APNN-TC: Accelerating Arbitrary Precision Neural Networks on Ampere GPU Tensor Cores

Boyuan Feng[†◇], Yuke Wang[†◇], Tong Geng[*], Ang Li[*], Yufei Ding[†]
[†]{boyuan, yuke_wang, yufeiding}@cs.ucsb.edu, [*]{tong.geng, ang.li}@pnnl.gov
[†]University of California, Santa Barbara, [*]Pacific Northwest National Lab.

ABSTRACT

Over the years, accelerating neural networks with quantization has been widely studied. Unfortunately, prior efforts with diverse precisions (e.g., 1-bit weights and 2-bit activations) are usually restricted by limited precision support on GPUs (e.g., int1 and int4). To break such restrictions, we introduce the first Arbitrary Precision Neural Network framework (APNN-TC) to fully exploit quantization benefits on Ampere GPU Tensor Cores. Specifically, APNN-TC first incorporates a novel emulation algorithm to support arbitrary short bit-width computation with int1 compute primitives and XOR/AND Boolean operations. Second, APNN-TC integrates arbitrary precision layer designs to efficiently map our emulation algorithm to Tensor Cores with novel batching strategies and specialized memory organization. Third, APNN-TC embodies a novel arbitrary precision NN design to minimize memory access across layers and further improve performance. Extensive evaluations show that APNN-TC can achieve significant speedup over CUTLASS kernels and various NN models, such as ResNet and VGG.

1 INTRODUCTION

Over the recent years, demands to improve the performance of deep neural network (DNNs) have never been satisfied. Prior work approaches faster and more efficient DNNs from different aspects, such as model pruning [28, 29, 31], kernel factorization [3, 14, 40], and data quantization [45, 50]. Among those efforts, quantization-based DNN acceleration [45, 46, 50] finds its strengths in minimum modification of the original model architecture, lower memory consumption, and better runtime performance.

To accelerate quantized DNNs, many specialized cores have been introduced to support low-precision dense matrix-matrix multiplications, such as Tensor Processing Units (TPUs) [20], Neural Network Processors (NNPs) [13], and GPU Tensor Cores [4]. For example, NVIDIA introduces Tensor Cores in Volta architecture [5] that supports FP16 matrix-matrix multiplication. In Turing architecture, NVIDIA extends architecture support for more precisions (*e.g.*, int1 and int4) and bit-level operations (*e.g.*, XOR) [26]. Recently in the Ampere architecture, we find there is additional support for more precision and bit-level operations (*e.g.*, AND). However, these specialized cores still support a limited range of precisions with only architecture-level efforts, while quantized DNNs usually require arbitrary precisions (*e.g.*, 1-bit weight and 2-bit activations). In this paper, our key question is *whether we can support arbitrary precision neural networks with the limited precisions on Tensor Cores.*

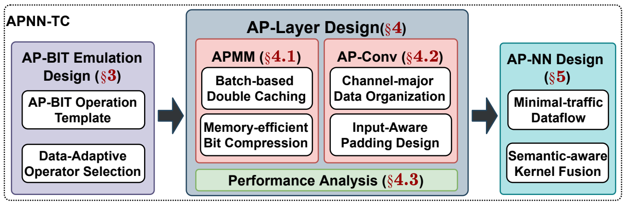

APNN-TC
AP-BIT Emulation Design (§3)
AP-BIT Operation Template
Data-Adaptive Operator Selection
AP-Layer Design(§4)
APMM (§4.1)
Batch-based Double Caching
Memory-efficient Bit Compression
AP-Conv (§4.2)
Channel-major Data Organization
Input-Aware Padding Design
Performance Analysis (§4.3)
AP-NN Design (§5)
Minimal-traffic Dataflow
Semantic-aware Kernel Fusion

Figure 1: The overview of APNN framework.

We identify two major challenges in accelerating arbitrary precision DNNs on Ampere GPU Tensor Cores.

Lack of mathematical emulation design. To support arbitrary precisions (*e.g.*, int1 weights and int2 activations), one naive approach is to represent these low-precision values with the supported high-precision values (*e.g.*, int4). However, this approach introduces extra overhead and prevents efficient quantized DNNs on Tensor Cores. Another approach is to emulate with int1 compute primitives. However, with int1 precision, Tensor Cores only support two bit-level operations (*i.e.*, XOR and AND) and mathematical emulation designs are required to support multiplication and addition in quantized DNNs. Moreover, quantized DNNs may have diverse input data (*e.g.*, -1/+1 or 0/1), where different data may require different emulation designs.

Lack of efficient implementation for arbitrary precision NN layers. To accelerate APNN on Tensor Cores, we need to efficiently map arbitrary precision NN layers to Tensor Cores with specialized compute primitives and memory architectures. Existing works on accelerating binary neural networks simply split NN layers into small matrix tiles (*e.g.*, 8 × 8) to match Tensor Core compute primitives and improve the parallelism. However, naively borrowing these strategies fails to exploit the data locality during NN layer computation especially for our emulation workload. Moreover, arbitrary precision computation usually computes at the bit-level (*e.g.*, int3 or int5) while existing hardware devices such as CPUs and GPUs usually operate at the word or byte level. Specialized bit operations and data organization are required to support efficient bit-level computation and avoid uncoalesced memory access.

Lack of efficient NN framework designs. One standard approach to build quantized neural networks is to stack a sequence of NN layers, such as a convolution layer followed by a pooling

[◇] The first two authors contribute equally.

SC '21, November 14–19, 2021, St. Louis, MO

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

layer and a quantization layer. However, this approach ignores the data reuse opportunity across NN layers and leads to unnecessary memory overhead. For example, on NNs with n 2-bit activations, there are two semantic equivalent implementations – quantization after reading 32-bit activations from the previous layer or quantization to 2-bit ones before writing to global memory for the next layer. While these two implementations provide the same semantic, the former requires memory access of $32n$ bits while the latter only requires memory access of $2n$ bits.

To this end, we propose APNN-TC to accelerate Arbitrary Precision Neural Networks on Ampere GPU Tensor Cores, as illustrated in Figure 1. First, we propose an *AP-BIT emulation design* to support arbitrary-precision computation with 1-bit compute primitives. Our AP-BIT algorithm can adaptively select operators (*e.g.*, `XOR` or `AND`) to support diverse input data (*e.g.*, -1/1 or 0/1). Second, we build efficient *AP-Layer design* including an arbitrary-precision matrix-matrix multiplication (APMM) layer for fully connected layers and an arbitrary-precision convolution (APConv) layer for convolution layers. We propose a set of memory and computation designs (*e.g.*, batch-based double caching and channel-major data organization) to fully exploit Tensor Core computation and minimize memory access. We also incorporate a performance analysis to automatically tune the hyper-parameters in APMM and APConv. Third, we propose an efficient *APNN design* to improve the performance at the framework level. It includes a minimal-traffic dataflow to support various precisions over APNN layers and a semantic-aware kernel fusion to minimize the data movement across layers.

In summary, we make following contributions in this paper.

- We develop APNN-TC to accelerate neural network on Ampere GPU Tensor Cores with arbitrary precision.
- We propose three novel techniques: a) an AP-BIT emulation design to support arbitrary-precision computation; b) an efficient AP-Layer design to achieve high performance at the layer level; c) an efficient APNN design to minimize the data movement across layers.
- Extensive experiments show that APNN-TC can achieve up to 3.78× speedup over CUTLASS kernels and 3.08× speedup over CUBLAS kernels. APNN-TC can also consistently outperform NNs implemented with built-in int8, half, or single precision. For example, with 2-bit weights and 8-bit activations, APNN-TC can achieve more than 4× latency reduction and 3× higher throughput than the single-precision NN with only 2% accuracy drop.

2 RELATED WORKS

2.1 APNN algorithm designs

Arbitrary precision (lower than INT8) neural network (APNN) algorithms have been widely studied [6, 10, 11, 24, 26, 27, 36, 44, 47, 49] to fully explore the spectrum of NN performance and NN accuracy and cater to diverse application requirements. In addition to widely supported precisions on modern GPUs (*e.g.*, `int1`, `int4`, and `int8`), these APNNs usually utilize more diverse precisions such as `int2`, `int3`, and `int5`. APNNs may also have different precisions for weights and activations (*e.g.*, 1-bit weights and 2-bit activations). Comparing with INT8 quantized neural networks, APNNs provide better performance and memory efficiency at the cost of (slightly) degraded accuracy. Popular APNNs include DoReFa-Net [49] for 1-bit weights and 2-bit activations, LQ-Nets [47] for 1-4 bits, HAQ [44] for 1-8 bits, OLAccel [36] for 4 bits, O3BNN [10], BSTC [24], and TCBNN [26] for 1 bits. In this paper, we follow LQ-Nets [47] that starts from a full-precision NN and adopts the quantization error minimization (QEM) strategy to generate quantized NNs.

2.2 APNN Hardware Supports.

While many APNN algorithms have been designed, the hardware supports are still limited. One direction is to build FPGA and ASIC based implementations [10, 36, 44] to demonstrate the performance benefits of APNNs. However, these implementations usually require specialized hardware designs to support arbitrary-precision computation and cannot be applied to GPUs. Another direction is to utilize built-in precisions on GPUs for quantized neural networks. Taking the most famous Pytorch [37] framework as an example, it supports FP32, FP16, and BF16 models on GPUs and int8 quantization on x86 CPUs with AVX2 support. Recently, BSTC [24] and BTC [25] accelerates binary neural networks on GPUs by exploiting the int1 compute primitive. However, existing works can only build on the limited precision supported on GPUs (*e.g.*, `int1`, `int4`, and `int8`) and cannot fully exploit the performance benefits from APNNs. In this paper, we build the first generalized framework to accelerate arbitrary-precision neural networks on Ampere GPU Tensor Cores.

