantized Neural Network on Modern Computer Architectures. In *ICPP '20: International Conference on Parallel Processing, August 17–20, 2020, Edmonton, AB, Canada.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Deep neural networks have been successfully adopted in many fields, such as computer vision, natural language processing, automatic driving, etc. With the increasing demand for higher accuracy, the size of DNN models becomes larger, which requires tremendous computation resources and memory footprint. The prohibitive resource requirements of modern DNN models prevent their pervasive deployment on edge devices, where resources are constrained by a limited power budget. Even in the cloud, deploying large DNN models are also a burden for service operators due to the potential of draining resources.

Since it is unsustainable with ever-increasing model size, several methods have been proposed to reduce model size without severely affecting its accuracy, such as network pruning [9, 10] and model quantization [7, 8]. Particularly, the quantization method takes advantage of the insensitivity of deep neural networks to data precision, which converts floating-point data to low-bit data, and accelerates model inference through low-bit computations. For example, the commonly used 8-bit quantization can reduce the model size to one forth than in full precision theoretically. In terms of accuracy, it has been proven that an 8-bit quantized model can almost reach the same accuracy as a full-precision one. Moreover, even lower-bit quantized models (e.g., 2∼4-bit) only loss the accuracy slightly compared to full-precision one [7].

Currently, both edge devices such as ARM CPU and cloud accelerators such as NVIDIA GPU provide architecture support for low-bit arithmetic instructions. For example, on ARMv8.1 CPU, there are many instructions that allow to perform low-bit calculation efficiently, such as `SMLAL`, `MLA`, `SADDW` and etc. On the NVIDIA Turing GPU, the Tensor Core unit can be used for 1-bit, 4-bit and 8-bit mixed-precision matrix multiplication through `mma` instructions [27], which can significantly boost the model inference. The availability of architecture support for low-bit computation on modern processors stimulates the performance optimization of quantized neural networks.

Several neural network inference frameworks and libraries have provided optimized implementation for low-bit computation in quantized neural networks leveraging the architecture support. For example, QNNPACK [6], ncnn [24] and gemmlowp [14] support 8-bit convolution and GEMM on ARM CPU. Whereas on NVIDIA GPU, tensorRT [23] supports 8-bit convolution using Tensor Core, cuDNN [2] supports 16-bit convolution using Tensor Core and 8-bit convolution with *dp4a* instruction, and CUTLASS [26] only supports low-bit GEMM with Tensor Core. Although the existing works [26, 29] have explored low-bit GEMM on ARM CPU and NVIDIA GPU, it does not intuitively transform to efficient low-bit convolution implementation. To the best of our knowledge, there is no public framework or library that can support extremely low-bit convolution covering a wide range of bit width on ARM CPU (2∼8-bit) and NVIDIA GPU (4-bit/8-bit). The missing support for extremely low-bit convolution motivates this paper to provide efficient implementations on processors such as ARM CPU and NVIDIA GPU, where DNN models are widely deployed.

To implement the extremely low-bit convolution efficiently, there are several challenges that need to be addressed on ARM CPU and NVIDIA GPU, respectively. For example, on ARM CPU, no instruction can directly multiply low-bit data and accumulate to the 32-bit data type. Therefore, customized instruction scheme needs to be designed to accelerate low-bit computation through instruction combinations such as `SMLAL`, `MLA` and `SADDW`. In addition, the register allocation method needs to be designed to accommodate the instruction scheme with reduced memory access overhead. Whereas on NVIDIA GPU, although Tensor Core provides high-performance micro GEMM kernel, we still need to adapt the computation to GPU thread organization, optimize the data access across memory hierarchy and eliminate unnecessary data access, to achieve satisfactory performance on low-bit convolution [22].

This paper presents the computation optimizations for achieving extremely low-bit convolution for quantized neural networks on ARM CPU and NVIDIA GPU efficiently. On ARM CPU, we propose two instruction schemes that use `SMLAL` and `SADDW` instructions to optimize 4∼8-bit convolutions, and `MLA` and `SADDW` instructions to optimize 2∼3-bit convolutions respectively. We propose register allocation methods tailored for the above instruction schemes. We also re-design and optimize GEMM computation with data padding and packing. We implement the winograd algorithm to accelerate the convolution kernels with 4∼6-bit input. On NVIDIA GPU, we propose a data partition mechanism that adapts to the GPU thread organization. In addition, we present multi-level memory access optimizations, including coalesced memory access on global memory, reordering access on shared memory, overlapped computation and memory access using registers, and in-place calculation. Moreover, we explore quantization fusion between different kernels to reduce the amount of memory access. The experiment results demonstrate our implementations of extremely low-bit convolutions outperform the state-of-the-art frameworks and libraries.

To the best of our knowledge, this is the first work to present efficient implementations of extremely low-bit convolutions covering a wide range of bit width on diverse architectures such as ARM CPU (2∼8-bit) and NVIDIA GPU (4-bit/8-bit).

Specifically, this paper makes the following contributions:

- On ARM CPU, we propose two instruction schemes that use available instructions wisely to optimize 2∼3-bit and 4∼8-bit convolution kernels, and design register allocation methods tailored for the instruction schemes. We also re-design the GEMM computation with data padding and packing optimizations. In addition, we implement the winograd algorithm to optimize 4∼6-bit convolution kernels.
- On NVIDIA GPUs, we propose a data partition mechanism to adapt the computation to GPU thread organization better. In addition, we propose multi-level memory access optimizations to improve the performance of data access across the GPU memory hierarchy. Moreover, we exploit quantization fusion to eliminate the overhead of storing intermediate results between consecutive kernels such as convolution and dequantization.
- We evaluate our implementations on Raspberry Pi 3B and NVIDIA RTX 2080Ti with ResNet-50. On ARM CPU, our 2-bit and 4-bit convolution kernels achieve 1.60× and 1.38× speedup on average, respectively, compared to 8-bit convolution in ncnn. On NVIDIA GPU, our 4-bit and 8-bit convolution kernels achieve 5.26× and 4.31× speedup on average, respectively, compared to 8-bit convolution in cuDNN.

The rest of the paper is organized as follows: Section 2 introduces the background of our work. Section 3 and Section 4 present the details of our extremely low-bit convolution optimizations on ARM and GPU, respectively. Section 5 presents the experimental results. Section 6 discusses the related work. Section 7 concludes this paper.

2 BACKGROUND

2.1 Quantized Deep Neural Network

Quantized Neural Networks (QNNs), serving as an effective technique to accelerate the DNN inference, have attracted considerable research interests in recent years. On the large scale ImageNet dataset [4], the algorithms proposed in [8] and [7] have achieved the same accuracy as the full-precision counterpart using 4-bit and even 3-bit, with minor accuracy degradation (less than 1%). For more complex tasks such as object detection and face recognition, QNNs have also been proved to be promising in terms of accuracy. For example, Li *et al.* [18] applied 4-bit full quantization to object detection and the proposed training techniques guarantee an outstanding performance (32.5% mAP on COCO dataset [20]). [34] devises a rotation consistent loss to retain the compactness of cluster for the open-set problem and the verification accuracy for 3-bit (98.73%) is competitive to that of the full-precision model (98.9%), which firstly proves the feasibility of extremely low-bit face recognition. Although QNNs can perform well under extremely low-bit setting, there are little efforts to provide an efficient implementation for them. As far as we know, the existing high performance inference libraries [14, 23, 24] mainly focus on the 8-bit convolution operation, and none of them supports lower bit-width such as 2-bit and 4-bit. It is necessary to further explore the specific optimization of lower bit-width for higher performance.

2.2 Convolution Algorithms

The convolution neural networks (CNNs) take a large portion of the DNN family [21]. The convolution layers dominate the inference time of CNN. We elaborate on the popular algorithms used to implement convolution calculations.

Direct convolution algorithm. This algorithm uses the definition of convolution to calculate the output through nested loops. It is simple to implement but inefficient. In real cases, the method is generally optimized to use better the cache and SIMD instructions of the hardware platform. The direct convolution algorithm is adopted in cuDNN, DNNL [13], TVM [1] and etc.

GEMM-based convolution algorithm. The algorithm converts convolution to matrix multiplication (GEMM) through im2col operation and exploits the highly optimized BLAS [5] libraries on various platforms. The GEMM-based methods can be further divided into two categories, such as explicit GEMM and implicit GEMM [15]. On ARM CPU, most frameworks such as Caffe and ncnn adopt the explicit GEMM method, which performs global im2col transformation and applies GEMM on the entire matrices. However, on GPU, the explicit GEMM method requires extra global memory space to store the transformed matrix. Therefore, the implicit GEMM method [2] has been proposed to divide the matrix into small tiles and perform the local conversion in each thread block, avoiding global matrix transformation and reducing memory footprint. Moreover, the implicit-precomp GEMM method [2] has been proposed to reduce computation overhead through precomputing the indexes of data in the input matrix, which has been adopted by cuDNN and TensorRT.