2.3 Tensor Cores

Tensor Cores are specialized cores for accelerating neural networks in terms of matrix-matrix multiplications. Tensor Cores are introduced in recent NVIDIA GPUs since Volta architecture [34]. Different from CUDA Cores that compute scalar values with individual threads, Tensor Cores compute at the matrix level with all threads in a warp [38]. For example, the 1-bit Tensor Core compute primitive takes two `int1` input matrices A and B of shape 8×128 and generates an `int32` output matrix C of shape 8×8 [25]. In Volta architecture, Tensor Cores support only half-precision computation [19]. To support more quantized neural networks, Tensor Cores add more precisions including `int1`, `int4`, and `int8` in Turing architecture [18]. Regarding `int1` precision, Tensor Cores support only `XOR` logical operation in Turing architecture and recently add `AND` logical operation in Ampere architecture [33]. Despite these hardware efforts on supporting more precisions, arbitrary precisions are still not supported. This is the first work to support arbitrary precision computation on Ampere GPU Tensor Cores with `int1` precision and support for both `XOR` and `AND` operations.

3 AP-BIT EMULATION DESIGN

In this section, we design an AP-BIT emulation on Tensor Cores to support arbitrary-precision computation. We first design an AP-Bit operation template that supports arbitrary-precision computation with 1-bit compute primitive on Tensor Cores. Then, we propose a data adaptive operator selection to automatically support various input data (*e.g.*, -1/+1 and 0/1) with bitwise `XOR` and `AND` on Tensor Cores. Here, we focus on the algorithm design on small matrices (*i.e.*, input matrices of 8×128 and output matrix of 8×8) that can fit directly on Tensor Core compute primitives. We will discuss the efficient computation of large matrices in the next section.

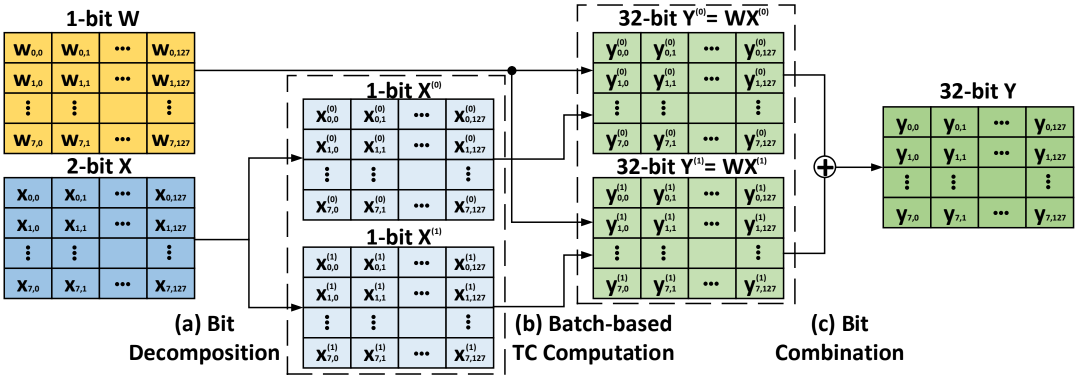


Figure 2: Illustration of AP-Bit Operation Template with 1-bit weight W and 2-bit feature X, which can be generalized to arbitrary weight bits and feature bits. Note that $X^{(0)}$ and $X^{(1)}$ in the dashed box are batched into a single large matrix during computation, which will be discussed in Section 4.1

3.1 AP-Bit Operation Template Design

The AP-Bit operation template takes a matrix W with p-bit elements and a matrix X with q-bit elements, and computes with 1-bit operations on Tensor Cores to generate a 32-bit output matrix $Y = WX$. Our key observation is that each arbitrary-bit scalar digit can be decomposed to a sequence of 1-bit scalar digits and the arbitrary computation can be conducted with only 1-bit operations and shift operations. Formally, to support scalar-level arbitrary precision computation wx of a 1-bit weight w and a 2-bit feature $x = x^{(1)}x^{(0)}$ with $w, x^{(i)} \in \texttt{int1}$, we can first decompose 1-bit values $x^{(1)}$ and $x^{(0)}$ from the 2-bit feature x as:

$$x^{(1)} = (x \gg 1)\&1, \quad x^{(0)} = (x \gg 0)\&1$$

Suppose we have an 1-bit operation $OP(a, b)$ (*e.g.*, the bmma API of Tensor Cores) that takes 1-bit inputs and generate 32-bit outputs, we can compute wx as

$$wx = OP(w, x^{(1)}) * 2 + OP(w, x^{(0)})$$

We illustrate our AP-Bit operation template in Figure 2. Here, we focus on a 1-bit weight matrix W of shape 8×128 and a 2-bit feature matrix X of shape 8×128 to illustrate our algorithm design. A naive approach is to use 4-bit integers to represent each 1-bit element $w_{i,j}$ and 2-bit element $x_{i,j}$, and then use the *int*4 compute primitive on Tensor Cores. However, this approach would lead to unnecessary memory and computation overhead. Instead, we propose to exploit the *int*1 compute primitive on Tensor Cores to support arbitrary-precision computation by dynamically adjusting the memory and computation requirement. In particular, the first step is to conduct **bit decomposition** by splitting a 2-bit $x_{i,j}$ to two 1-bit elements $x^{(0)}_{i,j}$ and $x^{(0)}_{i,j}$:

$$x^{(1)}_{i,j} = (x_{i,j} \gg 1)\&1, \quad x^{(0)}_{i,j} = (x_{i,j} \gg 0)\&1$$

These 1-bit elements are then packed into 1-bit matrix X^0 and $X^{(1)}$. The second step is to conduct **batch-based Tensor Core computation** on these 1-bit matrices with the bmma API and generate 32-bit output matrices

$$Y^{(0)} = \texttt{bmma}(W, X^{(0)}), \quad Y^{(1)} = \texttt{bmma}(W, X^{(1)})$$

These matrices can be computed directly with the bmma API since all of them have the shape of 8×128. We also note that Tensor Core primitives for int1, int4, and int8 generate 32-bit output matrices to accumulate a large number of bit-operation outputs and avoid overflow. The third step is to conduct **bit combination** and generate the final output matrix Y

$$Y_{i,j} = Y^{(1)}_{i,j} * 2 + Y^{(0)}_{i,j} \tag{1}$$

Here, $Y_{i,j}$, $Y^{(1)}_{i,j}$ and $Y^{(0)}_{i,j}$ refer to the $(i, j)^{th}$ scalar elements of matrix Y, $Y^{(1)}$ and $Y^{(0)}$, respectively. For notation simplicity, we abbreviate Equation 1 as $Y = Y^{(1)} * 2 + Y^{(0)}$ in the following sections to represent the scalar multiplication and elementwise addition. We note that $Y = WX$ mathematically.

It is not hard to see that this computation can be generalized to matrices with arbitrary bits p and q. Formally, given a p-bit weight matrix W and a q-bit weight matrix X, we can first decompose into 1-bit matrices $W^{(s)}, s \in \{0, 1, ..., p-1\}$ and $X^{(t)}, t \in \{0, 1, ..., q-1\}$. For each element, we have

$$w^{(s)}_{i,j} = (w_{i,j} \gg s)\&1, \quad x^{(t)}_{i,j} = (x_{i,j} \gg t)\&1 \tag{2}$$

Then, we compute the bmma API for pq times for each combination of s and t:

$$Y^{(s,t)} = \texttt{bmma}(W^{(s)}, X^{(t)})$$

Finally, we conduct bit combination to generate the 32-bit output matrix Y:

$$Y = \sum_{s=0}^{p-1} \sum_{t=0}^{q-1} Y^{(s,t)} * 2^{s+t}$$

Cost Analysis. The cost of arbitrary-precision computation comes from three parts: bit decomposition, tensor core computation, and bit combination. Given a p-bit weight matrix and a q-bit data matrix of shape $n \times n$, bit decomposition shows complexity of $O((p+q)n^2)$ since we need $O(pn^2)$ operations to split each p-bit element from A into p 1-bit elements and another $O(qn^2)$ operations to split each q-bit element from B into q 1-bit elements. The bit combination shows complexity of $O(pqn^2)$, since we have pq matrices $Y^{(s,t)}$ of shape $n \times n$ and need to add elementwisely. This overhead is negligible compared with the $O(n^3)$ complexity in the Tensor Core computation. Note that only 1-bit compute primitives are used for this expensive matrix-matrix multiplication, which significantly reduces the overall latency.

3.2 Data Adaptive Operator Selection

While we compute with bit-0 and bit-1 in arbitrary-precision computation, these two values may actually encode diverse values. For example, the 1-bit weight matrix in neural networks may encode −1 and 1, instead of 0 and 1, in order to improve the accuracy of neural networks. In this case, the bit-0 indicates the value −1 and the bit-1 indicates the value 1. To support this diversity in the encoded data, we introduce *data adaptive operator selection* by adopting different bit operations in Tensor Cores (*i.e.*, XOR and AND). In particular, we support three cases, where we first conduct bit operations and then accumulate with popc (*i.e.*, population count [35] that counts the number of set bits). The *Case-I* is that both W and X encode 0 and 1, where we choose logical AND operation. For example, given a 1-bit vector $W = [0, 1]$ and a two-bit vector $X = [1, 1]$, we use AND operation to compute as

$$WX = \texttt{popc}(\texttt{AND}([0, 1], [1, 1])) = \texttt{popc}([0, 1]) = 1$$

The *Case-II* is that both W and X encodes −1 and +1, where we select logical XOR operation. For example, given two 1-bit vectors $W = [-1, 1]$ and $X = [1, 1]$, we first map −1 to 0 and compute as

$$WX = n - \texttt{popc}(\texttt{XOR}([0, 1], [1, 1])) = n - 2 * \texttt{popc}([0, 1]) = 0$$

Here, n(=2) is the length of the vector.