Fast convolution algorithm. The commonly used fast convolution algorithms include FFT-based convolution algorithm [33] and winograd convolution algorithm [17]. The FFT-based convolution algorithm uses FFT, IFFT, and GEMM operations to speedup convolution calculations, which achieves better performance with large kernels, and has been used in cuDNN. The winograd convolution algorithm is based on the Coppersmith–Winograd matrix multiplication algorithm. It trades off the increasing number of additions with the decreasing number of multiplications through matrix transformation for better performance. It is usually applied to convolution kernel with a size of 3×3, and has been adopted in various deep learning frameworks and libraries, such as ncnn.

In this paper, we focus on optimizing the extremely low-bit convolution using GEMM-based algorithm, which has been widely adopted in existing frameworks and libraries. Specifically, we choose the explicit GEMM on ARM CPU [16] and the implicit-precomp GEMM on NVIDIA GPU [15] to implement the GEMM-based convolution. We also implement winograd convolution algorithm in cases (e.g., 4∼6-bit on ARM CPU) where it can be applied to boost the performance.

2.3 Architecture Support for Low-bit Computation

ARM CPUs have been widely used as mobile or edge devices to deploy quantized models. ARMv8 architecture introduces the aarch64 execution state and supports A64 instruction set. There are 31 64-bit general-purpose registers and 32 128-bit vector registers on ARMv8 architecture. In the latest ARMv8.2 architecture, `SDOT` instruction is introduced to support dot product calculation with 8-bit input and 32-bit output. However, ARMv8.1 is still the dominant architecture among existing ARM devices [30], so we focus our extremely low-bit convolution optimization on ARMv8.1 specifically. Currently, there is no instruction available on ARMv8.1 architecture that can directly support low-bit (e.g., 2∼7-bit) multiply-accumulate operation heavily used in convolution. The existing lowest-bit multiply-accumulate instructions are 8-bit `SMLAL` and `MLA`. These two instructions both take in 8-bit input data, but generate 16-bit and 8-bit output, respectively. Therefore, we can leverage the above two instructions to realize extremely low-bit multiply-accumulate operation. In addition, the `SADDW` instruction supports accumulating 8-bit and 16-bit input data to 16-bit and 32-bit computation results, respectively. Therefore, we can use `SADDW` instruction to increase the bit width for the final results in 32-bit.

NVIDIA GPU is the most widely used computing platform for both training and deploying DNN models. NVIDIA Volta and Turing architectures introduce Tensor Cores, which can provide mixed-precision computing capabilities for DNN inference. Turing Tensor Cores support mixed-precision matrix multiplication for 1 bit, 4 bits and 8 bits. Developers can use Tensor Core through CUDA WMMA API, `mma` instructions in PTX ISA [27], and libraries such as cuBLAS and cuDNN. The fragments of WMMA API are transparent to developers, making it difficult to utilize data locality. Moreover, current libraries do not support 4-bit and 8-bit calculations with Tensor Core. Compared to WMMA API, `mma` instructions support more flexible sizes for matrix multiplication and enable programmers to manage registers explicitly. Therefore, we choose to use `mma` instructions to utilize Tensor Core.

The practical demands [8, 18, 34] motivate our selection of 2∼8-bit convolution for optimization. On ARM, the 8-bit instructions (e.g., `MLA`/`SMLAL`) support 8/16-bit output, which leaves optimization opportunities for wider bit range of 2∼8-bit. Whereas on GPU, the natively supported 4/8-bit instructions (e.g., `mma.m8n8k32`/`mma.m8n8k16`) accumulate results to 32-bit registers, which leaves no room for performance optimization other than 4/8-bit convolution.

3 OPTIMIZATION METHODS ON ARM CPU

3.1 Optimization Consideration

Although there are libraries and frameworks, such as gemmlowp and ncnn, which have implemented 8-bit optimization with 16-bit `SMLAL` instruction, to provide efficient implementation towards extremely low-bit convolution (e.g., below 8-bit), there are still challenges to address on ARM CPU.

Re-designing and optimizing GEMM. On ARM CPU, the load instruction is much slower than arithmetic instruction. To improve the arithmetic intensity of GEMM-based convolution, we need to re-design the GEMM algorithm to perform more arithmetic calculations per load instruction. In addition, we also need to apply data padding and packing optimization to enable continuous data access for better performance.

Determining appropriate low-bit instructions. The ARM CPU provides no available instructions to support extremely low-bit convolution. Instead, we need to utilize 8-bit (e.g., `SMLAL` and `MLA`) and 16-bit (e.g., `SADDW`) instructions for realizing convolution below 8-bit. In addition, we need to consider the numerical range of the quantized data to determine the appropriate instruction schemes for efficient low-bit convolution.

Optimizing register allocation. On ARM CPU, the overhead of data movement between registers is much smaller than between cache and registers. With such a performance feature, using the registers efficiently can reduce the number of cache accesses, and thus increase the performance of low-bit convolution. Therefore, we need to optimize the register allocation method, specially tailored for different low-bit instruction schemes.

Applying winograd algorithm. The core computation of winograd convolution is still the multiplication and addition. However, before the multiply-add calculation, the *input* and the *weight* of the convolution are applied with two linear transformations, which increases the numerical range of *input* and *weight*. Therefore, we can only apply winograd algorithm for further optimizing the convolution on a limited range of low bits (e.g., 4 to 6-bit).

3.2 Re-desgining GEMM Computation

Re-desgining GEMM algorithm. The GEMM-based convolution is an effective method to convert direct convolution to GEMM, and we optimize it on ARM CPU. However, traditional GEMM requires a large number of load instructions. We re-design the GEMM algorithm to reduce the load instructions and improve the computation intensity for better performance.

Fig. 1 (a) shows a traditional GEMM, where A and B are $M \times K$ and $K \times N$ matrices, respectively, and C is a $M \times N$ matrix. The re-designed GEMM algorithm is shown in Fig. 1 (b). First, we allocate *Buffer A*, *Buffer B* and *Buffer C* in registers. When traversing the common dimension K, we read data from column k of *Matrix A* ($0 <= k < K$) into *Buffer A* as v_a, and read data from row k of *Matrix B* ($0 <= k < K$) with each element replicated M times into *Buffer B* as v_{b_i}. Then we perform element-wise multiplication between v_a and each vector in v_{b_i} to obtain a $M \times N$ temporary matrix stored in *Buffer C*, and accumulate the data in *Buffer C* into *Matrix C*. By traversing the dimension K and accumulating all temporary results, we can derive *Matrix C*.

Here, we briefly analyze the number of load and arithmetic instructions required by the traditional and our re-designed GEMM. We define θ_1 as the amount of data that a single SIMD instruction can operate on, β_1 represents the two load instructions reading the two matrices, and β_2 represents the number of multiply-accumulate instructions required to operate on two SIMD registers. LD is the number of load instructions, CAL is the number of arithmetic instructions, and δ is the number of reduced sum instructions (constant number), which is much smaller than K. For traditional GEMM in Fig. 1 (a), LD and CAL is calculated using Eq. 1 and Eq. 2 respectively.

$$LD = \beta_1 \times \frac{M \times N \times K}{\theta_1} \quad (1)$$

$$CAL = \beta_2 \times \frac{M \times N \times K}{\theta_1} + \beta_2 \times \frac{M \times N}{\theta_1} \times \delta \approx \beta_2 \times \frac{M \times N \times K}{\theta_1} \quad (2)$$

Whereas for the re-designed GEMM in Fig. 1 (b), θ_2 represents the maximum number of elements operated by a single load-replicate instruction (e.g., 4 with `LD4R`). LD and CAL is calculated using Eq. 3 and Eq. 4 respectively.

$$LD = \beta_1 \times \frac{M \times N \times K}{\theta_2 \times \theta_1} = \beta_1 \times \frac{M \times N \times K}{4 \times \theta_1} \quad (3)$$

$$CAL = \beta_2 \times \theta_2 \times \frac{M \times N \times K}{\theta_2 \times \theta_1} = \beta_2 \times \frac{M \times N \times K}{\theta_1} \quad (4)$$

As we can see, the re-designed GEMM requires much few load instructions than traditional GEMM, and the $\frac{CAL}{LD}$ is about 4× of traditional GEMM. The above advantage enables the re-designed GEMM to achieve better performance on ARM CPU.