The *Case-III* is that W encodes −1 and +1, while X encodes 0 and 1. For example, we may need to compute the multiplication of two 1-bit vectors $W = [-1, 1]$ and $X = [1, 0]$. This case happens frequently in neural networks with a 1-bit weight matrix W and a q-bit feature matrix X with $q > 1$. In this case, naively adopting XOR or AND does not work, since there are three values −1, 0, and 1 that cannot be easily encoded with 1 bit. To this end, we incorporate a linear transformation on W and compute with only AND operation. Our key observation is that W can be transformed into a vector with only 0 and 1 by adding a constant vector $\mathbf{J}_2 = [1, 1]$:

$$\hat{W} = \frac{W + \mathbf{J}_2}{2} = [0, 1]$$

Then, we compute $\hat{W}X = 0$ with AND operation as Case-II. Finally, we recover the value WX by another linear transformation:

$$WX = 2\hat{W}X - \mathbf{J}_2 X = 2 * 0 - 1 = -1$$

Note that $\mathbf{J}_2$ is a constant vector that can be cached in Tensor Core fragment and does not introduce extra memory overhead.

4 ARBITRARY PRECISION LAYER DESIGN

In this section, we propose the Arbitrary-Precision Matrix Multiplication (APMM) for fully connected layers and Arbitrary-Precision Convolution (APConv) for convolution layers.

4.1 Arbitrary-Precision Matrix Multiplication (APMM)

APMM takes the decomposed 1-bit weight matrix $W^{(s)}, s \in \{0, ..., p-1\}$, the decomposed 1-bit feature matrix $X^{(t)}, t \in \{0, ..., q-1\}$, and computes output matrix $Y = \sum_{s=0}^{p-1} \sum_{t=0}^{q-1} Y^{(s,t)} * 2^{s+t}$. By default, APMM generates 32-bit output to avoid data overflow for large matrices and match the 32-bit output in Tensor Core compute primitives. APMM also supports arbitrary-precision output (*e.g.*, int2) when APMM is used as a hidden layer in neural networks (NNs) and the output is consumed by the next APMM-based NN layer.

Considering that APMM essentially computes an arbitrary precision GEneral Matrix-Matrix multiplication (GEMM) kernel with multiple Binary Matrix-MAtrix multiplication (BMMA) kernels, one naive strategy is to build upon existing BMMA kernels [24, 25]. In particular, we can use existing BMMA kernels to multiply each pair of $W^{(s)}$ and $X^{(t)}$ and accumulate $W^{(s)}X^{(t)}$ to the output matrix Y. However, this approach shows significant inefficiency due to two reasons. First, this approach ignores the data reuse opportunity since the same weight matrix tile from $W^{(s)}$ can be multiplied with different feature matrix tiles from X_{t1} and X_{t2}. Second, this approach requires extra communication across BMMA kernels, such that reducing $W^{(s)}X^{(t)}$ into Y leads to significant global memory access.

We show our efficient APMM design in Figure 3. It includes a *batch-based double caching* to facilitate the data reuse and a *memory-efficient bit combination* to accelerate the accumulation and optionally generate the arbitrary-precision output. Here, we illustrate the design with 1-bit W and 2-bit X for notation simplicity while arbitrary-precision W and X are supported.

(a) Batch-based Double Caching. Batch-based double caching exploits two GPU memory hierarchies (*i.e.*, shared memory and fragment located in registers) to cache matrix tiles and facilitate data reuse in APMM computation, as illustrated in Figure 3(a). Considering the limited size of shared memory and fragment, we tile weight matrices $W^{(s)}$ and feature matrices $X^{(t)}$ such that these tiles can be cached in GPU memory hierarchies. Formally, given $W^{(s)}$ of shape $M \times K$ and $X^{(t)}$ of shape $N \times K$, we first tile $W^{(s)}$ along the M dimension into block matrix tiles of shape $b_m \times b_k$. Similarly, we tile $X^{(t)}$ along the N dimension into block matrix tiles of shape $b_n \times b_k$. Here, each GPU block will multiply one pair of block matrix tiles and generate an output matrix tile of shape $b_m \times b_n$. Considering that Tensor Cores compute at the warp level, we further tile $W^{(s)}$ into warp matrix tiles of shape $w_m \times w_k$ and X_s into $w_n \times w_k$ such that each warp computes an output tile of shape $w_m \times w_n$. To match with the $8 \times 8 \times 128$ bmma compute primitive of Tensor Cores, each warp will slide along w_m, w_n, and K dimension during computation. Note that these tiling sizes have a significant impact on the performance, which will be analyzed in Section 4.3.

Batch-based double caching first adopts a batch strategy to improve inter-thread parallelism and achieve high performance. Existing works on binary neural networks [24, 25] report that the GEMM size in NN workload is usually small (*e.g.*, 512×512) and use small matrix tiling sizes (*e.g.*, 32×32) to improve the inter-thread parallelism. However, this approach leads to low intra-thread parallelism and prevents data reuse. Instead, our batch strategy virtually transforms multiple small BMMAs into a large BMMA. In particular, given $W^{(s)}, s \in \{1, ..., p-1\}$ of shape $M{\times}K$ and $X^{(t)}, t \in \{1, ..., q-1\}$ of shape $N \times K$, we batch these small matrices into W_B of shape $pM \times K$ and X_B of shape $qN \times K$ and compute using a single large BMMA. Here, we implement a "virtual" batch strategy during the data loading procedure by dynamically deciding the global memory

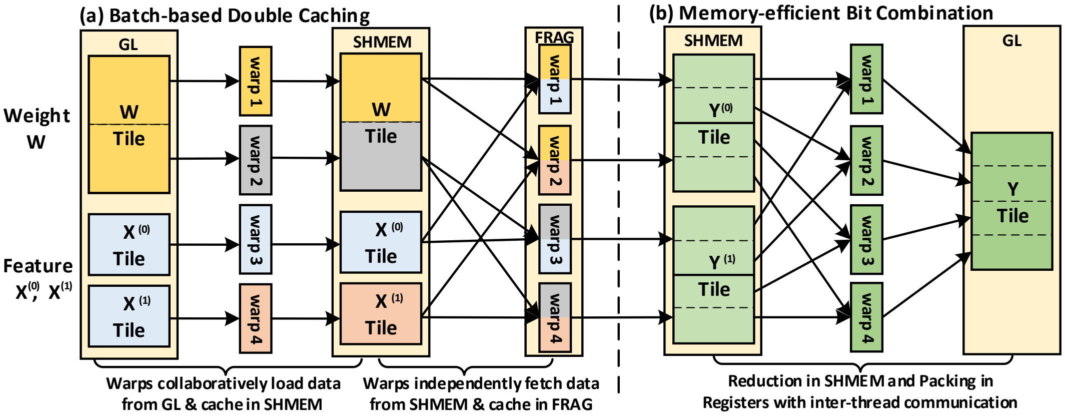

(a) Batch-based Double Caching
(b) Memory-efficient Bit Combination
GL
SHMEM
FRAG
Weight W
Feature X(0), X(1)
W Tile
X (0) Tile
X (1) Tile
warp 1
warp 2
warp 3
warp 4
Y (0) Tile
Y (1) Tile
Y Tile
Warps collaboratively load data from GL & cache in SHMEM
Warps independently fetch data from SHMEM & cache in FRAG
Reduction in SHMEM and Packing in Registers with inter-thread communication

Figure 3: Illustration of APMM. GL: GLobal memory. SHMEM: SHared Memory. FRAG: FRAGment.

address of the corresponding matrix tile such that no additional memory movement is involved.

Batch-based double caching then exploits two GPU memory hierarchies to facilitate data reuse at different levels. The first level is shared memory caching to reuse matrix tiles from $W^{(s)}$ and $X^{(t)}$. Here, a naive strategy is that each warp independently loads a weight tile and a feature tile for computation. However, we observe that the same weight tile may be multiplied with feature tiles from different 1-bit feature matrices $X^{(0)}$ and $X^{(1)}$, as illustrated in Figure 3(a). To this end, our design requires all warps to first collaboratively load $b_m \times b_k$ weight data and $b_n \times b_k$ feature data from global memory to shared memory. Then, each warp fetches its own matrix tiles from shared memory. In this way, we can significantly reduce global memory access by exploiting fast shared memory.

The second level is fragment caching to continuously store output tiles in the same Tensor Core fragment. Since Tensor Core compute primitives require to accumulate in 32-bit Tensor Core fragments, the output tiles usually consume a large memory space compared with the 1-bit input data. Moving output tiles between shared memory and Tensor Core fragment may lead to heavy shared memory access. Moreover, existing dissecting works [18, 19] reveal that fragment is composed of registers and one GPU block of 8 warps can provide up to 256 KB Fragment, which is much larger than shared memory. To this end, as iterating through the K dimension during computation, we continuously use multiple fragments to cache output tiles of shape $b_m \times b_n$ for reducing shared memory access and caching more feature and weight tiles in shared memory.

(b) Memory-efficient Bit Combination. Bit combination consumes 32-bit BMMA outputs $Y^{(s,t)} \in \texttt{int32}^{M\times N}$ and generates 32-bit APMM outputs $Y \in \texttt{int32}^{M\times N}$ as $Y = \sum_{s=0}^{p-1}\sum_{t=0}^{q-1} Y^{(s,t)} * 2^{s+t}$. 'Bit combination can also generate arbitrary precision output when it is utilized as a NN hidden layer and its output is consumed by the next NN layer. Overall, bit combination takes only $O(pqMN)$ computation complexity, which is significantly lower than the computation complexity of GEMM operations.

However, there are two potential memory bottlenecks in bit combination, which have a significant performance impact. The first one is global memory access when reducing 32-bit BMMA outputs to 32-bit APMM outputs. In a naive implementation that independently conducts BMMA and bit combination, bit combination usually introduces similar latency as the BMMA kernel. The main reason is that, while Tensor Cores provide significantly higher computation throughput than CUDA Cores, the global memory bandwidth remains the same. The second one is the shared memory access when converting 32-bit APMM outputs to arbitrary-precision outputs. In this procedure, we usually need to pack low-bit values (*e.g.*, 2-bit) in registers from different threads to a single memory-aligned value (*e.g.*, 32-bit) before storing to global memory. Relying on shared memory for data exchange across threads may lead to heavy shared memory access.

Memory-efficient bit combination includes two novel designs to mitigate memory overhead. The first design includes a semantic-aware workload allocation and an in-shared-memory reduction. In particular, at the data loading phase of BMMA, we load feature tiles and weight tiles of the same spatial location such that their multiplication outputs can be reduced directly. As illustrated in Figure 3, instead of loading a $b_n \times b_k$ feature tile of $X^{(0)}$ or $X^{(1)}$, we load two $0.5b_n \times b_k$ feature tiles of both $X^{(0)}$ and $X^{(1)}$ with the same matrix index. In this way, we can reduce $WX^{(1)}$ and $WX^{(0)}$ directly in shared memory and mitigate global memory access while not degrading the BMMA performance.

The second design incorporates an element-wise routine and an inter-thread communication to pack low-bit values and mitigate shared memory overhead. The element-wise routine is a user-defined interface to provide diverse support of quantization and batch normalization across NN layers. This routine applies to individual 32-bit reduced values in registers. Given a 32-bit value in a register, this routine may quantize it into a p-bit value that is still stored in the 32-bit register with the first $32 - p$ bits as zeros. This routine also includes bit decomposition (Equation 2) that splits this p-bit value in a register to 1-bit values in p registers. After that, we use a `__ballot_sync` API to enable inter-thread communication and directly pack the 1-bit values across threads into 32-bit values that can be stored to the global memory.

4.2 Arbitrary-Precision Convolution (APConv)

APConv takes the decomposed 1-bit weight matrix $W^{(s)}$ of shape $C_{out} \times C_{in} \times K \times K$, the decomposed 1-bit feature matrix $X^{(t)}$ of

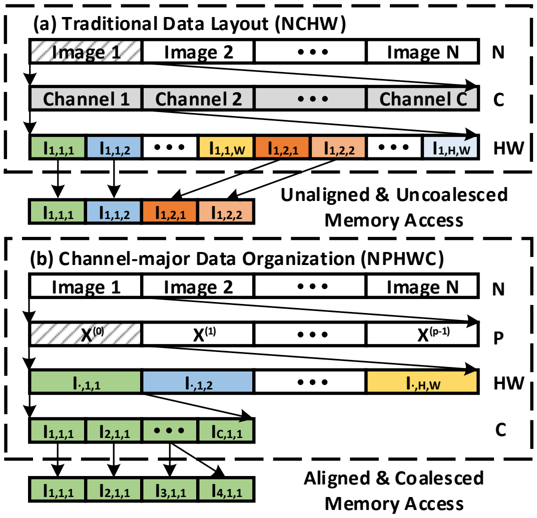


Figure 4: Illustration of Channel Major Data Organization (NPHWC). P indicates the number of bits. I_{chw} indicates the image pixel at the c-th channel, h-th height, and w-th width.

shape $BS \times C_{in} \times Height \times Width$, and generates output matrix Y. Here, C_{out} is the number of output channels, C_{in} is the number of input channels, K is the kernel size, BS is the batch size.

Existing works on bit-level convolution usually adopt a direct convolution design [24, 25] to improve the GPU utilization. However, these methods ignore the data reuse opportunity and introduce heavy global memory access. In addition, APConv on a p-bit weight and a q-bit feature usually has pq times workload than the BConv on the same weight and feature size, which can easily contribute to high GPU utilization. To this end, APConv incorporates the batch-based double caching design as APMM to mitigate the global memory access.

However, there are still two key challenges that distinguish APConv from APMM. The first is the data organization where naively reading the $K \times K$ feature map may easily lead to un-coalesced memory access. The second is the data padding where simply padding zeros may lead to erroneous results. To tackle these challenges, we propose *channel-major data organization* and *input-aware padding design*.

(a) Channel-Major Data Organization. Channel-major data organization transforms un-coalesced and unaligned memory access to a coalesced and aligned one for improving performance. Traditional data organization for 32-bit convolution usually employs a NCHW design, as illustrated in Figure 4(a). However, naively borrowing this design to APConv leads to un-aligned and un-coalesced memory access due to two reasons. First, multiple P-bit (*e.g.*, 3-bit) elements usually cannot be packed into an aligned 32-bit element, which is required for valid GPU reads and writes. Using a 32-bit element to store a P-bit element will introduce extra memory overhead. Second, convolution operations usually read only K continuous elements (or KP bits) due to the $K \times K$ kernel size, which may lead to un-coalesced memory access.

We design a channel-major data organization as illustrated in Figure 4(b). There are two key design choices. First, we split a P-bit feature matrix into P 1-bit feature matrices and store each 1-bit feature matrix consecutively. In this way, we can provide aligned memory access for each 1-bit feature matrix and support arbitrary precision P. Second, we consecutively store all channels of elements with the same spatial location. Since convolution layers usually have $128C, C \in \mathbb{N}$ channels, this usually leads to coalesced memory access during computation.

(b) Input-aware Padding Design. Input-aware padding design adaptively adjusts padding values according to input values. As mentioned in Section 3.2, when the weight W encodes -1 and 1 with 0 and 1, we cannot naively padding 0 since 0 represents -1.

We propose three padding strategies according to the input data. First, when both weight and feature encode 0 and 1, we simply pad zeros for features. In this case, padding 0 for features will only add extra 0's for arbitrary weight values, which does not change the computation result. Second, when both weight and feature encode -1 and 1, we pad 1 for features and use an extra `counter` flag to track the number of 0's when the convolution weight moves outside the input image frame. We will subtract `counter` to amend the corresponding convolution results. Third, when weight encodes -1 and 1 and feature encodes 0 and 1, we pad 0 to features and do not change the convolution results.

4.3 Performance Analysis

In our APNN-TC kernel design, there are six tuning knobs – the block tiling sizes b_m, b_n, b_k, and the warp tiling sizes w_m, w_n, w_k. These tiling sizes bring a trade-off between the Thread-Level Parallelism (TLP) and the Instruction Level Parallelism (ILP), especially the compute intensity (CI). Here, we focus on block tiling sizes, since we empirically observe that utilizing 8 warps per block and splitting the block workload evenly across warps provide the best performance (*i.e.*, $w_m = b_m/4$, $w_n = b_n/2$, $w_k = b_k$). In this subsection, we first analyze the performance impact of individual tuning knobs. Then, we propose an autotuning strategy to maximize the performance. Since APMM and APConv share the same batch-based double caching strategy, we use the same autotuning for these two kernels.

4.3.1 *Performance Model.*

TLP refers to the thread-level parallelism in terms of the number of threads in use. Intuitively, larger TLP can improve GPU utilization and kernel performance. Formally, given a p-bit weight matrix of shape $M \times K$, a q-bit feature matrix of shape $K \times N$ and the matrix tiling size $b_m \times b_n$, we define the TLP as

$$TLP = \frac{pM \times qN}{b_m \times b_n} \qquad (3)$$

We ignore the number of threads for each block since it is a constant in our evaluation. Intuitively, smaller $b_m \times b_n$ may improve TLP, which suggests a small $b_m \times b_n$ especially for small matrices.

Compute intensity (CI) refers to the ratio of computation over memory access on each thread block. We aim to improve CI for two reasons. First, a higher CI indicates less memory access and better performance. While the amount of computation remains the same, the amount of memory access may be reduced significantly by data reusing and hyper-parameter tuning. Second, a higher CI on a thread block provides more opportunities for latency hiding.

Formally, for a matrix tile, we compute the amount of global memory access as $b_m \times b_k + b_n \times b_k$ when reading a $b_m \times b_k$ weight tile and a $b_m \times b_k$ feature tile. We compute the amount of computation as $2 \times b_m \times b_n \times b_k$ from the matrix-matrix multiplication. Finally, we compute CI as

$$CI = \frac{2 \times b_m \times b_n}{b_m + b_n} \tag{4}$$

Note that CI can be increased when b_m and b_n are increased. We also observe that CI is independent of b_k such that we can use smaller b_k to leave space for larger b_m and b_n, especially when the shared memory size is a limiting factor. In our evaluation, we fix b_k as 128 by default.

4.3.2 *Auto-tuning.* During APNN-TC kernel design, there is a large search space on the complex interaction between matrix size (M, N, and K), weight bit p, feature bit q, and block tiling size b_m and b_n. Note that the selected parameters may also be different on various GPUs according to computation and memory capabilities. To this end, we propose a heuristic algorithm to provide a faster search procedure in this large search space. Formally, given the matrix size M, N, K, the weight bit p, the feature bit q, the algorithm selects $b_m, b_n \in \{16, 32, 64, 128\}$ in two steps. First, we compute the TLP of each combination of b_m and b_n. We put these combinations in a priority queue, where a higher TLP leads to a high priority. Second, we pop individual combinations in the priority queue. We stick to the first combination with the highest TLP if its TLP is already smaller than a threshold T. Otherwise, we continuously pop and select combinations in the priority queue to improve CI while ensuring TLP is larger than T. We empirically set T as 64 in our evaluation. Note that different block tiling sizes share the same data layout such that there is no overhead when consecutively executing two layers with different block tiling sizes.

5 ARBITRARY PRECISION NEURAL NETWORK DESIGN

In this section, we introduce our Arbitrary Precision Neural Network (APNN) design. We first introduce a minimal-traffic dataflow on supporting various precisions across layers in APNN. Then, we incorporate a semantic-aware kernel fusion to minimize the memory access across layers.