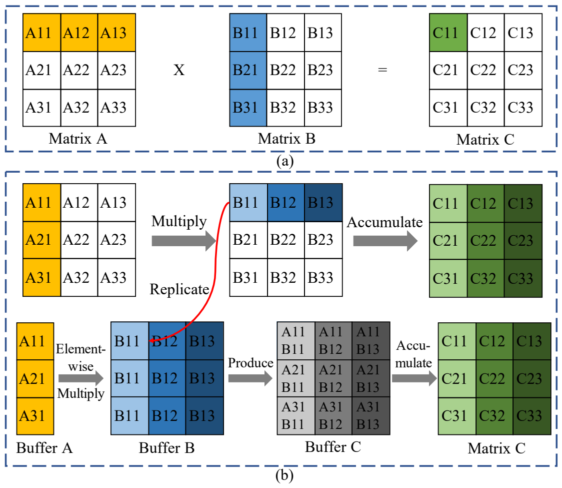

Figure 1: The GEMM computation, (a) traditional GEMM and (b) our re-designed GEMM.

Data padding and packing optimization. To optimize data access of the GEMM kernel, we take n_a elements from the k-th column of *Matrix A* and n_b elements from the k-th row of *Matrix B*. When M is not a multiple of n_a, and N is not a multiple of n_b, it is necessary to pad the matrix for better performance (e.g., zero padding). Taking Fig. 2 for example, *Matrix A* and *Matrix B* are both 3×3 matrices. Assuming n_a and n_b is set to 4, and both *Matrix A* and *Matrix B* are in row-major order, we need to first zero padding both matrices to a multiple of 4, and then perform data packing to allow continuous data access. As shown in Fig. 2, we allocate *Buffer A* and *Buffer B*, and copy n_a elements from *Matrix A* into *Buffer A* in column-major order each time, and copy n_b elements from *Matrix B* into *Buffer B* in row-major order each time. After the data padding and packing optimization, continuous data access can be achieved for improving the performance of matrix multiplication addition kernel.

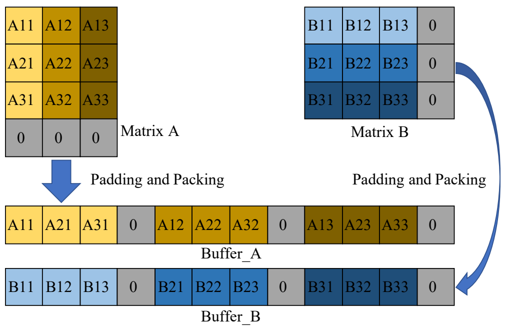

Figure 2: Illustrative example for data padding and packing optimization.

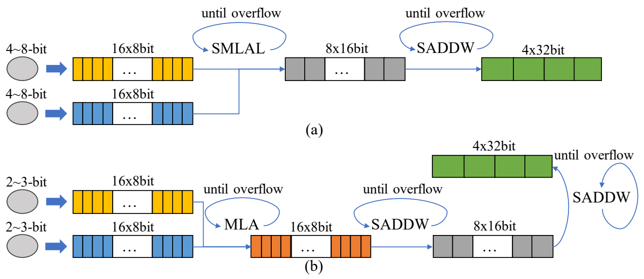

Figure 3: The optimized instruction schemes for 4~8-bit GEMM (a), and for 2~3-bit GEMM (b).

3.3 Instruction and Register Allocation Optimization

Optimized instruction schemes for GEMM. We propose two instruction schemes for optimizing the low-bit GEMM as shown in Fig. 3. For 4 to 8-bit GEMM, we choose SMLAL and SADDW instructions, and for 2 to 3-bit GEMM, we choose MLA and SADDW instructions. As shown in Fig. 3 (a), 4 to 8-bit input data is loaded into 8-bit registers. The SMLAL instruction multiplies the data in 8-bit registers and accumulates the results to a 16-bit register. The above process continues until the data in 16-bit register saturates. Then, the SADDW instruction accumulates data in a 16-bit register to a 32-bit register for the final results. To achieve better performance, we need to control the ratio of SMLAL to SADDW instruction for higher computation intensity during the low-bit GEMM. Consider two b-bit signed numbers, the SMLAL instruction needs to be executed $\frac{2^{16-1}-1}{(-2^{b-1})^2}$ times before transferring the result to a 16-bit register by SADDW. For 8-bit data, we adjust its value range to [-127,127], so that two SMLAL instructions are executed before applying SADDW instruction. Similarly, we can adjust the data range of other low-bit data. In our implementation, for 4, 5, 6, 7 and 8-bit GEMM, the ratio of SMLAL (8-bit register) to SADDW (16-bit register) instruction is 511/1, 127/1, 31/1, 8/1 and 2/1, respectively. Based on the above quantitative analysis, our instruction scheme achieves better performance towards lower-bit GEMM. Fig. 3 (b) shows the instruction scheme for 2 to 3-bit GEMM. The 2 to 3-bit input data is first loaded into 8-bit registers, and then MLA instruction multiplies the data in 8-bit registers and accumulates results into an 8-bit register. The above process continues until the data in the 8-bit register saturates. We control the ratio of MLA (8-bit register) to SADDW (16-bit register) as 31/1 and 7/1 for 2 and 3-bit GEMM, respectively. A similar trend is also observed that our instruction scheme works better with lower-bit GEMM.

Register allocation optimization. We also propose register allocation optimizations to accommodate different instruction schemes for low-bit GEMM. For 4 to 8-bit GEMM, v_0 ~v_1 are used to read *Matrix A*, v_2 ~v_9 are used to read *Matrix B*, v_{10} ~v_{17} are used to store temporary 16-bit results, and v_{18} ~v_{31} and x_0 ~x_3 are used to store the final 32-bit results. Alg. 1 shows the detailed process of GEMM kernel for 4 to 8-bit data, assuming n_a and n_b equals to 16 and 4 respectively during the data padding and packing optimization. The LD1 instruction fetches 16 8-bit data into a single register (line 3 and line 6) each time. The LD4R instruction fetches 4 8-bit data to 4 registers each time and replicates the data 16 times in each register (line 4 and line 7). We put v_0 and v_2 ~v_5 into a group, and v_1 and v_6 ~v_9 into another group. Both groups can get data from *Matrix A* and *B* and accumulate the results to v_{10} ~v_{17} (line 5 and line 8). In addition, we interleave the {LD1, LD4R} and SMLAL (2) instructions (line 3-8) for realizing data prefetching. To further expose instruction-level parallelism, we apply loop unrolling to loop K with unrolling_factor set to 32, 24, 16, 8 and 2 for 4, 5, 6, 7 and 8-bit GEMM, respectively. For 2 to 3-bit GEMM, we choose the *MLA* instruction, that exhibits twice computation throughput than SMLAL instruction. Due to the much better performance with MLA instruction, we use a simpler register allocation mechanism in this scenario, In the simplified mechanism, v_0 ~v_3 are used to read *Matrix A*, v_4 ~v_7 are used to read *Matrix B*. The MLA instruction multiplies the input data and accumulates results to v_8 ~v_{11}, which are used to store temporary 8-bit results. SADDW instruction accumulates data in v_8 ~v_{11} into v_{12} ~v_{19}, which store temporary 16-bit results. Then the SADDW instruction continues to accumulate 16-bit results to final 32-bit register in v_{20} ~v_{31} and x_0 ~x_7.

3.4 Winograd Optimization

Winograd fast convolution algorithm can be applied to the smaller convolution kernel for acceleration. In this paper, we choose to implement $F(2 \times 2, 3 \times 3)$ winograd algorithm, which is suitable for convolution calculation with a kernel size of 3 and stride of 1, and can calculate 2×2 results at one time.

As showed in Eq. 5, the winograd algorithm uses three transformation matrices G, B and A, where G applies to the *weight* g and B apply to the *input* data d. The transformed GgG^T and $B^T dB$ can be calculated directly by element-wise multiplication. After A^T and A transformation, the convolution results can be obtained.

$$Y = A^T [[GgG^T] \odot [B^T dB]]A \tag{5}$$

In the winograd algorithm with kernel size of $F(2 \times 2, 3 \times 3)$, the transformation of matrix G applied to the *weight* will increase the numerical range of *weight* by 9/4 times. Similarly, the numerical range of *input* will increase by 4× after the transformation of matrix B. For the $F(2 \times 2, 3 \times 3)$ winograd, in order to ensure that the data is still within the range of 8-bit precision, we limit the *activation*

Algorithm 1 The 4~8-bit GEMM kernel with register allocation optimization

```
Input: Padding_and_Packing { Matrix A and Matrix B }
1:  while $k > 0$ do
2:      ...
3:      LD1 { $v_0$ } addr_Matrix_A
4:      LD4R { $v_2$ ~$v_5$ } addr_Matrix_B
5:      SMLAL(2) { $v_{10}$ ~$v_{17}$ } { $v_1$ } { $v_6$ ~$v_9$ }
6:      LD1 { $v_1$ } addr_Matrix_A
7:      LD4R { $v_6$ ~$v_9$ } addr_Matrix_B
8:      SMLAL(2) { $v_{10}$ ~$v_{17}$ } { $v_0$ } { $v_2$ ~$v_5$ }
9:      ...
10:     MOV { $v_0, v_1$ } { { $x_0, x_1$ }, { $x_2, x_3$ } }
11:     SADDW(2) { $v_{18}$ ~$v_{31}$ } { $v_{10}$ ~$v_{16}$ }
12:     SADDW(2) { $v_0, v_1$ } { $v_{17}$ }
13:     MOV { { $x_0, x_1$ }, { $x_2, x_3$ } } { $v_0, v_1$ }
14:     $k \leftarrow k - unrolling\_factor$
15: end while
16: MOV { $v_0, v_1$ } { { $x_0, x_1$ }, { $x_2, x_3$ } }
17: ST1 { { $v_{18}$ ~$v_{31}$ }, { $v_0, v_1$ } } addr_Matrix_C
```

and *weight* to no more than 6 bits. And we do not apply winograd algorithm with $F(4 \times 4, 3 \times 3)$, due to the unacceptable increment of numerical range after G and B transformation. In addition, considering that the maximum speedup of $F(2 \times 2, 3 \times 3)$ winograd algorithm is 2.25× theoretically without taking linear transformation overhead into account. In the meanwhile, MLA instruction (used in GEMM-based convolution) is 2× faster than SMLAL instruction (used in winograd-based convolution), which offsets the performance advantage of winograd at 2 to 3-bit convolution. Therefore, we focus on applying $F(2 \times 2, 3 \times 3)$ on 4 to 6-bit convolution.

4 OPTIMIZATION METHODS ON NVIDIA GPU

4.1 Optimization Consideration

To achieve extremely low-bit convolution efficiently on GPU, we need to address the following challenges regarding GPU computation and memory hierarchies.

Adapting computation to GPU thread organization. The threads on GPU are organized at different granularities such as grid, block, and warp. In order to fully utilize GPU's computation resources, especially the powerful Tensor Cores, we need to design a data partition mechanism adapting the calculation of GEMM to the thread organization efficiently.

Optimizing data access across memory hierarchy. There are various memory layers on GPU that differ in terms of latency, bandwidth and capacity. It is essential to optimize the data access pattern to exploit the characteristics of memory hierarchy for higher memory bandwidth and lower memory latency. Besides, adopting software prefetching during GEMM is also essential to hide the long latency of global memory access.

Eliminating intermediate results. Eliminating the need for storing the intermediate results into global memory on GPU can deliver significant performance benefits for consecutive calculations in quantized neural networks. We exploit several ways of quantization fusion across different computation kernels for better performance.

4.2 Data Partition along with Thread Hierarchy

After converting the convolution to GEMM, we apply a data partition mechanism to assign the calculation to the GPU execution units at different thread hierarchy. Our implementation of implicit-precomp GEMM-based convolution kernel is shown in Alg. 2. We store the offsets of elements instead of the pointers in the precomputed buffer. The advantage is that the offset calculation process only needs to be done once for a specific shape and can be regarded as the pre-processing of the kernel. When loading the data of matrix *A*, we first get the offset of the data from the precomputed buffer and then load it directly from the input array.

Algorithm 2 Implicit-precomp GEMM-based Conv2D

```
Input: Shape of convolution and pointers of input, weight and
    output. The precomputed buffer.
Tiling Parameters: MTile, NTile, KTile, KStep, blockRowWarpNum and blockColWarpNum.
1:  compute KTileNum, KStepNum, MFrag, NFrag, warpRowNum
    and warpColNum;
2:  for k_outer in KTileNum do
3:      load A_Tile to shared memory by precomputed buffer;
4:      load B_Tile to shared memory;
5:      __syncthreads();
6:      for k_inner in KStepNum do
7:          load A_Fragment to register;
8:          load B_Fragment to register;
9:          for row in WarpRowNum do
10:             for col in WarpColNum do
11:                 compute C_Fragment by mma instruction;
12:             end for
13:         end for
14:     end for
15:     add bias and re-quantize on register;
16:     store C_Fragment to global memory;
17: end for
```

We propose a data partition mechanism that partitions the matrices at grid-level, block-level and warp-level and stores the sub-matrices on global memory, shared memory and registers correspondingly, as shown in Fig. 4. At the grid-level, we divide the matrix C into tiles with parameters *MTile* and *NTile*, and assign each tile to one thread block. Similarly, the matrix A and B are also partitioned based on parameter *KTile*. At the block-level, we partition the sub-matrix C_Tile into fragments based on parameters *blockRowWarpNum* and *blockColWarpNum*, which represents the number of rows and columns of the warps in the thread block, respectively. Then we assign each fragment to one warp. To avoid the value of *KTile* is too large, which prevents the matrix A_Fragment and B_Fragment from being accommodated in the warp due to insufficient registers, we use the factor *KStep* to split *KTile* to fit in the warp. At the warp-level, each warp executes on Tensor Core through mma instruction to perform the matrix multiplication (line 9~13 in Alg. 2).

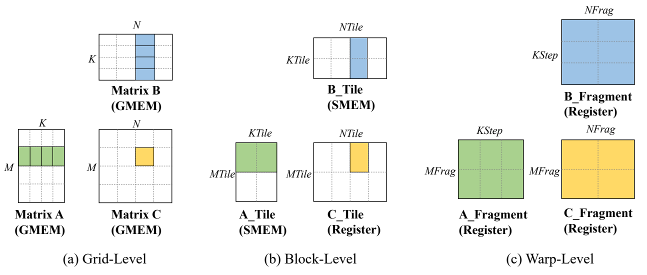

Figure 4: Data partition mechanism along the thread hierarchy on GPU. *GMEM* and *SMEM* represent global memory and shared memory, respectively.

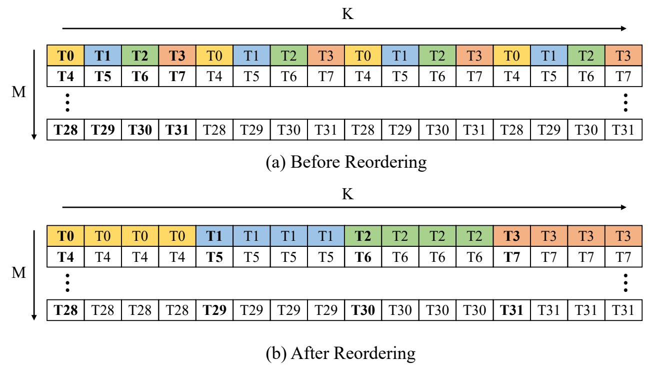
K
M
T0 T1 T2 T3 T0 T1 T2 T3 T0 T1 T2 T3 T0 T1 T2 T3
T4 T5 T6 T7 T4 T5 T6 T7 T4 T5 T6 T7 T4 T5 T6 T7
T28 T29 T30 T31 T28 T29 T30 T31 T28 T29 T30 T31 T28 T29 T30 T31
(a) Before Reordering
K
M
T0 T0 T0 T0 T1 T1 T1 T1 T2 T2 T2 T2 T3 T3 T3 T3
T4 T4 T4 T4 T5 T5 T5 T5 T6 T6 T6 T6 T7 T7 T7 T7
T28 T28 T28 T28 T29 T29 T29 T29 T30 T30 T30 T30 T31 T31 T31 T31
(b) After Reordering

Figure 5: Reordering memory access of matrix A on shared memory. The blocks in bold font represent the data accessed by the first `mma` instruction.

4.3 Multi-level Memory Access Optimization

We propose multi-level memory access optimizations, including coalesced access on global memory, reordering access on shared memory, overlapped computation, and data access using registers and in-place calculation. These optimizations reduce the data access latency across layers of GPU memory.

Coalesced access on global memory. To optimize access to global memory, we implement coalesced access by using appropriate vector types to minimize memory transactions and maximize the throughput of global memory. We allow each thread access consecutive 16 bytes to achieve coalesced access, where the memory requests in the warp is divided into four independent 128-bytes memory requests for four quarter-warps. In addition, we use built-in vector types in CUDA, such as *int4* and *int2*, to implement automatic alignment and coalesced access. Since matrix A and B are read-only, we further improve the performance of data access by using read-only cache with *__ldg*() function.

Reordering memory access on shared memory. We reorder the access pattern of matrix A and B on shared memory to increase continuous data accesses of each thread. Here, we take 8-bit data with `mma8816` instruction for example. For this instruction, the matrix A is accessed by thread 0~31 through the data blocks (in bold font) T0~T31 as shown in Fig. 5 (a), where each data block is 4 bytes.

The common approach is that each thread performs stride memory access with four 4-bytes data blocks, as shown in Fig. 5 (a). Each thread needs four `LDS.32` instructions to perform the access. Our optimization is to adjust the access order of threads so that each thread achieves continuous memory access, as shown in Fig. 5 (b). In this case, each thread accesses 4 blocks consecutively with a size of 16 bytes. And each thread only needs one `LDS.128` instruction to perform the access, and the number of access instructions is reduced to one-quarter of the original. The same reordering is also applied to matrix B to guarantee the correct results.