5.1 Minimal-Traffic Dataflow

Given an `int8` RGB image, APNN computes a sequence of NN layers with p-bit weights and q-bit activations and finally generates an `int32` output logits. Here, all intermediate layers compute at arbitrary precision by taking a p-bit weights and q-bit activations and generate 32-bit outputs. Note that the `int1` Tensor Core compute primitive can only generate $int32$ outputs and an extra quantization layer is required to quantizing into q-bit activations for the next layer. For performance consideration, during the initialization of an APNN, we quantize all weights before the model inference computation. To effectively maintain and transfer arbitrary-bit data, we pack the data bit-by-bit for both weight and feature map, following the data organization discussed in Section 4.2.

The input layer and the output layer have different precisions from the intermediate layers. As is the common practice with `int8` image inputs, the input layer requires an extra quantization layer that quantizes 8-bit inputs into q-bit activations. The output of the input layer will also be the quantized arbitrary-bit feature map serving as the input for the following intermediate layers. In the output layer, Tensor Core computation results will be directly used for the final softmax logits computation. Thus, we do not apply quantization after the output layer.

5.2 Semantic-aware Kernel Fusion

Besides APMM and APConv discussed previously, there are still multiple important layers in APNN, including quantization, Batch Normalization (BN), pooling, and ReLU. Given all scalars $x_{i,j}$ in the i^{th} layer, quantization element-wisely converts `int32` values $x_{i,j}$ to `q`-bit values $y_{i,j}$:

$$y_{i,j} = \lfloor (x_{i,j} - z_i)/s_i \rfloor$$

Here, z_i is a 32-bit scalar zero-point, s_i is the scaling scalar, and $\lfloor \cdot \rfloor$ is the floor function. BN [17] is another major component in NNs for tackling the covariate shift problem and facilitating NN training:

$$y_{i,j} = \frac{x_{i,j} - \mathbb{E}[x_{i,*}]}{\sqrt{Var[x_{i,*} + \epsilon]}} \cdot \gamma_j + \beta_j \tag{5}$$

where $\mathbb{E}$ and Var are expectation and variance across the batch, γ_j and β_j are two learned parameters. Pooling splits the feature map spatially into $k \times k$ grids and generates 1 scalar output for each grid by computing the average or the maximum value in each grid. ReLU takes individual input values $x_{i,j}$ and generates output values $y_{i,j} = max(x_{i,j}, 0)$.

While these operations have linear time complexity to the size of feature maps and consume significantly less computation than APConv and APMM kernels, these operations may still introduce heavy latency due to the expensive memory access. Indeed, while Tensor Cores provides significantly improved computation capability, Tensor Cores share the same memory bandwidth with CUDA Cores on GPUs. Moreover, we observe that these values are usually computed element-wisely and do not require heavy communication across GPU threads.

We propose a semantic-aware kernel fusion to minimize memory access. We first fuse APMM/APConv with its following quantization, BN, pooling, and ReLU kernels into a single kernel to minimize the global memory access. In particular, these following layers can be seamlessly applied once the convolution results become available at the shared memory. This can improve the computation intensity for individual convolution kernels meanwhile reducing the global memory access from invoking an additional batch normalization kernel. Second, considering that these following layers usually compute at scalar level, we can further reduce shared memory access by directly reusing values in registers. For example, when a APMM layer is followed by a BN layer, a quantization layer, and a ReLU layer, we directly compute the output scalar as

$$\lfloor max(\frac{x_{i,j} - \mathbb{E}[x_{i,*}]}{\sqrt{Var[x_{i,*} + \epsilon]}} \cdot \gamma_j + \beta_j - z_i, 0)/s_i \rfloor$$

Note that we only need to load a scalar once to a register and avoids unnecessary shared memory access.

6 EVALUATION

In this section, we evaluate APNN-TC under diverse precisions and show the benefits of arbitrary-precision computation in performance and accuracy.

Environments. We evaluate on both Nvidia RTX 3090 and Nvidia Tesla A100. The RTX3090 GPU is in a ubuntu 16.04 system with Intel(R) Xeon(R) Silver 4110 CPU @ 2.10GHz, 64 GB DDR3 DRAM, gcc-7.5.0, and using CUDA-11.1, CUTLASS-2.5, and CUBLAS-11.1. The A100 GPU is in a Linux 3.10.0 system with AMD EPYC 7742 64-core CPU, 1TB DDR4, gcc-9.1.0, and using CUDA-11.1, CUTLASS-2.5, and cuBLAS-11.3. All results reported are the average of 200 times execution.

6.1 APLayer Evaluation

6.1.1 APMM Performance. We compare our APMM designs with NVIDIA implementations of low-bit gemm (*i.e.*, int1, int4, and int8) that are accelerated by Tensor Cores. For int8, we compare with cublas implementation, namely cublass-gemm-int8. Since int1 and int4 are not supported in cublas, we compare with cutlass implementation, namely cutlass-gemm-int1 and cutlass-gemm-int4. Following popular settings in NNs, we compute matrix multiplication of a matrix with shape $B \times K$ and a matrix with shape $K \times N$, where $B = 64$ is a popular batch size and $K = N \in \{128, 256, ..., 1024\}$ covers typical fully connected layer dimensions. According to the precision of our APMM kernel, we name it APMM-wxay, where x indicates the weight bit and y indicates the activation bit. For example, APMM-w1a2 indicates 1-bit weights and 2-bit activations. While our APMM is general to support arbitrary precision, we show 8 popular bit combinations due to page limits. If both weight bits and activation bits are less than 4 (*e.g.*, w1a2, w1a3, w1a4, w2a2), we compare it against cutlass-gemm-int4. If either weight bits or activation bits are larger than 4, we compare it against cublas-gemm-int8. For each matrix size, we also show the speedup of cutlass-gemm-int1 against cutlass-gemm-int4 and cublas-gemm-int8 as the performance benefit when sticking to binary neural networks [24, 25]. Since Tensor Core compute primitive supports only 32-bit outputs, all of these gemm kernels take low-bit input (*e.g.*, int1, int4, and int8) and generate 32-bit outputs. We will study the performance of quantization in later sections.

Figure 5 shows the results of APMM on RTX 3090. We compare APMM with cutlass-gemm-int4 in Figure 5(a) and cublas-gemm-int8 in Figure 5(b). Overall, we have three major observations. First, APMM can usually achieve significant speedup over baselines. For example, APMM-w1a2 can achieve up to 2.35× speedup over cutlass-gemm-int4, while APMM-w5a1 can achieve up to 3× speedup over cublas-gemm-int8. This result demonstrates the performance benefits of emulating arbitrary-precision with int1 compute primitives over sticking to int4 or int8 compute primitives. Second, APMMs with various weight and activation bits usually show similar performance on small matrices. For example, APMM-w1a2, APMM-w1a3, APMM-w1a4, and APMM-w2a2 achieves almost the same speedup when N=128 and N=256, even if these kernels have different computation overhead (*e.g.*, 2× from APMM-w1a2 and 4× from APMM-w2a2). This benefit comes from our batch-based double caching (Section 4.1(a)), where individual small BMMAs are batched into a large BMMA and computed simultaneously. Surprisingly, our arbitrary precision computation can even outperform cutlass-gemm-int1 in such cases due to the improved GPU utilization. Third, we observe a smaller speedup over cublas-gemm-int8 on large matrix sizes, when peak int1 performance is achieved. Our investigation shows that, on RTX 3090, cutlass-gemm-int1 is only 5.9× faster than cublas-gemm-int8, such that emulation is slower than built-in int8 compute primitives on large matrices when peak int1 performance is achieved (*e.g.*, 64×1024×1024 for APMM-w2a8). We argue that NN workload can still benefit significantly from our APMM since the fully connected layers in neural networks usually have small matrix sizes (*e.g.*, $1 \times 512 \times 512$ in ResNet-18). We also show the results of APMM on A100 in Figure 6 with similar observations.

Table 1: APNN Evaluation Setting. We list the dataset, network, input size, output size, and the model accuracy under precisions of BNN (*i.e.*, int1), w1a2 (*i.e.*, 1-bit weights with 2-bit activations), and single-precision floating point.

Dataset	Network	Input Size	Output Size	Binary	w1a2	Single
ImageNet	AlexNet [21]	224x224x3	1000	46.1% [15]	55.7% [47]	57.0% [22]
ImageNet	VGG-Variant [2]	224x224x3	1000	53.4% [15]	68.8% [47]	69.8% [41]
ImageNet	ResNet-18 [12]	224x224x3	1000	51.2% [15]	62.6% [47]	69.6% [12]

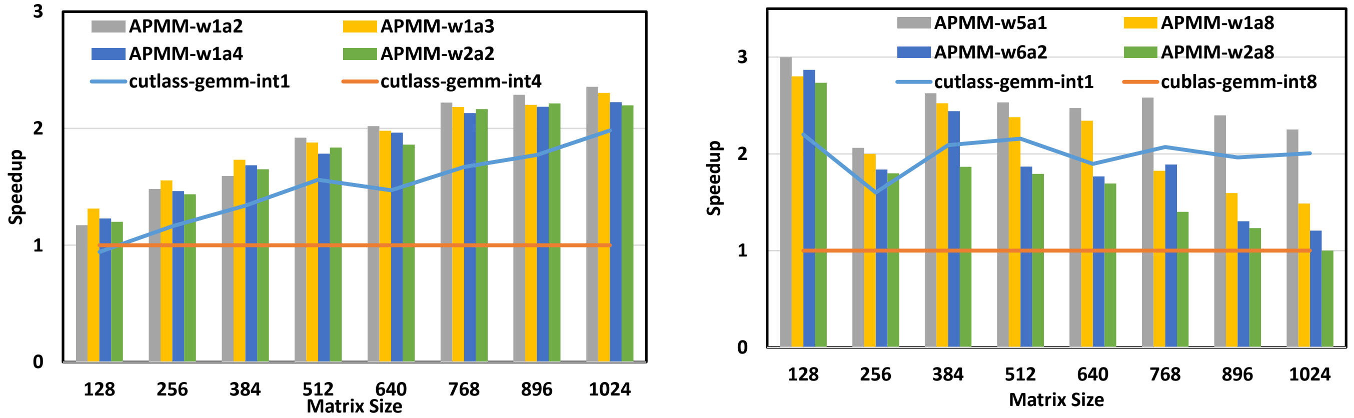
6.1.2 APConv Performance. We compare our APConv designs with NVIDIA implementations of low-bit convolution that are accelerated by Tensor Cores. Since cublas does not support int1, int4, AND int8 convolution, we use kernels from cutlass. We name these kernels as cutlass-conv-int1, cutlass-conv-int4, and cutlass-conv-int8. Similar to APMM, we evaluate 8 types of precision with the name APConv-wxay. Since convolution kernels have much more hyperparameters than matrix-multiplication kernels, we show the performance under various input and output channels while fixing the input size as 16 (medium feature size), filter size as 3 (most frequently used), stride as 1 (most frequently used), and batch as 1 (for inference).