Overlapped computation and memory access using registers. We utilize register to overlap `mma` calculation and global memory access. Specifically, we set up a temporal buffer of registers to prefetch the data required for the next iteration, as shown in Fig. 6. At the beginning of each iteration, the data of A_Tile and B_Tile in this iteration is already in the temporal buffer, so we load the data directly from the temporal buffer to shared memory (II). Then each warp loads the data of A_Fragment and B_Fragment from shared memory to register and executes `mma` calculation (III and IV). In the meantime, the data in the temporary buffer is no longer needed so that we can load the data required for the next iteration from global memory in advance (I). With the help of the temporal buffer using registers, we overlap memory access to global memory (I) with `mma` calculation (IV), and thus improve the computation performance.

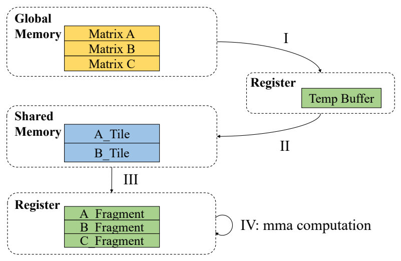

Figure 6: The overlapped computation and data access using register buffer on GPU.

In-place calculation of bias and re-quantization. To further optimize memory access, we perform the in-place calculation of bias and re-quantization on registers. After finishing the `mma` calculation as shown in Alg. 2 (line 2~14), we directly apply bias and re-quantization on the temporary data generated by `mma` calculation (line 15), instead of writing it back to global memory first. After the in-place calculation, due to the reduced data type from *int32* to *int8*, the amount of global memory accesses used to store C_Fragment also reduces.

4.4 Quantization Fusion

To take advantage of computation fusion for better performance, we exploit two ways of fusion in quantized neural networks. Generally, the quantized neural network adds more layers around the convolution layer. For example, the original convolution layer becomes: quantization -> convolution (+re-quantization) -> dequantization -> quantization -> ReLU -> dequantization. We apply the quantization fusion regarding the above layer sequence.

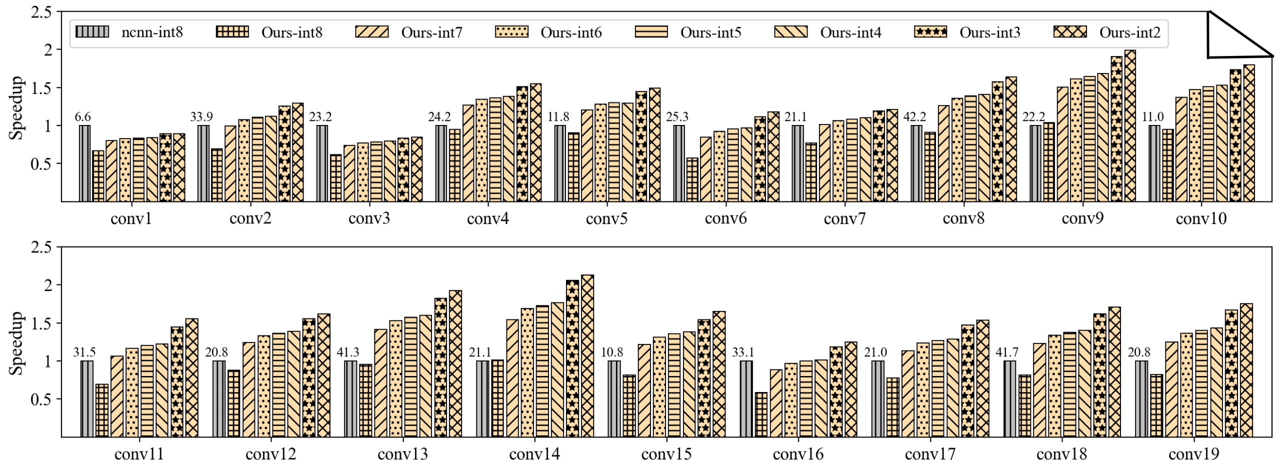

Figure 7: The performance comparison between our optimized 2∼8-bit convolution kernels and ncnn 8-bit convolution kernel (baseline) on Raspberry Pi 3B. The time above the first bar of each layer is the absolute execution time (ms) in baseline.

Fusion of convolution and dequantization. We combine the calculation process of convolution and dequantization, skip storing the intermediate results with *int8* data type, and directly transform the results from *int32* to *fp32*. The fusion not only reduces accesses to global memory but also reduces the overhead of launching multiple kernels.

Fusion of convolution and ReLU. Fusing the convolution and ReLU kernels in quantized neural network enables us to skip the dequantization and quantization calculations between the two kernels, which eliminates the overhead of unnecessary computation and memory access. We can fuse convolution and ReLU kernels by changing the truncated range of re-quantization in convolution kernel.

5 EVALUATION

5.1 Experiment Setup

We evaluate our implementation of extremely low-bit convolution kernels on Raspberry Pi 3B and NVIDIA RTX 2080Ti. The specific hardware and software configurations are shown in Tab. 1. We evaluate all convolution layers in ResNet-50 [11] adopted from Caffe Model Zoo. In addition, we also evaluate representative and non-repetitive convolution layers from SCR-ResNet-50 (convolution layers with different shapes from ResNet-50 [11]) searched by CRNAS [19] and DenseNet-121 [12]. We use the NCHW layout on ARM CPU and the NHWC layout on NVIDIA GPU. We choose the 8-bit convolution kernels from ncnn and cuDNN as the baselines on ARM CPU and NVIDIA GPU, respectively. Currently, cuDNN does not support the 8-bit convolution with Tensor Core. Thus we choose the 8-bit kernels with dp4a in cuDNN as the baseline on GPU. In addition, we also compare with 8-bit convolution in TensorRT through execution profile with *trtexec* tools. The above frameworks and libraries represent the state-of-the-art implementation of low-bit convolution to our knowledge. To determine the optimal tiling parameters in the data partition mechanism on GPU through auto-search, we use C++ template to generate multiple kernels with different combinations of tiling parameters and choose the best ones through profile runs. All experiments in Section 5.3 use the optimal tiling parameters by default. Note that the optimal tiling parameters only need to be determined once per convolution shape with negligible overhead. Note that our optimizations do not affect the accuracy of the model, which can be guaranteed from two aspects: *1) model quantization*, which has been proven by existing studies [7, 18, 34] with slight accuracy loss using low-bit linear quantization (e.g., 2∼4-bit). We apply the same quantization scheme and thus can directly enjoy their accuracy results; *2) low-bit convolution*, which has been implemented with overflow well addressed. Therefore, our optimized low-bit convolution kernels guarantee the same results as 32-bit computation. In sum, there is no accuracy loss for the interest of this paper.

Table 1: Hardware and software configurations.

Platform	ARM CPU	NVIDIA GPU
Device	Raspberry Pi 3B	RTX 2080Ti
Architecture	ARM Cortex-A53	NVIDIA Turing TU102
Software	Ubuntu 16.04 LTS for Raspberry Pi, gcc 5.4.0, ncnn with commit 6f2ef19	Ubuntu 16.04 LTS, gcc 5.4.0, CUDA 10.2, cuDNN 7.6.5, TensorRT 7

5.2 Performance Improvement on ARM CPU

On Raspberry Pi 3B, we evaluate the performance of 2∼8-bit convolution kernels. Fig. 7 shows the performance comparison between our implementation and ncnn. We choose the batch size to 1 due to the limited computation and memory resources on edge devices such as ARM CPU, which is commonly adopted for performance evaluation in existing works [31, 32]. The x-axis indicates the convolution layer, and the y-axis shows the speedup compared to the baseline. As shown in Fig. 7, the highest speedups achieved by our implementations from 2∼7-bit are 2.13×, 2.06×, 1.76×, 1.73×, 1.69× and 1.54×, all of them appear in conv14 layer. The highest speedup of our 8-bit implementation is 1.04× in conv9 layer. The performance of our optimized 2∼8-bit convolution kernels exceeds ncnn in 17, 17, 16, 15, 15, 14 and 2 out of 19 layers, respectively. In the layers with better performance, our 2∼8-bit implementations achieve average speedups of 1.60×, 1.54×, 1.38×, 1.38×, 1.34×, 1.27× and 1.03×, respectively.

We notice that for 8-bit implementation, our optimization achieves lower performance compared to ncnn for most of the cases. The reason can be explained as follows. In ncnn, it stores the 8-bit input into a 16-bit register, and uses 16-bit `SMLAL` instruction to compute and accumulate the result to a 32-bit register. In our optimization, we can apply two 8-bit `SMLAL` instructions to compute and accumulate in a 16-bit register, and then apply one 16-bit `SADDW` instruction to accumulate in a 32-bit register. Since our optimization relies on the high ratio of `SMLAL` to `SADDW` instructions (e.g., with lower-bit input), and there is little room to perform more `SMLAL` instructions with 8-bit input (due to data overflow), it constrains the performance speedup of our optimization.