Figure 7 and Figure 8 show the speedup of APConv on RTX 3090 and A100, respectively. We observe that APConv can achieve 3.78× speedup over cutlass-conv-int4 and 3.08× speedup over cutlass-conv-int8. This result shows the significant performance benefit from emulating arbitrary precision with int1 over utilizing int4 or int8. Similar to APMM, we also observe a smaller speedup over cutlass-conv-int8 on larges channels due to the limitation of peak int1 performance. Since RTX3090 and A100 provide similar performance, we will focus on RTX3090 in the following evaluations.

6.2 APNN Evaluation

In this section, we evaluate the overall APNN performance on three mainstream neural network models with ImageNet dataset. The details of our evaluated NN models and their corresponding binarized neural network, low-bit (1-bit weight with 2-bit activation), single-precision accuracy precision are listed in Table 1.

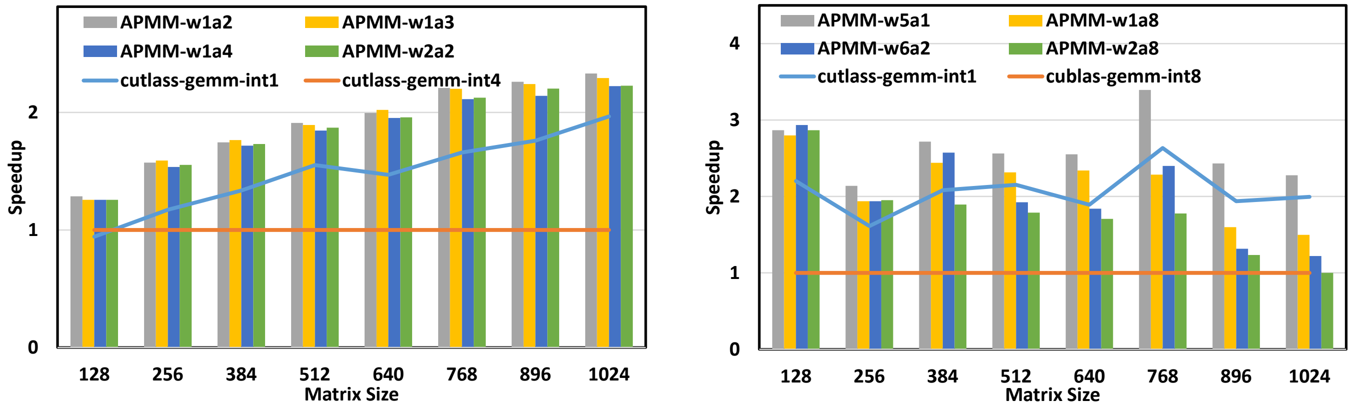
We consider two types of configurations for evaluation. In the first setting, we focus on a specific low-bit configuration (1-bit weights and 2-bit activations, *i.e.*, w1a2) across different neural network models. We choose several baselines including neural networks built with single-precision floating-point implementation from CUTLASS [32] running on CUDA Cores, half-precision implementation from CUTLASS running on Tensor Cores, INT8 precision



(a) Over CUTLASS-GEMM-INT4

(b) Over CUBLAS-GEMM-INT8.

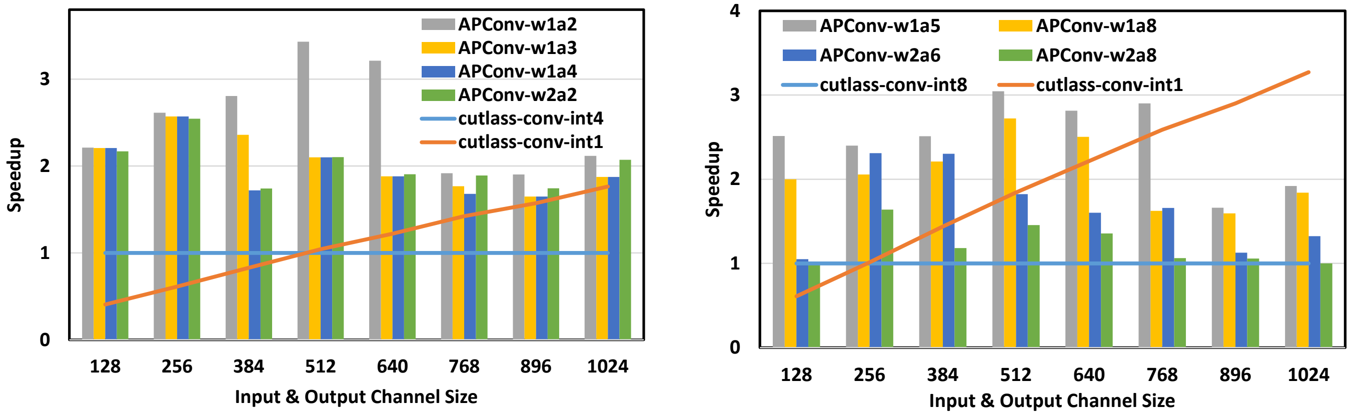
Figure 5: APMM Performance on RTX 3090.



(a) Over CUTLASS-GEMM-INT4.

(b) Over CUBLAS-GEMM-INT8.

Figure 6: APMM Performance on A100.

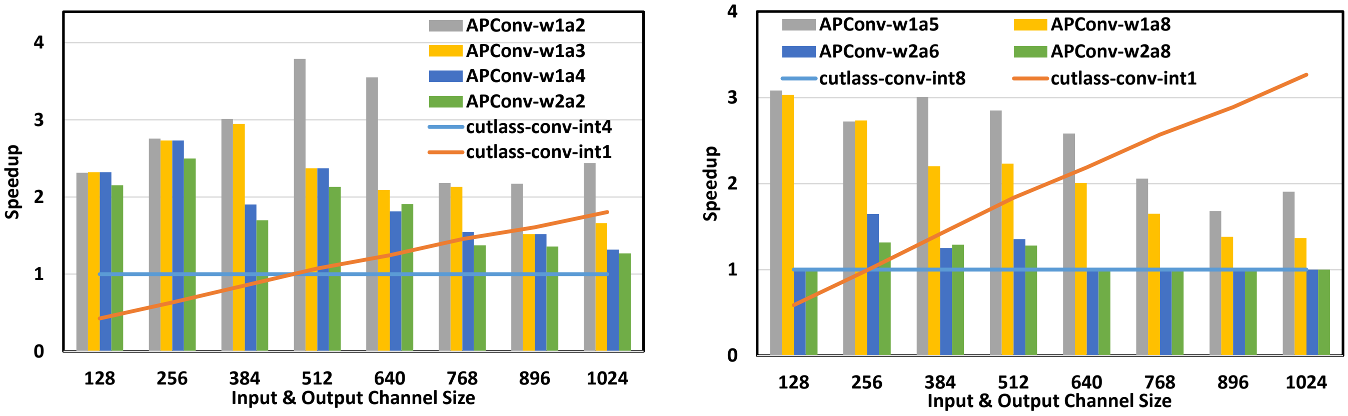


(a) Over CUTLASS-Conv-INT4.

(b) Over CUTLASS-Conv-INT8.

Figure 7: APConv Performance on RTX 3090.

implementation from CUTLASS running on Tensor Cores, and the 1-bit binarized neural network running on Tensor Cores. As shown in Table 2, our APNN design running on Tensor Cores can achieve a significant speedup (more than 2× on average) compared with CUTLASS INT8, half and single precision implementations. This indicates the practical usage of our APNN design in latency-sensitive applications. Meanwhile, on large batch sizes for throughput performance evaluation, our APNN design also demonstrates its high throughput advantage over these "standardized" bit (*e.g.*, 8-bit and



(a) Over CUTLASS-Conv-INT4.

(b) Over CUTLASS-Conv-INT8.

Figure 8: APConv Performance on A100.

Table 2: APNN Inference Performance on NVIDIA Ampere RTX3090 GPU. Note that latency is measured under a batch of 8 images, throughput is measured under a batch of 512 for AlexNet and ResNet18, and 256 for VGG-variant model.

	ImageNet-AlexNet		ImageNet-VGG		ImageNet-ResNet18	
Schemes	**8 Latency**	**Throughput**	**8 Latency**	**Throughput**	**8 Latency**	**Throughput**
CUTLASS-Single	25.22ms	3.29×10^2fps	116.84ms	6.85×10^1fps	24.02ms	5.22×10^2fps
CUTLASS-Half-TC	14.37ms	6.21×10^2fps	31.42ms	2.79×10^2fps	12.52ms	1.13×10^3fps
CUTLASS-INT8-TC	3.78ms	2.40×10^3fps	23.53ms	3.51×10^2fps	6.6ms	3.13×10^3fps
BNN	0.69ms	1.37×10^4fps	2.17ms	3.91×10^3fps	0.68ms	1.89×10^4fps
APNN-w1a2	2.87ms	3.79×10^3fps	7.50ms	1.07×10^3fps	3.66ms	4.37×10^3fps

Table 3: Case Study: APNN Evaluation on ResNet-18 and ImageNet with various precision.