In addition, our optimized kernels show inferior performance on two layers (e.g., conv1 and conv3) across all low-bits in Fig. 7. This is because the convolution size of these layers is the smallest (1x1 kernel with 64 channels) compared to other layers. After applying the matrix blocking for GEMM, the small block size constrains the performance speedup due to the limited computation intensity.

As shown in Fig. 8, we compare the performance of our winograd optimized convolution kernels with GEMM-based convolution kernels and 8-bit convolution kernel in ncnn (baseline). Due to the restriction of winograd algorithm (convolution kernel (3 × 3) with the stride of 1), we show the performance results for all the layers in ResNet-50 where winograd can be applied. The time above the first bar of each layer is the absolute execution time (ms) in the baseline. The performance of 4∼6-bit winograd implementations outperforms the baseline and GEMM-based implementations in all cases. The 4∼6-bit winograd implementations achieve a maximum speedup of 1.73×, 1.66× and 1.52×, and an average speedup of 1.50×, 1.44× and 1.34×, respectively.

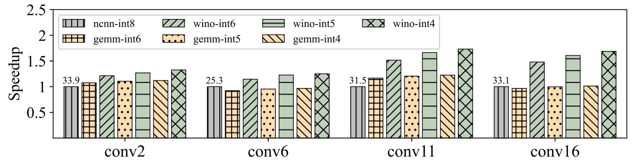

Figure 8: The performance comparison of our GEMM-based and winograd-based convolution kernels at 4∼6-bit input on Raspberry Pi 3B.

We also compare the performance of our GEMM-based convolution kernel at 2-bit with TVM's popcount-based solution [3] (baseline), as shown in Fig. 9. For the 2-bit convolution, both input data and weights are stored in 2-bit (A^2W^2). For each convolution layer implemented in TVM, we enable performance auto-tuning for 100 trials according to [3]. The time above the first bar of each layer is the absolute execution time (ms) in the baseline. Our 2-bit implementation outperforms TVM in most cases (16 out of 19 cases), with the highest speedup of 2.11× (in conv11). For the cases our implementation surpasses TVM, the average speedup is 1.78×.

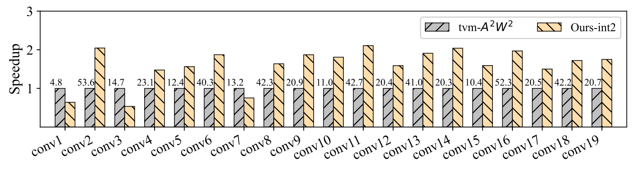

Figure 9: The performance comparison of our implementation with TVM at 2-bit convolution (A^2W^2) on Raspberry Pi 3B. The performance of TVM is chosen as baseline.

5.3 Performance Improvement on NVIDIA GPU

We evaluate the performance of our optimized 4-bit and 8-bit convolution kernels in the batch size of 1 and 16 on NVIDIA GPU. Fig. 10 shows the performance comparison between our implementations and cuDNN on RTX 2080Ti. With the batch size of 1, the performance of our 4-bit and 8-bit convolution kernels exceed cuDNN in 18 out of 19 layers by an average speedup of 5.26× and 4.31×, respectively. With the batch size of 16, our 4-bit and 8-bit convolution kernels outperform cuDNN in 17 and 16 out of 19 layers by an average speedup of 3.45× and 2.44×, respectively. In most cases, our 4-bit (17 out of 19 layers) and 8-bit (15 out of 19 layers) convolution kernels outperform 8-bit convolution kernels in TensorRT by an average of 1.78× and 1.44× with the batch size of 1, respectively. With the batch size of 16, our 4-bit kernels also outperform TensorRT in 12 layers by an average speedup of 1.46×. These results demonstrate the effectiveness of our extremely low-bit convolution optimization, especially in small batch sizes. In addition, our 4-bit convolution kernels outperform 8-bit convolution kernels by 1.18× and 1.32× on average with the batch size of 1 and 16, respectively. The results reveal that our optimizations are more effective towards lower bit convolution.

For the speedup of 8-bit implementation compared to cuDNN, we believe the speedup comes from the utilization of Tensor Core in our implementation. Whereas compared to TensorRT, our kernels achieves similar performance. After a thorough investigation of convolution kernels optimized by our approach and TensorRT with NVIDIA Nsight Compute kernel profiler [25], we have the following observations. For the cases where our implementations achieve better performance than TensorRT, we observe higher memory bandwidth utilization with our approach. Since the TensorRT implementation is proprietary, our best guess for the reason is that our data partition scheme along with the auto-search for optimal tiling size is more effective in optimizing GPU memory access compared to TensorRT (especially with small batch size). Whereas for the cases TensorRT achieves better performance, we observe higher instruction-per-cycle and SM utilization compared to our approach. The reason could be TensorRT has applied many low-level optimizations with heavily-tuned SASS code.

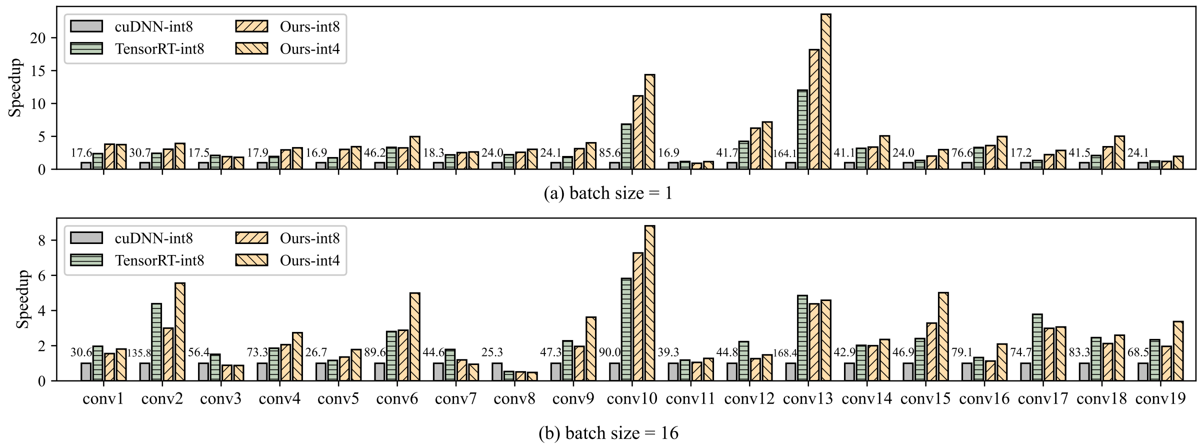

Figure 10: The performance comparison of our 8-bit and 4-bit convolution kernels and 8-bit convolution kernels of cuDNN and TensorRT on RTX 2080Ti GPU. The 8-bit convolution with dp4a instruction in cuDNN is chosen as baseline. The time above the first bar of each layer is the absolute execution time (us) in baseline.

We also notice that our implementation achieves better speedup with small batch size (e.g., batch size = 1). This is because, with small batch size, the tiling size has a significant impact on performance, where our auto-search method with profile runs is more effective in determining the optimal tiling size, thus improving both memory access efficiency and thread-level parallelism. Whereas for large batch size, the performance of calculation becomes dominant, where the low-level optimizations (e.g., SASS code) applied by TensorRT can better utilize the hardware features to achieve higher performance.

Fig. 11 shows the performance improvement after using profile runs to determine the optimal tiling parameters under the batch size of 1. The *w/o profile* and *w/ profile* indicates the performance using default parameters and optimal parameters, respectively. The default parameters are usually selected based on programmer experience. The average speedup of 4-bit and 8-bit convolution kernels with the profile runs enabled is 2.29× and 2.91×, respectively. The performance results with the batch size of 16 show similar tendency, which are omitted for brevity.

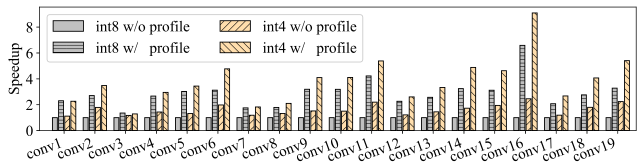

Figure 11: The performance improvement after profile runs on RTX 2080Ti GPU (batch size = 1). The baseline is the 8-bit implementation without profile runs.

Moreover, we evaluate the performance improvement of quantization fusion, including the fusion of convolution and dequantization and fusion of convolution and ReLU, as shown in Fig.12. The experiments use 8-bit convolution kernels with the batch size

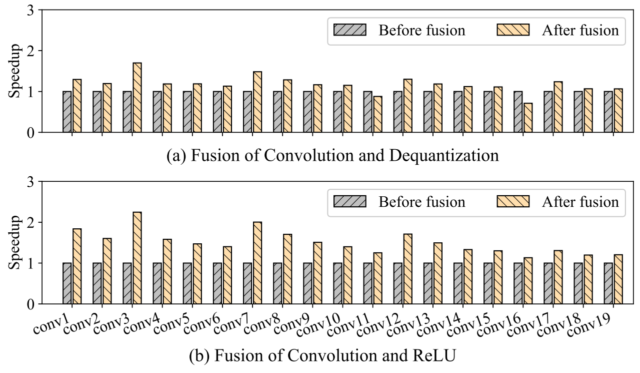

Figure 12: The performance improvement of quantization fusion on RTX 2080Ti GPU (batch size = 1). The baseline is the implementation without quantization fusion.