Precision	Accuracy (%)	8 Latency (ms)	Throughput (fps)
Float	69.8	24.02	5.22×10^2
Half	NA	12.52	1.13×10^3
INT8	NA	6.6	3.13×10^3
BNN	51.2	0.68	1.89×10^4
w1a2	59.6	3.66	4.37×10^3
w2a2	62.6	3.65	4.38×10^3
w2a8	67.7	4.71	1.67×10^3

half) precision baselines. Compared with the 1-bit binarized neural network running on Tensor Cores, our APNN design would demonstrate its significant accuracy improvement (an average 11.67%) as listed in Table 1. This can demonstrate the application of our APNN design in some application settings, where the BNN model accuracy performance fails to meet the demands. Overall, from the study, we can see that using our APNN design for arbitrary-bit precision computation is a potential way for balancing NN model accuracy and runtime performance.

In the second setting, we shift our focus towards the precision and model runtime performance tradeoff on ResNet18, which is popularly used in many workloads [43]. We select several low-bit settings for comparison, including the 1-bit weight with 2-bit activation, 2-bit weight with 2-bit activation, and 2-bit weight with 8-bit activation. As shown in Table 3, APNN-TC significantly reduces latency and improves throughput for w1a2 and w2a2 than INT8 which shows that APNN-TC can bring benefits for many arbitrary-precision computations. On w2a8, we only trade 2% model accuracy loss for more than 4× speedup in latency performance and 3× higher throughput, comparing with the full-precision floating-point design. Comparing with INT8, APNN-TC with w2a8 shows lower throughput since we need to compute 16 (=2*8) 1-bit matrices to emulate arbitrary-precision computation, which require more computation than w1a2 with 2 1-bit matrices and w2a2 with 4 1-bit matrices. This also matches the performance on individual kernels (e.g., Figure 5, 6, 7, 8). We also note that APNN-TC can still achieve lower latency on w2a8 than INT8. This result indicates that APNN-TC can still bring benefits for latency-sensitive applications.

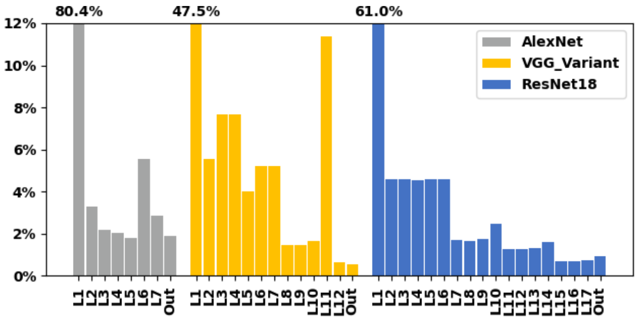


Figure 9: Per-layer latency breakdown of our APNN design on the 3 models.

6.3 Additional Studies

We perform several additional studies in this subsection, including the latency breakdown from individual NN layers and the benefit from kernel fusion. We show results from RTX 3090 and skip results from A100 since we observe similar trend on these two GPUs.

Latency Breakdown. Figure 9 illustrates the percentage breakdown of the latency for the inference of 8 images over three NNs on RTX-3090 GPU. Clearly, the first layer introduces the most delay since the input feature size for this layer is significantly larger than other layers. This percentage can be as high as 80.4% for AlexNet and 47.5% for VGG_Variant. On other layers, we observe a roughly balanced latency.

Benefits from Kernel Fusion. Figure 10 investigates the performance benefits from fusing APConv-w1a2, pooling, and quantization into one kernel. Specifically, in the "w/o Fusion" implementation, we implement three global functions for APConv-w1a2 with 32-bit output, 2 × 2 pooling, and quantizing into 2-bit outputs, respectively. Here, each function read and write data to the global

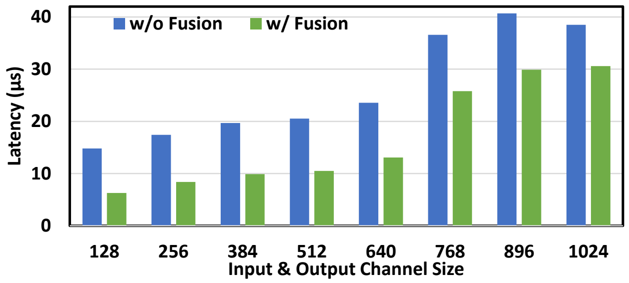


Figure 10: Speedup from APNN Kernel Fusion

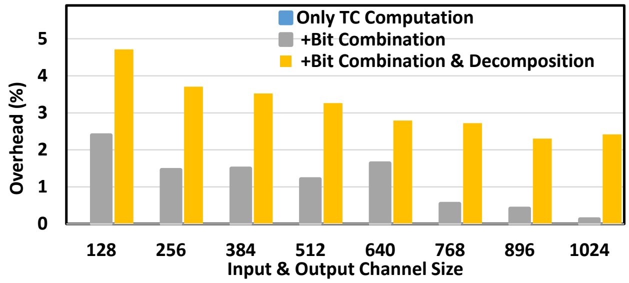


Figure 11: Overhead from bit combination and bit decomposition, relative to TC Computation.

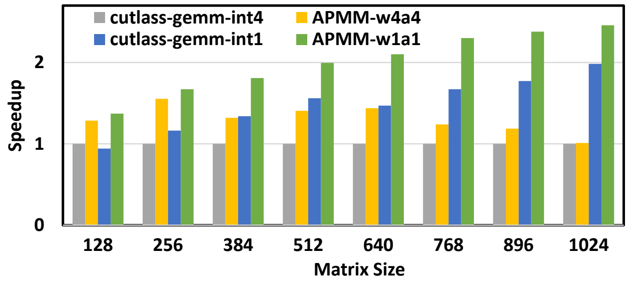


Figure 12: Comparing APMM and cutlass-gemm on same bits.

memory. In the "w/ Fusion" implementation, we conduct the same workload in a single kernel. Overall, we observe a latency reduction of 1.77× on average. The main reason is that, in "w/ Fusion", data across APConv, pooling, and quantization can be cached in shared memory and global memory access is significantly reduced.

Overhead from bit combination and bit decomposition. We show the overhead from bit combination and bit decomposition in Figure 11. We profile the overhead on APConv designs following the same setting as Section 6.1.2. We show results from APConv-w1a2 since we observe similar overhead across bit settings. On average, we empirically observe 1.16% overhead from bit combination and another 2.02% overhead from bit decomposition, compared to only TC computation. The main reason is that bit combination and bit decomposition introduce only quadratic time complexity, which is significantly smaller than the cubic time complexity from TC computation. Due to this difference in time complexity, the overhead from bit combination decreases from 2.4% to 0.12% as the channel size increases from 128 to 1024. We also observe similar trend for bit decomposition.

Comparing APMM and cutlass GEM under the same bits. Figure 12 shows the performance comparison between APMM and cutlass-gemm when using the same bits. Overall, we observe that APMM-w4a4 can achieve 1.3× speedup over cutlass-gemm-int4. The main reason is that APMM-w4a4 can achieve better parallelism by using 16 int1 computations to emulate 1 int4 computation and achieving better GPU utilization, especially for small matrix sizes. We note that this speedup of APMM-w4a4 over cutlass-gemm-int4 decreases as the matrix size increases where more int1 computation resources are required for emulation. We also observe that APMM-w1a1 can achieve 1.35× speedup over cutlass-gemm-int1. This shows the benefit from our kernel-level optimizations.

Table 4: Raw latency of a typical fully-connected layer with batch size $M = 64$, input dimension $K = 1024$, and output dimension $N = 1024$. Unit: microsecond.

w1a2	w1a3	w1a4	w2a2	cutlass-gemm-int4	cutlass-gemm-int1
6.67	6.81	7.06	7.15	15.61	7.92

Raw latency of a typical fully-connected layer. Table 4 shows the raw latency of a typical fully-connected layer with batch size $M = 64$, input dimension $K = 1024$, and output dimension $N = 1024$. Overall, we observe that we require only around 7 microsecond for such a layer. Comparing with cutlass-gemm-int4, we can achieve 2.27× speedup on average by using arbitrary-precision computation. We also note that the arbitrary-precision computation is even slightly faster than the cutlass-gemm-int1, which matches the result in Section 6.1.1.

7 DISCUSSION

Practical usage of APNN. Arbitrary-precision neural networks have been widely studied to provide diverse tradeoffs between precision and efficiency [6, 10, 11, 24, 26, 27, 36, 44, 47, 49]. While arbitrary-precision may slightly reduce the precision, it shows merit in many practical usages such as smart sensors [23, 30, 39], mask detection [8], and intelligent agriculture [9]. In these usages, when a certain accuracy bar is surpassed, other essential metrics such as real-time processing and resource consumption are more important. For example, BinaryCoP [8] utilizes low-power binary neural networks to detect facial-mask wear at entrances to corporate buildings and airports. Another example is XpulpNN [9] that uses quantized neural network on energy-efficient IoT devices.

Generality to other NNs. This paper reports the results of APNN-TC on two most time-consuming kernels, GEMM and Convolution, from the computer vision domain and showcases the performance on popular vision models (e.g., AlexNet, VGG, and ResNet). Yet, we expect that APNN-TC applies to NNs from various domains such as natural language processing (NLP). Intuitively, APNN-TC accelerates GEMM and dot products which is the building block of many NLP NNs [7, 42, 48], such as the attention layer and the feed-forward layer.

Generality to other processors. APNN-TC utilizes population count (i.e., `popc()`) and two logical operations (i.e., `XOR` and `AND`) to support arbitrary-precision computation on Nvidia GPUs. Considering the wide support for `popc()` and logical operations, APNN-TC can be easily adapted to diverse processors. For example, AMD GPUs [1] supports population count (i.e. `popcnt()` on AMD GPUs) and logical operations (e.g., bitwise `XOR`). Xeon phi [16] also supports population count and logical operations.