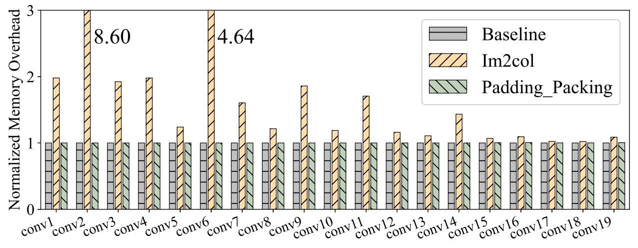

Figure 13: The space overhead of each layer for ResNet-50 after applying im2col, data padding and packing operations. The baseline is space occupation of activation and weight for each layer.

of 1. Among all cases, the fusion of convolution and dequantization achieves a 1.18× speedup on average, whereas the fusion of

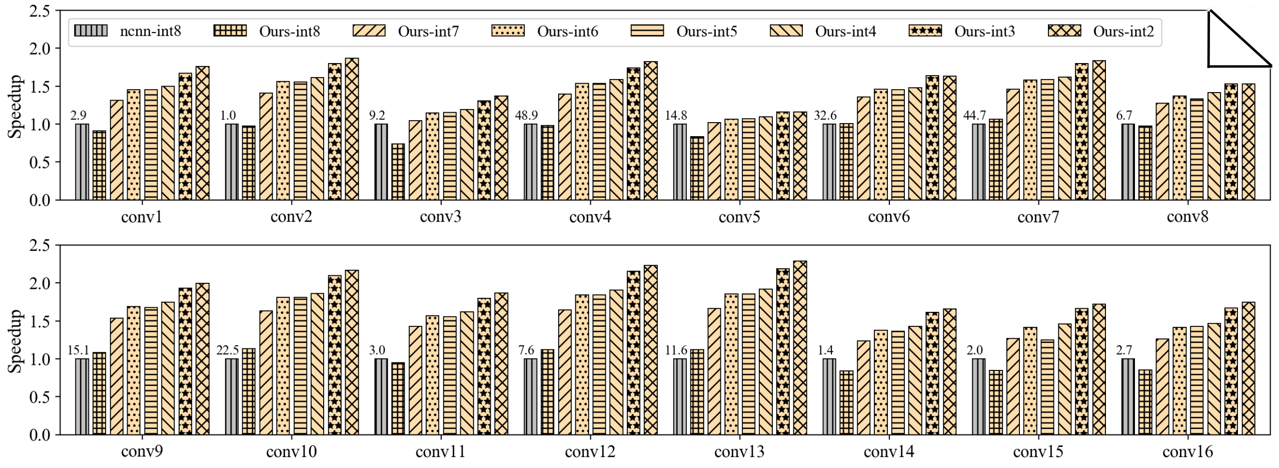

Figure 14: The performance comparison between our optimized 2~8-bit convolution kernels and ncnn 8-bit convolution kernel (baseline) in DenseNet-121 on Raspberry Pi 3B. The time above the first bar of each layer is the absolute execution time (ms) in baseline.

convolution and ReLU achieves a 1.51× speedup on average. The performance results with the batch size of 16 show similar tendency, which are omitted for brevity.

5.4 Discussion of Space Overhead

On GPU, we use the implicit-precomp GEMM method, which avoids using global memory to store the transformed matrix. Instead, only the pre-computed buffer occupies few global memory spaces ranging from 0.5 KB to 50 KB, which is negligible. Besides, our optimization uses shared memory and registers for caching, which does not consume extra global memory space.

To understand the space overhead of our optimization on ARM, we have analyzed the space occupation after applying im2col, data padding and packing operations for each layer of ResNet-50. The baseline is the space occupation of activation and weight for each layer. Fig 13 shows the space overhead for each layer of ResNet-50 compared to the baseline. The minimum, maximum and average space overhead of *im2col* is 1.0218× (*conv18*), 8.6034× (*conv2*) and 1.9445×, respectively. The minimum, maximum and average space overhead of data padding and packing is 1.0× (*conv1~14*), 1.0058× (*conv2*) and 1.0010×, respectively. In total, the space overhead of our optimization on ARM ranges from 1.0232× to 8.6034×, with 1.9455× on average. Note that the space overhead of *im2col* is determined by convolution kernel size, stride, and input size, whereas the space overhead of data padding and packing is determined by the size of matrix generated through *im2col* and layer weight.

5.5 Applying to more CNN Models

To further demonstrate the applicability of our optimization, we have evaluated two more CNN models, including SCR-ResNet-50 and DenseNet-121, on both ARM CPU and NVIDIA GPU.

Fig. 14 and Fig. 15 show the performance comparison for the convolution layers in DenseNet-121 and SCR-ResNet-50 on ARM CPU, respectively. For DenseNet-121 in Fig. 14, our 2~7-bit kernels

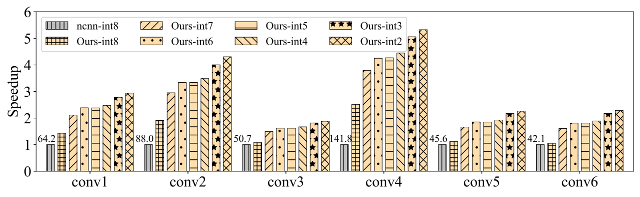

Figure 15: The performance comparison between our optimized 2~8-bit convolution kernels and ncnn 8-bit convolution kernel (baseline) in SCR-ResNet-50 on Raspberry Pi 3B. The time above the first bar of each layer is the absolute execution time (ms) in baseline.

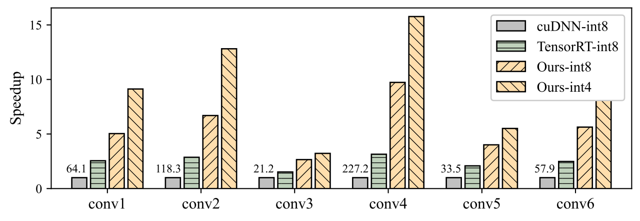

Figure 16: The performance comparison when applying our approach to optimize convolution kernels in SCR-ResNet-50 on RTX 2080Ti GPU (batch size = 1). The time above the first bar of each layer is the absolute execution time (us) in baseline.

achieve an average speedup of 1.79×, 1.74×, 1.56×, 1.50×, 1.51× and 1.37× compared to ncnn, respectively. For the 8-bit convolution kernels, our optimization exceeds ncnn in 6 out of 16 layers, with an average speedup of 1.09×. Whereas for SCR-ResNet-50 in Fig. 15, our 2~8-bit kernels outperform ncnn across all convolution layers,

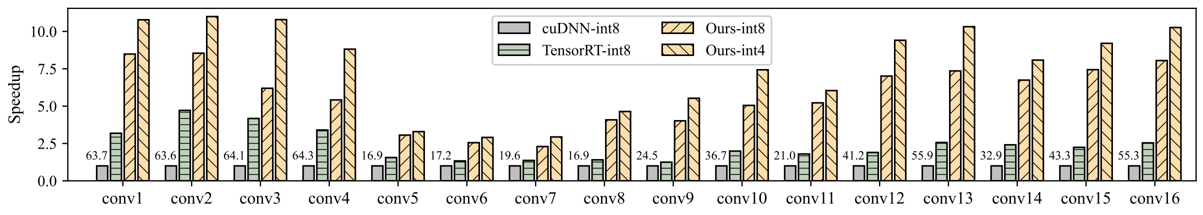

Figure 17: The performance comparison when applying our approach to optimize convolution kernels in DenseNet-121 on RTX 2080Ti GPU (batch size = 1). The time above the first bar of each layer is the absolute execution time (us) in baseline.

with an average speedup of 3.17×, 3.00×, 2.65×, 2.54×, 2.54×, 2.27× and 1.52×, respectively.

Fig. 16 and Fig. 17 show the performance comparison for the convolution layers in SCR-ResNet-50 and DenseNet-121 on NVIDIA GPU, respectively. Our 4-bit and 8-bit kernels outperform 8-bit kernels of TensorRT and cuDNN across all convolution layers. Specifically, compared to TensorRT, our 4-bit and 8-bit kernels achieve an average speedup of 3.53× and 2.22× for SCR-ResNet-50, and 3.29× and 2.53× for DenseNet-121. We observe that our optimization achieves better performance speedup on SCR-ResNet-50 and DenseNet-121 compared to ResNet-50. We believe the reason is that the convolution shapes in SCR-ResNet-50 and DenseNet-121 are not commonly used (e.g., input size for *conv15* in DenseNet-121 is $1 \times 14 \times 14 \times 736$), and thus out of the radar of TensorRT for heavy optimization (e.g., with SASS code). However, our optimization can determine the optimal tiling size through auto-search with profile runs, and better adapts to the unusual convolution shapes for higher performance speedup.

6 RELATED WORK

Low-bit computation optimization on ARM CPU. QNNPACK is a high-performance kernel library, providing efficient implementations for convolution, deconvolution and fully connected layers, which supports 8-bit quantization. ncnn is a high performance neural network inference framework. It provides the 8-bit convolution using the GEMM-based method and winograd method, and supports ARM Neon assembly-level optimization. gemmlowp is a low-precision GEMM library that provides efficient implementations on ARM with Neon and Intel x86 with SSE 4.1. Lai *et al.* [16] implements the CMSIS-NN kernels for low-bit neural network on the Cortex-M processor. However, none of the above work supports convolution kernels on ARM CPU below 8 bits.

Both Tulloch *et al.* [31] and Cowan *et al.* [3] use `popcount` instructions to optimize low-bit convolution kernels on ARM CPU. The former proposes a low-bit convolution implementation, and the latter presents an automated approach to generate high-performance convolution kernels. However, the `popcount` instructions are usually used to realize low-bit convolution below 4-bit and in the cases of higher bits, using `popcount` instruction cannot achieve better performance compared to 8-bit baseline [31], whereas our optimization methods can be applied to a wider range of low-bit convolution, covering 2~8-bit specifically.

Low-bit computation optimization on NVIDIA GPU. The cuDNN library provides highly tuned kernels for DNN, such as 8-bit convolution using `dp4a` instruction. TensorRT is an SDK for high performance deep learning inference on GPU, which supports fast *int8* inference. The above library and framework represent the state-of-the-art 8-bit convolution implementation, but none of them supports 4-bit convolution on GPU. CUTLASS [26] contains a collection of CUDA C++ templates for high performance GEMM, which supports low-bit computation with Tensor Core. But it doesn't optimize convolution kernels specifically. The MLPerf Inference Benchmark [28] contains the only public reported 4-bit convolution implementation in ResNet-50 on GPU, but it only provides a model binary file, which cannot be used individually and compared with our 4-bit implementation. In addition, there are research works using Tensor Core to accelerate low-bit computation in general. Markidis *et al.* [22] analyze the performance and errors of GEMM calculation using Tensor Core. Zhu *et al.* [35] propose an algorithm and hardware co-design for sparse neural networks on Tensor Core.

7 CONCLUSION

In this paper, we explore extremely low-bit convolution optimizations and provide efficient implementations on ARM CPU and NVIDIA GPU. On ARM CPU, for different bit width, we re-design the GEMM computation with data padding and packing optimizations. In addition, we propose different instruction schemes and corresponding register allocation methods for further optimizing the low-bit GEMM. We also implement winograd algorithm for accelerating the convolution at certain low-bit range. On NVIDIA GPU, we propose a data partition mechanism and multi-level memory access optimizations. We also implement quantization fusion to eliminate the overhead of storing intermediate data. Our evaluation demonstrates that our extremely low-bit convolution kernels archive significant speedups compared to the state-of-the-art implementations such as ncnn/TVM and cuDNN/TensorRT. In the future, we would like to integrate our low-bit convolution optimizations into deep learning frameworks such as TVM to enable end-to-end optimization as well as explore auto-tuning for better portability.

ACKNOWLEDGMENTS

This work was supported by SenseTime Research Fund for Young Scholars and National Natural Science Foundation of China (No. 61502019 and No. 61732002).

REFERENCES

[1] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. TVM:

an automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation*. 579–594.

[2] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759* (2014).

[3] Meghan Cowan, Thierry Moreau, Tianqi Chen, James Bornholt, and Luis Ceze. 2020. Automatic generation of high-performance quantized machine learning kernels. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*. 305–316.

[4] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 248–255.

[5] Jack J Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain S Duff. 1990. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software (TOMS)* 16, 1 (1990), 1–17.

[6] Marat Dukhan, Yiming Wu, and Hao Lu. 2018. QNNPACK: open source library for optimized mobile deep learning. https://github.com/pytorch/QNNPACK.

[7] Steven K. Esser, Jeffrey L. McKinstry, Deepika Bablani, Rathinakumar Appuswamy, and Dharmendra S. Modha. 2020. LEARNED STEP SIZE QUANTIZATION. In *International Conference on Learning Representations*.

[8] Ruihao Gong, Xianglong Liu, Shenghu Jiang, Tianxiang Li, Peng Hu, Jiazhen Lin, Fengwei Yu, and Junjie Yan. 2019. Differentiable soft quantization: Bridging full-precision and low-bit neural networks. In *Proceedings of the IEEE International Conference on Computer Vision*. 4852–4861.

[9] Song Han, Jeff Pool, John Tran, and William Dally. 2015. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*. 1135–1143.

[10] Babak Hassibi and David G Stork. 1993. Second order derivatives for network pruning: Optimal brain surgeon. In *Advances in neural information processing systems*. 164–171.

[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[12] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. 2017. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4700–4708.

[13] Intel. 2016. Deep Neural Network Library. https://github.com/intel/mkl-dnn.

[14] Benoit Jacob et al. 2017. gemmlowp: a small self-contained low-precision GEMM library.(2017).

[15] Heehoon Kim, Hyoungwook Nam, Wookeun Jung, and Jaejin Lee. 2017. Performance analysis of CNN frameworks for GPUs. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 55–64.

[16] Liangzhen Lai, Naveen Suda, and Vikas Chandra. 2018. Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus. *arXiv preprint arXiv:1801.06601* (2018).

[17] Andrew Lavin and Scott Gray. 2016. Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 4013–4021.

[18] Rundong Li, Yan Wang, Feng Liang, Hongwei Qin, Junjie Yan, and Rui Fan. 2019. Fully Quantized Network for Object Detection. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[19] Feng Liang, Chen Lin, Ronghao Guo, Ming Sun, Wei Wu, Junjie Yan, and Wanli Ouyang. 2019. Computation Reallocation for Object Detection. *arXiv preprint arXiv:1912.11234* (2019).

[20] Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, and Piotr Dollár. 2014. Microsoft COCO: Common Objects in Context.

[21] Weibo Liu, Zidong Wang, Xiaohui Liu, Nianyin Zeng, Yurong Liu, and Fuad E Alsaadi. 2017. A survey of deep neural network architectures and their applications. *Neurocomputing* 234 (2017), 11–26.

[22] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S Vetter. 2018. Nvidia tensor core programmability, performance & precision. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 522–531.

[23] Szymon Migacz. 2017. 8-bit inference with TensorRT. In *GPU Technology Conference*.

[24] nihui et al. 2017. NCNN. https://github.com/Tencent/ncnn.

[25] NVIDIA. 2019. NVIDIA Nsight Compute. https://developer.nvidia.com/nsight-compute.

[26] CUTLASS NVIDIA. 2017. CUDA Templates for Linear Algebra Subroutines. https://github.com/NVIDIA/cutlass.

[27] PTX NVIDIA. 2019. Parallel Thread Execution ISA version 6.5. *NVIDIA Corporation (November 2019)* (2019).

[28] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, et al. 2019. Mlperf inference benchmark. *arXiv preprint arXiv:1911.02549* (2019).

[29] Günther Schindler, Manfred Mücke, and Holger Fröning. 2017. Linking application description with efficient simd code generation for low-precision signed-integer gemm. In *European Conference on Parallel Processing*. Springer, 688–699.

[30] SoftBank. 2017. Q4 2016 Roadshow Slides - Arm.(2017).

[31] Andrew Tulloch and Yangqing Jia. 2017. High performance ultra-low-precision convolutions on mobile devices. *arXiv preprint arXiv:1712.02427* (2017).

[32] Yaman Umuroglu and Magnus Jahre. 2017. 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*. 1–2.

[33] Nicolas Vasilache, Jeff Johnson, Michael Mathieu, Soumith Chintala, Serkan Piantino, and Yann LeCun. 2014. Fast convolutional nets with fbfft: A GPU performance evaluation. *arXiv preprint arXiv:1412.7580* (2014).

[34] Yudong Wu, Yichao Wu, Ruihao Gong, Yuanhao Lv, Ken Chen, Ding Liang, Xiaolin Hu, Xianglong Liu, and Junjie Yan. 2020. Rotation Consistent Margin Loss for Efficient Low-bit Face Recognition. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[35] Maohua Zhu, Tao Zhang, Zhenyu Gu, and Yuan Xie. 2019. Sparse tensor core: Algorithm and hardware co-design for vector-wise sparse neural networks on modern gpus. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 359–371.