8 CONCLUSION

In this paper, we design and implement APNN-TC that accelerates arbitrary-precision neural networks on Ampere GPU Tensor Cores. Specifically, APNN-TC contains an int1-based emulation design on Tensor Cores to enable arbitrary-precision computation, an efficient AP-Layer design for efficiently mapping NN layers towards Tensor Cores, and an APNN design to minimize the memory access across NN layers. Extensive evaluations on two Ampere GPUs show that APNN-TC can achieve significant speedup over CUTLASS kernels and various mainstream NN models, such as ResNet and VGG.

9 ACKNOWLEDGEMENTS

We thank all anonymous reviewers for their valuable comments. This work was supported in part by NSF 1925717 and 2124039.

REFERENCES

[1] AMD. 2013. AMD Accelerated Parallel Processing OpenCL Programming Guide. http://developer.amd.com/wordpress/media/2013/07/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide-rev-2.7.pdf.

[2] Zhaowei Cai, Xiaodong He, Jian Sun, and Nuno Vasconcelos. 2017. Deep learning with low precision by half-wave gaussian quantization. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 5918–5926.

[3] François Chollet. 2017. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*.

[4] Jack Choquette, Wishwesh Gandhi, Olivier Giroux, Nick Stam, and Ronny Krashinsky. 2021. NVIDIA A100 Tensor Core GPU: Performance and Innovation. *IEEE Micro* 41, 2 (2021), 29–35.

[5] Jack Choquette, Olivier Giroux, and Denis Foley. 2018. Volta: Performance and programmability. *Ieee Micro* 38, 2 (2018), 42–52.

[6] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2015. BinaryConnect: Training Deep Neural Networks with binary weights during propagations. In *NIPS*. 3123–3131.

[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *NAACL-HLT (1)*. Association for Computational Linguistics, 4171–4186.

[8] Nael Fasfous, Manoj Rohit Vemparala, Alexander Frickenstein, Lukas Frickenstein, and Walter Stechele. 2021. BinaryCoP: Binary Neural Network-based COVID-19 Face-Mask Wear and Positioning Predictor on Edge Devices. *CoRR* abs/2102.03456 (2021).

[9] Angelo Garofalo, Giuseppe Tagliavini, Francesco Conti, Davide Rossi, and Luca Benini. 2020. XpulpNN: Accelerating Quantized Neural Networks on RISC-V Processors Through ISA Extensions. In *DATE*. IEEE, 186–191.

[10] Tong Geng, Tianqi Wang, Chunshu Wu, Chen Yang, Wei Wu, Ang Li, and Martin C. Herbordt. 2019. O3BNN: an out-of-order architecture for high-performance binarized neural network inference with fine-grained pruning. In *ICS*. ACM, 461–472.

[11] Song Han, Huizi Mao, and William J. Dally. 2016. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. In *ICLR*.

[12] 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.

[13] Brian Hickmann, Jieasheng Chen, Michael Rotzin, Andrew Yang, Maciej Urbanski, and Sasikanth Avancha. 2020. Intel Nervana Neural Network Processor-T (NNP-T) Fused Floating Point Many-Term Dot Product. In *2020 IEEE 27th Symposium on Computer Arithmetic (ARITH)*. IEEE, 133–136.

[14] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *arXiv e-prints*, Article arXiv:1704.04861 (April 2017), arXiv:1704.04861 pages. arXiv:1704.04861 [cs.CV]

[15] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized neural networks. In *Proceedings of the 30th international conference on neural information processing systems*. Citeseer, 4114–4122.

[16] Intel. 2012. Intel Xeon Phi Coprocessor Instruction Set Architecture Reference Manual. https://software.intel.com /content/dam/develop/external/us/en/documents/327364001en.pdf.

[17] Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*. PMLR, 448–456.

[18] Zhe Jia, Marco Maggioni, Jeffrey Smith, and Daniele Paolo Scarpazza. 2019. Dissecting the NVidia Turing T4 GPU via microbenchmarking. *arXiv preprint arXiv:1903.07486* (2019).

[19] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P Scarpazza. 2018. Dissecting the NVIDIA volta GPU architecture via microbenchmarking. *arXiv preprint arXiv:1804.06826* (2018).

[20] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 1–12.

[21] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems* 25 (2012), 1097–1105.

[22] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (Eds.), Vol. 25. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

[23] Jaeha Kung, David C. Zhang, Gooitzen S. van der Wal, Sek M. Chai, and Saibal Mukhopadhyay. 2018. Efficient Object Detection Using Embedded Binarized Neural Networks. *J. Signal Process. Syst.* 90, 6 (2018), 877–890.

[24] Ang Li, Tong Geng, Tianqi Wang, Martin Herbordt, Shuaiwen Leon Song, and Kevin Barker. 2019. BSTC: A novel binarized-soft-tensor-core design for accelerating bit-based approximated neural nets. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–30.

[25] Ang Li and Simon Su. 2020. Accelerating Binarized Neural Networks via Bit-Tensor-Cores in Turing GPUs. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* (2020).

[26] A. Li and S. Su. 2021. Accelerating Binarized Neural Networks via Bit-Tensor-Cores in Turing GPUs. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 32, 7 (2021), 1878–1891. https://doi.org/10.1109/TPDS.2020.3045828

[27] Ang Li and Simon Su. 2021. Accelerating Binarized Neural Networks via Bit-Tensor-Cores in Turing GPUs. *IEEE Trans. Parallel Distributed Syst.* 32, 7 (2021), 1878–1891.

[28] Ning Liu, Xiaolong Ma, Zhiyuan Xu, Yanzhi Wang, Jian Tang, and Jieping Ye. 2020. AutoCompress: An automatic DNN structured pruning framework for ultra-high compression rates. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 4876–4883.

[29] Xiaolong Ma, Fu-Ming Guo, Wei Niu, Xue Lin, Jian Tang, Kaisheng Ma, Bin Ren, and Yanzhi Wang. 2020. Pconv: The missing but desirable sparsity in dnn weight pruning for real-time execution on mobile devices. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 5117–5124.

[30] Bradley McDanel, Surat Teerapittayanon, and H. T. Kung. 2017. Embedded Binarized Neural Networks. In *EWSN*. Junction Publishing, Canada / ACM, 168–173.

[31] Wei Niu, Pu Zhao, Zheng Zhan, Xue Lin, Yanzhi Wang, and Bin Ren. 2020. Towards Real-Time DNN Inference on Mobile Platforms with Model Pruning and Compiler Optimization. *IJCAI* (2020).

[32] NVIDIA. [n.d.]. CUDA Template Library for Dense Linear Algebra at All Levels and Scales (CUTLASS).

[33] Nvidia. [n.d.]. NVIDIA A100 Tensor Core GPU Architecture. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf.

[34] Nvidia. [n.d.]. NVIDIA TESLA V100 GPU ARCHITECTURE. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[35] NVIDIA. 2021. CUDA Programming Guide: Sub-byte Operations. https://docs.nvidia.com/cuda/cuda-c-programming-guide/#wmma-subbyte.

[36] Eunhyeok Park, Dongyoung Kim, and Sungjoo Yoo. 2018. Energy-Efficient Neural Network Accelerator Based on Outlier-Aware Low-Precision Computation. In *ISCA*. IEEE Computer Society, 688–698.

[37] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems (NIPS) 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d Alch'e-Buc, E. Fox, and R. Garnett (Eds.). http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[38] Md Aamir Raihan, Negar Goli, and Tor M Aamodt. 2019. Modeling deep learning accelerator enabled gpus. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 79–92.

[39] Advanced Grid Research. 2018. Sesor Technologies and Data Analytics. https://www.smartgrid.gov/files/Sensor_Technologies_MYPP_12_19_18_final.pdf.

[40] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. Mobilenetv2: Inverted residuals and linear bottlenecks. In

Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR).

[41] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[42] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *NIPS*. 5998–6008.

[43] Guangting Wang, Chong Luo, Xiaoyan Sun, Zhiwei Xiong, and Wenjun Zeng. 2020. Tracking by Instance Detection: A Meta-Learning Approach. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR).*

[44] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. 2019. HAQ: Hardware-Aware Automated Quantization With Mixed Precision. In *CVPR*. Computer Vision Foundation / IEEE, 8612–8620.

[45] J. Wu, C. Leng, Y. Wang, Q. Hu, and J. Cheng. 2016. Quantized Convolutional Neural Networks for Mobile Devices. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 4820–4828. https://doi.org/10.1109/CVPR.2016.521

[46] Zhaohui Yang, Yunhe Wang, Kai Han, Chunjing Xu, Chao Xu, Dacheng Tao, and Chang Xu. 2020. Searching for Low-Bit Weights in Quantized Neural Networks. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.). https://proceedings.neurips.cc/paper/2020/hash/2a084e55c87b1ebcdaad1f62fdbbac8e-Abstract.html

[47] Dongqing Zhang, Jiaolong Yang, Dongqiangzi Ye, and Gang Hua. 2018. Lq-nets: Learned quantization for highly accurate and compact deep neural networks. In *Proceedings of the European conference on computer vision (ECCV)*. 365–382.

[48] Shiyue Zhang, Benjamin Frey, and Mohit Bansal. 2020. ChrEn: Cherokee-English Machine Translation for Endangered Language Revitalization. In *EMNLP (1)*. Association for Computational Linguistics, 577–595.

[49] Shuchang Zhou, Zekun Ni, Xinyu Zhou, He Wen, Yuxin Wu, and Yuheng Zou. 2016. DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients. *CoRR* abs/1606.06160 (2016).

[50] B. Zhuang, L. Liu, M. Tan, C. Shen, and I. Reid. 2020. Training Quantized Neural Networks With a Full-Precision Auxiliary Module. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 1485–1494. https://doi.org/10.1109/CVPR42600.2020.00156